# 華南師範大學

## 本科学生实验（实践）报告

院　　系：计 算 机 学 院

实验课程：编译原理实验

实验项目：比特大战

指导老师：王欣明

开课时间：2017 ~ 2018 年度第 2 学期

专　　业：软件技术与应用

班　　级：2015 级软工 4 班

学　　生：詹萍

学　　号：20152100027

华南师范大学教务处

# 目录

# 一、实验内容

## 1、游戏介绍

这个游戏来源于《自私的基因》。一个 N 回合的比特大战由 A 和 B 两方进行，每个回合 A 和 B 可以选择 0（背叛）或 1（合作），共进行 N 个回合。如果第n（$1 \le n \le N$）个回合 A 和 B 的选择分别为 $A_n$ 和 $B_n$，则他们在这个回合分别得分 $S_A(n)$ 和 $S_B(n)$ 由下表决定：

| | $B_n = 0$ | $B_n = 1$ |
|---|---|---|
| $A_n = 0$ | $S_A(n) = 1, S_B(n) = 1$ | $S_A(n) = 5, S_B(n) = 0$ |
| $A_n = 1$ | $S_A(n) = 0, S_B(n) = 5$ | $S_A(n) = 3, S_B(n) = 3$ |

A 和 B 的总分为 N 个回合各自得分的和。

## 2、游戏的策略

对于 N 个回合的比特大战，每回合的得分可能是 0,1,5，因此，总得分总是在 0 和 $5 * N$ 之间。双方不同的策略可能导致不同的得分。以下使一些可能的策略：

**（T1）永远合作：**每次都选择 1；

**（T2）随机：**每次咦某个概率随机选择 1，否则选择 0；

**（T3）针锋相对：**第一次选择 1，以后每次都选择对方的上一次选择；

**（T4）老实人探测器：**基本上和“针锋相对”一样，只是会随机的选择一次 0；

**（T5）永不原谅：**一直选 1，一旦对方选择 0，则一直选择 0。

这些策略可以用如下的算法描述：

**T1 永远合作**

```
Strategy T1://T1 表示策略的名字
Return 1;
```

**T2 随机**

```
Strategy T2://以 0.75 的概率返回 1
i = RANDOM(3);//随机函数，RANDOM(k)等概率返回 0 到 k 之间的一个整数
if(i == 3)
    return 0;
```

```
else
    return 1;
```

**T3 针锋相对**

```
Strategy T3:
//CUR 标识当前的回合数
//A[1…N]和 B[1…N]是两个预定义数组，分别保存自己和对手每次的选择
//在第 CUR 回合可以访问数组的前 CUR-1 个元素
if(CUR == 1)
    return 1;
else
    return B[CUR-1];
```

**T4 老实人探测器**

```
Strategy T4:
if(CUR == 1)
    return 1
else {
    i = RANDOM(9);
    if(i == 9)
        return 0;
    else
        return B[CUR-1];
}
```

**T5 永不原谅**

```
Strategy T5:
i = 1;
k = 1;
while((k < CUR)&&(i == 1)){
    if(B[k] == 0)
        i = 0;
    k = k + 1;
}
return i;
```

**3、问题描述**

（B1）设计一种程序设计语言可以用来描述比特大战的策略。这个语言应该至少可以描述上述 T1-T5 的策略，并用这个语言描述更多的策略。

（B2）实现对这个语言的语法分析。定义相应的语法树，并且可以输出语法分析的结果。当出现可能的输入错误时，可以指出出错的位置和可能的错误原因。

（B3)实现一个程序:用户可以输入若干的策略,每个策略保存为一个文本文件，模拟这些策略两两之间的 N 回合（例如 N = 200）的比特大战，并以所有对战得分的总和为这些策略排序。

