MUA最终设计报告

目录

第一阶段实现功能

数据类型

四种基本类型： `word`，`number`，`bool` 和 `list`：

- `number`：数字的字面量以[0~9]或'-'开头，不区分整数，浮点数

```
>>> 23.0
23.0
>>> 23
23
```

- `word`：字的字面量以双引号"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字的一部分，包括其中可能有的"和[]等符号。

```
>>> "hello
hello
>>> "[jfsioj]][[[
[jfsioj]][[[
```

- `bool`：字面量为 `false` 或 `true`：

```
>>> true
true
>>> false
false
>>>
```

- `list`：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。默认在 `print` 和在命令行输入时显示 `[` 和 `]`。

```
>>> [print "a [any thing]]
[print "a [any thing]]
```

- 数字和布尔都是字的特例，因此，作为字面量，`"12` 和 `12` 等价，`"true` 和 `true` 等价

```
>>> eq "12 12
true
>>> eq "true true
true
```

基本操作

- `make <word> <value>`： 将value绑定到word上，这里的word必须以字母开头。基本操作的字不能用做这里的word。绑定后的word称作名字，位于命名空间。

```
>>> make "a 233
>>> make "b true
>>> make "c ["hello]
>>> :a
233
>>> :b
true
>>> :c
["hello]
```

- `thing <word>`：返回word所绑定的值:

```
>>> thing "a
233
>>> thing "b
```

- `:<word>`：与thing相同

```
>>> make "c [a b c]
>>> :c
[a b c]
```

- `erase <word>`：清除word所绑定的值

```
>>> make "a 23
>>> erase "a
>>> isname "a
false
```

- `isname <word>`：返回word是否是一个名字，true/false

```
>>> make "a 23
>>> isname "a
true
```

- `print <value>`：输出value。输出 `list` 时默认加 []

```
>>> print [a b c]
[a b c]
>>> print "hello
hello
```

- `read`：返回一个从标准输入读取的数字或字。为了支持以数字开头的 `word` 输入，要求输入 `word` 时必须使用

```
>>> make "a read
hello
>>> make "b read
233
>>> :a
hello
>>> :b
233
```

- 运算符
  - `add`, `sub`, `mul`, `div`, `mod`： `<operator> <number> <number>`

```
>>> add 123 345
468.0
>>> sub 123 234
-111.0
>>> mul 123 321
39483.0
>>> div 50 2
25.0
>>> mod 50 23
4.0
```

  - `eq`, `gt`, `lt`： `<operator> <number|word> <number|word>`

```
>>> eq "hi "hi
true
>>> eq 23 "23
true
>>> gt "hi "hello
true
>>> lt 23 24
true
```

  - `and`, `or`： `<operator> <bool> <bool>`

```
>>> and true false
false
>>> and true true
true
>>> or true false
true
>>> or false false
false
>>> or eq "23 "24 eq 23 23
true
```

  - `not`： `not <bool>`

```
>>> not eq "good "bad
true
>>> not true
false
```

第二阶段实现功能

函数定义和调用

定义

```
make <word> [<list1> <list2>]
    word为函数名
    list1为参数表
    list2为操作表
```

```
>>> make "test [
... [a b]
... [print add :a :b]
... ]
```

调用

```
<functionName> <arglist>
    <functionName>为make中定义的函数名，不需要双引号"
    <arglist>是参数表，<arglist>中的值和函数定义时的<list1>中名字进行一一对应绑定
```

```
>>> test 23 "24
47.0
>>>
>>> test -1 23
22.0
```

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

```
>>> make "test [
... [a b]
... [make "c add :a :b
... output :c]
... ]
>>> print test 24 25
49.0
```

- `stop`： 停止执行

```
>>> make "test [
... []
... [print "a
... stop
... print "b]
... ]
>>> test
a
```

- `export`： 将本地make的值输出到全局

```
>>> make "test [
... []
... [make "a "hello
... export "a]
... ]
>>> isname "a
false
>>> test
>>> isname "a
true
>>> :a
hello
```

