MatchaScript: Language Reference Manual
Programming Languages & Translators
Spring 2017

Language Guru: Kimberly Hou - kjh2146
Systems Architect: Rebecca Mahany - rlm2175
Manager: Jordi Orbay - jao2154
Tester: Ruijia Yang - ry2277

Table of Contents

Introduction

MatchaScript is a general-purpose statically typed programming language that is convenient for both imperative and functional programming. It will be compiled to LLVM using our OCaml compiler, and will from LLVM be compiled to native code.

Features include:

- Optimized for event-driven programming
- Garbage collection, implemented similarly to OCaml
- No type inference
- Lexical scoping

Lexical Structure

Tokens in MatchaScript include identifiers, keywords, constants, literals, operators, and separators. Tokens are separated by whitespace (blanks, tabs, and newlines) or comments.

Comments in MatchaScript are C-style comments, beginning with /* and terminating with */.

Identifiers are sequences of letters and, optionally, one or more digits or underscores. Identifiers must begin with a letter. CamelCase is suggested.

The following are reserved keywords in MatchaScript:

For declarations, *const* is reserved to declare constants, and *fun* to declare functions.

The following are reserved for primitives: *int, float, double, char, String, Boolean*, and *null*.

The following are reserved for control flow: *while, do/while, for, for/in, for/each, if/else if/else, switch, try/catch/finally, continue, break*.

The keywords *class* and *constructor* are reserved for implementing classes.

The keyword *let* is reserved for block scoping.

The keyword *log* is used for print statements in MatchaScript.

Unlike JavaScript, semicolons are mandatory for all MatchaScript statements.

A simple sample program with this structure can be seen below:

```
/****
Similar to JavaScript's console.log(), log.nl() adds a newline at the end of its print
statement while log() does not.
****/

function void helloWorld() {
	log.nl("Hello World!"); // prints "Hello World!"
	String world = "World.";
	log("Goodbye " + world); // prints "Goodbye World."
}
```

Types, Values, and Variables

Primitives - strict type system

Name	*Description*
int	Integer
char	Character
String	Sequence of characters surrounded by ¡ and !
Boolean	Boolean
float	Single-precision floating point number

double	Double-precision floating point number
Matrix	Matrix of int, float, or double.
null	Nullable type; All variables are automatically assigned to it when not explicitly assigned

We will implement the **const** keyword for constant variables.

Expressions

Expressions in MatchaScript are assignments, function declarations, and function calls.

The following is a list of all operators in MatchaScript, in their order of precedence:

Operator	*Operation*
++	Pre- or post-increment
--	Pre- or post-decrement
-	Negate number
+	Convert to number
~	Invert bits
!	Invert Boolean value
delete	Remove a property
typeof	Determine type of operand
void	Return undefined value
*, /, %	Multiply, divide, modulo
+, -	Add, subtract
+	Concatenate strings
<<	Shift left
>>	Shift right with sign extension
>>>	Shift right with zero extension
<, <=, >, >=	Numeric or alphabetic comparison

instanceof	Returns object class
in	Test existence of property
==	Test strict equality
!=	Test strict inequality
&	Compute bitwise AND
^	Compute bitwise XOR
|	Compute bitwise OR
&&	Compute logical AND
||	Compute logical OR
?:	Choose second or third operand
=	Assignment
*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=	Operate and assign
eval	Evaluate or execute the argument(s)

For arithmetic expressions, MatchaScript, like JavaScript, will attempt to convert operands to numbers; if they cannot, they will be converted to NaNs.

Notes on differences from JavaScript:

- We will not implement the comma operator.
- We implement strict equality and inequality only (=== and !== in JavaScript), but for simplicity's sake they will be implemented using the following operators: == and !=.
- We will implement eval() as an operator, rather than a function.

Statements

For the most part, MatchaScript syntax for statements is similar to JavaScript syntax.

Conditional Statements

if	Completes block of statements when a boolean expression is evaluated as TRUE
else if	Follows a preceding if when the if is false. Acts

	similar to an if.
else	Follows a preceding if when the if is false. Completes block of statements as long as preceding if is false.
switch	Completes one of multiple statements based off of the value of a particular evaluation
OHMSS	Completes block of statement if the reserved variable OHMSS is set to true. Used normally for testing and print statements.

