

# MBat 脚本解释器

## 一、功能需求分析

本课题主要目的是实现一个解释器，将定义的一种脚本语法顺利解释执行，并能够在语法错误时提示错误位置，在运行时出错时终止脚本运行，能够编译为更快速的二进制脚本运行；脚本语法的基本需求是实现输出、输入、变量、判断、算数运算及跳转功能，实现了这些基本功能后，就能够使用这些基本语法组合写出功能多样的脚本来执行，以完成目标任务。例如使用脚本编写“基于记录”类型的文字游戏、问卷等程序。

## 二、总体设计

### 1. 使用独立于程序逻辑的 Parser 类来完成对脚本的解释执行任务

采用这种设计可以解耦，将脚本的解释、执行任务封装进独立的类，类的使用和具体的任务逻辑分离，可以根据不同需求组装出不同的程序，实现代码复用；

如本程序是实现一个完整的解释器，且不支持交互式输入模式，但通过对主程序逻辑的少许修改，即可实现一个交互式脚本解释器，因为 Parser 类中相关接口时能够实现该功能的。

### 2. 在命令定义上使用枚举-字符数组对应的方式，解析、执行使用 switch

这种方案有明显的缺点，那就是并不十分清晰，关于同一个命令相关的代码分散于 4 处，在修改、增加命令的时候需要在代码中跳转，但实现起来相对较为简单；

另一种较好的实现方式是通过定义一个虚基类接口，定义命令的公共接口，然后再解析和执行的时候使用这些接口来完成解析和执行的的操作，这样子定义新命令、新语法时只需要编写一个新的类派生于基类并实现接口即可，但相对来说实现起来稍复杂一点。鉴于时间原由以及本人在实现过程中已使用简单方式实现大半，故先使用第一种方案完成了本项目需求。

### 3. 本脚本解释器没有“块级作用域”的概念

没有块级作用域意味着在脚本中定义的所有变量都是全局的，且不存在所谓的“函数调用”过程以及判断语句的块级分支，但函数调用过程和判断语句的块分支均可通过跳转命令来模拟实现，目前没有在解释器级别上提供该程度的语法。

添加块级作用域的概念需要对 Parser 的大体框架进行修改，如添加相应的栈来存储当前块的信息，在块进入和退出时进行相应的 push 和 pop 操作。

#### 4. 编译过程

Parser 类通过提供 compile 接口来完成对已解析的动作进行编译，其本身不负责任何加密、附加文件头等操作，该接口只负责将已解析的动作序列化以供 Parser 类提供的另一接口 load 来加载；而该程序实现的编译功能实际上不但附加了文件头，且对编译后的内容进行了简单的加密，这两个过程都是在主函数中完成操作的。使用这种模式的原因是不同的业务逻辑由不同的实体来完成，作为 Parser 类，其职责就是负责 parse 和 execute，并提供序列化和反序列化的功能，因此加密和文件头的识别不属于其职责，而是本脚本解释器特有的功能，因此这部分代码在本脚本解释器的主函数中实现。

#### 5. 自动识别文件头判断是否是编译后的文件

如第四点所提，解释器主函数在输出编译后的文件时会为其加上特定文件头，并且进行加密混淆，在使用解释器打开脚本的时候，解释器主函数会先判断文件头是否为特定的文件头来决定调用 Parser 的什么过程去正确解释文件，因此不需要在参数中额外说明某个脚本是编译后的脚本。

#### 6. 系统安全性

系统安全性能得到保证，不会发生因一行内容过长而导致缓冲区溢出攻击问题，实现方法是：对于不定长度参数进行处理时，buffer 是根据长度通过 new 来动态申请的缓冲区，但这种方案牺牲了一部分性能（因为涉及到系统的内存分配）。其次，在长度的处理上使用的并不是传统 int 类型而是更安全可靠的大小\_t，确保在任何硬件情况下都能处理机器理论极限数目的数据；再次，对于已编译的文件，在无法得知密钥的情况和文件结构的情况下，理论上是非常困难去还原其内容的，因此脚本的编译保密性也能够得到保障。

#### 7. 运行环境

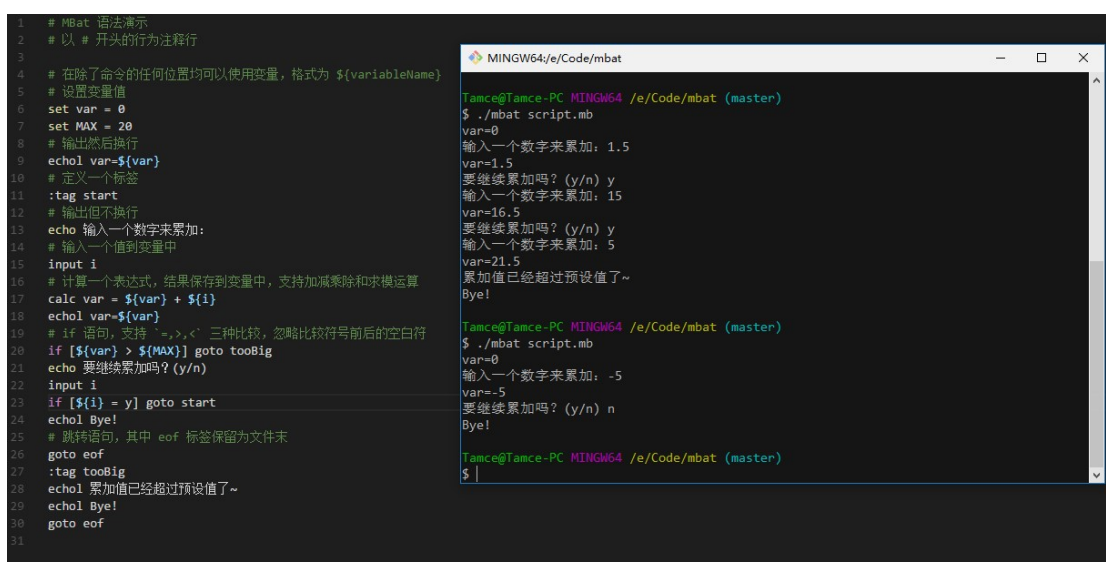
本程序使用的所有类库仅有标准模板库 STL 的内容，任何地方不涉及系统特性、不调用系统 API，因此本程序能够在任何支持标准 C++ 的机器上编译、运行。

本程序使用了 C++11 特性，因此编译该程序必须要求编译器支持 C++11 标准，本人在 Windows10, MinGW-W64(g++) 4.8.2 下编译测试通过，使用编译指令：

```
g++ main.cpp --std=c++11
```