## 二、实现思路

### 1、设计描述比特大战的语言

#### （1）文法规则

```
program -> func id stmt-list endf
stmt-list -> {stmt}
stmt -> if-stmt | while-stmt | assign-stmt | return-stmt
if-stmt -> if exp then stmt-list [else stmt-list] endi
while-stmt -> while exp do stmt-list endw
assign-stmt -> id is exp
return-stmt -> return exp
exp -> logic-exp [logic logic-exp]
logic -> and | or
logic-exp -> simple-exp [comp simple-exp]
comp -> > | < | =
simple-exp -> term {addop term}
addop -> + | -
term -> factor {mulop factor}
mulop -> * | /
factor -> (exp) | num | id | random | const
random -> random(exp)
```

const -> current | my | opponent

my -> my(exp)

opponent -> opponent(exp)

**（2）文法解释**

[]:表示出现 0 或 1 次。

{}:表示出现次数 ≥ 0。

current:当前的回合数。

my(exp):自己的前面第 exp 回合的结果。

opponent(exp):对手前面第 exp 回合的结果。

**（3）文法单词**

关键字：func endf、if then else endi、while do endw、is、return、and or、comp、addop mulop、random、current my opponent

数字：num -> [0-9]+

标识符：id -> [a-zA-Z][0-9]*

**2、对策略进行语法分析**

**（1）输入处理**

用户的策略保存在文件中，程序从文件读入用户的策略，根据单词之间的间隔（空白）来分割文件的内容，转换成单词流保存起来。

相应功能代码如下：

根据文件的字符串来获取单词序列 tokens。

```
//根据输入的策略文件来获取单词(Token)序列
public static TokenList getTokens(String code) throws
CompileException{
    List<String> tokens = new ArrayList<>();
    String token = "";
    for(int i=0;i<code.length();i++){
        //判断当前字符是否为空格或者是否为运算符
```

```
        if((code.charAt(i) == ' ' || isOp(code.charAt(i)))){
            //判断是不是一个单词(Token)
            if(isToken(token)) {
                tokens.add(token);
                token = "";
            }
            //如果没有单词而是一个空串表明出错
            else if(!token.matches("[\\s]?")){
                throw new CompileException(token + " isn't a token");
            }
        }
        //将不是运算符和空格的字符连接成为一个 token
        else token += code.charAt(i);
        //如果是运算符直接将其作为一个 token
        if(isOp(code.charAt(i)))
            tokens.add(String.valueOf(code.charAt(i)));
    }
    //最终返回得到的单词序列
    return new TokenList(tokens);
}
```

**（2）生成语法树**

我们要根据上述文法规则来生成语法树，语法树上不同的结点是由程序的单词流来形成的。根据不同的单词类型，语法树的结点类型有以下 18 种：

ProgramTree、StmtListTree、StmtTree、IfTree、WhileTree、AssignTree、ReturnTree、ExpTree、LogicExpTree、SimpleExpTree、TermTree、FactorTree、RandomTree、ConstTree、CurrentTree、MyTree、OpponentTree、Leaf

对于这些不同的结点，目前有一个共同的操作，就是生成语法树，我们可以定义一个接口：Tree，其包含方法 grow()，用来根据单词流生成语法树，上述的结点都继承自这个接口，则都需要实现方法 grow()。

接口 Tree 的 grow()方法：tokens 是用来编译的单词流。

```
//根据单词流生长语法树
void grow(TokenList tokens) throws CompileException;
```

