图灵机执行器

该作业作为图灵机的第二次小作业，我们将关注以下几件事情

- 两种接受方式
- 图灵机与编程

在开始作业前，务必仔细阅读代码指导

1. 实验背景

接受

上回说到，对于一个磁带，图灵机有两种接受的方式。磁带，实际上就是一个字符串，如果要严格地定义的话，**字符串**就是定义在Σ上的一个`List`，每一个元素都是Σ中的一个元素，字符串的长度就是其中元素的个数。

如果Σ={a, b}， 那么定义在Σ上的一个字符串S可以是$aaaabbbababaabbbaba$，ϵ表示一个空的字符串，长度为0，Σ^*表示所有定义在Σ上的字符串。**语言**是Σ^*的一个子集。被图灵机所接受的磁带，也就是被图灵机所接受的字符串，就是**被图灵机所定义的语言**。接下来我们讨论一下两种接受方式

实际上两种接受方式是完全等价的。如果迁移函数中有任何以FinalState开始，前往下一个状态的函数，我们将这些函数去掉，此时，对于之前所有可以接受的字符串，现在都会在新的图灵机上halts，但是等价意味着我们不能引入新的可以接受的字符串，我们还需要保证这个新的图灵机不会接受以前不能接受的字符串。

也就是说，如果现在的图灵机上存在一个状态q，他不是FinalState，但是$\delta(q, Z)$也是不存在的，那么这个图灵机就可以在这个状态上停机，也就是说我们引入了新的字符串，要解决这个问题很简单，**留给大家自己思考**

同理，从Halting到FinalState也可以用类似的证明思路证明他们是等价的。

图灵机与算法

能被图灵机所定义的语言也称为**递归可枚举语言(recursively enumerable languages)**，如果一个图灵机以FinalState来接受字符串，同时，不管这个字符串能否以FinalState的方式接受，图灵机一定会停机，那么我们就称这个图灵机为**算法**。被**算法**所定义的语言称为**递归语言**。

如果大家在wiki上搜索算法，最开始的一段话如下

> In mathematics and computer science, an **algorithm** is a **finite sequence** of well-defined, computer-implementable instructions, typically to solve **a class of problems or to perform a computation**. Algorithms are always **unambiguous** and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.

我们很容易可以把标粗的词和刚刚被我们称为算法的图灵机对应起来。有限的序列表示不会陷入死循环，它一定会停止，对应着这段话

> 不管这个字符串能否以FinalState的方式接受，它一定保证图灵机可以停下来

而一类问题则可以看成是被我们定义好的 Σ 所产生的所有字符串

无歧义，表示每一步要么只有一个迁移函数可以选择，要么没有函数可以选择，对应着我们确定性图灵机的概念

到这里，我相信大家对图灵机的认识已经没有那么抽象了！我们通过定义，描绘出了一台机器，这台机器在加上特别的约束后就可以成为**算法**，更神奇的是，设计这台机器的过程，其实就是我们所谓的编程。

图灵机与编程

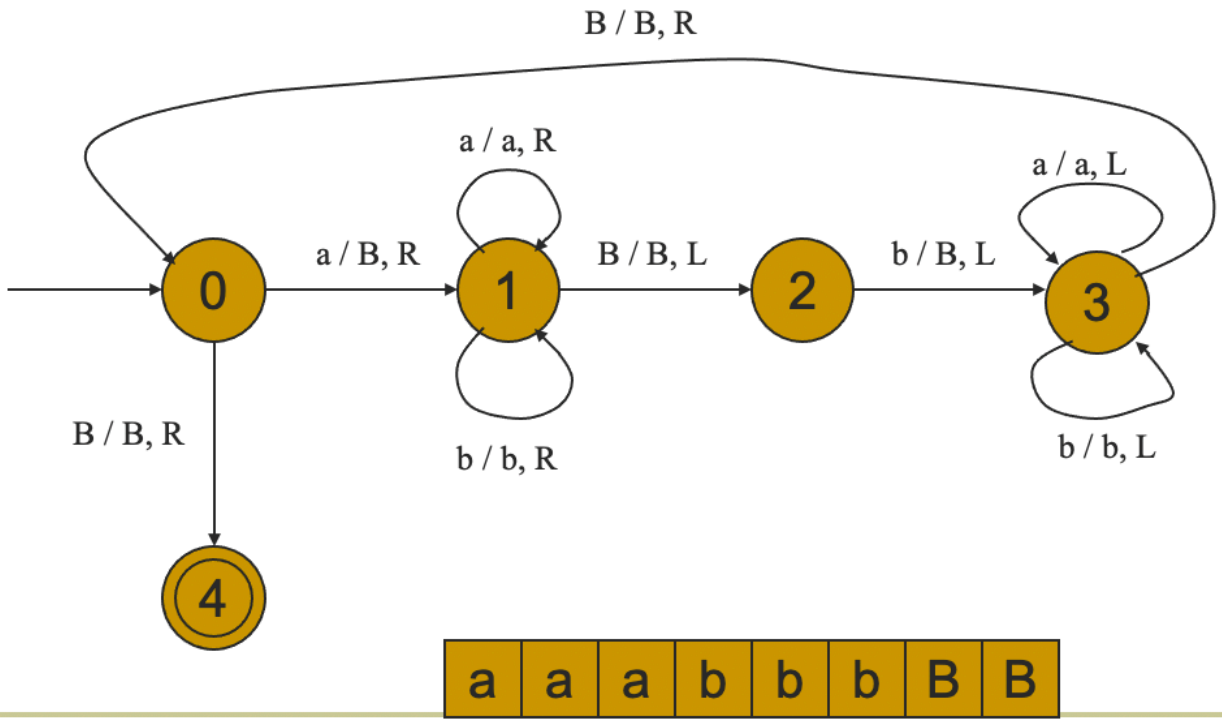
设计一个图灵机其实就是写代码的过程，我们的迁移函数和状态定义了我们的图灵机（也就是代码）会怎么样来运行，磁带就是我们给我们的图灵机（代码）提供的输入，这样我们就把图灵机和编程联系了起来。但是设计图灵机难度和写代码的难度是天差地别的，接下来我们会给出几个例子，并且在最后给出一些造成图灵机设计困难的原因。接下来的每个栗子，我们都会给出两种描述，一种是纯图灵机的描述，一种是和编程相关的描述。

