Marble Project Proposal

Huaxuan Gao (hg2579), Qiwen Luo (ql2427), Yixin Pan (yp2601), Xindi Xu (xx2391)

1. Introduction

Marble is a programming language that incorporates matrix manipulation functionalities natively so that the compiled code can solve linear algebra problems efficiently. With standard library classes Image and Pixels, it can process images swiftly as well. This programming language would be useful in applications such as Computer Vision and Robotics.

With Marble, developers can define matrices using Matlab-like `[]` literal syntax, i.e. `M = [0,0,0;0,0,0]`, as well as generator functions, i.e. `M = Matrix.zeros(2,3)`, to create a 2-by-3 matrix with all 0's. We'll include a bare minimum number of matrix manipulation functions in the language to speed up compiling. This language is flexible - developers can add methods to any class. Developers can extend the Matrix class and define their own methods which can be used later by Matrix objects. Marble also includes syntactic sugar to make reading and writing matrix-related code more natural. For example, `A^T` is the same as `A.transpose()`, which means "A transpose".

Due to the time constraint, our language will deploy C libraries for accessing the file system and reading/displaying images.

2. Language Details

2.1 Data Types

2.1.1 Primitive types

- int, float, boolean, null

2.1.2 Built-in types (non-primitive types)

- String
- Matrix: implemented using array of arrays

Java 复制代码

```
// when the parser sees [], we'll create a matrix
A = [1,2,3,4;5,6,7,8];
// equivalent to:
A = [
  1,2,3,4;
  5,6,7,8;
];
// Initialize matrix with generator function
Matrix.zero(5,5);
```

2.2 Comments

- Single-line comment

```Java
// This is a comment
print("Hello World");
```

- Multi-line comment

```Java
/* The code below will print the words Hello World
to the screen */
print("Hello World");
```

2.3 Object and Class

2.3.1 Every object is an instance of Object

In this language, all objects are instances of the Object class. Thus, they inherit methods defined in the Object class.

- `Object.typeOf(a)` checks the type of the object

It returns the type of the variable `a`. If `a` is a primitive type, it will return the exact type, as a string, i.e. “boolean”. If a is an object, it will return "object".

- `Object.classOf(a)` checks the class of the object

It returns the name of the class which `a` instantiated from as a string. For example, if `a` is an object of `DirectedGraph`, it will return "DirectedGraph". Suppose `DirectedGraph` is a subclass of `Graph` `Object.classOf(a)` will return "DirectedGraph".

If `a` is a primitive type, it will return "Object" since all objects are an instance of the Object class.

- Other methods in the `Object` class

`Object.toString()`: override this method to provide a custom string to print to the screen upon calling `print(a)`

2.3.2 Define a Class

Developers can declare a class using the `class` keyword. All class is a subclass of Object by default.

- Constructor:

Classes can have multiple constructors but their function signature should be different (overloading). If a class doesn't have a constructor, it will use the constructor from the superclass.

- Variables & methods:

Classes can have class or instances variables and methods.

```java
class Filter {
    // instance variable
  Matrix filter;

  constructor(Matrix m){
    self.filter = m;
  }

  // define getter and setter
  get matrix(){
    return self.filter;
    }

    // instance method
    apply(Matrix img){
        return img * self.filter;
    }
}
```

2.3.3 Extends a class

One can add methods to a class by doing the following. In this way, developers can easily add instances variables and methods to a class. Note that this programming language doesn't have static/class variables or methods.

```java
Matrix.eigenvalues = function (self) {
  double values = [];
  // compute eigenvalues
  return values;
}

// later
Matrix a = [1,2,3; 4,5,6];
double values = a.eigenvalues();
```

2.4 Functions

2.4.1 Function is an Object (First-class Function)

In this language, functions are treated like any other variables. Thus, a function can be passed as an argument to other functions, can be returned by another function, and can be assigned as a value to a variable.

All functions are instances of the Function class. Thus, their type is "object" i.e. `Object.typeOf(foo) == "object"` and their class is Function, i.e. `Object.classOf(foo) == "Function"`.

Example (assign a function to a variable):

```java
// assign an anonymous function in a variable addOne
Function addOne = function (int x) {
  return x+1;
}

// invoke the function we defined
add();
```

Example (pass a function as an argument):

```java
function map(int[] array, Function fn){
  for(int i = 0; i < array.length; i++){
    array[i] = fn(array[i]);
  }
}

// pass addOne as an argument
map([1,2,3,4,5], addOne);
// result: [2,3,4,5,6]
```

Example (return a function):

```java
function greeting(string message){
  return func (string name){
    print(message, name);
  }
}

Function sayHi = greeting(“Hi”);
sayHi(“PLT”);
// prints “Hi, PLT”
```

2.4.2 Function returns null by default

Return values for the functions are optional. When a function doesn't have a “return” statement, it returns `null` by default.