各个结点实现 grow()方法的依据是上述文法。各个结点对应的文法如下：

**ProgramTree:** program -> func id stmt-list endf

**StmtListTree:** stmt-list -> {stmt}

**StmtTree:** stmt -> if-stmt | while-stmt | assign-stmt | return-stmt

**IfTree:** if-stmt -> if exp then stmt-list [else stmt-list] endi

**WhileTree:** while-stmt -> while exp do stmt-list endw

**AssignTree:** assign-stmt -> id is exp

**ReturnTree:** return-stmt -> return exp

**ExpTree:** exp -> logic-exp [logic logic-exp]

**LogicExpTree:** logic-exp -> simple-exp [comp simple-exp]

**SimpleExpTree:** simple-exp -> term {addop term}

**TermTree:** term -> factor {mulop factor}

**FactorTree:** factor -> (exp) | num | id | random | const

**RandomTree:** random -> random(exp)

**ConstTree:** const -> current | my | opponent

**CurrentTree:** current

**MyTree:** my -> my(exp)

**OpponentTree:** opponent -> opponent(exp)

**Leaf:** id / num

ProgramTree 实现 grow()的代码：

```
//从树根开始根据单词流生长语法树
@Override
public void grow(TokenList tokens) throws CompileException{
    //program -> func id stmt-list endf
    //句子以 func 为开始符号
    if(!tokens.read().equals("func")) throw new CompileException("not
start with 'func'");
    //tokens.read() 读取下一个单词
    String value = tokens.read();
    //判断是不是一个 id
    if(isId(value)) id = new Leaf(value);
    else throw new CompileException("function name is not an id");
    //根据中间的 stmtlist 单词流继续生长语法树
    stmtList = new StmtListTree();
    stmtList.grow(tokens);
    //句子应该以 endf 结尾
    if(!tokens.read().equals("endf")) throw new
CompileException("function is not finished by 'endf'");
}
```

其他结点实现 grow()的代码可以类比 ProgramTree。

### （3）打印语法树

在生成语法树之后，就可以将这棵语法树以缩进的方式打印出来。对于上述定义的不同的结点，增加一个共同的操作，打印语法树。可以在一个接口 Tree 增加方法 print()，用来打印编译好的语法树，上述的结点都继承自这个接口，则都需要实现方法 print()。

接口 Tree 的 print()方法：deep 表示当前节点在树中的深度。

```
//输出语法树
void print(int deep);
```

ProgramTree 实现 print()的代码：

```
//根据层数 deep 来输出每一层的单词
@Override
public void print(int deep) {
    Parser.printWord(deep, "func " + id.getValue());
    stmtList.print(deep + 1);
    Parser.printWord(deep, "endf");
}
```

如果是叶子结点就输出 token 或者 id 或者 num 的值；如果是非叶子结点，就进入下一层继续打印输出。

### 3、进行比特大战

对两个策略进行多轮对战。每次运行前需要标识出当前是玩家 1 还是玩家 2 在进行游戏，我们定义一些值来区分两个玩家：

```
//玩家 1 回合
public static final int FIRST_RUN = 1;
//玩家 2 回合
public static final int SECOND_RUN = 2;
//要取得玩家 1 的历史回合数
public static final int MY_CURRENT = 1;
//要取得玩家 2 的历史回合数
public static final int OPPONENT_CURRENT = 2;
```

将上述的值保存在 localVal 中，每次在运行玩家的策略之前，设置好 localVal，作为参数传递到整棵语法树的运行过程中：

```
//标识当前玩家信息
private Map<String, Integer> localVal;
```

对于上述定义的不同的结点，增加一个共同的操作，运行语法树。可以在一个接口 Tree 增加方法 run()，用来运行编译好的语法树，上述的结点都继承自这个接口，则都需要实现方法 run()。

接口 Tree 的 run()方法：localVal 保存当前玩家信息。

```
//运行当前策略对应的语法树
Integer run(Map<String, Integer> localVal) throws RunningException;
```

ProgramTree 实现 run()的代码：

```
//运行当前策略对应的语法树
@Override
public Integer run(Map<String, Integer> localVal) throws
RunningException {
    return stmtList.run(localVal);
}
```

注意到在策略 3、4、5 需要使用到对手的历史对战结果，因此在 OpponentTree 中需要获取对手的信息：

```
//运行当前策略对应的语法树
@Override
public Integer run(Map<String, Integer> localVal) throws
RunningException {
    //OPPONENT_CURRENT = 2
    //更新 current 的值
    localVal.put("current", Program.OPPONENT_CURRENT);
    int arg = exp.run(localVal);
    int index = localVal.get("my");
    //FIRST_RUN = 1
    //玩家 1 的回合
    if(index == Program.FIRST_RUN){
        //arg = 0 获取玩家 2 的对战轮数
        if (arg == 0) return ((List)
GlobalValue.getGlobalVal("history2")).size();
        if (arg < 0 || arg >= ((List)
GlobalValue.getGlobalVal("history2")).size()){
            throw new RunningException("current index out of array
'history2'");
        }
        return (Integer) ((List)
GlobalValue.getGlobalVal("history2")).get(arg);
    }
    //SECOND_RUN = 2
    //玩家 2 的回合
    if(index == Program.SECOND_RUN){
```

```
        //arg = 0 获取玩家 1 的对战轮数
        if (arg == 0) return ((List)
GlobalValue.getGlobalVal("history1")).size();
        if (arg < 0 || arg >= ((List)
GlobalValue.getGlobalVal("history1")).size()){
            throw new RunningException("current index out of array
'history1'");
        }
        return (Integer) ((List)
GlobalValue.getGlobalVal("history1")).get(arg);
    }
    else {
        throw new RunningException("my-stmt cannot run in single
mode");
    }
```

