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

Project Proposal: MatchaScript

Summary

MatchaScript is a general-purpose, statically typed, compiled programming language based on JavaScript.

- *general-purpose*: Like JavaScript, MatchaScript is a programming language that is applicable to a wide range of domains and is not optimized for a particular kind of program or application.
- *statically typed*: All types contained within a program have to be explicitly determined and checked at compile time.

As an interpreted language, JavaScript requires a runtime environment. We are experimenting with LLVM's support for JIT compilation, but for now, we plan to implement static compilation of MatchaScript to LLVM.

Syntactically, MatchaScript resembles JavaScript and C-based languages. While it is an imperative language, MatchaScript, like JavaScript, can also be declarative in practice for certain situations, such as use of currying and higher-order functions. This ability to be both imperative and declarative is one of the characteristics that defines JavaScript; MatchaScript hopes to preserve that characteristic.

While JavaScript contains massive capabilities for both client-side and server-side scripting, we will not implement components and functions specifically for the browser. Rather, our goal is that by linking open-source GUI libraries to OpenGL, we may give other developers the capability to manipulate GUI components through leveraging MatchaScript and perhaps building additional libraries in the future.

Purpose

As MatchaScript is a general-purpose programming language, there is no specific application domain the language is optimized for. The goal is that MatchaScript applications may be used to solve problems from a wide range of fields.

That being said, MatchaScript, like JavaScript, can be useful for server-side scripting. There will be support for working with text, arrays, dates, and regular expressions. And despite its lack of browser integration, MatchaScript can still be used to write programs like desktop widgets. https://en.wikipedia.org/wiki/JavaScript#Uses_outside_Web_pages provides examples of applications for JavaScript as a non-browser scripting language.

Parts of MatchaScript

The following provides a brief overview of what we plan to implement in MatchaScript and highlights our departures from JavaScript:

- Lexical Structure
    - Case-sensitivity, whitespace, and comment style are the same as C and Java.
    - Unlike JavaScript, semicolons are mandatory for all MatchaScript statements.
- Types, Values, and Variables
    - Primitives - strict type system
        - Integer - **int**
        - Float - **float**
        - String - **String**
        - Boolean - **Boolean**
        - Null / Undefined - **null**
    - When variables are declared but not yet explicitly assigned, they are automatically assigned a null value. There is no undefined type in MatchaScript.
    - We will implement the **const** keyword for constant variables.
- Expressions
    - Operators, as well as their order of precedence, are mostly the same as in JavaScript.
    - We will implement **===** and **!==** (strict equality and inequality), but not **==** and **!=**.
    - We will implement **eval()** as an operator, rather than a function.
    - We will not implement the comma operator.
    - For arithmetic expressions, MatchaScript, like JavaScript, will attempt to convert operands to numbers; if they cannot, they will be converted to NaNs.
- Statements
    - For the most part, MatchaScript syntax for statements is similar to JavaScript syntax.
    - We will implement a lot of traditional control flow elements from JavaScript (**while, do/while, for, for/in, for/each, if/else if/else, switch, try/catch/finally**), but we will not implement jump statements.
    - There will be support for multiple **catch** clauses.
    - Unlike in JavaScript, **with** cannot be used to temporarily define or extend scope; **with**, as it is prone to misuse, will not be included in MatchaScript.
    - **use strict** will not be implemented in MatchaScript.
- Scope
    - JavaScript has two kinds of scope: global and function. Later, JavaScript introduced the **let** keyword to achieve block scoping. We plan to implement block scoping (as the lack of it was a common complaint about JavaScript), but possibly in a different way in order to maximize intuitiveness and ease of use.
- Arrays

    - Arrays are typed.
        - To initialize an empty array:
            - **<type>[] <name> = new <type>[];**
        - To initialize an array of a fixed size:
            - **<type>[] <name> = new <type>[](<size>);**
    - Built-in array functions include **push()** and **pop()**.
- Functions
    - Functions are seen as variables.
    - Function declarations and function expressions are supported with the following syntax:
        - Function declaration:
            - **function <return type> <name> (<parameter(s) type and name>)**
        - Anonymous function/function expression:
            - **function <return type>(<parameter type and name>)**
    - We will eliminate arrow function expressions.
- Classes and Inheritance
    - Departing from JavaScript, we will implement MatchaScript based on classical OOP. Because MatchaScript is *not* dynamically typed, JavaScript's prototype-based inheritance would not be an appropriate choice.
- RegExp
    - We will implement regex searching/matching via string methods like **search()**, **replace()**, **match()**, and **split()**, but we will not implement the RegExp object.
- Garbage Collection
    - If we decide to develop a runtime environment for MochaScript, then we will integrate our own implementation of garbage collection with LLVM for our runtime environment.
    - However, if we compile to LLVM completely statically, then there will be no garbage collection, and heap-allocated objects will have to be explicitly freed.

Example Program

A simple but interesting program in MatchaScript involves a mix of dynamic programming and nested currying to present a solution to the given problem below.

```
/****
Problem: Given an mxn matrix, have the algorithm first count all possible paths from
the top left to bottom right where only right, down, and diagonally right / down
movements are allowed from one cell to another. Then guess the number of ways and see
how it compares with the answer.
****/

function fun count(int m) {
    return function fun(int n) {
        return function String(int x) {
            int i, j;
```

```
            // initialize 2D matrix
            int[] paths = new int[](m);
            for (i = 0; i < m; i++) {
                paths[i] = new int[](n);
            }

            for (i = 0; i < m; i++) {
                paths[i][0] = 1;
            }

            for (i = 0; i < n; i++) {
                paths[0][i] = 1;
            }

            /* Example: Build a 4x3 matrix from the bottom up
              | 1 | 1 | 1 | 1 |
              | 1 |   |   |   |
              | 1 |   |   |   |
            */

            for (i = 1; i < m; i++) {
                for (j = 1; j < n; j++) {
                    paths[i][j] = paths[i-1][j] + paths[i][j-1] + paths[i-1][j-1];
                }
            }

            /* Each cell now shows how many ways one can get to said cell
              | 1 | 1 |  1 |  1 |
              | 1 | 3 |  5 |  7 |
              | 1 | 5 | 13 | 25 |
            */

            if (paths[m-1][n-1] > x) {
                return "There are more than " + x + " ways to traverse this maze."
            } else if (paths[m-1][n-1] < x) {
                return "There are fewer than " + x + " ways to traverse this maze."
            } else {
                return "Bingo! There are " + x + " ways to traverse this maze."
            }

        };
    };

}

function void main() {
    fun guessWays = count(4)(3);
    log.nl(guessWays(20)); // prints "There are more than 20 ways to traverse this
maze."
}
```