Example:

```java
function foo() {
  print(“foo”)
}

// `foo` doesn’t have an explicit return type, so it returns `null` by default.
// It is okay to assign `null` to a variable with a specific type
// i.e. int. Here `a` is null.
int a = foo()
```

2.4.3 `main` function

When a file is being compiled, the compiler will first look for a function with the name `main`. It will only read the code written in the `main` function and other code related to those codes. Regardless of the position of the `main` function, this function will be called first.

```java
function main() {
  print("Execution starts here.");
}
```

2.5 Loops

2.5.1 While-loop

```java
int i = 0;
while(i < 10){
  i = i + 1;
}
```

2.5.2 For-loop

```java
int n = 1;
for(int i = 0; i < 10; i = i + 1){
  n = n * 10;
}
```

2.6 Operators

2.6.1 Assignment Operator

The equal sign `=` is used to indicate storing values in variables.

Example:

`int x = 1;`

2.6.2 Arithmetic Operator

The following standard arithmetic operators are provided:

- addition `+`, subtraction and negation `-`, multiplication `*`, division `/`, modular `%`

The resulting type will depend on the operands. When the operands are integer and double type, the result will be automatically cast to a double type.

Example:

`1/3 will evaluates to 0`

`1/3.0 will evaluates to 0.3333`

2.6.3 Comparison Operators

The following comparison operators are provided:

- greater than `>`, less than `<`, greater than or equal to `>=`, less than or equal to `<=`, equal to `==`, not equal `!=`

Note that the comparison operators will be performed on the values (not the reference address) of the operands.

2.6.4 Logical Operators

The following logical operators are provided:

- negate `!`, and `&&`, or `||`

2.7 Keywords & Separators

The following keywords will be reserved. If used as a variable name, the compiler will throw an error indicating that the keyword cannot be used in variable assignments.

- `if`, `elif`, `else`: Reserved for conditional statements.
- `for`, `while`, `break`, `continue`: Reserved for flow control.
- `main`: Used to indicate the starting point to execute the program.
- `return`: Reserved for functions return statements.
- `null`: Evaluates to false when used as boolean.
- `function`, `class`, `constructor`: Reserved for function/class/constructor declarations.
- `try`, `catch`, `throw`: Reserved for exception handling.
- `import`, `export`: Reserved for modules, check 2.11 for details.
- Separators: `( ) [ ] { } , ;`

2.8 Memory

2.8.1 Call–by–value for primitive types

Call–by–value example:

```
class Example{
  function swap(a, b){
    c = a;
    a = b;
    b = c;
  }
}
a = 1;
b = 2;
Example.swap(a,b);
a
> 1
b
> 2
```

2.8.2 Call-by-reference for non-primitive types

Call-by-reference example:

```
class Example{
  function swap(a, b){
    c = a;
    a = b;
    b = c;
  }
}

a = [1,2,3];
b = [2,5,6];
Example.swap(a,b);
a
> [2,5,6]
b
> [2,5,6]
```

2.9 Scope

We choose to use static scoping in our language since we want to facilitate modular coding. In this scoping, a variable always refers to its top top-level environment. For example,

```java
class Example{
  function meth(){
    int a = 0;
  }
  function meth2(){
    int b = a + 2; // Invalid since no "a" declared in meth2's top level env
  }
}
```

```java
class Example{
  int a = 10;
  function meth(){
    return a;
  }
  function meth2(){
    int a = 20;
    return meth();
  }
  function main(){
    print(meth2()); // Output is 10
  }
}
```

2.10 Modules

A program can use pre-defined functions and classes defined in other files, these files are called modules.

2.10.1 export

A module can define which functions or classes will be visible to other modules by the export keyword.

`export functionName;`

`export className;`

Note that if a function is defined inside a class, it cannot be export alone without the class.

2.10.2 import

A module can use the functions and classes exported by other modules.

`import functionName from moduleName;`

`import className from moduleName;`

`import * from moduleName;`

3. Code Samples

Java 复制代码

```
import readImage, showImage from Image;

image = readImage("path/to/file");
blur = [
 0, 0, 0, 0, 0;
 0, 1, 1, 1, 0;
 0, 1, 1, 1, 0;
 0, 1, 1, 1, 0;
 0, 0, 0, 0, 0;
];

blur = [Color(100,200,100), Color(100,200,100);];

// apply blur filter to image
blurred_image = image * blur;

// print out the image
showImage(blurred_image);
```