MUA 解释器报告

1. 整体架构与简介：

D:\NEW\NEW4\PPL\MUA_INTERPRETER\SRC\COM\COMPANY

```
|   CmdExecute.java
|   Data.java
|   MUA.java
|   MyArray.java
|   OperatorBind.java
|   OperExecute.java
|   Parse.java
|   tree.txt
|
└──DetailCommand
        AddExecute.java
        AndExecute.java
        ButfirstExecute.java
        ButlastExecute.java
        DivExecute.java
        EqExecute.java
        ErallExecute.java
        EraseExecute.java
        ExportExecute.java
        FirstExecute.java
        GtExecute.java
        IfExecute.java
        IntExecute.java
        IsboolExecute.java
        IsemptyExecute.java
        IslistExecute.java
```

IsnameExecute.java

IsnumberExecute.java

IswordExecute.java

JoinExecute.java

JudgeType.java

LastExecute.java

ListExecute.java

LoadExecute.java

LtExecute.java

MakeExecute.java

ModExecute.java

MulExecute.java

NotExecute.java

OrExecute.java

OutputExecute.java

PiExecute.java

PoallExecute.java

PrintExecute.java

RandomExecute.java

ReadExecute.java

ReadlistExecute.java

RepeatExecute.java

RunExecute.java

SaveExecute.java

SentenceExecute.java

SqrtExecute.java

StopExecute.java

SubExecute.java

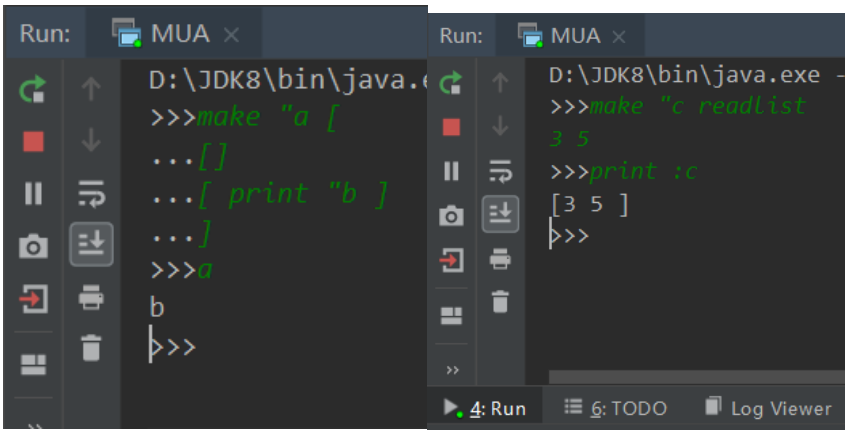
ThingExecute.java

WaitExecute.java

WordExecute.java

程序入口位于 MUA 类中的 main 方法，com.company 包中的类是解释器的整体架构所在处，com.company.DetailCommand 包中的类是解释器的具体方法执行对应的类。

程序的大致流程是从 main 开始，构建一个全局的命名空间（CmdExecute 类）对象，再用全局的命名空间初始化 OperatorBind 对象，以获取操作名的初始化，之后初始化 Parse 类对象，用于准备对命令的语法解析。语义解析的过程在 CmdExecute 类以及各个具体的操作类中完成。用一个 StringBuffer 对象来记录用户的输入，如果发现没有平衡的方括号，即提供多行输入的操作方式，直到用户输入的指令达成平衡的方括号为止。这样的输入方式在 readlist 中也相同。



2. 重要类分析

a) Parse 类：

本类对于用户输入的指令进行初步的处理，包括去掉注释，去掉空白符号，以及提取出 list 类型的对象。为了提取 list 类型的对象，同时实现了平衡方括号的方法。经过 Parse 类中 parseCmd 方法的解析，最终返回一个 List<String>对象，其中包括已经分割好的语素，包括操作名，word 以及 list。本类不对语义进行解析，仅仅抛出括号不匹配的异常。

```
>>>make "a ]
List Error: Brackets are not matched
```