Loop Structures

while	Completes an entire block of statements until a boolean expression is evaluated as FALSE
for	A while loop that also increments a designated variables that may or may not be used in the boolean expression

Error Structures

try	A structure that may or may not throw an exception
catch	A structure that may or may not handle particular exceptions. Multiple catch clauses can be used.
finally	A structure that always executes when the try block exits. Useful for mandatory clean up procedures

Will not be implemented in MachaScript:

with	With blocks allow a series of statements to be performed on a specified object. Will not be used because it is prone to misuse.
use strict	Blocks that do not allow the use of undeclared variables. Will not be used because is rarely helpful to anyone.

As stated previously: unlike in JavaScript, semicolons are mandatory for all MatchaScript statements.

Functions

Function Declaration

A function returns to its caller by either *return;* or *return <type>;* , the first case being valid if the function return type is void and the function is to end early, and the second being valid otherwise.

Functions are declared using the *function* keyword. They may also be assigned to variables using the *fun* keyword, or they can be anonymous functions. Below is an example of a simple function call. Note that the void keyword is similar to C-based languages unlike JavaScript.

```
/*********/
function void add(int x, int y) {
	int z = x + y;
	log.nl(z); // this is the same as log.nl(x + y);
}

function void main() {
	add(x,y);
}
/*********/
```

Functions can be assigned to variables using the *fun* keyword.

```
/*********/
function int factorial(int x) {
	if (x === 1) {
		return x;
	} else {
		return x * factorial(x - 1);
	}
}

function void main() {
	fun f = factorial(10);
	log.nl(f); // prints 3628800
}
/*********/
```

Closures

MatchaScript supports closures, meaning when functions are nested within each other, the inner function has access to variables in the outer function. Note that the inner function stores references to these outer variables, not actual values.

Closures are implemented by

```
/*********/
function String myName(String firstName) {
	String intro = "My name is ";

	function String mySurname(String lastName) {
		return intro + firstName + " " + lastName;
	}

	return lastName;
}

function void main() {
	fun theName = myName("Stephen"); // outer function returns
	log.nl(theName("Edwards")); // closure (inner function) returns; statement
//prints "My name is Stephen Edwards"
}
/*********/
```

Currying

Currying and higher-order functions are supported, allowing for detailed custom functions. Below, note the use of anonymous functions which may be utilized as well.

```
/*********/
fun motto = function fun(String statement) {
	return function fun(String name) {
		return function void(String punctuation) {
			log.nl(statement + ", " + name + punctuation);
		};
	};
};

fun pokemonMotto = motto("Gotta catch 'em all");
pokemonMotto("Ash")("!"); // prints "Gotta catch 'em all, Ash!"

fun edisonQuote = motto("Vision without execution is hallucination")("Thomas");
edisonQuote("?"); // prints "Vision without execution is hallucination, Thomas?"
```

```
/*********/
```

Classes

As MatchaScript is statically typed, the language implements classical OOP inheritance. This is a departure from JavaScript, which uses prototype-based inheritance. Since variables are loosely typed in JavaScript, if it cannot find attributes for a given object in its own definition, it will fall back on each of its prototypes and look at their attributes to match which properties an object the user declared can be assigned to without error. This fallback mechanism is one of the defining characteristics of prototypical inheritance, where objects with loosely defined types can morph into other types at runtime. For the purposes of a statically typed language, much of the power of prototypical inheritance would be nullified because all variables have to be of one type throughout the runtime of a program. Thus, a classical OOP inheritance model is better suited for MatchaScript. Below is an example of class creation:

```
class Animal {
  constructor(String name, int weight) {
       this.name = name;
    this.weight = weight;
  }
}

class BunnyRabbit extends Animal {
       constructor(String name, int weight, String ears) {
              super(name, weight);
              this.ears = ears;
       }
       function int getWeight() {
              return this.weight;
       }
       function void setWeight(int newWeight) {
              this.weight = newWeight;
       }
       function String getName() {
              return this.name;
       }
       function String getEars() {
              return this.ears;
       }
       function void setEars(String newEars) {
              this.ears = newEars;
       }
}

BunnyRabbit bugsBunny = new BunnyRabbit("Bugs Bunny", 15, "Floppy");
```

Note that even though the syntax resembles the ES2015 syntactic sugar keyword set to implement classes, the underlying mechanism for MatchaScript inheritance is still object-oriented based, not prototypical.