Language Manual

1. Introduction
2. Language Details
   - 2.1 Data Types
     - 2.1.1 Primitive types (call by value)
     - 2.1.2 Built-in types (call by reference, non-primitive types)
   - 2.2 Comments
     - 2.2.1 Single-line comment
     - 2.2.2 Multi-line comment
   - 2.3 Program
   - 2.4 Functions
     - 2.4.1 Define a Function
     - 2.4.2 Function Return Statements
     - 2.4.3 main Function
   - 2.5 Variables
     - 2.5.1 Define a Variable
   - 2.6 Loops
     - 2.6.1 While-loop
     - 2.6.2 For-loop
   - 2.7 If-elif-else
   - 2.8 Operators
     - 2.8.1 Assignment Operator
     - 2.8.2 Arithmetic Operator
     - 2.8.3 Comparison Operators
     - 2.8.4 Logical Operators
   - 2.9 Keywords & Separators
   - 2.10 Memory
     - 2.10.1 Call-by-value for primitive types
     - 2.10.2 Call-by-reference for non-primitive types

2.11 Scope

3. Code Sample

Manager: Xindi Xu (xx2391)
Language Guru: Qiwen Luo (ql2427)
System Architect: Huaxuan Gao (hg2579)
Tester: Yixin Pan (yp2601)

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 = 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.

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

In order to accomplish certain operations and functions efficiently, we create the following primitive types as building blocks (each of which contains simple values of a kind).

2.1.1 Primitive types (call by value)

- `int`: Integer under a range of `-2^30 to 2^30 -1`
- `float`: OCaml float type (IEEE 754 with a 53-bit mantissa and exponents from -1022 to 1023)

- `boolean`: `true/false`
- `null`: type of variables after declaration and before assignment, type of defined functions, type of variables assigned to functions without return statements

2.1.2 Built-in types (call by reference, non-primitive types)

- `matrix`:
  - accessor: `a[1][0]`
  - dimension: `rows(a) cols(a)`
  - initialization:
    - `matrix A = [1,2;3,4];`
    - `mat_init()`
  - All entries in the matrix need to be `float`. If entries are integers, we will cast them to floats

Java 复制代码

```
// two ways to initialize a matrix
matrix A = [1,2;3,4];
matrix B = mat_init(1,2,1.0); // [0.0, 0.0]

// access an element in the matrix
A[1][0]; // returns 3

// getting the matrix dimension, number of rows, number of cols
rows(A); // returns 2
cols(A); // returns 2
```

2.2 Comments

2.2.1 Single-line comment

The content after the symbol `//` within a line is recognized as a comment in our language and our interpreter will skip the content during the execution.

Java 复制代码

```
// This is a comment
```

2.2.2 Multi-line comment

Any content after `/*` and before `*/` is recognized as a comment in our language and our interpreter will skip the content during the execution.

```
/*
This is also a comment
*/
```

2.3 Program

When developers write code in Marble, the file that contains the code is a Marble program. A program consists of a collection of function declarations, variable declarations, and one and only one `main()`.

- Function declarations and variable declarations are optional and can be in any order before the `main()` function
- One `main()` function is required for every program and it should be at the end of the file

```
matrix numbers;
function get(matrix m, int row, int col){
    return m[row][col];
}
main(){
    numbers = [1,2,3,4;5,6,7,8];
    return get(numbers, 0, 1); // 2
}
```

2.4 Functions

2.4.1 Define a Function

A function is a collection of input parameters and statements. A function declaration creates one function and binds the corresponding identifier to it.

- A function must have a name; adding parenthesis `()` to the end of its name will invoke the function
- Input parameters are optional and multiple input parameters are separated by commas `,`. Each input parameter must have a type and a name
- Inside the curly braces `{}` is a collection of 0 or more statements
- A function can have 0 or more `return` statements
- Within the same scope, functions must have different names

```java
function fib(int n){
    if(n == 1){
        return 1;
    }
    if(n == 2){
        return 2;
    }
    return fib(n-1) + fib(n-2);
}
```

2.4.2 Function Return Statements

- Once any `return` statement is executed, the function will terminate
- The value returned by the function will be available in the context where the function is invoked
- Return values for the functions are optional. A function without `return` statements returns `null` by default

Example:

```java
function get(matrix m, int row, int col){
    return m[row][col];
}
function set(matrix m, int row, int col, int val){
    m[row][col] = val;
}
matrix m = [1,2;1,2];
float a = get(m,0,0); // a = 1
boolean b = set(m,1,1,2); // b = null
```

2.4.3 `main` Function

`main` function is a special type of function. In particular, its name must be "main" and it lacks input parameters.

- `main` function must be at the end of the program and one program must have one `main` function.
- Return statements are allowed in the `main` function and they will terminate the program. The value returned from the `main` function is useless

Code will start executing from `main`.

```java
main(){
    // ...
}
```

2.5 Variables

2.5.1 Define a Variable

A variable has a type, a name, and an optional value. A variable declaration creates one variable, binds corresponding identifiers to it, and gives it a type and an initial value.