下图代码不会出现异常，是因为最后一个右括号为带引号的 word 类型。

```
>>>make "a [ "[ "] ] "]
>>>print :a
[ "[ "] ]
>>>
```

对外提供的接口：

public static int checkBracket(String listContent)//用于方括号匹配的检测

public static List<String> parseCmd(String cmd) throws Exception//用于初步处理指令

b) CmdExecute 类：

这是程序执行的主体部分，这个类中有一个命令栈，一个执行栈，以及一个命名空间，对于主程序的运行或者用户定义的函数提供了统一的接口。以 getCmdSet 方法来接收一个被 Parse 之后的 List<String>。对于 getCmdSet，需要传入一个参数为 callSpace，即调用此方法的命名空间类，为 CmdExecute 类型，目的是为了记录下执行此段代码的父空间，以便 output 指令的输出。

每一个 String 有以下可能：

1) 是一个字，传入 Data 的构造函数，构造一个新的 Data 对象

2) 是直接一个 list，直接执行。

```
>>>[make "b 1 poall wait 1234 poall]
0th name: a
0th value: [ [n] [ if lt thing "n 2 [ output 1 ] [output mul thing "n a sub thing "n 1 ] ] ]
1th name: b
1th value: 1
0th name: a
0th value: [ [n] [ if lt thing "n 2 [ output 1 ] [output mul thing "n a sub thing "n 1 ] ] ]
1th name: b
1th value: 1
```

3) 非法字符。

```
>>>ppppp
Data Type Error: Get a illegal operator.
```

然后对传入的指令进行语义分析。因为被 Parse 的指令很可能不止一条，所以需要识别成一组完整的语句。这里使用了类似于括号平衡的方法来做，获取每一个关键字以及用户定义的函数所需参数个数并压栈，发现有返回值的函数或者操作名或者字，就对最近的那个操作名所需参数-1 直到 0 为止。当整个堆栈都是 0 时，前面的所有内容即为一条指令。为此实现了 MyArray 类。

下图样例分别表示了一行多条指令与多个操作综合成的一条指令的正确运行结果。

```
>>>make "a 1 print :a make "b 2 print :b
1
2
>>>print add mul 1 2 sub 3 4
1.0
```

截取出单条指令后，调用 dealOneIntruction 方法来处理单条指令。如果本方法返回了 true，则代表处理中断，直接跳出主循环（抛异常或者 stop），否则继续执行。对于单

条指令的每一个语素，如果是操作名就直接调用对应类的 execute 方法来执行。如果是用户定义函数，重新 new 一个 CmdExecute 类与对应的 OperatorBind 类，将用户定义的函数直接放进新的 CmdExecute 类中的 getCmdSet 方法，即可完成函数的操作。其他情况均为值，压入执行栈中。

另外，全局的命名空间被设计为对所有的命名空间可见，也就是说：

```
>>>make "a true
>>>make "b [ [ ] [ print not :a ] ]
>>>b
false
>>>make "a false
>>>b
true
>>>make "b [ [ ] [make "a true print :a ] ]
>>>b
true
>>>print :a
false
```

上图中，全局的 a 为 true，在函数 b 中的 a 也是 true；在全局中改变 a 的值，函数 b 中的 a 也随之改变；在 b 中改变 a 的值，全局中的 a 的值不变。这样的设计主要是为了递归函数的实现。如果命名空间完全隔离，那么函数的子命名空间中将会因为没有自身的定义而报错。