### 三、详细设计与程序实现

在此项目中，脚本的每一行均对应一个动作(Action)，Parser 的职责是从文本解析生成 Action 序列和执行 Action 任务；我们先通过一个例子看一下 MBat 的脚本语法：



```
1 # MBat 语法演示
2 # 以 # 开头的行为注释行
3
4 # 在除了命令的任何位置均可以使用变量，格式为 ${variableName}
5 # 设置变量值
6 set var = 0
7 set MAX = 20
8 # 输出然后换行
9 echo! var=${var}
10 # 定义一个标签
11 :tag start
12 # 输出但不换行
13 echo 输入一个数字来累加:
14 # 输入一个值到变量中
15 input i
16 # 计算一个表达式，结果保存到变量中，支持加减乘除和求模运算
17 calc var = ${var} + ${i}
18 echo! var=${var}
19 # if 语句，支持 '<','>','=' 三种比较，忽略比较符号前后的空白符
20 if [ ${var} > ${MAX} ] goto tooBig
21 echo 要继续累加吗? (y/n)
22 input i
23 if [ ${i} = y ] goto start
24 echo! Bye!
25 # 跳转语句，其中 eof 标签保留为文件末
26 goto eof
27 :tag tooBig
28 echo! 累加值已经超过预设值了~
29 echo! Bye!
30 goto eof
31
```

```
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat script.mb
var=0
输入一个数字来累加: 1.5
var=1.5
要继续累加吗? (y/n) y
输入一个数字来累加: 15
var=16.5
要继续累加吗? (y/n) y
输入一个数字来累加: 5
var=21.5
累加值已经超过预设值了~
Bye!

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat script.mb
var=0
输入一个数字来累加: -5
var=-5
要继续累加吗? (y/n) n
Bye!

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$
```

图 3.1. MBat 脚本语法概览

图 3.1 中展示了目前 MBat 脚本所定义的所有命令，并使用该语法编写了一个简单的累加器程序。

看完了基本的语法，就让我们从解析语法的 Parser 说起，纵观 Parser，其公共接口如下图 3.2:

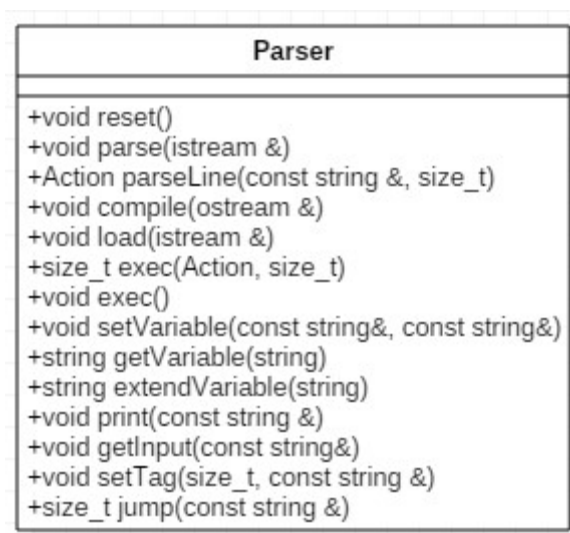


图 3.2. Parser 的公共接口

该类提供 reset 接口来重置类的状态，以便重复使用一个实例进行操作；在解析方面，主要提供两种方式进行解析；也提供了两种执行 Action 的方法，另外其他一些公共接口都是提供了手动设置变量、取得变量、设置跳转标签等功能。

除了这些公共接口，Parser 类内部存在一个 `vector<Action>` 来保存已解析的命令序列，同时使用一个 `map<string, size_t>` 映射来保存标签和行号（命令 id）的对应关系，此外还有一个 `map<string, string>` 映射来存储所有的变量信息。

下图（图 3.3）展示了调用 `parse` 时 Parser 的行为：

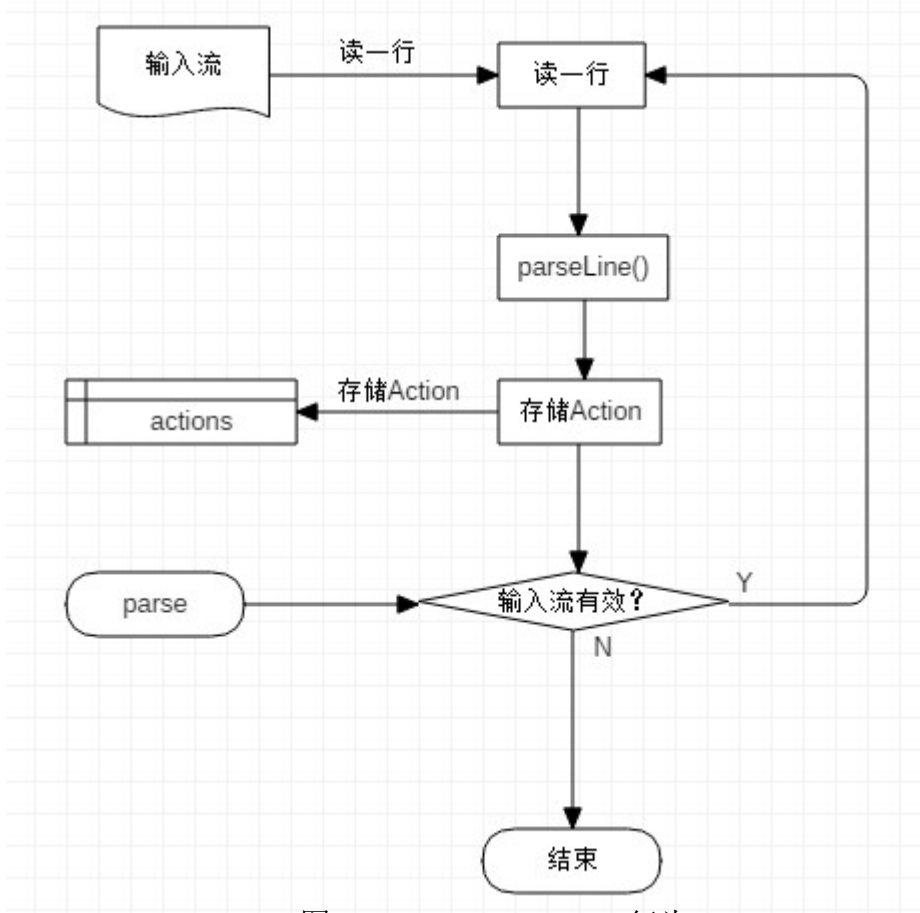


图 3.3. `parse(istream &)` 行为

调用 `parse` 时 Parser 不断的从输入流中读取一行内容并交给 `parseLine` 解析产生 Action，之后将生成的 Action 存储到内部私有变量 `actions` 中，同时，行数对应 Action 在 `actions` 中的下标，这样，在之后的跳转中便可以通过 `actions` 的下标来跳转到指定命令。

在介绍 `parseLine()` 之前，先介绍一下 Action 的组成，Action 定义为一个结构体，其结构可以由下图（图 3.4）表示：

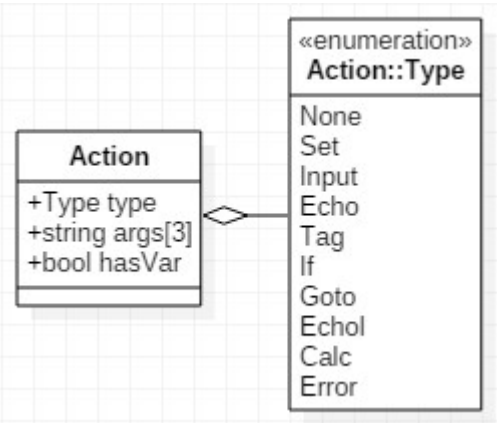


图 3.4. Action 定义

使用枚举类型 `Type` 来表示命令类型，使用 `args` 的 `string` 数组来表示命令附属的其他参数，考虑过直接使用 `stringstream`，也可使用 `vector<string>`，最后因实现已经写了大半，便没有进行更改。若使用上述两种实现方案，其他逻辑实际变动不大。

