The Language Reference Manual for Graphr

October 17, 2007

1 Language

1.1 Characters

1.1.1 Escape sequences

The following escape sequences allow special characters to be put into source code.

Escape Sequence	Name	Meaning
\t	Tab	Produces a tab character.
\n	Newline	Produces a new line character
\\	Backslash	Produces a single backslash.
\#	Number sign	Prouduces the number sign character.
\"	Double Quote	Produces a double quote.

1.1.2 Comments

Comments in source code are ignored by the compiler. Multiline comments start with /* and continue until the first */ is encountered.

1.2 Identifiers

1.2.1 Keywords

The following keywords are reserved and can not be used.

for	if	nil	else
foreach	while	return	def
true	false	in	or
and	puts	STDIN	

1.2.2 Variables

Variable names start with a character a-z or A-Z then continue with zero or more of the following characters a-z, A-Z, 0-9, or underscore _. Variable names are case sensitive.

Variables in graphr are dynamically typed. Variable scope is static. Variables definied outside of a function have global scope. Variables defined within a function are accessible in the function, conditional expressions within the function, and any descendent functions.

1.2.3 Numbers

The underlying structure storing numbers has the precision of a double in C. Here are the number of bits and the possible values a number can store.

64 bits 2.2250738585072014 * (10^-308) to 1.7976931348623157 * (10^+308)

Numbers can be assigned to variables using the following syntax.

```
some_integer = 12;
some_decimal = 3.2;
some_hex = #ffffff;
```

Numbers must start with a digit, a number sign, or a decimal. Numbers that start with a digit can be followed by any number of digits and optional decimal point and zero or more digits afterward. Numbers that start with a decimal must have one or more digits following. Numbers that start with a number sign must have six characters following in the range of a-f or 0-9.

1.2.4 Arrays

Arrays are single or multidimensional matricies. They are defined by assinging a set of values seperated by commas and bracketed by [] characters to a variable. Examples:

```
my_array = [2, 4, 2];
my_multi_dimensional_array = [[2, 3, 1], [4, 5]];
my_array_holding_different_types = ["hello world", 23.34, some_variable];
```

Memory for arrays is allocated at run time. They should start with a minimal number of elements and increase in size dynamically. Arrays are zero indexed and elements within an array can be referenced by including an expression that evaluates to a number within the postfix brackets on an array name. Examples:

```
puts my_array[2];
puts my_array[some_function_that_returns_a_number()];
```

1.2.5 Associative Arrays

Associative arrays are arrays that have their elements dereferenced by a hashed value rather than an integer. They are defined by assigning a set of key, value pairs separated by commas and enclosed in curly braces. Each key value pair is specified with the key on the left and the value on the right with the two seperated by =>. Examples:

```
assoc_array = {"a string" => "some value", 3 => "another value"};
```

1.2.6 Constants

Constants are varibles that can be assigned only once. A variable is specified as a constant if all of the letters are capitalized. Numbers and underscores are allowed in constant names after the first character. Example:

MY_CONSTANT = "test";

1.2.7 Strings

Strings are an array of characters surrounded by double quotes. Examples:

my_string = "hello world";
my_string = "\tThis is indented";

Strings can interpolate values into them like in perl. Examples:

my_string = "some val: #{my_variable}";
my_string = "some val returned by a function: #{some_function()}";

Either a variable name or function call can be enclosed in the #{} block in the string. The value returned will be inserted into the string at that location.

1.3 Functions

1.3.1 Function Definition

A function is declared using the def keyword. They are declared like this:

def function_name(parameter-list, ...) { body... }
Funtion_name is the name of the function. Allowable function names are the same as those for variables.

Parameter-list is the list of parameters that the function takes separated by commas. If no parameters are given the parameter list should be defined either with an empty set of parenthesis or no parenthesis at all.

All functions return a value. The value they return is specified by a return statement within the function body or the value returned by the last statement exectued within the function body. A return statement without a value returns the nil value. Examples:

return 3;
return;
return "some string";
return nil;

Functions by at the top level of the script have global scope. Functions defined within functions are only accessible by the parent function.

1.3.2 Program Startup

The entire program is recognized as one expression, which returns the final value computed. All command line arguments are stored withing a global array called **args.**

1.4 Operators

1.4.1 Postfix

Postfix operators are operators that are attached to the end of an expression.

operand++;

After the result is obtained, the value of the operand is incremented by 1 and the subsequent value is returned.

operand–;

After the result is obtained, the value of the operand is decremented by 1 and the subsequent value is returned.

1.4.2 Prefix

Prefix operators are operators that are attached to the beginning of an expression.

!operand

Returns the logical NOT operation on the operand. A true operand returns false, a false operand returns true.