类型判断

- `isnumber <value>`： 返回value是否是数字

```
>>> isnumber 23
true
>>> isnumber "23
true
>>> isnumber "hello
false
```

- `isword <value>`：返回value是否是字

```
>>> isword 23
true
>>> isword true
true
>>> isword [hi]
false
```

- `islist <value>`：返回value是否是表

```
>>> islist "hello
false
>>> islist 23
false
>>> islist [yes]
true
```

- `isbool <value>`：返回value是否是布尔量

```
>>> isbool true
true
>>> isbool "false
true
>>> isbool "hello
false
```

- `isempty <word|list>`：返回word或list是否是空

```
>>> isempty []
true
>>> isempty [not]
false
>>> isempty "
true
>>> isempty 23
false
```

数值计算

- `random <number>`： 返回[0,number)的一个随机数

```
>>> random 3
2.0
>>> random 3
0.0
>>> random 3
1.0
>>> random 3
1.0
```

- `sqrt <number>`： 返回number的平方根

```
>>> sqrt 25
5.0
>>> sqrt 3
1.7320508075688772
>>> sqrt 2
1.4142135623730951
```

- `int <number>`: floor the int

```
>>> int 1.4
1.0
>>> int -2.4
-3.0
>>> int 1
1.0
```

其他

- `readlist`： 返回一个从标准输入读取的一行，构成一个表，行中每个以空格分隔的部分是list的一个元素

```
>>> make "c readlist
a 23 true [hello 23]
>>> :c
[a 23 true [hello 23]]
```

- `repeat` 运行list中的代码number次：

```
>>> repeat 5 [print :a make "a add :a 1.0]
0.0
1.0
2.0
3.0
4.0
```

第三阶段实现功能

字表处理

- `word <word> <word|number|bool>`：将两个word合并为一个word，第二个值可以是word、number或bool

```
>>> word "hello "yes
helloyes
>>> word "hello true
```

- `if <bool> <list1> <list2>`：如果bool为真，则执行list1，否则执行list2。list均可以为空表

```
>>> if eq 23 23 [
...     print "hello
... ] [
...     print "hi
... ]
hello
>>>
>>> if eq 23 24 [
...     print "hello
... ] [
...     print "hi
... ]
hi
```

- `sentence <value1> <value2>`：将value1和value2合并成一个表，两个值的元素并列，value1的在value2的前面

```
>>> sentence [1 2] [3 4]
[1 2 3 4]
>>> sentence 1 [2 3]
[1 2 3]
>>> sentence [[1]] [2 3]
[[1] 2 3]
```

- `list <value1> <value2>`：将两个值合并为一个表，如果值为表，则不打开这个表

```
>>> list [1 2] [3 4]
[[1 2] [3 4]]
>>> list "hello "hi
[hello hi]
```

- `join <list> <value>`：将value作为list的最后一个元素加入到list中（如果value是表，则整个value成为表的最后一个元素）

```
>>> join [1 2] 3
[1 2 3]
>>> join [1 2] [3 4]
[1 2 [3 4]]
```

- `first <word|list>`：返回word的第一个字符，或list的第一个元素

```
>>> first [1 2 3]
1
>>> first "hello
h
```

- `last <word|list>`：返回word的最后一个字符，list的最后一个元素

```
>>> last [1 2 3]
3
>>> last "hello
o
```

- `butfirst <word|list>`：返回除第一个元素外剩下的表，或除第一个字符外剩下的字

```
>>> butfirst [1 2 3]
[2 3]
>>> butfirst "hello
ello
```

- `butlast <word|list>`：返回除最后一个元素外剩下的表，或除最后一个字符外剩下的字

```
>>> butlast [1 2 3]
[1 2]
>>> butlast "hello
hell
```

其他操作

- `wait <number>`：等待number个ms

```
>>> wait 1000
>>> wait 2000
```

- `save <word>`：保存当前命名空间在word文件中

```
>>> poall
a
b
>>> save "ns.txt
```

- `load <word>`：从word文件中装载内容，加入当前命名空间

```
[MUA Interpreter | Zhixuan Lin]
[Welcome, and enjoy.]
>>> poall
>>> load "ns.txt
>>> poall
a
b
```

