# MUA解释器第二段实验报告

3130000240 陈泓宇

## 一、实验要求

### 1)基本数据类型value

数字number，单词word，列表list，布尔bool

1. 数字的字面量以[0~9]或'-'开头，不区分整数，浮点数
2. 单词的字面量以双引号"开头，不含空格，采用Unicode编码。在"后的任何内容，直到空格（包括空格、tab和回车）为止的字符都是这个单词的一部分，包括其中可能有的"和[]等符号
3. 列表的字面量以方括号[]包含，其中的元素以空格分隔；元素可是任意类型；元素类型可不一致

### 2)基本操作

基本形式：操作名 参数

操作名是一个不含空格的词，与参数间以空格分隔。参数可以有多个，多个参数间以空格分隔。每个操作所需的参数数量是确定的，所以不需要括号或语句结束符号。有的操作有返回值，有的没有。

一个程序就是操作的序列。

**基本操作有**：

1. `//`：注释
2. `make <word> <value>`： 将value绑定到word上。基本操作的单词不能用做这里的word。绑定后的word称作名字，位于命名空间。
3. `thing <word>`：返回word所绑定的值
4. `:<word>`：与thing相同
5. `erase <word>`：清除word所绑定的值
6. `isname <word>`：返回word是否是一个名字，true/false
7. `print <value>`：输出value
8. `read`：返回一个从标准输入读取的数字或单词
9. `readlinst`：返回一个从标准输入读取的一行，构成一个列表，行中每个以空格分隔的部分是list的一个元素

**运算符operator**

1. `add`, `sub`, `mul`, `div`, `mod`：`<operator> <number> <number>`
2. `eq`, `gt`, `lt`：`<operator> <number|word> <number|word>`
3. `and`, `or`：`<operator> <bool> <bool>`
4. `not`：`not <bool>`
5. `repeat <number> <list>`：运行list中的代码number次

### 3)函数定义和调用

**定义**

make <word> [<list1> <list2>]

*word为函数名*

*list1为参数列表*

*list2为操作列表*

*调用*

<functionName> <arglist>

*<functionName>为make中定义的函数名，不需要双引号"*

*<arglist>是参数列表，<arglist>中的值和函数定义时的<list1>中名字进行一一对应绑定*

### 4)函数相关的操作

\* `output <value>`：设定value为返回给调用者的值，但是不停止执行

\* `stop`：停止执行

### 5)表达式计算

允许使用以下运算符对数字进行计算：

+-\*/%()

## 接口与类

### 1）接口

Fragment

public interface Fragment {  
 boolean hasNextInstruction();  
 Value nextRawInstruction() throws MuaExceptions;  
}

Executable

public interface Executable {  
 Value execute(Context context) throws MuaExceptions, Function.FunctionStop;  
 boolean needsMoreArguments();  
 void addArgument(Value argument) throws MuaExceptions.InvalidArgumentTypeException;  
 Executable clone();  
}

Context

继承自Fragment

public interface Context extends Fragment

读取

读取下一句，并判断输入指令的类型并返回为Value类型的值（可以直接区分NumberValue和WordValue, ListValue先保留不动等到运行时处理）

default Value nextInstruction(Fragment fragment)  
 throws MuaExceptions, Function.FunctionStop {  
 Value instruction = fragment.nextRawInstruction();  
 if (instruction instanceof NumberValue)  
 return instruction;  
 else if (instruction instanceof WordValue) {  
 String instructionStr = instruction.toString();  
 if (instructionStr.startsWith(":")) {  
 return getSymbol(instructionStr.substring(1));  
 } else if (Expression.*isExpression*(instructionStr)) {  
 return Expression.*evaluate*(this, (WordValue) instruction);  
 } else if (instructionStr.contains("\"")) {  
 return new WordValue(instructionStr.substring(1));  
 } else if (isExecutable(instructionStr)) {  
 return (Value) getExecutable(instructionStr);  
 } else {  
 throw new MuaExceptions.UnknownOperatorException(instructionStr);  
 }  
 } else {  
 return instruction;  
 }  
}

Run函数

将读取的指令转换成操作符压入临时栈，并判断是否可执行。如果可执行则执行，如果不能执行则读取下一行指令

default void run(Fragment fragment) throws MuaExceptions, Function.FunctionStop {  
 Stack<Executable> opStack = new Stack<>();  
 while (fragment.hasNextInstruction()) {  
 Executable op = (Executable) nextInstruction(fragment);  
 opStack.push(op);  
 while (!opStack.isEmpty()) {  
 // If the operator needs no more arguments, it is finished  
 // and will be popped from the stack, and add to the argument  
 // list of the last layer, or set it the root.  
 while (!opStack.isEmpty() && !opStack.peek().needsMoreArguments()) {  
 Executable op2 = opStack.pop();  
 Value result = op2.execute(this);  
 if (!opStack.isEmpty())  
 opStack.peek().addArgument(result);  
 }  
 if (opStack.isEmpty())  
 break;  
 // Executable op2 = opStack.peek();  
 Value nextInstruction = nextInstruction(fragment);  
 if (nextInstruction instanceof Executable)  
 opStack.push((Executable) nextInstruction);  
 else  
 opStack.peek().addArgument(nextInstruction);  
 }  
 }  
}