1.4.3 Normal

There are several normal operators which return the result defined for each:

expression1+expression2

The result of this is the sum of the two expressions.

expression1-expression2

The result of this is the difference of the two expressions.

expression1*expression2

The result of this is the product of two expressions.

expression1/expression2

The result of this is the quotient of the two expressions.

expression1%expression2

The result of this is the value of the remainder after dividing expression1 by expression2. Also called the modulo operator.

expression1^expression2

The result of this is the value of expression1 raised to the power expression2.

1.4.4 Boolean

The boolean operators return either **true** or **false**. Everything which does not evaluate to either **false** or **nil** is considered **true**.

expression1&&expression2

Returns the logical AND operation of expression1 and expression2. Can also be written:

expression1 **and** expression2

expression1||expression2

Returns the logical OR operation of expression1 and expression2. Can also be written:

expression1 **or** expression2

expression1<expression2

Returns true if expression1 is less than expression2, otherwise the result is false.

expression1>expression2

Returns true if expression1 is greater than expression2, otherwise the result is false.

expression1<=expression2

Returns true if expression1 is less than or equal to expression2, otherwise the result is false.

expression1>=expression2

Returns true if expression1 is greater than or equal to expression2, otherwise the result is false.

expression1==expression2

Returns true if expression1 is equal to expression2, otherwise the result is false. String equality is based on case sensitive lexical comparison.

expression1!=expression2

Returns true if expression1 is not equal to expression2, otherwise the result is false.

1.4.5 Assingment

An assignment operator stores the value of the right expression into the left expression. All assignment operators return the value of the right expression.

expression1=expression2

The value of expression2 is stored in expression1.

expression1*=expression2

The value of expression1 times expression2 is stored in expression1.

expression1/=expression2

The value of expression1 divided by expression2 is stored in expression1.

expression1%=expression2

The value of the remainder of expression1 divided by expression2 is stored in expression1.

expression1+=expression2

The value of expression1 plus expression2 is stored in expression1.

expression1-=expression2

The value of expression1 minus expression2 is stored in expression1.

expression1||=expression2

The value of expression2 is stored in expression1 if expression1 equals nil.

1.4.6 Precedence

The operators have a set order of precedence. Items inside parenthesis are evaluated first and have the highest precedence. The following chart shows the order of precedence with the items at the top having highest precedence.

Operator	Name
!	Logical NOT
++ –	Incerement/Decrement operators
^	Exponentiation operator
* / %	Multiplicative operators
+ -	Additive operators
< > <= >=	Inequality comparators
== != ||=	Equality comparators
&& and	Logical AND
|| or	Logical OR
+=, -=, ...	Assignment

1.5 Conditional Expressions

All conditional expressions return the value of the last statement executed in the block. This enables the assignment of variables with a conditional expression as the right hand side.

1.5.1 if

The if expression evaluates an expression. If that expression is true, then the expression immediately following is executed. If an else clause is given and if the expression is false, then the else's expression is executed. For multi-line expressions, the expressions can be grouped into a block within { } brackets.

Syntax:

```
if( expression1 ) expression2;
or
if( expression1 ) expression2;
else expression3 ;
```

1.5.2 while

The while statement provides an iterative loop.

Syntax:

```
while( expression1 ) expression2...
```

expression2 is executed repeatedly as long as expression1 is true. The test on expression1 takes place before each execution of expression2.

1.5.3 for

The for statement allows for a controlled loop.

Syntax:

```
for( expression1 ; expression2 ; expression3 ) expression4...
```

expression1 is evaluated before the first iteration. After each iteration, expression3 is evaluated. expression1, expression2 or expression3 can be replaced with nil. If expression2 is nil, it is assumed to be false. expression4 is executed repeatedly until the value of expression2 is false. The test on expression2 occurs before each execution of expression4.

1.5.4 foreach

The foreach statement allows for iterating through an array.

Syntax:

```
foreach(element in myarray) { ... }
foreach(element in expression) { ... }
```

In the above example element represents a new variable that will be in scope of the foreach block. "in" is a keyword seperating the new variable and the expression or variable on the right hand side.

1.5.5 return

The return statement causes the current function to terminate. It can return a value to the calling function. Using the return statement without an expression returns the last evaluated expression. Reaching the } at the end of the function is the same as returning without an expression.

Syntax:

```
return expression;
```

2 Library

The library consists of a number of built in functions to enable input, output, and drawing graphics.

2.1 Graphics

There are a number of built in functions for creating charts. Each function takes an associative array called options. The return value of the function is the resulting canvas, which is just an associative array. This can later be passed to one of the IO functions to create an image for the chart.

In each of the following descriptions the right hand operand is the type the option should be. To the right of that is the information on if it is required. All numbers except for colors represent a number of pixels. Color numbers will be a hexadecimal number that corresponds to the color's hex value.

create_canvas(options);
add_line(options);
add_line_from_points(options);
add_data_point(options);
add_circle(options);
add_rectangle(options);
add_label(options);

2.1.1 create_canvas(options);

Returns a canvas which is the base for a graph.