- `erall`：清除当前命名空间的全部内容
- `poall`：列出当前命名空间的全部名字

```
>>> make "a "hi
>>> make "b "hello
>>> poall
a
b
>>> erall
>>> poall
```

```
>>> make "func [
... []
... [make "c 23
... make "d 24
... poall
... erall
... poall
... ]]
>>> func
c
d
```

既有名字

系统提供了一些常用的量，或可以由其他操作实现但是常用的操作，作为固有的名字。这些名字是可以被删除（erase）的。

- `pi`：3.14159
- `run <list>`：运行list中的代码

```
>>> :pi
3.1415926535
>>> run [print "hello]
hello
```

设计思想

MUA解释器在**实现**上（并非语言语义上，但也有部分重合）有两个核心概念：

- 对象（object）
- 命名空间/域（scope）

对象

在解释器层面上对象共有六种:

- `number`
- `word`
- `list`
- `bool`
- `none`
- `expr`

基础的对象类型有 `number`, `word`, `list`, `bool` 。所有的操作符在内部都是是 `expr` 对象。 `none` 是语义上无返回类型的 `expr` 的返回值，不能作为参数。

Scope/域

域（Scope）在解释器实现中定义为名（name）到对象（object）的映射。这个映射的定义域在程序运行过程中可变。域在运行时会动态创建或删除。多个域可以在在程序运行中共存。除global scope外，任何一个scope都有其lexical enclosing scope。实际上在MUA中只有两种（而不是两个）域，global和function。第三阶段增加了一个builtin域，用来存放mua内置的名字，作为global的lexical enclosing scope。

域是一个运行时概念。在程序运行的任何一个时刻，都有且只有一个当前域。

解释器

解释器在接收到一条语句时，按后缀表达式方式并对其进行evaluation。evaluation有两个语义：

- 返回一个 `object` （可以是 none）
- 修改当前scope的内容

如果这个 `object` 是基础类型，evaluation直接输出其字符串表达。解释器顺序接收 `object` 并顺序进行evaluation，就构成了程序执行的概念。

设计实现

代码结构：

```
├── MUA.java
└── lib
    ├── Bool.java
    ├── Expr.java
    ├── Func.java
    ├── List.java
    ├── MuaObject.java
    ├── None.java
    ├── Number.java
    ├── Scope.java
```

```
├── Word.java
├── error
|   ├── ArgError.java
|   ├── ArithmeticError.java
|   ├── IOError.java
|   ├── IndexError.java
|   ├── InputError.java
|   ├── MuaError.java
|   ├── NameError.java
|   ├── SyntaxError.java
|   └── TypeError.java
├── operation
|   ├── OpButLast.java
|   ├── OpButfirst.java
|   ├── OpErall.java
|   ├── OpErase.java
|   ├── OpExport.java
|   ├── OpFirst.java
|   ├── OpIf.java
|   ├── OpInt.java
|   ├── OpIsbool.java
|   ├── OpIsempty.java
|   ├── OpIslist.java
|   ├── OpIsname.java
|   ├── OpIsnumber.java
|   ├── OpIsword.java
|   ├── OpJoin.java
|   ├── OpLast.java
|   ├── OpList.java
```

```
|   ├── OpLoad.java
|   ├── OpMake.java
|   ├── OpOutput.java
|   ├── OpPoall.java
|   ├── OpPrint.java
|   ├── OpRandom.java
|   ├── OpRead.java
|   ├── OpReadlist.java
|   ├── OpRepeat.java
|   ├── OpSave.java
|   ├── OpSentence.java
|   ├── OpSqrt.java
|   ├── OpStop.java
|   ├── OpThing.java
|   ├── OpWait.java
|   ├── OpWord.java
|   └── operator
|       ├── OpAdd.java
|       ├── OpAnd.java
|       ├── OpDiv.java
|       ├── OpEq.java
|       ├── OpGt.java
|       ├── OpLt.java
|       ├── OpMod.java
|       ├── OpMul.java
|       ├── OpNot.java
|       ├── OpOr.java
|       └── OpSub.java
└── util
```

```
        ├── ArgUtil.java
        ├── Interpreter.java
        ├── OpRun.java
        ├── ParserUtil.java
        └── RunUtil.java
```