知道了 `Action` 的定义，`parseLine` 就不难解释了，其逻辑可以用以下流程图（图 3.5）表示：

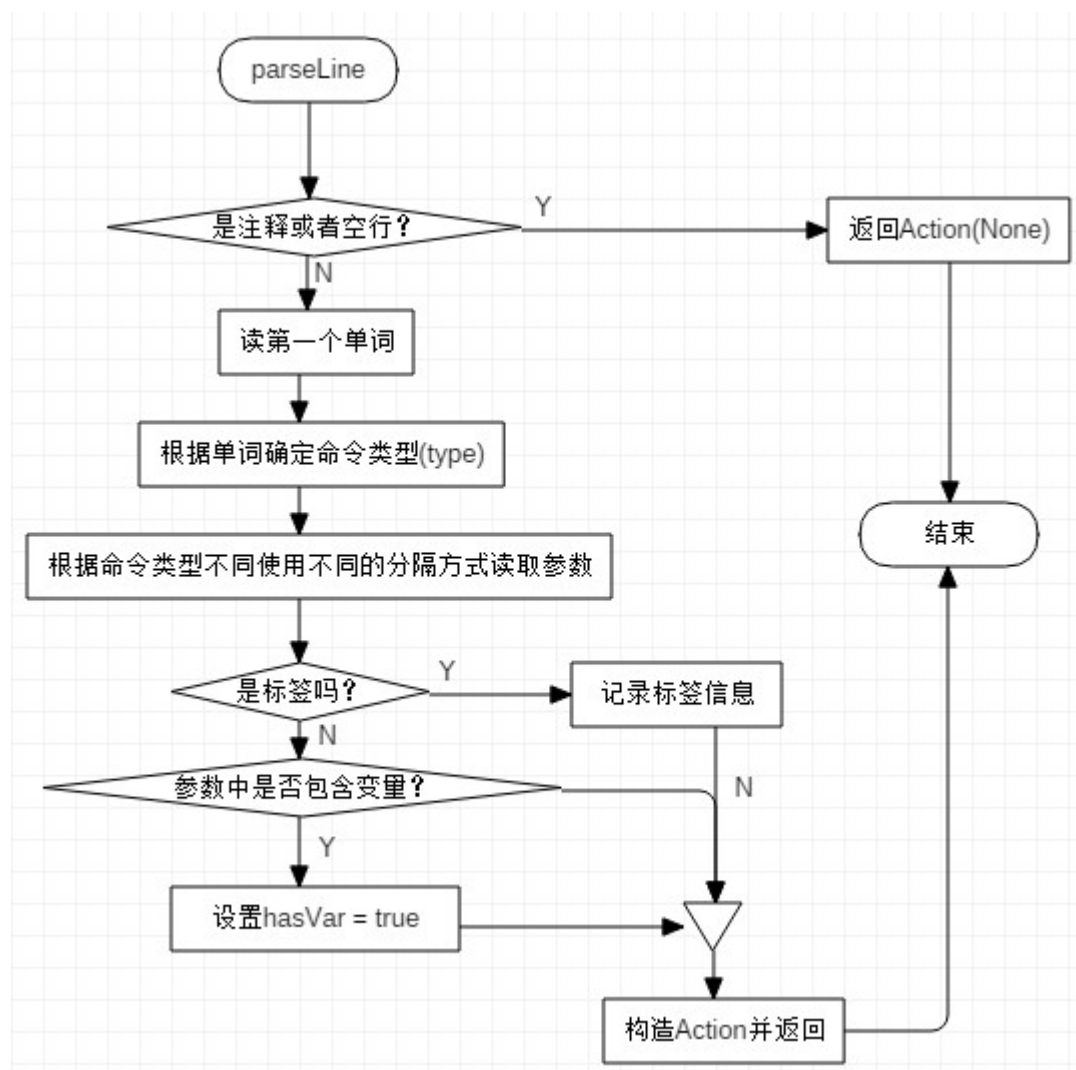


图 3.5. `parseLine` 基本逻辑

在解析的过程中首先判断其类型，之后根据类型执行相应的逻辑来分隔后续内容，最后在分隔出的各个部分检查是否存在变量并打上标记，以便在实际执行之前对变量的值进行替换；目前版本的代码在此处不同类型的命令分词逻辑是通过 `switch-case` 来完成的，更好的实现方式应该是编写一组派生于一个虚基类（`Action`）的详细命令，分词完成后逐个调用其接口检查是否是那个命令并转交给其进行接下来的分词逻辑判断。

解析的部分到此基本已经大部分涉及，其中的细节便不过多深入阐述；接下来简单介绍一下执行部分。

执行部分相对来说较为简单一些，其核心在于接受 Action 的 exec 函数，这个函数根据 Action 的类型、是否含有变量和具体的参数来决定作何动作，如图 3.6 所示：

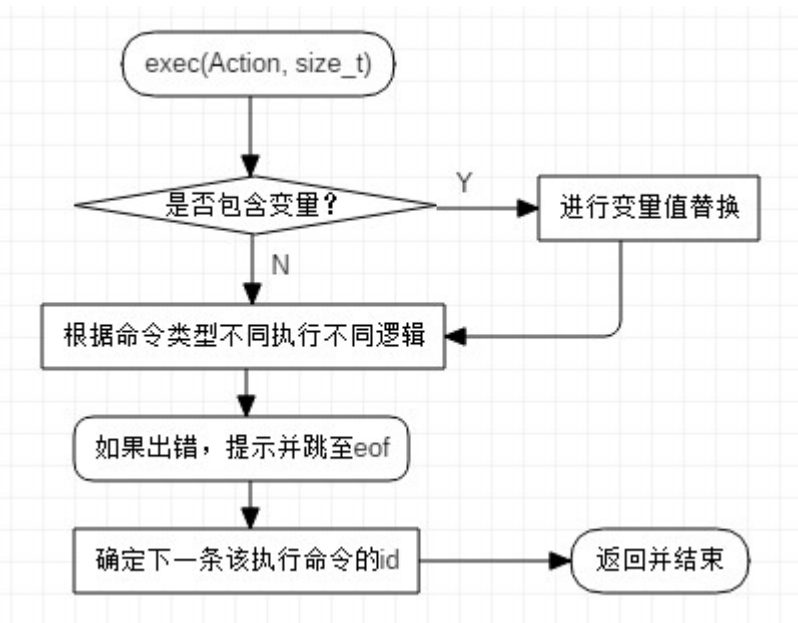


图 3.6. exec(Action, size\_t) 基本逻辑

此部分类似解析，也可以通过命令类的方式来拆分代码。在执行部分唯独有一点较为特殊，那就是这个函数被设计为接受一个 size\_t 且返回一个 size\_t 的样子而不是单纯的 Action，第一个原因是可以在发生执行时错误时给出错误行数信息，其次可以通过返回值来确定 Action 的执行情况，在此返回值设计为需要执行的下一条命令的位置，因此在处理跳转指令时只需要找到需要跳转的目标位置，并将其返回即可；因此默认不带参数的 exec 函数的行为就显而易见了，那就是从第一条保存的 Action 开始，逐条传递给带参数的 exec 执行，并且需要执行的下一条命令的位置由其返回值给出。