"width" => Number (required)
"height" => Number (required)
"color" => Number (optional, defaults to #ffffff)

2.1.2 add_line(options);

Adds a line to the canvas passed in the options.

"canvas" => Canvas (required)
"bottom" => Number (required)
"left" => Number (required)
"width" => Number (optional, defaults to 1. this is in pixels)
"length" => Number (required)
"direction" => String (must be either "horizontal" or "vertical", optional, defaults to horizontal)
"color" => Number (optional, defaults to #000000)

2.1.3 add_line_from_points(options);

Adds a line to the canvas with the start and end points.

"canvas" => Canvas (required)
"start_x" => Number (required)
"start_y" => Number (required)
"end_x" => Number (required)
"end_y" => Number (required)
"width" => Number (optional, defaults to 1 pixel)
"color" => Number (optional, defaults to #000000)

2.1.4 add_data_point(options)

Adds a data point at a specific location to the canvas.

"canvas" => Canvas (required)
"x" => Number (required)
"y" => Number (required)
"width" => Number (optional, defaults to 2 pixels)
"color" => Number (optional, defaults to #000000)
"label" => Label (optional)

2.1.5 add_ciricle(options)

Adds a circle to the canvas.

"canvas" => Canvas (required)
"x" => Number (required)
"y" => Number (required)
"radius" => Number (required)
"width" => Number (optional, defaults to 1 pixel)
"label" => Label (optional)
"border_color" => (optional, defaults to #000000)
"background_color" => (optional, defaults to #ffffff)

2.1.6 add_rectangle(options)

Adds a rectangle to the canvas.

"canvas" => Canvas (required)
"top" => Number (required)
"bottom" => Number (required)
"height" => Number (required)
"width" => Number (required)
"border_color" => Number (optional, defaults to #000000)
"background_color" => Number (optional, defaults to #ffffff)
"label" => Label (optional)

2.1.7 add_label(options)

Adds a label to the canvas.

"canvas" => Canvas (required)
"top" => Number (required)
"bottom" => Number (required)
"direction" => String (must be either "horizontal" or "vertical", optional, defaults to horizontal)
"size" => Number (optional, defaults to 10)
"text" => String (required)

2.2 Input Output

The Input Output library contains the functions necessary for reading input and printing output. This gives the user the ability to take in data from sources outside of the source code, such as the standard input or other files. This also allows the user to write data to files and also output png files for graphs.

2.2.1 Functions List:

load_csv(); save_csv();
open();
close();
read_line();
read_char();
read_token();
read_file();
puts();
write_file();
write_image();
include();

2.2.2 load_csv

Declaration:

def load_csv(*filename*);

Reads a csv file named *filename* and loads all the data into a table, returning a reference to the table. It returns nil if the file is not found, the file is unreadable, or if the file is not in csv format.

2.2.3 save_csv

Declaration:

def save_csv(*filename, array*);

Takes an array and writes it in csv format to a file named *filename*. Returns nil if it fails and 1 if it succeeds.

2.2.4 open

Declaration:

def open(*filename*);

Opens a file called *filename*, returns the reference to the file handle if successful. Returns nil if unsuccessful.

2.2.5 close

Declaration:

def close(*filename*);

Closes a *filehandle*, preventing future reads and writes to the file until it is opened again.

2.2.6 read_line

Declaration:

def read_line(*filehandle*);

Returns from the *filehandle* until a newline character and returns a string if successful, a nil if unsuccessful or the end of the file is reached. The filehandle can be STDIN to read from the standard input.

2.2.7 read_char

Declaration:

def read_char(*filehandle*);

Returns from the *filehandle* the next character if succesful and a nil if end of file is reached. The filehandle can be STDIN to read from teh standard input.

2.2.8 read_token

Declaration:

def read_token(*filehandle*);

Returns from the *filehandle* the next token, ignoring whitespace, \t, and \n if successful. Otherwise, returns nil if end of file is reached. The *filehandle* can be STDIN to read from teh standard input.

2.2.9 read_file

Declaration:

def readfile(*filename*);

Reads from filename and returns the entire file in an array with each line of the file as a string in the array if successful. Otherwise, returns nil.

2.2.10 puts

Declaration:

def puts(*string*);

Prints *string* to the standard output. Returns true if successful, otherwise returns nil.

2.2.11 write_file

Declaration:

def write_file(*filehandle, string*);

Returns true if writing string to the filehandle is succesful, otherwise returns nil.

2.2.12 write_image

Declaration:

def write_image(*filename, canvas*);

Creates a png image *filename* using the *canvas*. Returns 1 if successful, and nil otherwise.

2.2.13 include

Declaratoin:

def include(*filename*);

Returns true if *filename* is successfully found. *Filename* should be a Graphr header file or a Graphr source code. Acts as if the entire content of *filename* is copied into the code with brackets around it. If *filename* is not found, returns nil.