`MusObject` 抽象类与基础类型

`MuaObject` 是所有类型和操作的父类。类接口定义:

```
// method implementation omitted
abstract public class MuaObject {

    // type name
    abstract public String getTypeString();

    // get object value
    abstract public Object getValue();

    // string representation
    @Override
    abstract public String toString();
    public Scope enclosingScope = null;
}
```

基础类型实现了 `MuaObject` 所有的接口。例如， `List` 的定义:

```
public class List extends MuaObject {
    public List(ArrayList<MuaObject> list) {
```

```
        this.value = list;
    }

    @Override
    public String toString() {
        String content = "";
        boolean first = true;
        for (MuaObject v: value) {
            content += (first ? "" : " ") +
v.toString();
            first = false;
        }
        return "[" + content + "]";
    }

    @Override
    public String getTypeString() {
        return "list";
    }

    @Override
    public ArrayList<MuaObject> getValue() {
        return value;
    }

    private ArrayList<MuaObject> value;
}
```

其他基础类型的定义类似。需要注意的是，在用户层面上实际上只有 word 和 list 两种类型。 number 和 bool 是解释器的内部类型。

Scope 类

Scope 类可以看作是一个简单的hash表。具体的存储很自然的用 HashMap 实现。对外接口如下：

```
// method implementation omitted
public class Scope {
    enum Type {
        GLOBAL,
        FUNCTION,
        BULITIN // new builtin scope
    }

    // by default create a global scope
    public Scope() {
        this("global", Type.GLOBAL, null);
    }

    // create a function scope
    public Scope(String name, Type type, Scope
enclosing) {}

    public String getScopeName();

    public Type getScopeType();
```

```
    public Scope getEnclosingScope();

    public void addName(Word name, MuaObject
value);

    public MuaObject getName(Word name) throws
NameError;

    public void removeName(Word name) throws
NameError;

    public boolean hasName(Word name);
    public void setReturnValue(MuaObject o);

    public MuaObject getReturnValue();

    private String scopeName = "global";
    private Type scopeType = Type.GLOBAL;
    private Scope enclosingScope = null;
    private MuaObject returnValue = new None();
    private HashMap<String, MuaObject> scope =
new HashMap<>();
}
```

Expr 类与操作定义

`Expr` 类继承自 `MuaObject`，是所有操作的父类。接口如下：

```
// method implementation omitted
abstract public class Expr extends MuaObject {

    @Override
    public String getTypeString() {
        return "expr";
    }

    // get operation name
    abstract public String getOpName();

    // evaluation
    public MuaObject eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception;
    abstract public int getArgNum();

    @Override
    public Expr getValue();

    @Override
    public String toString();

}
```

`Expr` 类最核心的method就是 `eval` 。仔细看 `eval` 的接口。

```
// evaluation
public MuaObject eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception;
```

任何操作都要override这个method，并做两件事

- `eval` 的结果返回一个 `MuaObject` 。对于没有返回类型的操作，返回值为 `None` 。
- 根据当前 `Scope` 和操作语义，修改 `Scope` 。

以 `Make` 的 `eval` 实现为例，可以看到一般操作的 `eval` 实现：

- 检查参数类型
- 得到参数
- 根据语义修改当前scope，返回值

```
@Override
public None eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
    super.eval(scope, arglist);
    // check argument types
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    Word word = (Word) arglist.get(0);
    if
(!Character.isLetter(word.getValue().charAt(0))
)
        throw new SyntaxError("<word> in make
must start with a letter");
    MuaObject value = arglist.get(1);
    scope.addName(word, value);

    return new None();
}
```