Parser 的编译和载入功能是将已解析的 Action 序列和标签信息以二进制方式保存，但由于 Action 中含有 string 类的实例（存在动态内存分配），因此在编译和载入过程中对其进行了单独处理，如在 32 位情况下编译运行时（int/size\_t 类型占用 4 字节）序列化的格式如下图（图 3.7）所示：

|                     |   |   |   |                 |   |   |   |                     |   |    |    |                    |    |    |    |    |    |    |    |                     |    |                |    |    |    |    |    |              |    |    |    |  |  |  |
|---------------------|---|---|---|-----------------|---|---|---|---------------------|---|----|----|--------------------|----|----|----|----|----|----|----|---------------------|----|----------------|----|----|----|----|----|--------------|----|----|----|--|--|--|
| 0                   | 1 | 2 | 3 | 4               | 5 | 6 | 7 | 8                   | 9 | 10 | 11 | 12                 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20                  | 21 | 22             | 23 | 24 | 25 | 26 | 27 | 28           | 29 | 30 | 31 |  |  |  |
| n-tags(4)           |   |   |   | tag1-id(4)      |   |   |   | tag1-len(4)         |   |    |    | tag1-name(tag-len) |    |    |    |    |    |    |    |                     |    |                |    |    |    |    |    |              |    |    |    |  |  |  |
| tag2-id(4)          |   |   |   | tag2-len(4)     |   |   |   | tag2-name(tag-len)  |   |    |    |                    |    |    |    |    |    |    |    |                     |    | .....(剩余的tags) |    |    |    |    |    |              |    |    |    |  |  |  |
| n-actions(4)        |   |   |   | action1-type(n) |   |   |   | action1.arg1.len(4) |   |    |    | action.arg1        |    |    |    |    |    |    |    | action1.arg2.len(4) |    |                |    |    |    |    |    | action1.arg2 |    |    |    |  |  |  |
| action1.arg3.len(4) |   |   |   | action1.arg3    |   |   |   |                     |   |    |    |                    |    |    |    |    |    |    |    |                     |    |                |    |    |    |    |    |              |    |    |    |  |  |  |
| .....(剩余的actions)   |   |   |   |                 |   |   |   |                     |   |    |    |                    |    |    |    |    |    |    |    |                     |    |                |    |    |    |    |    |              |    |    |    |  |  |  |

图 3.7. 序列化格式

首先是一个 `size_t(4)`，表示随后接有 `n` 个 `tag` 数据，之后便跟着 `n` 个 `tag` 数据，每个 `tag` 数据又由一个 `size_t(4)` 的行 `id`、一个表示标签名长度的 `size_t(4)` 和标签名组成，标签部分数据结束后跟着一个 `size_t(4)`，表示随后接着 `n` 个 `Action` 数据，每个 `Action` 又由标签类型(`int, 4`)和 3 个参数组成，每个参数又由表示参数长度的 `size_t(4)` 和参数内容组成。当然，值得注意的是，通过解释器编译出来的二进制文件还是经过加密且增加了文件头的，这是跟本解释器主函数相关的部分，并不属于 `Parser` 的部分。

而后是主程序的主要流程：

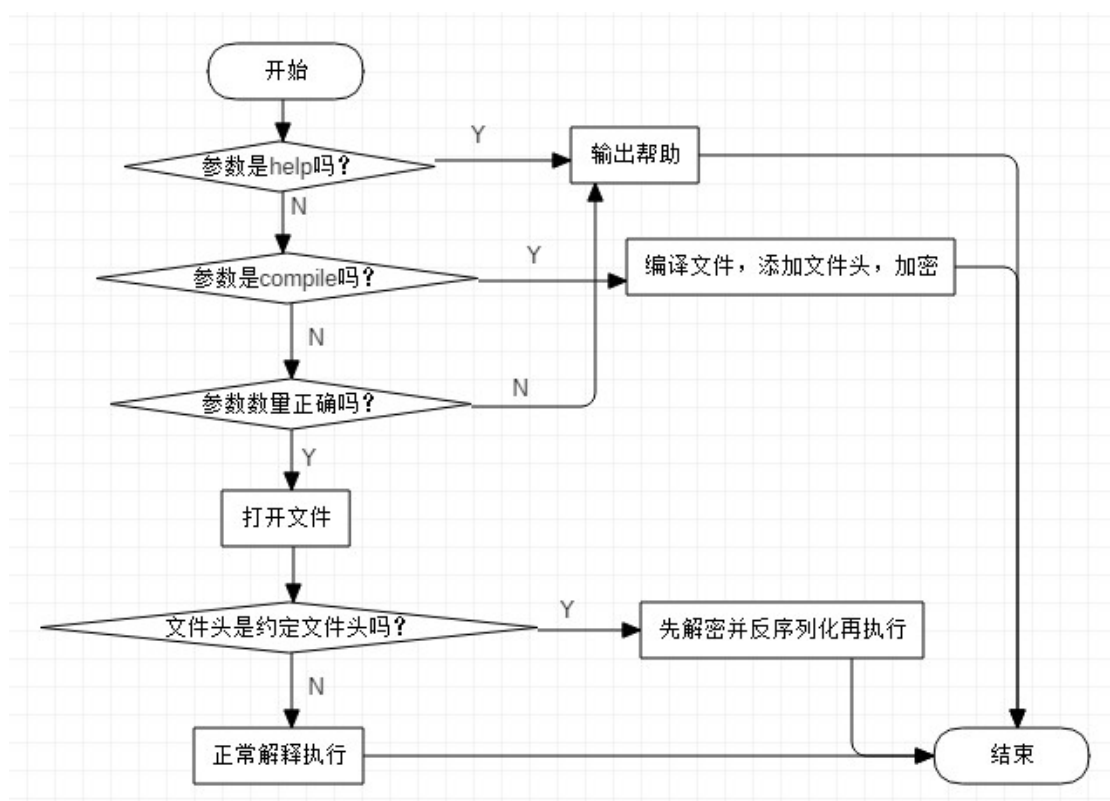
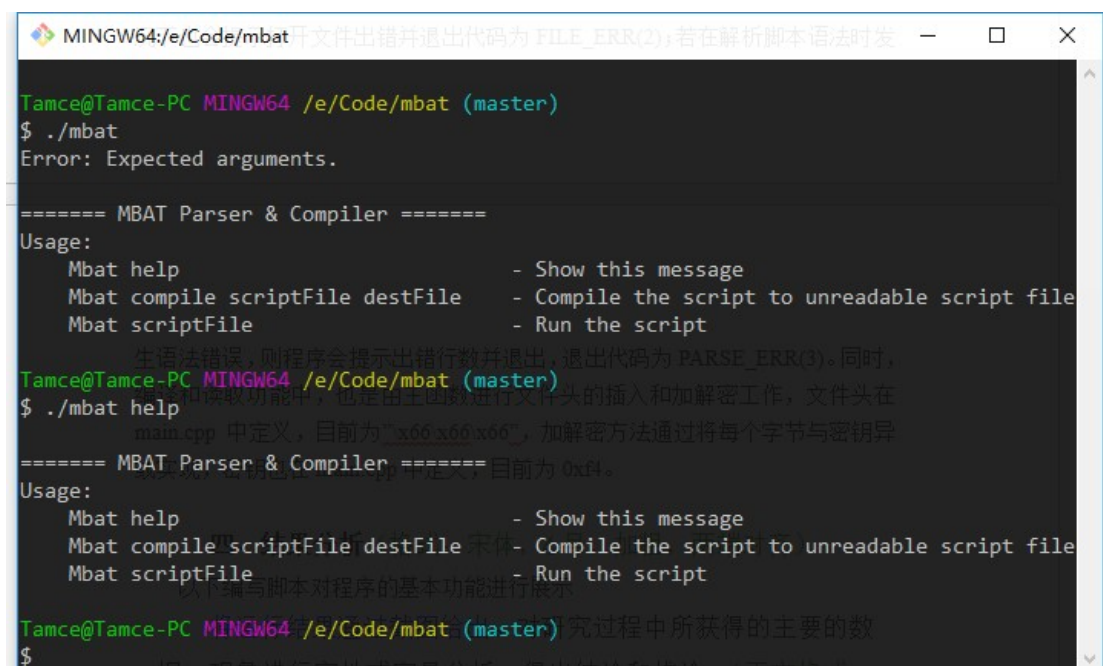