对外提供的接口：

public void getCmdSet(List<String> cmdSet, CmdExecute callSpace)//将输入的命令解析为单条命令，并调用单条命令的解析方法

public void pushToExecuteStack(Data item) //压入执行栈，用于操作返回值

public List<Data> popExecuteStack(int num)//取出执行栈中的 num 个值，用于操作或者函数取参数。

public void pushToCmdStack(Data item)压入命令栈，用于 read

public void addOneBind(String key, Data value)增加绑定，用于 make

public Data deleteOneBind(String key)删除绑定，用于 erase

public Data getValue(String key)取名字的值，用于 thing

public void deleteAllBind()//删除所有绑定，用于 erall

public void traverse()//遍历命名空间，用于 poall

public void writeToFile() throws Exception//将命名空间写入文件，用于 save

public void loadFromFile() throws Exception//将命名空间从文件中载入，用于 load

public CmdExecute getCallSpace()//获取父空间

对内提供的接口：

private boolean dealOneIntruction(CmdExecute callSpace) throws Exception/对单条命令的每一个语素解析并执行。如果执行遇到问题或者遇到 stop，返回 true，否则返回 false。

private void clearStack()//清空栈，用于中途退出程序（抛出异常）以及初始化。

c) Data 类：

将语言中出现的所有语素全部封装为 Data 类，这也是 MUA 仿 lisp 语言的特点：代码即数据。向外提供接口为：

public Data(String org)//构造函数，要么为 word 类型，要么此关键字存在于 Keywords 列表中并且不带”标记，那么为 oper 类型。

public Data(double num)//构造函数，构造一个 number 类型

public Data(boolean bool)//构造函数，构造一个 boolean 类型

public Data(List<String> list)//构造函数，构造一个 list 类型

@Override

public String toString()直接返回用于构造 Data 的字符串值，比如”a

public String getLiteral()返回字面量，比如 a

public int getType()返回类型

public double getNumberValue() throws Exception//获取 double 类型的值

public boolean getBooleanValue() throws Exception//获取 boolean 类型的值

public List<String> getListValue() throws Exceptio//如果为列表类型，返回列表的值。

public boolean isWord()判定是否为 word

public boolean isFunction(CmdExecute cmdExecute) throws Exception 判定这个 word 是否为已经被定义过的函数。因为例如”a 不是一个函数，那么在程序中输入 a，会被判定为非法字符。

public boolean getHaveReturnValue(CmdExecute cmdExecute) throws Exception 判定是否为有返回值的函数

对内提供接口为：

private List<String> checkAndGetValue()//用于检查列表类型的 Data 是否符合列表的语法，如果符合，则返回值。

默认访问权限（供 OperatorBind 类使用）的接口为：

void notOperator()//将制定操作符变为普通的 word 类型，而不是操作符

d) OperatorBind 类：

将系统定义的操作名与对应操作绑定，以及定义操作名的命名空间（为全局，只有一个）。

对外提供接口：

public OperatorBind(CmdExecute cmdExecute) //构造函数，对于每一个 CmdExecute 类，由于操作时所用的命令栈与数据栈不同，所以要有不同的操作名对应的类

public static int typeToNum(String choice)类型名字对应的类型数字

public OperExecute getExecuteClass(String key)获取对应的执行类

public static void addOneOperBind(String key)增加一个操作名，调用 Data 提供的接口将此操作名的类型改变为 word

public static Data deleteOneOperBind(String key)删除一个操作名，调用 Data 提供的接口将此操作名的类型改变为 word

除了与操作名的操作类绑定的方法之外，其余方法均为静态方法，因为这些方法对每个类都是一样的。

e) OperExecute 类及其子类：

OperExecute 实际上与抽象类类似，代码中并不会直接实例化 OperExecute 类，而是会针对每一个具体的操作初始化它的每一个具体的子类。

对外提供接口：

public OperExecute(CmdExecute cmdExecute)//构造函数，初始化时，需要输入这个操作名活动的命名空间。

public void execute(Data nowCmd) throws Exception//执行 为空函数，子类继承这个函数写具体操作

public int getNeedValue()//记录某个操作所需要的参数个数

public boolean getHaveReturnValue()//记录某个操作是否有返回值。

对于具体类的操作，基本流程大同小异，即在对应的 cmdExecute 类中，运用 CmdExecute 类提供的对命令栈与数据栈的接口进行操作。

由于指令的特殊性，OperExecute 类中有两个特例：

StopExecute 类：stop 直接中断函数执行，转化为代码操作就是直接 return true，故

提取出来最为方便

OutputExecute 类：output 指令要求将指令返回给它的调用者，考虑到其它的操作名与函数不会需要这样的操作，故执行 output 指令时，CmdExecute 类通过 OutputExecute 类中的独有接口 public void getFatherSpace(CmdExecute fatherSpace)接口，将 callSpace 传递给它。