解释器实现

由于所有操作也看作是`MuaObject`，使解释器的实现十分简明。由于语法定义的后缀表达式，使用stack来实现是非常自然的选择。核心是`evalObj`和`reduceObj`函数

```
public static MuaObject
evalObj(ArrayList<String> tokens, Scope scope)
throws Exception {
    // do evaluation
    Stack<MuaObject> opStack = new Stack<>();
    for (int i = tokens.size() - 1; i >= 0; i--
) {
        reduceObj(tokens.get(i), opStack,
scope);
    }
    if (opStack.size() != 1) {
        throw new SyntaxError("more than one
statement per line");
    }
    return  opStack.pop();
}
```

`reduce` 函数实际执行的操作是解析传入的token。若是字面量，则将对应的 `MuaObject` 压栈。若是一个操作名，则按参数数量 pop，evaluation，并压栈。一个合法的语句的解析结果应该是一个 `MuaObject` 。以 `print eq 23 32` 为例：

- `32` 被解析为 `new Word(32)`，压栈
- `23` 被解析为 `new Word(32)`，压栈
- `eq` 被解析为一操作名，于是将栈中两个元素取出，evaluation，并压栈
- `print` 被解析为一操作名，于是将栈中一个元素（`false`）取出，evaluation，压栈。

最后，解释器取出栈中唯一的`MuaObject`，并输出其字符串形式。

Error处理

所有的error类继承自`MuaError`：

```
├── ArgError.java
├── ArithmeticError.java
├── IOError.java
├── IndexError.java
├── InputError.java
├── MuaError.java
├── NameError.java
├── SyntaxError.java
└── TypeError.java
```

出现错误时，抛出对应的`MuaError`对象。该对象最终会被解释器`catch`，并打印出对应的信息。

函数

在定义时，无需任何特殊处理。函数本身只是一个嵌套的`list`。为了支持函数调用，需要定义一个类`Func`。这个类继承`Expr`，与其他操作的地位几乎相同：

```
// implementation omitted
```

```
public class Func extends Expr {
    // look up function name in the scope,
parse the nested list
    public Func(String , Scope scope) throws
Exception;

    // check function syntax, setup enclosing
scope, etc.
    private void setUp(MuaObject o) throws
Exception;

    @Override
    public MuaObject eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception;

    @Override
    public String getOpName();

    @Override
    public int getArgNum();

    final private ArrayList<Class> argtypes =
new ArrayList<>(Arrays.asList(
    ));

    private String name;
    private ArrayList<Word> argNames = new
ArrayList<>();
    private List body;
```

```
    private Scope lexicalEnclosingScope;
}
```

函数创建时有两个参数：`name`和`scope`。将做以下操作：

- 在`scope`内查找`name`，获取绑定的对象
- 检查这个对象是否是`List`，是否符合函数的语法要求
- `setup`设置函数的各种信息，如参数表，函数体，所在的`scope`等。

需要关注的是函数的`eval`的实现：

```
public MuaObject eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
        super.eval(scope, arglist);
        ArgUtil.argCheck(name, argtypes,
arglist);
        Scope local = new Scope(name,
Scope.Type.FUNCTIONAL, lexicalEnclosingScope);
        for (int i = 0; i < argNames.size();
i++) {
            local.addName(argNames.get(i),
arglist.get(i));
        }
        try {
              RunUtil.runList(local, body);
        }
```

```
            catch (OpStop.StopSignal e) {
                // stop execution
            }
            return local.getReturnValue();
        }
```

步骤：

1. 创建一个该函数专用的 `scope` 。
2. 在这个 `scope` 中加入初始变量，即传入的参数
3. 在local scope中运行函数体。

这个实现记录了函数定义时（`make`）时的lexical enclosing scope，故支持嵌套函数定义。

在实际运行时，当解释器遇到一个非字面量，会检查是否是既有操作。若不是，则当作函数并试图用绑定的对象创建这个函数，并按一般操作的执行方法执行：

```
 // operation
MuaObject o = scope.getName(new Word(token));
Expr expr;

if (o instanceof Expr) { // built-in operations
    expr = (Expr) o;
}
else {  // functions, o instance of list
    expr = new Func(token, scope);
}
```

```
int argNum = expr.getArgNum();
ArrayList<MuaObject> arglist = new ArrayList<>
();
for (int i = 0; i < argNum; i++) {
    if (!opStack.isEmpty()) {
        arglist.add(opStack.pop());
    }
}
// evaluation, push
opStack.push(expr.eval(scope, arglist));
```