图 3.8. 主程序逻辑

在主程序中对传入参数进行分析决定执行什么操作，参数有误的情况下会提示参数有错误并且退出代码为预先定义的 `INVALID_ARG(1)`；在打开文件出错的情况下也会提示打开文件出错并退出代码为 `FILE_ERR(2)`；若在解析脚本语法时发生语法错误，则程序会提示出错行数并退出，退出代码为 `PARSE_ERR(3)`。同时，编译和读取功能中，也是由主函数进行文件头的插入和加解密工作，文件头在 `main.cpp` 中定义，目前为 `"\x66\x66\x66"`，加解密方法通过将每个字节与密钥异或实现，密钥也在 `main.cpp` 中定义，目前为 `0xf4`。

## 四、结果分析

以下编写各种脚本对程序的基本功能进行展示，需要执行一个脚本文件时，可以通过直接拖入文件执行，也可以选择使用参数指定；不带参数运行程序和使用 `help` 命令运行程序时会显示帮助信息，并且不带参数时会提示参数错误，如下图所示：



```
MINGW64:/e/Code/mbat
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat
Error: Expected arguments.

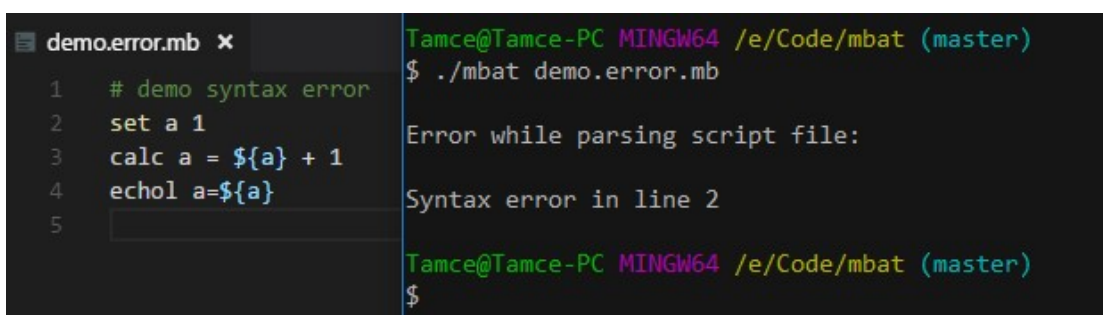
===== MBAT Parser & Compiler =====
Usage:
  Mbat help                    - Show this message
  Mbat compile scriptFile destFile - Compile the script to unreadable script file
  Mbat scriptFile              - Run the script

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat help
===== MBAT Parser & Compiler =====
Usage:
  Mbat help                    - Show this message
  Mbat compile scriptFile destFile - Compile the script to unreadable script file
  Mbat scriptFile              - Run the script

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$
```

图 4.1 不带参数和使用 `help` 参数运行

接下来编写一个带有语法错误的脚本，将其保存在 `demo.error.mb`，脚本内容和执行结果如图 4.2 所示：



```
demo.error.mb x
1  # demo syntax error
2  set a 1
3  calc a = ${a} + 1
4  echol a=${a}
5

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat demo.error.mb
Error while parsing script file:
Syntax error in line 2

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$
```

图 4.2 `demo.error.mb` 内容和执行结果



图 4.3 展示了尝试跳转到一个未定义标签的脚本的执行结果

|   |  |
|---|--|
| <pre>demo.error.mb x 1 2   set a = 1 3   calc a = \${a} + 1 4   echo! a=\${a} 5   goto notExist 6</pre> | <pre>Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$ ./mbat demo.error.mb a=2 Runtime Error: Tag `notExist` not found! Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$  </pre> |
|---|--|

图 4.3 跳转到未定义标签

可以看到，在跳转之前的所有命令都能正确执行，而在执行跳转命令时发现标签未定义，程序抛出异常随后终止运行。同样的，在进行表达式的计算时和对判断条件进行计算时也会发生运行时错误，如图 4.4 所示

|  |  |
|--|--|
| <pre>demo.error.mb x 1   set a = 1 2   calc a = \${a} + 1 3   echo! a = "\${a}" 4   calc a = a + 1 5   echo! a = "\${a}"</pre> | <pre>Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$ ./mbat demo.error.mb a = "2" Runtime Error: Cannot evaluate expression `a + 1`.</pre> |
|--|--|

