YAGL: Yet Another Graphics Language

Final Report

Edgar Aroutiounian, Jeff Barg, Robert Cohen

August 14, 2014

Abstract

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

1 The Proposal

1.1 Description of YAGL

YAGL is a new programming language for constructing graphics, with the intent of allowing programmers to construct graphics with various different output formats.

The language features a static type system with a void type that can conditionally downcast to the other primitives (Int, Array, etc..). Types are indicated by the type name declarator followed by the variable name. All types are objects. Each object holds an enumeration for what its type is, along with the underlying data. There is no string type. There are integer and array literals.

There are limited control structures: an iterated for loop, an if statement and break. A for loop can iterate on Stream and Array types and will yield a void type as the iterated item which needs to be downcast.

1.2 What Does YAGL Solve?

YAGL allows users to take JSON formatted data and produce clear and concise graphs. The properties of these graphs (such as color) can be manipulated utilizing algorithms. The output of the YAGL program can be determined by the user (SVG, ASCII, or PDF). YAGL is essentially and clean and minimal language for producing graphs quickly and easily.

1.3 Examples of YAGL Syntax

YAGL will have a void type, which cannot be operated upon. The void type can be conditionally downcast using the syntax:

```
if let b = a as Int {
      # scope of b is only in this function
}
```

This allows us to safely operate on JSON streams.

YAGL does not have a string type. In order to pipe in filenames to open JSON files, the open open keyword takes in $[n] which specifies the *nth* command line parameter. *open* constructs a Stream.

```
Stream s = open $0
```

YAGL has an iterated for loop and array literals:


```
Stream s = open $0
```

YAGL has an iterated for loop and array literals:


```
for a in [[4,1,5], 3, 5, [2, 3]] {
      if (let int_a = a as Int) {
      }
      elif (let int_a = a as Array) { }
}
```

Table 1 YAGL PRIMITIVES

KEYWORDS	OPERATORS	TYPES
if	++	Void
elif	+	Array
for	-	Canvas
in	*	Int
break	/	Stream
let		
in		
func		

2 Language Tutorial

2.1 Example YAGL Program

```
func makeGraph(Array: param1)
{
	Canvas myGraph = new Canvas()
	# 0 is ascii art graph, 1 is SVG meant for a browser, 2 is PDF?
	myGraph.graphType = 0
	# Opens up command line argument, presumably JSON.
	Stream s = open $0
	for (item in s)
	{	# Say there are only 3 enums of types
		# 0 is integer, 1 is array, 2 is canvas Object
		if (let integer_item = item as Int)
		{
			myGraph.addRectAllParams(item)
		}
		elif (let array_item = item as Array)
		{
			myGraph.addRect(item[0], item[1], item[2], item[3])
```

```
            }
        }
}

# Notice Last item is a nested Array
myData = [3, 4, 5, 1, [3, 1, 5, 6]]
build makeGraph(myData)

## Standard Library - preview graph with predefined parameters,
spits out ascii art graph to standard output preview(data)
```

2.2 More 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 gm[``xCoord'
      #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>)
```

3 Language Reference Manual

3.1 Introduction

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

3.2 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 SEMI, 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 a semi-colon character.

3.2.1 Tokens

A logical line may consist of the following tokens assuming correct syntax is used:
SEMI, LPAREN, RPAREN, LBRACE, RBRACE, COMMA, COLON, IN, PLUS, MINUS, TIMES, DIVIDE, ASSIGN, FUNC, EQ, NEQ, LT, LEQ, GT, GEQ, RETURN, IF, ELSE, FOR, WHILE, INTLITERAL, INT, DICT, ARRAY, STRING, ID, STRINGLITERAL, EOF.

3.2.2 Comments

Comments are introduced by the '#' character and last until they encounter the next NEW-LINE token.

3.2.3 Identifiers

An identifier is a sequence of ASCII alphatic letters and the underscore character where upper and lower case are distinct; ASCII digits may not be included in an identifier. Identifiers must start with a lowercase letter.

3.2.4 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‘. Identifiers cannot have the same name as keywords.

3.2.5 String Literals

YAGL string literals begin with a double-quote () followed by a finite sequence of non-double-quote 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.

3.2.6 Integer Literals

YAGL supports integer literals. An integer literal means any sequence of digits that doesnt lead with a 0. Integer literals are in base 10. Integer literals are the only type of numeric literal recognized by the language. Integer literals can only be positive. Additionally, integer literals will use the system implementation of integer (so either 16, 32, or 64 bits). Any integer literal larger than the system int limit is not legal syntax and will not compile.

3.2.7 Operators

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

3.3 Meaning of Identifiers

Identifiers either refer to functions, builtin Objects or user defined variables. Identifiers must start with a lowercase letter.

3.3.1 Basic Types

YAGL features four built-in 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 key-value storage system and are mainly used as a YAGL container for JSON data. Dictionaries may have only Strings as keys and their values are integers. Arrays and Dictionaries are iterable and hence may be used in the declaration of a for loop.

3.4 Data Model

As YAGLs 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 EmcaScripts 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 users 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.

3.5 Expressions

The precedence of expression operators will first prioritize function calls, then parentheses, then multiplication/division/modulo, then addition and subtraction.

3.6 Array References

An array identifier followed by a set of square brackets enclosing an integer value to denote the index denotes array indexing, i.e. myArray[4]. Array indexing returns an integer value. Indexing out of bounds in an array causes a compile time error. Arrays start at index 0, and a legal index is any index from 0 to the number of elements in the array minus 1.

3.7 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 integer value associated with the key.

3.8 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. return is a statement that takes an expression, evaluates it, then returns it as the value of the expression where the function is called.

3.9 Operators

3.9.1 Multiplicative Operators

The multiplicator operators *, /, division by 0 will return 0. Division will also truncate decimal parts of numbers to the nearest integer lower than the mathematical division.

3.9.2 Additive Operators

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

3.9.3 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.

3.9.4 Equality Operators

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

3.9.5 Logical And

The && operators groups left to right and returns 1 if both its operands compare unequal to zero with 0 otherwise, logical and is only defined for integers.

3.9.6 Logical Or

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

3.9.7 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.

3.9.8 Comma Operator

A pair of expressions separated bys 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.

3.10 Scope

3.10.1 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.

3.10.2 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. The order of the dictionary iteration is left up to implementation details. The order of an array is from index 0 to the last index in the array.

While loops execute after theinalogously, 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.

3.10.3 If Statements

In an *if* statement, the expression is evaluated, including side-effects, and łsymbols NAME, INTEGER, STRING, NEWLINE, OPERATORs, which includes +, -, /, *,

```
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
exefinition | ( 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)*
```

4 Project Plan

4.1 Plan for YAGL Developing

Table 2 YAGL Development Calendar

Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Proposal, Plan	Parser	Syntactic Analysis	Bytecode	Compiler	Test Suite and Final Project

5 Architectural Design

Wut?

6 Test Plan

7 Lessons Learned

The following lessons were learned during the course of this project:

1. Start earlier!
2. Finish earlier.
3. Utilize a better project management suite than TODO.txt

8 Complete Listing of Code

Why?