该实现自然支持递归，函数参数等。

Name Lookup/Binding规则

任何对象创建时，我们都记录其lexical enclosing scope。在函数调用时，会设置其local scope的lexical enclosing scope。在查找名字时，我们先查看本地scope。若不存在，查看其enclosing scope：

```
public MuaObject getName(Word name) throws
NameError {
    MuaObject ret = scope.get(name.toString());
    if (ret == null) {
        if (enclosingScope == null) { // final,
or global scope
            throw new NameError("name '" +
name.getValue() + "' not found");
        }
        else // enclosing escope
            return
enclosingScope.getName(name);
    }
    return ret;
}
```

任何local scope的enclosing scope都是global scope。然而，以上记录lexical enclosing scope的方式支持了嵌套函数定义。

在添加或修改名字时，我们只对local scope进行操作：

```
public void addName(Word name, MuaObject value)
{
    scope.put(name.getValue(), value);
    // set lexical enclosing scope
    if (value.enclosingScope == null) {
        value.enclosingScope = this;
    }
}
```

按照要求，删除名字时，会查找enclosing scope中的内容：

```
public void removeName(Word name) throws
NameError {
    MuaObject succeed =
scope.remove(name.getValue());
    // remove global
    if (succeed == null) {
        if (enclosingScope != null) {
            enclosingScope.removeName(name);
        }
        throw new NameError("name '" +
name.getValue() + "' not found");
    }
}
```

函数相关操作

- `output <value>`：设定value为返回给调用者的值，但是不停止执行。任意一个`scope`都有一个字段`returnValue`，`output`只要设置该字段即可。之后，调用链的上一个层次即可取得这个返回值：

```
// Output.eval
@Override
public None eval(Scope scope,
ArrayList<MuaObject> arglist) throws
Exception {
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    // set current scope return value
    scope.setReturnValue(arglist.get(0));
    return new None();
}
```

- `stop`：停止执行

  `stop`在实现上只要丢出一个`StopSignal`异常即可终止当前层面运行。上一个层面会catch这个异常：

```
// Stop.eval
@Override
public None eval(Scope scope,
ArrayList<MuaObject> arglist) throws
Exception {
    super.eval(scope, arglist);
```

```
        ArgUtil.argCheck(getOpName(), argtypes,
    arglist);
        throw new StopSignal();
    }

    // Runutil.runList
    try {
        ArrayList<ArrayList<String>> exprs =
    parseExpr(tokens, scope);
        for (ArrayList<String> expr: exprs) {
            evalObj(expr, scope);
        }
    }
    catch (Stop.StopSignal s) {
        // catch stop signal
    }
```

- `export`：将本地make的值输出到全局。得到当前scope的lexical enclosing scope，并加入名字即可：

```
// Export.eval
@Override
public None eval(Scope scope,
ArrayList<MuaObject> arglist) throws
Exception {
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    Word word = (Word) arglist.get(0);
    if (scope.getEnclosingScope() != null)
{

 scope.getEnclosingScope().addName(word,
scope.getName(word));
    }
    return new None();
}
```

类型判断

对于 `word` 和 `list`，使用 `instance of` 操作符可以自然实现：

```java
@Override
public Word eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    MuaObject obj = (MuaObject) arglist.get(0);
    if (obj instanceof Word) {
        return new Word(true);
    }
    else
        return new Word(false);
}
```

`Argutil.typeCast` 实现了检查 `Word` 能否转换为 `Number` 和 `Bool` 的方法。用这些方法就可以实现 `isnumber`, `isbool`：

```
@Override
public Word eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    MuaObject obj = (MuaObject) arglist.get(0);
    if (ArgUtil.typeCast(Number.class, obj) ==
null) {
        return new Word(false);
    }
    else
        return new Word(true);
}
```