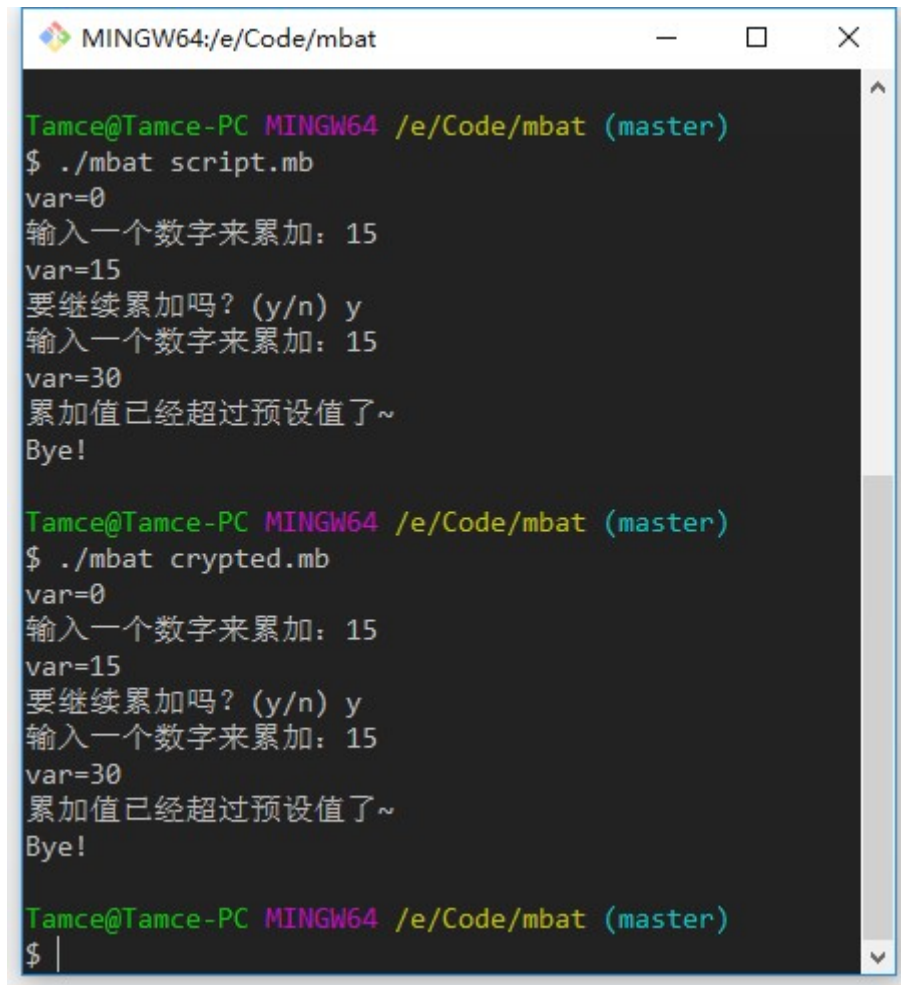
图 4.4 发生运行时错误

接下来展示编译功能，还是以本文第三节所介绍语法的脚本为例，脚本内容以及执行的编译指令和编译后输出的文件内容展示如图 4.5 所示：

|   |  |
|---|--|
| <pre>script.mb x 1 # Mbat 语法演示 2 # 以 # 开头的行为注释行 3 4 # 在除了命令的任何位置均可以使用变量，格式为 \${variableName} 5 # 设置变量值 6 set var = 0 7 set MAX = 20 8 # 输出然后换行 9 echo! var=\${var} 10 # 定义一个标签 11 :tag start 12 # 输出但不换行 13 echo 输入一个数字来累加: 14 # 输入一个值到变量中 15 input i 16 # 计算一个表达式，结果保存到变量中，支持加减乘除和求模运算 17 calc var = \${var} + \${i} 18 echo! var=\${var} 19 # if 语句，支持 '&gt;', '&lt;', '&lt;=' 三种比较，忽略比较符号前后的空白符 20 if [\${var}] &gt; \${MAX}] goto tooBig 21 echo 要继续累加吗? (y/n) 22 input i 23 if [\${i}] = y] goto start 24 echo! Bye! 25 # 跳转语句，其中 eof 标签保留为文件末 26 goto eof 27 :tag tooBig 28 echo! 累加值已经超过预设值了~ 29 echo! Bye! 30 goto eof 31</pre> | <pre>MINGW64/e/Code/mbat Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$ ./mbat compile script.mb crypted.mb Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$ cat crypted.mb [乱码输出] Tamce@Tamce-PC MINGW64 /e/Code/mbat (master) \$  </pre> |
|---|--|

图 4.5 编译前和编译后的文件比较

可以看到编译后的文件内容完全不可读，为二进制文件，但是编译后的文件依然可以执行，且执行效果和编译前的完全一致，如图 4.6 所示：



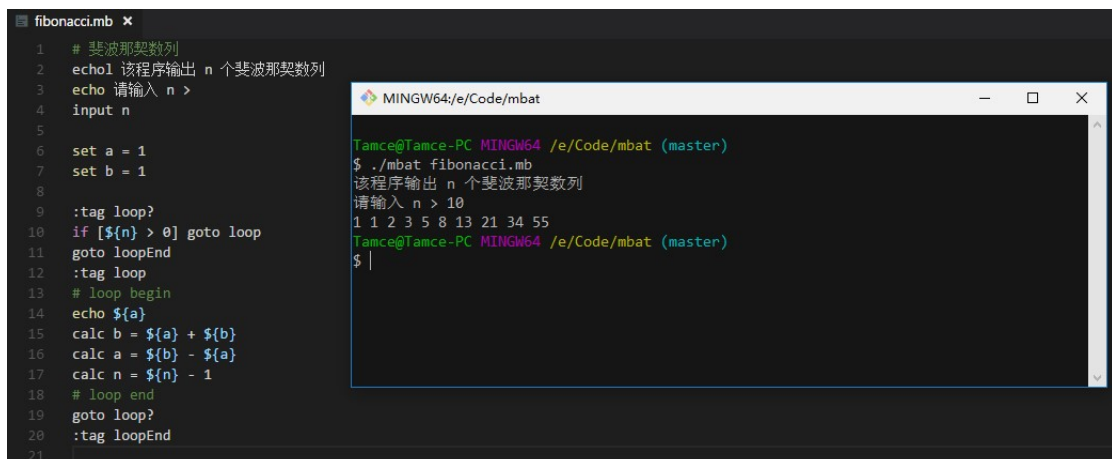
```
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat script.mb
var=0
输入一个数字来累加: 15
var=15
要继续累加吗? (y/n) y
输入一个数字来累加: 15
var=30
累加值已经超过预设值了~
Bye!

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat crypted.mb
var=0
输入一个数字来累加: 15
var=15
要继续累加吗? (y/n) y
输入一个数字来累加: 15
var=30
累加值已经超过预设值了~
Bye!

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ |
```

图 4.6 编译前和编译后文件的执行效果

通过脚本定义的指令，我们几乎可以组合出任何一般语言能够进行的基本操作，包括分支、循环、函数调用等，但缺点也十分明显，那就是阅读起来十分困难，甚至跟汇编语言有的一拼，这是因为该脚本语言目前定义的指令还十分少的缘故；下图（4.7）展示了使用 MBat 脚本语言编写的输出斐波那契数列的程序：

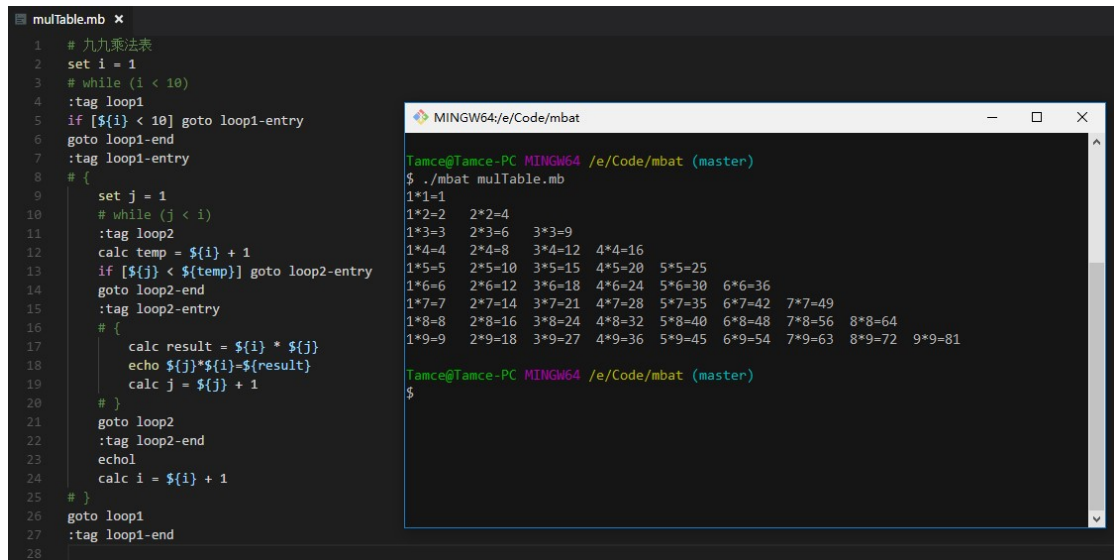


```
1 # 斐波那契数列
2 echo! 该程序输出 n 个斐波那契数列
3 echo 请输入 n >
4 input n
5
6 set a = 1
7 set b = 1
8
9 :tag loop?
10 if [{n} > 0] goto loop
11 goto loopEnd
12 :tag loop
13 # loop begin
14 echo ${a}
15 calc b = ${a} + ${b}
16 calc a = ${b} - ${a}
17 calc n = ${n} - 1
18 # loop end
19 goto loop?
20 :tag loopEnd
21
```

```
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat fibonacci.mb
该程序输出 n 个斐波那契数列
请输入 n > 10
1 1 2 3 5 8 13 21 34 55
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ |
```