针对类名生成类的方法，在 OperatorBind 类中初始化时完成。

```
try
{
    mainNameSpace.put(s, new Data(s,  isOp: true));
    String className = "com.company.DetailCommand." + s.substring(0, 1).toUpperCase() + s.substring(1) + "Execute";
    Class executeClass = Class.forName(className);
    Constructor constructor = executeClass.getConstructor(cmdExecute.getClass());
    OperExecute getExecuteClass = (OperExecute) constructor.newInstance(cmdExecute);
    detailExecute.put(s, getExecuteClass);
}
catch (Exception e)
{
    e.printStackTrace();
}
```

这样设计的优点是，减少了大量的 if else，用 HashMap 存储操作的名字对应的类，减少了代码的冗余。但是，由于每一个操作需要对应一个类，故需要大量的类与之匹配，又使得不必要的代码多出许多。总体上而言，使得开发更加的模块化。并且，针对不必要的代码，可以有相对的优化，比如 isbool，islist，isname，isword 四个操作，虽然仍然需要 4 个类，但是可以由新建的 JudgeType 类代替其完成操作。在 isbool 的操作类中，只需要声明自身是否有返回值以及需要几个参数，然后将操作交与 JudgeType 类执行。

```
public class IsboolExecute extends OperExecute
{
    public IsboolExecute(CmdExecute cmdExecute)
    {
        super(cmdExecute);
        this.haveReturnValue = true;
        this.needValue = 1;
    }

    @Override
    public void execute(Data nowCmd) throws Exception
    {
        JudgeType j = new JudgeType(cmdExecute);
        j.execute(nowCmd);
    }
}
```

JudgeType 类：通过调用者的操作名，与一个 static 的表对应取得其类型，返回结

果。

```
@Override
public void execute(Data nowCmd) throws Exception
{
    List<Data> getDataSet = cmdExecute.popExecuteStack( num: 1);
    if (getDataSet.size() == 1)
    {
        Data d = getDataSet.get(0);
        String judge = nowCmd.getLiteral().substring(2).toLowerCase();
        //System.out.println("Output From JudgeType: now type is: " + judge);
        if (d.getType() == typeToNum(judge) || (typeToNum(judge) == 1 && d.isWord()))
        {
            cmdExecute.pushToExecuteStack(new Data( org: "true"));
        }
        else
        {
            cmdExecute.pushToExecuteStack(new Data( org: "false"));
        }
    }
    else
    {
        throw new Exception("Operator Error: Lack of parameter.");
    }
}
```

f) MyArray 类：

MyArray 类实际上是对一个 ArrayList 中常用操作的封装。为了达成之前提到的像平衡括号一样平衡指令，将一行中的多条指令分割出来而设计。

对外提供接口：

public void add(int num)//往数组最后添加一个数，这个数代表的是新进来的操作名所需要的参数数量。例如 print，则添加一个 2。

public void minusOne()//对这个数组中所有不为 0 的数减 1。代表的是新进来的是一个字或者一个有返回值的函数时，贡献一个参数，为之前需求参数的指令减少一个参数的需求。

public boolean checkZero()//如果在这个操作名或者参数输入进来之后，整个 list 的值全部为 0，则说明前面的指令为一条指令，截取出来，跳出循环，开始执行。

g) 异常设计：

检测到的所有异常都直接抛出并输出。这样就会防止一条指令报出多个错误的情况。

```
>>>make "a qwe
Data Type Error: Get a illegal operator.
```