数值计算

数值计算的实现实现十分简单，直接使用java自带的数值计算功能即可。例如 sqrt 的实现：

```
@Override
public Word eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    MuaObject obj = (MuaObject) arglist.get(0);
    double d = Math.sqrt(((Number)
obj).getValue());
    return new Word(d);
}
```

字表处理

`word`，`sentence`，`join`，`first`，`butfirst`等操作的处理基本类似，这里介绍`butfirst`的处理：

```
MuaObject obj = (MuaObject) arglist.get(0);
if (obj instanceof List) {
    List l = (List)obj;
    ArrayList<MuaObject> list =
(ArrayList<MuaObject>) l.getValue().clone();
    if (list.size() == 0) {
//                throw new IndexError("list
index out of range");
        return new List(list);
    }
    list.remove(0);
```

```
        return new List(list);
    }
    else {
        Word w = (Word)obj;
        if (w.getValue().length() == 0) {
    //                throw new IndexError("word
    index out of range");
             return new Word(w.getValue());
        }
        return new Word(w.getValue().substring(1));
    }
```

该操作首先检测参数类型。若为list，则去除其内部表示的第一个元素，并返回新表。若为word，则去除其第一个字符，并返回新word。

`if`的实现较为简单。后缀表达时的处理方式使得`bool`值可以即刻得到：

```
Bool cond = (Bool) arglist.get(0);
List listA = (List) arglist.get(1);
List listB = (List) arglist.get(2);
if (cond.getValue()) {
    RunUtil.runList(scope, listA);
}
else {
    RunUtil.runList(scope, listB);
}
```

其他操作

前面说过，任何一个操作执行时都有一个当前scope。`save`操作将这个scope串行化并保存，`load`操作将这个对象加载，并将所有名字注入当前scope：

```
// save
public None eval(Scope scope,
ArrayList<MuaObject> arglist) throws Exception
{
    super.eval(scope, arglist);
    ArgUtil.argCheck(getOpName(), argtypes,
arglist);
    Word w = (Word)arglist.get(0);
    File file = new File(w.getValue());
    try {
        ObjectOutputStream output =
            new ObjectOutputStream((new
FileOutputStream(file)));
```

```
            output.writeObject(scope);
            output.close();
        } catch (IOException e1) {
            throw new IOError(e1.getMessage());
        }

        return new None();
    }

    // load
    public None eval(Scope scope,
    ArrayList<MuaObject> arglist) throws Exception
    {
        super.eval(scope, arglist);
        ArgUtil.argCheck(getOpName(), argtypes,
    arglist);
        Word w = (Word)arglist.get(0);
        File file = new File(w.getValue());
        try {
            ObjectInputStream input =
                new ObjectInputStream(new
    FileInputStream(file));
            Scope newScope = (Scope)
    input.readObject();
            scope.addAllName(newScope);
            input.close();

        } catch (IOException e1) {
            throw new IOError(e1.getMessage());
```

```
        }

        return new None();
    }
```

`erall`和`poall`的实现较为简单，这里不列出了。

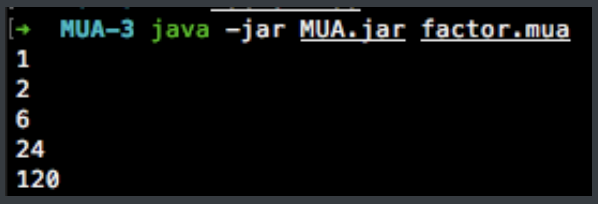
样例程序

递归计算阶乘：

```
make "factor [
[n]
[
    if eq :n 1 [
        output 1
    ] [
        output mul :n factor sub :n 1
    ]
]
]

print factor 1
print factor 2
print factor 3
print factor 4
print factor 5
```

结果


MUA-3 java -jar MUA.jar factor.mua
1
2
6
24
120