图 4.7 使用 MBat 语言编写的斐波那契数列程序

例如下图（图 4.8）展示了使用 MBat 脚本语言编写的输出九九乘法表的程序：



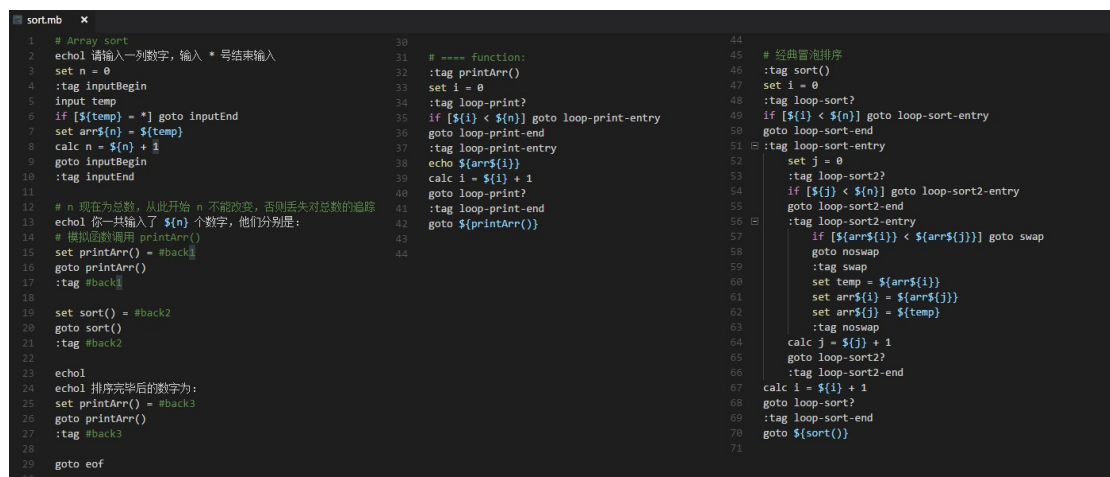
```
mulTable.mb
1 # 九九乘法表
2 set i = 1
3 # while (i < 10)
4 :tag loop1
5 if [${i} < 10] goto loop1-entry
6 goto loop1-end
7 :tag loop1-entry
8 # {
9   set j = 1
10  # while (j < i)
11  :tag loop2
12  calc temp = ${i} + 1
13  if [${j} < ${temp}] goto loop2-entry
14  goto loop2-end
15  :tag loop2-entry
16  # {
17    calc result = ${i} * ${j}
18    echo ${j}*${i}=${result}
19    calc j = ${j} + 1
20  # }
21  goto loop2
22 :tag loop2-end
23 echol
24 calc i = ${i} + 1
25 # }
26 goto loop1
27 :tag loop1-end
28
```

```
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat mulTable.mb
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$
```