如图，qwe 是一个非法参数，直接抛出后，make 就不会在运行，不会继续报出 make

缺少参数的错误。

h) 特殊情况：

由于 word 的字面量定义为”后面到空白字符的所有字符，故定义列表时，右括号前需要加一个空格以分割字与列表的边界，否则为语法错误。

错误：

```
>>>make "a ["b]
...]
```

正确：

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

3. 测试情况

a) 类型与命名空间：

```
>>>make "a 1
>>>print isname "a
true
>>>print isname :a
false
>>>print isword "a
true
>>>print isword :a
true
>>>print isnumber "a
false
>>>print isnumber :a
true
>>>make "b [ [ ] [ print isname "a ] ]
>>>b
true
>>>make "b [ [ ] [ make "c 1 print isname "c ] ]
>>>b
true
>>>print isname "c
false
>>>print isname :b
false
>>>print islist :b
true
>>>print islist "b
false
>>>print isword :b
false
```

如图所示，全局名字 a 在函数 b 中也是已经绑定的名字（isname 返回 true），在函数 b 中定义的名字 c，在全局中 isname 返回 false。说明全局命名空间对所有命名空间可见，子命名空间对全局命名空间不可见。

类型系统：可以看出，a 的字面量为 1 时，a 的 isword 和 isnumber 都返回 true，说明 number 作为 word 的一个特例。但是 b 的字面量是一个 list，此时:b 的 islist 返回 true，而 isword 返回 false，说明 list 与 word 是分离开的。

```
>>>make "a [ ]
>>>print isempty :a
true
>>>make "a []
>>>print isempty :a
true
>>>make "a "
>>>print isempty :a
true
```

同时，isempty 也会正确返回空的 list 或者 word。

b) read 与 readlist

```
>>>make "c read
1
>>>print :c
1
>>>make read 2
"d
>>>print :d
2
```

```
>>>make "c readlist
1 3 5 6 9
>>>print :c
[1 3 5 6 9 ]
```

直接替换 read 和 readlist 所在的位置。

c) 运算符

布尔值计算与比较计算：

```
>>>print and gt 2 1 lt 1 2
true
```

(2>1) && (1<2)的结果

数值计算：

```
>>>print mul div add 3 2 sub 1.25 pi mod 12 5
-5.2865578693057165
```

(3 + 2) / (1.25-π) * (12%5)的结果

d) make，poall，erall，erase

```
>>>make "a 1
>>>make "b :a
>>>make "c [ asd [asd [asd ] asd ] asd ]
>>>poall
0th name: a
0th value: 1
1th name: b
1th value: 1
2th name: c
2th value: [ asd [asd [asd ] asd ] asd ]
>>>erase "b
>>>print :b
Namespace Error: This name is not in namespace
>>>poall
0th name: a
0th value: 1
1th name: c
1th value: [ asd [asd [asd ] asd ] asd ]
>>>erall
>>>poall
```

先 make3 个名字，然后 poall 全部输出，再 erase 一个，可以看到”b 已经不是一个名字了，再 erall，之后所有的全局名字都被移除。

e) stop && export

```
>>>make "func [ [ a b ] [ make "c add :a :b output add :a :c export "c stop output add :a :b export "b ] ]
>>>print func 4 5
13.0
>>>print :c
9.0
>>>print :b
Namespace Error: This name is not in namespace
```

在执行了 func 函数之后，先是 make 了 c 这个名字，然后输出了 c 并且导出了 c，并且 stop 后续的代码不再运行。

f) random sqrt int

```
>>>print random 456
376.45443477614344
>>>print random 456
250.1116231029532
>>>print random 456
262.45752254876896
>>>print sqrt 456
21.354156504062622
>>>print int sqrt 456
21.0
```

每次 random 返回值并不相同。

g) word sentence join list first last butfirst butlast

```
>>>make "a "test
>>>print word "a :a
atest
>>>print word :a "a
testa
```