4、异常处理

（1）编译异常

在生成语法树的过程中出现的错误均属于编译错误。

异常信息（如：对一个策略文件，在生成语法树时不符合文法规则）通过参数传递到以下函数，在 cmd 窗口抛出错误提示信息。

```
public class CompileException  extends Exception {

    public CompileException(String message) {
        super("compile error: " + message);
    }

}
```

（2）运行时异常

在进行比特大战，即运行语法树的过程中出现的错误均属于运行时错误。

异常信息（如：无法获取对手信息，访问数组越界等）通过参数传递到以下函数，在 cmd 窗口抛出错误提示信息。

```
public class RunningException  extends Exception {

    public RunningException(String message) {
```

```
        super("runtime error: " + message);
    }

}
```

### （3）命令异常

用户输入命令格式出错、输入的文件名对应的文件不存在等情况。

异常信息（如：文件不存在、命令不存在等）通过参数传递到以下函数，在 cmd 窗口抛出错误提示信息。

```
public class CmdException  extends Exception {

    public CmdException(String message){
        super("error input format: " + message);
    }

}
```

## 三、实验结果

### 1、运行程序

```
          ******   welcome to bitwar   ******
 ========================================================
 | command | format                                       |
 ========================================================
 |  load   | load / load file                             |
 |  list   | list                                         |
 |  show   | show file                                    |
 |  run    | run file                                     |
 |  battle | battle times / battle file1 file2 times      |
 |  exit   | exit                                         |
 ========================================================
 input:
```

2、加载策略文件并编译生成语法树

（1）载入单个策略文件

```
input:load t1.txt
t1.txt: compile success
successfully load strategy 't1.txt'
```

（2）载入全部策略文件

```
t1.txt: compile success
t2.txt: compile success
t3.txt: compile success
t4.txt: compile success
t5.txt: compile success
successfully load strategies from default 'strategy'
```

3、列出编译好的文件

```
input:list
t5.txt
t1.txt
t3.txt
t4.txt
t2.txt
```

4、打印语法树

（1）Strategy 1

```
input:show t1.txt
------- t1.txt -------
func t1
    return
        1
    -return
endf
------- end -------
```

（2）Strategy 2

```
input:show t2.txt
------- t2.txt -------
func t2
    if
        random
            4
        -random
        =
        3
        then
            return
                0
            -return
        -then
        else
            return
                1
            -return
        -else
    endi
endf
------- end -------
```

（3）Strategy 3

```
input:show t3.txt
------- t3.txt -------
func t3
    if
        current
        =
        1
        then
            return
                1
            -return
        -then
        else
            return
                opponent
                    current
                    -
                    1
                -opponent
            -return
        -else
    endi
endf
------- end -------
```

（4）Strategy 4

```
input:show t4.txt
------- t4.txt -------
func t4
    if
        current
        =
        1
        then
            return
                1
            -return
        -then
        else
            if
                random
                    4
                -random
                =
                3
                then
                    return
                        0
                    -return
                -then
                else
                    return
                        opponent
                            current
                            -
                            1
                        -opponent
                    -return
                -else
            endi
        -else
    endi
endf
------- end -------
```

