YAGL: Yet Another Graphing Language

Language Reference Manual

Edgar Aroutiounian, Jeff Barg, Robert Cohen

July 23, 2014

Abstract

YAGL is a programming language for generating graphs in SVG format from JSON formatted data.

1 Introduction

This manual describes the YAGL programming language as specified by the YAGL team.

2 Lexical Conventions

A YAGL program is written in the 7-bit ASCII character set and is divided into a number of logical lines. A logical line terminates by the token NEWLINE, where a logical line is constructed from one or more physical lines. A physical line is a sequence of ASCII characters that are terminated by an end-of-line character. All blank lines are ignored entirely.

2.1 Tokens

Besides the NEWLINE token, the following tokens also exist: NAME, INTEGER, STRING, and OPERATOR, where a LITERAL is either a String literal or an Integer literal.

2.2 Comments

Comments are introduced by the '#' character and last until they encounter a NEWLINE token, comments do not nest.

2.3 Identifiers

An identifier is a sequence of ASCII letters including underscores where upper and lower case count as different letters; ASCII digits may not be included in an identifier.

2.4 Keywords

The following is an enumeration of all the keywords required by a YAGL implementation: 'if', 'elif', 'for', 'in', 'break', 'func', 'else', 'return', 'continue', 'print', 'while'

2.5 String Literals

YAGL string literals begin with a double-quote (") followed by a finite sequence of ASCII characters and close with a double-quote ("). YAGL strings do not recognize escape sequences. An example of a YAGL string literal is String Bar = "Hello World". YAGL strings are not iterable and hence may not be used in the declaration of a for loop.

2.6 Integer Literals

YAGL supports plain integer literals. Integer literals are the only type of numeric literal recognized by the language, not that numeric literals do not include a sign. YAGL integer literals do not recognize prefixes that are commonly found in other languages such as b'123'. An example of a YAGL integer literal is: Int foo = 123

2.7 Operators

The following tokens are operators: '+', '-', '*', '/', '>', '<', '<=', '>=', '==', '%', '='

3 Meaning of Identifiers

Identifiers either refer to functions, builtin Objects or user defined variables.

3.1 Basic Types

YAGL features four builtin data types, String, Array, Integer, and Dictionary. String represents string objects, Arrays represent an ordered sequence of Integers or Dictionaries. Arrays must have all elements of the same type. Dictionaries represent an implementation of a hashtable and are mainly used as a YAGL container for JSON data. Dictionaries may have only Strings as keys but their values may be either Strings, Arrays, Integers or other Dictionaries. Arrays and Dictionaries are iterable and hence may be used in the declaration of a for loop.

4 Data Model

As YAGL's primarily purpose is a language to programmatically create SVGs (scalable vector graphics), it makes sense to have just one global "SVG" Object. This is similar in spirit to EmcaScript's global "this" object which represents the host environment. Although end users in YAGL may not get a handle on the "SVG" Object, they may interact with it using builtin library functions such as title, addRect, addCircle, etc. This data model simplifies the end user's experience by allowing them to focus on their algorithmic manipulation of their data. A call to makeGraph along with initial parameters executes the code from top to bottom. All code written after a call to makeGraph is not executed.

5 Expressions

The precedence of expression operators follows the regular laws of mathematics aside from division, which returns integer division as YAGL only features integers as its native numeric type.

6 Array References

An array identifier followed by a set of square brackets enclosing an integer literal to denote the index denotes array indexing, i.e. myArray[4]. Indexing out of bounds in an array causes a compile time error.

7 Dictionary References

A Dictionary identifier followed by a set of square brackets enclosing a string literal performs a lookup. Performing a lookup on a Dictionary where the key does not exist returns -1, else it returns the value associated with the key.

8 Function Calls

A function call is a postfix expression which is performed by the identifier of the function followed by a possibly empty set of parentheses. Functions may return an explicit value to their caller if they have a return expression defined in their body, else they return 0.

9 Operators

9.1 Multiplicative Operators

The multiplicator operators *, /, division by 0 causes an exception.

9.2 Additive Operators

The additive operators +, - group left to right where + denotes addition and - denotes subtraction

9.3 Relational Operators

The relational operators group left to right and return back 1 if the operator evaluates to true and 0 if the operator evaluates to false.

9.4 Equality Operators

The equality operator == is only valid for either Integer or String types and returns 1 if the operands are equal, 0 otherwise.

9.5 Logical And

The && operators groups left to right and returns 1 if both its operands compare unequal to zero with 9 otherwise, logical and is only defined for integers.

9.6 Logical Or

The || operators group left to right and returns 1 if either of its operands compares unequal to zero and 0 otherwise.

9.7 Assignment Expressions

There is only one assignment operator, =. The equals operator accepts a type declaration along with a NAME token for its left operand and an expression for its right operand.

9.8 Comma Operator

A pair of expressions separated by a comma is strictly evaluated left-to-right and the value of the left expression is discarded.

10 Type Specifiers

The type specifiers are: 'Array', 'Dict', 'Int', 'String' They are used in declaring variables, see Assignment Expressions.