上图是 word，第一个参数为 word 类型，第二个为除了 list 以外的类型，拼接起来以后结果正确。

```
>>>make "a [ 1 2 3 [ 4 5 ] ]
>>>make "b [ 2 6 8 [ 5 9 ] ]
>>>print sentence :a :b
[ 1 2 3 [ 4 5 ]  2 6 8 [ 5 9 ]  ]
>>>print join :a :b
[ 1 2 3 [ 4 5 ]  [ 2 6 8 [ 5 9 ] ] ]
>>>print list :a :b
[ [ 1 2 3 [ 4 5 ] ] [ 2 6 8 [ 5 9 ] ] ]
```

sentence 将两个 value 合并为一个 list，value 为 list 类型的话要打开

list 将两个 value 合并为一个 list，value 为 list 类型的话不打开

join 是将后一个 value 作为 list 的最后一个元素加入 list。

```
>>>make "a [ [ asd ] [ print :asd ] ]
>>>print butfirst :a
[ [ print thing "asd ]  ]
>>>print butlast :a
[ [ asd ]  ]
>>>print first :a
[ asd ]
>>>print last :a
[ print thing "asd ]
```

```
>>>print butfirst :a
weug5109
>>>print butlast :a
qweug510
>>>print first :a
q
>>>print last :a
9
```

对 list 的 first 和 last 返回的是 list 的元素类型，butfirst 和 butlast 返回的是 list。对 word 的操作均返回 word。

h) save && load

```
>>>make "a 1
>>>make "b 2
>>>make "c :a
>>>make "d [ 4889 [ 489 ] ]
>>>poall
0th name: a
0th value: 1
1th name: b
1th value: 2
2th name: c
2th value: 1
3th name: d
3th value: [ 4889 [ 489 ] ]
>>>save
>>>erall
>>>load
>>>poall
0th name: a
0th value: 1
1th name: b
1th value: 2
2th name: c
2th value: 1
3th name: d
3th value: [ 4889 [ 489 ] ]
>>>
```

save 是直接保存在一个默认文件中，load 是从默认的文件中将命名空间读出。先 make，然后 save，然后 erall，然后 load，结果正确。

i) wait

```
>>>make "i 0
>>>repeat 3 [ wait 2000 print :i make "i add :i 1 ]
0
1.0
2.0
```

本函数的运行结果是每两秒输出一个 i，i 自增。

j) pi run

```
>>>run [ print "a make "b 1 print :b ]
a
1
>>>erase run
Operator Error: Lack of parameter.
>>>erase "run
>>>run [ print "a ]
null
>>>print pi
3.14159
>>>erase "pi
>>>print pi
null
>>>
```

pi 和 run 都可以被移除。

k) 具体函数

i. 阶乘的递归定义：

```
>>>make "a [ [n] [ if lt :n 2 [ output 1 ] [output mul :n a sub :n 1 ] ] ]
>>>print a 9
362880.0
```

计算了 9 的阶乘，返回值为正确值。

ii. 阶乘的迭代定义

```
>>>make "fact [ [ n ] [ make "i 1
...make "ret 1
...repeat :n
...[make "ret mul :ret :i make "i add :i 1 ]
...output :ret ]
...]
>>>print fact 5
120.0
>>>print fact 9
362880.0
```

计算了 5 和 9 的阶乘，返回值为正确值。

iii. 在函数内定义函数

```
>>>make "fun [
... [m n]
... [
...  make "sub2 [ [ n m ] [ output sub :n :m ] ]
...  make "add2 [ [ a b ] [output add :a :b ] ]
...  make "square [ [ x ] [ output mul :x :x ] ]
...  make "m2 [ [ ] [ output square :m ] ]
... output add sub2 add2 square 3 square 6 mul 6 4  div 6 5
... ]
... ]
>>>fun 4 5
>>>print fun 4 5
22.2
```

如图，在函数内定义了 4 个函数，分别是加法，减法，平方，然后返回的算式是

(3^2 + 6^2) – 6 * 4 + 6 / 5 = 22.2