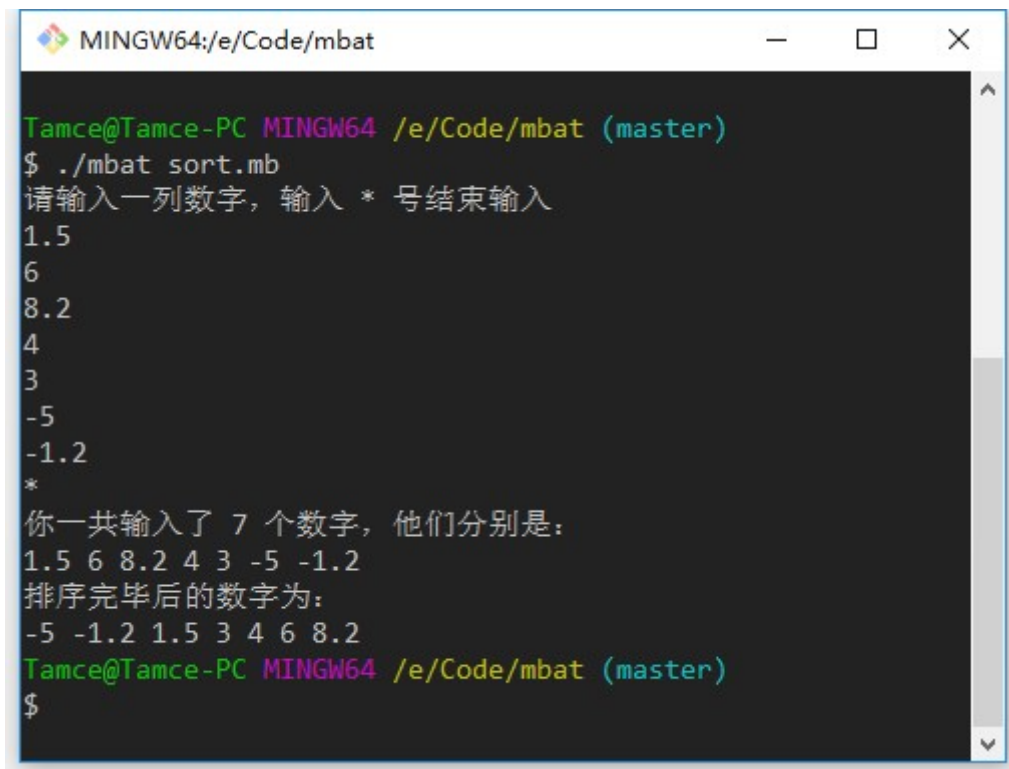
图 4.8 使用 MBat 语言编写的九九乘法表

图 4.9 和图 4.10 分别展示了 MBat 编写的冒泡排序程序源码和运行结果：



```
sort.mb
1 # Array sort
2 echol 请输入一系列数字，输入 * 号结束输入
3 set n = 0
4 :tag inputBegin
5 input temp
6 if [${temp} = *] goto inputEnd
7 set arr[${n}] = ${temp}
8 calc n = ${n} + 1
9 goto inputBegin
10 :tag inputEnd
11
12 # n 现在为总数，从此开始 n 不能改变，否则丢失对总数的追踪
13 echol 你一共输入了 ${n} 个数字，他们分别是：
14 # 模拟函数调用 printArr()
15 set printArr() = #back1
16 goto printArr()
17 :tag #back1
18
19 set sort() = #back2
20 goto sort()
21 :tag #back2
22
23 echol
24 echol 排序完毕后的数字为：
25 set printArr() = #back3
26 goto printArr()
27 :tag #back3
28
29 goto eof
30
31 # ---- function:
32 :tag printArr()
33 set i = 0
34 :tag loop-print?
35 if [${i} < ${n}] goto loop-print-entry
36 goto loop-print-end
37 :tag loop-print-entry
38 echo ${arr[${i}]}
39 calc i = ${i} + 1
40 goto loop-print?
41 :tag loop-print-end
42 goto ${printArr()}
43
44 # 经典冒泡排序
45 :tag sort()
46 set i = 0
47 :tag loop-sort?
48 if [${i} < ${n}] goto loop-sort-entry
49 goto loop-sort-end
50 :tag loop-sort-entry
51 set j = 0
52 :tag loop-sort2?
53 if [${j} < ${n}] goto loop-sort2-entry
54 goto loop-sort2-end
55 :tag loop-sort2-entry
56 if [${arr[${i}]} < ${arr[${j}]}] goto swap
57 goto noswap
58 :tag swap
59 set temp = ${arr[${i}]}
60 set arr[${i}] = ${arr[${j}]}
61 set arr[${j}] = ${temp}
62 :tag noswap
63 calc j = ${j} + 1
64 goto loop-sort2?
65 :tag loop-sort2-end
66 calc i = ${i} + 1
67 goto loop-sort?
68 :tag loop-sort-end
69 goto ${sort()}
70
71
```

图 4.9 MBat 语言编写的冒泡排序源码



```
MINGW64:/e/Code/mbat

Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$ ./mbat sort.mb
请输入一系列数字，输入 * 号结束输入
1.5
6
8.2
4
3
-5
-1.2
*
你一共输入了 7 个数字，他们分别是：
1.5 6 8.2 4 3 -5 -1.2
排序完毕后的数字为：
-5 -1.2 1.5 3 4 6 8.2
Tamce@Tamce-PC MINGW64 /e/Code/mbat (master)
$
```

图 4.10 MBat 语言编写的冒泡排序运行结果

除此之外，我还使用 Mbat 语言编写了一个简单的文字冒险游戏 MBWORLD，通过拖入脚本文件来打开，游戏打开时画面如下图（图 4.11）



图 4.11 MBWORLD 游戏开始画面

游戏进程的选项如下图（图 4.12）

```
D:\Webroot\mbat\MBat.exe
HP: 100
ATK: 15    DEF: 10
Score: 0
----- Tamce -----
##### 注意 #####
当程序提示 ... 的时候表示等待用户按下回车
当程序提示 > 的时候表示等待用户输入并按下回车
#####
准备好了就按下回车开始吧!
...
你推开了 MBWORD 的大门，走进了迷宫
来到大厅，你突然发现你面前有一个样子奇特的神秘生物
-----
                    要怎么做？
-----
a. 尝试向其表示友好    b. 不理它    c. 直接向其发起攻击
-----
> b
你选择不理睬神秘生物
当你准备偷偷的绕过神秘生物时，突然，神秘生物发现了你!
...
-----
                    要怎么做？
-----
a. 先发制人，对其发起攻击    b. 向迷宫深处跑
    c. 我是和平主义者，尝试与其交流
-----
>
```

图 4.12 MBWORLD 游戏进程中



下图（4.13）展示了使用 Mbat 编写的游戏的战斗系统：

```
D:\Webroot\mbat\MBat.exe
===== 战斗发生 =====
                    先攻
=====
对手 神秘生物
  HP: 15
 ATK: 14 DEF: 10
-----
玩家 Tamce
  HP: 100
 ATK: 15 DEF: 10
=====
...
Tamce 对 神秘生物 发起攻击! 造成了 5 点伤害! 神秘生物 剩余 HP 变为: 10!
...
神秘生物 对 Tamce 进行攻击! 造成了 4 点伤害! Tamce 剩余 HP 变为: 96!
...
Tamce 对 神秘生物 发起攻击! 造成了 5 点伤害! 神秘生物 剩余 HP 变为: 5!
...
神秘生物 对 Tamce 进行攻击! 造成了 4 点伤害! Tamce 剩余 HP 变为: 92!
...
Tamce 对 神秘生物 发起攻击! 造成了 5 点伤害! 神秘生物 剩余 HP 变为: 0!
...
神秘生物 倒下了! Tamce 获得了胜利!
...
战斗结束, Tamce 获得了 5 分! 现在总分为: 5!
由于战斗带来的经验, Tamce 的攻击力增加了!
由于战斗带来的经验, Tamce 的防御力增加了!
Tamce 获得了 2 点攻击力, 现在攻击力为 17!
Tamce 获得了 1 点防御力, 现在防御力为 11!
...

```

图 4.13 MBWORLD 的战斗系统

下图（图 4.14）展示了使用 Mbat 语言编写的该游戏的源代码的一部分，源代码可以直接拖入解释器运行，也可以从命令行将文件名传递给解释器，同样的，源代码经过编译后也可以正常解释运行。

```
mbworld.mb x
301 # ===== Functions =====
302
303 # Function: battle
304 # Required: battle.name battle.hp battle.atk battle.def battle.first battle.score
305 :tag battle()
306 # 先打印战斗信息
307 set battleInfo() = #3
308 goto battleInfo()
309 :tag #3
310 echo ...
311 input choice
312 # 战斗过程
313 if [{battle.first} > 0] goto battle-player-atk
314 goto battle-enemy-atk
315 # 玩家攻击回合
316 :tag battle-player-atk
317 calc battle.damage = ${player.atk} - ${battle.def}
318 calc battle.hp = ${battle.hp} - ${battle.damage}
319 echo ${player.name} 对 ${battle.name} 发起攻击! 造成了 ${battle.damage} 点伤害!
320 echol ${battle.name} 剩余 HP 变为: ${battle.hp}!
321 echo ...

```

图 4.14 MBWORLD 源代码的一部分

## 五、课程设计总结

在做这个课程设计的过程中，也遇到过一些奇怪的 BUG，最终也都通过在代码中插入特定的调试代码，输出标记执行顺序，顺利解决了一些 BUG，在设计上会先思考出简单的实现方式，而往往在实现了一部分之后又会冒出更好一些点子，于是就会纠结是否要改变架构、是否要考虑更强的拓展性等等。通过这个课程设计，因为自己定义的脚本语言能够做的事情不多，也更深刻感受到了能做事局限性所带来的不便利和高级语言的很多特性时多么方便，特别是关于块级作用域和函数、循环、分支控制的特性上，因为这个脚本设计的语法相对比较简单，虽然能够实现那些功能，但是明显工作量和阅读的复杂程度都上升了很多。同时，本次课设的代码也是用了 git 进行管理，在 github 上以 GPLv2 协议进行托管，所有相关的源代码以及脚本都可以在我的 github 仓库([github.com/tamce/mbat](https://github.com/tamce/mbat))上获得，本次课设也进一步熟悉掌握了 git 的使用，总之这次的课设是可以说是一次比较宝贵的经历。

## 参考文献

- [1] c++ community. cppreference.com . <http://en.cppreference.com/w/>,  
2017. 8. 24/2017. 8. 31