11 Function Declarators

A YAGL function has the form:

```
func functionName (argument_list)
  {
      #some code
      #optional return expression
  }
```

a function returns any primitive type: Array, Int, Dict, or String. Execution of function code terminates upon flow of execution reaching the return statement, at which point the return

expression evaluated value is returned to the caller. If no return statement is present, then the function returns 0 without need of an explicit return declaration. The parameter list is a comma-separated series of identifiers with type names (for example, (int a, int b, int c). The parameter list can also be empty and just be an empty set of parenthesis.

12 Scope

12.1 Lexical Scope

Identifiers are placed into non-intersecting namespaces. The two namespaces are functions and file level. The lexical scope of an object or function identifier that appears in a block begins at the end of its declarator and persists to the end of the block in which it appears. The scope of a parameter of a function begins at the start of the function block and extends to the end. If an identifier is reused at the head of a block or as a function parameter, any other declaration of the identifier is shadowed until the end of the block or function.

13 Statements

13.1 Compound Statements

Compound statements are a series of statements enclosed in braces that define their own lexical block.

13.2 Iteration Statements

For loops iterate through Arrays or Dictionaries. The type specifier is given as a parameter in the for loop. If the array contains no objects of the specified type, the loop simply does not execute. For an array, the variable specified will be bound to each of the elements of the array sequentially and the loop statement will run. For a Dictionary, the string key will be bound to the variable specified in the for loop; moreover, the only allowed type for the iterated variable is String. Example Code:

```
for (String a in myDict)
{
      #body code
```

```
}


for (Int i in myArray)
{
       #body code
}
```

While loops execute after their boolean expression evaluates to 1 and continue onward until their test condition evaluates to 0.

13.3 Break and Continue

The break keyword may only be used within the body of a *for* loop or a *while* loop. The break prematurely ends execution of the *for* or *while* loop. Analogously, the continue keyword may only be used within the body of a *for* or *while* loop and it signals the flow of control to move onto the next item in iteration.

13.4 If Statements

In an *if* statement, the expression is evaluated, including side-effects, and *if* the result is not 0, the first substatement is executed. If it is equal to 0, the else substatement is executed if it exists.

14 Print Statements

The print keyword takes an expression, evaluates it, then prints the result to standard output. If the expression evaluates to an int, it prints a string representation of the int, a string will output the ascii representation of the string, a dict will print "Dict(%i)" where %i would be replaced by the number of key-object pairs in the Dictionary, and similarly Array would print "Array(%i)" where %i is the number of elements in the array.

15 Example Code

Below is an example program written in the YAGL language.

```
# This is an example of a YAGL program.
# There is one global object always available, the Graph object that you
# don't have a handle on, but will manipulate with makeGraph
Array myJson = jsonArray("/path/to/json/data.json")

#Could also do jsonDict which returns Dict
func createGraph()
{
   for (Dict item in myJson)
   {  # Assuming everything in item["foo"] is string and adding a rect to the global singlet
svg object.
      addRect(item["xCoord"], item["yCoord"], item["height"], item["width"])
      #OR!
      addCircle(item["cx"], item["cy"], item["r"]
   }  #Builtin function that provides a title to the global singleton svg object.
   title("YAET, Yet Another Example Title")
}

createGraph()
# Need to call makeGraph for the graph to be actually made
makeGraph(<nameOfFile>, <width>, <height>)
```

16 Grammar

This is the comprehensive Grammar for the YAGL programming language. The grammar has terminal symbols NAME, INTEGER, STRING, NEWLINE, OPERATORs, which includes '+', '-', '/', '*', '%', and PUNCTUATION which includes '(', ')', '{', and '}'. Note that * after a regular expression denotes zero or more instances of the expression, + denotes one or more instances and ? denotes one or zero.

```
type_spec: Array | Dict | Int | String
flow_stmt: break_stmt | continue_stmt | return_stmt
break_stmt: 'break'
```

```
continue_stmt: 'continue'

return_stmt: 'return' [expr]

func_definition: 'func' NAME parameters '{' suite '}'

suite: simple_stmt | compound_stmt

simple_stmt: (expr_stmt | print_stmt) NEWLINE

expr_stmt: asn_stmt, NEWLINE | expr

asn_stmt: type_specifier, NAME '=' expr

expr: add_expr | mult_expr | div_expr | sub_expr | neg_int | mod_expr | INTEGER

func_expr: NAME parameters

add_expr: expr '+' expr

mod_expr: expr '%' expr

mult_expr: expr '*' expr

div_expr: expr '/' expr

sub_expr: expr '-' expr

neg_int: '-' INTEGER

print_stmt: 'print' (NAME | STRING | INTEGER | expr)

compound_stmt: func_definition | ( if_stmt | while_stmt | for_stmt | simple_stmt )+

if_stmt: 'if' '(' bool_expr ')' suite ('elif' suite)* ('else' suite)?

while_stmt: 'while' '(' bool_expr ')' suite

for_stmt: 'for' '(' type_spec NAME 'in' NAME ')' '{' suite '}'

bool_expr: 1 | 0 | logic_and | logic_or

logic_and: expr '&&' expr

logic_or: expr '||' expr

comp_op: '<' | '>' | '==' | '>=' | '<=' | '!='

parameters: '(' [args_list] ')'

args_list: (type_spec NAME)*
```