- One variable can only have one type in its lifetime. There's no way to change its type. A runtime error will be thrown if the variable and the value it is assigned to have mismatched types. See the "2.1 Data Types" section for more details.
- A variable declared without assigning an initial value will have a `null` value. Variables can be reassigned later in the program
- Variables must be declared before assigning a value to it or before using it
- Variable declarations end with a semicolon `;`
- Within the same scope, variables must have different names

```java
// declare a variable i with type i and assign 0 to it
int i = 0;
// declare a variable j with type int without assigning initial value
// j should be null
int j;

// assign a new value 1 to i
i = 1;
// assign a new value 1.5 to j
// Run-time error: type mismatch
j = 1.5;
```

2.6 Loops

2.6.1 While-loop

The format for while-loop is `while(expr){stmts}`. The expression is the condition part of the loop. The expression is of type boolean and the type check will be done during runtime. The loop-body is a statement list.

Example:

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

2.6.2 For-loop

The format for for-loop is `for(assigstmts;expr;expr){stmts}`. The assignment statement is the init part, such as `int i = 0`, `i = 0`, `i += 1` or `i -= 1`. The expression with type boolean is the condition part and the type check will be done during runtime. The part after the condition part is also an expression, which will be executed after each iteration. The loop-body is a statement list.

Example:

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

2.7 If-elif-else

The format is `if(expr){stmts}elif(expr){stmts}else{stmts}`.
The if-branch is required and can only have one. The elif-branch is optional and can have multiple elif. The else-branch is optional and can have zero or one else.

Example:

```java
if (a<=10) {
  // ...
}
elif (a<=20) {
  // ...
}
else {
    // ...
}
```

2.8 Operators

2.8.1 Assignment Operator

The equal sign `=` is used to indicate storing values in variables with the format `type ID = expr;` or `ID = expr;`. Type checking will be done during the runtime.
We also support `+=, -=` and the syntax is `expr += expr;` or `expr -= expr;`. This shortcut is only available for int and float.
If a variable is assigned a value before the declaration, the error will be caught during the complilation.

Example:
```
int x = 1;
x += 2;
```

2.8.2 Arithmetic Operator

The following standard arithmetic operators are provided (only applies to int/float):

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

2.8.3 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 `!=`
- ref equal `~=`

All comparison operators, except ref equal, will be performed on the values of the operands, not the reference addresses.
Non-primitive types can only use ref equal.

2.8.4 Logical Operators

The following logical operators are provided:

- negate `!`
- and `&&`

- or `||`

Only boolean expressions are allowed. Any other expressions will cause runtime errors.

2.9 Keywords & Separators

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

Keywords	Format	Remarks
`if, elif, else`	`if(expr){stmts}elif(expr){stmts}else{stmts}`	Reserved for conditional statements
`for, while`	`for(assignstmt;expr;expr){stmts}` `while(expr){stmts}`	Reserved for flow control
`main`	`main(){stmts}`	main function is used to indicate the starting point to execute the program
`function`	`function foo(){}`	Reserved for functions
`return`	`return expr;`	Reserved for function return statements
`null`		Evaluates to `false` when used as a boolean
`int`, `float`, `boolean`, `matrix`	`type ID;`	Built-in datatypes
`( ) [ ] { } , ;`		separators

2.10 Memory

2.10.1 Call-by-value for primitive types

Call-by-value example:

```java
function swap(int a, int b){
    int c = a;
    a = b;
    b = c;
}

int a = 1;
int b = 2;

main(){
    swap(a,b); // a = 1, b = 2
}
```

2.10.2 Call-by-reference for non-primitive types

Call-by-reference example:

```java
function swap(matrix a, matrix b){
    matrix c = a;
    a = b;
    b = c;
}


matrix a = [3,2,1;4,5,6];
matrix b = [1,2,3;2,5,6];

main(){
    // both a and b are pointing to the same matrix [1,2,3;2,5,6]
    swap(a, b);
}
```

2.11 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.

Example:

Java 复制代码

```
function meth(){
    int a = 0;
}
function meth2(){
    // Invalid since "a" is not declared in meth2's scope
    int b = a + 2;
}
```

Java 复制代码

```
int a = 10;
function meth(){
    return a;
}
function meth2(){
    int a = 20;
    return meth();
}
main(){
    return meth2(); // 10
}
```

3. Code Sample

Java 复制代码

```
function get(matrix mat, int r, int c){
    return mat[r][c];
}

function set(matrix mat, int r, int c, int val){
    mat[r][c] = val;
}

function mult(matrix mat, matrix m2){
    // initialize a matrix with same dimension
    matrix res = mat_init(rows(m1), cols(m2), 0);
    for(int i=0; i<rows(m1); i+=1){
        for(int j=0; j<cols(m2); j+=1){
            for(int k=0; k<rows(m2); k+=1){
                res[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
    return res;
}

main(){
    matrix m1 = [1,2,3;4,5,6];
    matrix m2 = [1,2,3;4,5,6];
    matrix res = mult(m1, m2);
    return res;
}
```