(5) Strategy 5

```
input:show t5.txt
------- t5.txt -------
func t5
    index
    is
    1
    while
        index
        <
        opponent
            0
        -opponent
        do
            if
                opponent
                    index
                -opponent
                =
                0
                then
                    return
                        0
                    -return
                -then
                else
                    index
                    is
                    index
                    +
                    1
                -else
            endi
        -do
    endw
    return
        1
    -return
endf
------- end -------
```

5、运行单个策略

（1）Strategy 1

```
input:run t1.txt
result: 1
```

（2）Strategy 2

```
input:run t2.txt
result: 1
```

（3）Strategy 3、4、5

```
input:run t3.txt
runtime error: my-stmt cannot run in single mode
```

由于 Strategy 3、4、5 均涉及到对手的数据，因此无法再单人模式运行，抛出自定义的异常：runtimeException。

6、进行比特大战

（1）两个策略之间进行对战

```
input:battle t2.txt t5.txt 10

result:      t2.txt  t5.txt
round 1:     1   1
round 2:     1   1
round 3:     1   1
round 4:     1   1
round 5:     1   1
round 6:     1   1
round 7:     0   0
round 8:     1   0
round 9:     0   0
round 10:    1   0

final:  t2.txt=20   t5.txt=30
```

（2）全部策略之间两两进行对战

```
input:battle 200

every battle result:
t5.txt vs t1.txt: 600   600
t5.txt vs t3.txt: 600   600
t5.txt vs t4.txt: 201   206
t5.txt vs t2.txt: 787   72
t1.txt vs t5.txt: 600   600
t1.txt vs t3.txt: 600   600
t1.txt vs t4.txt: 447   702
t1.txt vs t2.txt: 441   706
t3.txt vs t5.txt: 600   600
t3.txt vs t1.txt: 600   600
t3.txt vs t4.txt: 203   208
t3.txt vs t2.txt: 522   522
t4.txt vs t5.txt: 208   208
t4.txt vs t1.txt: 700   450
t4.txt vs t3.txt: 204   204
t4.txt vs t2.txt: 614   394
t2.txt vs t5.txt: 53    788
t2.txt vs t1.txt: 698   453
t2.txt vs t3.txt: 496   496
t2.txt vs t4.txt: 369   614
```

```
final battle result:
t5.txt=4384
t1.txt=4191
t3.txt=3825
t4.txt=3456
t2.txt=3310
```

**7、退出游戏**

```
input:exit
          ******   exit system! bye!   ******

Process finished with exit code 0
```

## 四、实验小结

这次的实验一开始不知道怎么下手，后来在网上查阅资料，无意中找到github上的一份代码（https://github.com/konginyan/bitwar），实现了上述一部分要求，就参考了这个程序的思想，并进行改进。

第一步是设计一个可以描述比特大战策略的文法。我们以类似程序设计语言的形式来描述比特大战的策略，因此也可以用类似编译程序设计语言的语法树来编译这些策略。

第二步是对用户编写的策略文件进行编译，生成语法树。考虑到这个语言生成的语法树的结点有多种类型，因此我们定义不同类型的结点；又因为这些结点有相同的操作：生成语法树、打印语法树、运行语法树（运行策略），因此让这些结点继承自一个接口，这个接口定义这三种操作。

在生成语法树的过程中，按照上述文法规则来检查是否存在语法错误，在每个结点，如果单词流还没读完，就根据文法继续调用下一个层次的结点来进行编译，直到读完整个单词流。如果没有出现错误，表示编译成功；否则，根据异常的信息，抛出编译异常。

在打印语法树的过程中，对于一棵已经编译好的语法树，根据编译时记录的情况（如：在当前选择的是 if 结点等）来输出每个结点的信息，并按照当前的深度来决定下一层的深度。

进行比特大战是整个程序比较复杂的部分，因为运行时不止涉及到单个玩家的信息。因此我们需要记录的信息有：①当前玩家是哪一个 ②玩家每轮对战的历史记录。并且在每个结点，都可以访问这些数据，因此，我们将其作为参数沿

着语法树逐层传递下去。最后对大战的结果进行统计，并进行排序即可。

完成这次的程序设计使我受益良多，对程序编译的过程有了更好的理解，并且学习了自定义异常的方法，以及功能模块划分的思想。