栗子1

描述1：构造一个图灵机来接受这个语言 $L = \{a^n b^n | n \geq 0\}$ ，也就是判定一个字符串是否符合这个形式

描述2：设计一个程序，来识别输入的字符串是否符合 $L = \{a^n b^n | n \geq 0\}$的形式。

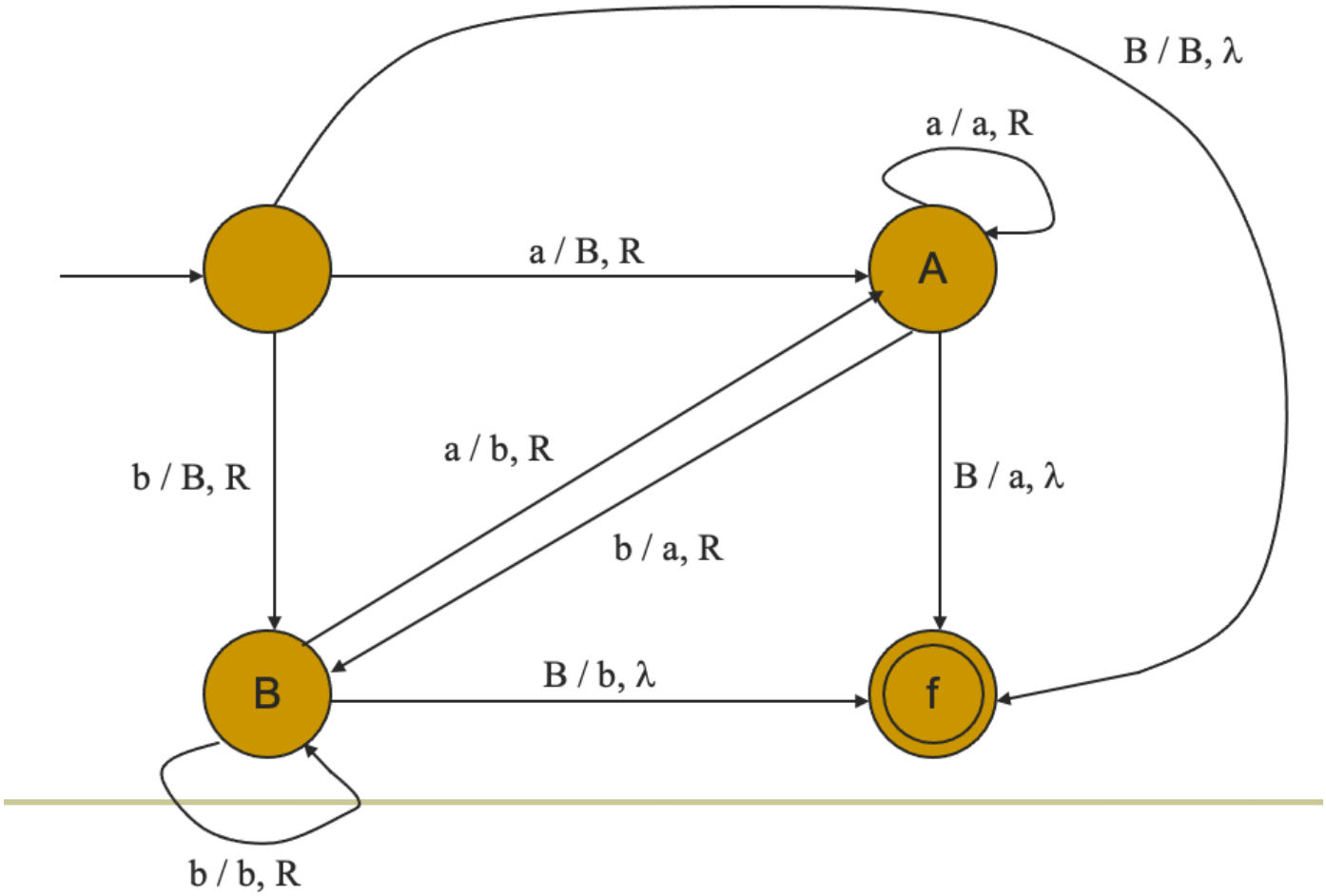
其中每个圆圈表示一个状态，圆圈里面的不同字符表示了不同的状态，如果圆圈里面还有一个小圈，表示这个状态是FinalState，如果从状态A到状态B有箭头从A指向B，表示可以从状态A转移到状态B，箭头上面标志了这次转换需要做的事情，`a / B, R`表示目前磁头指向的磁带上的字符是a时，将它重写为B，并且磁头向右移动一格。后面的图以同样的方式解释



栗子2

描述1：构造一个图灵机来让一个字符串集体右移一位，也就是`abbaBB->BabbaBB`，字符串中只会出现a和b

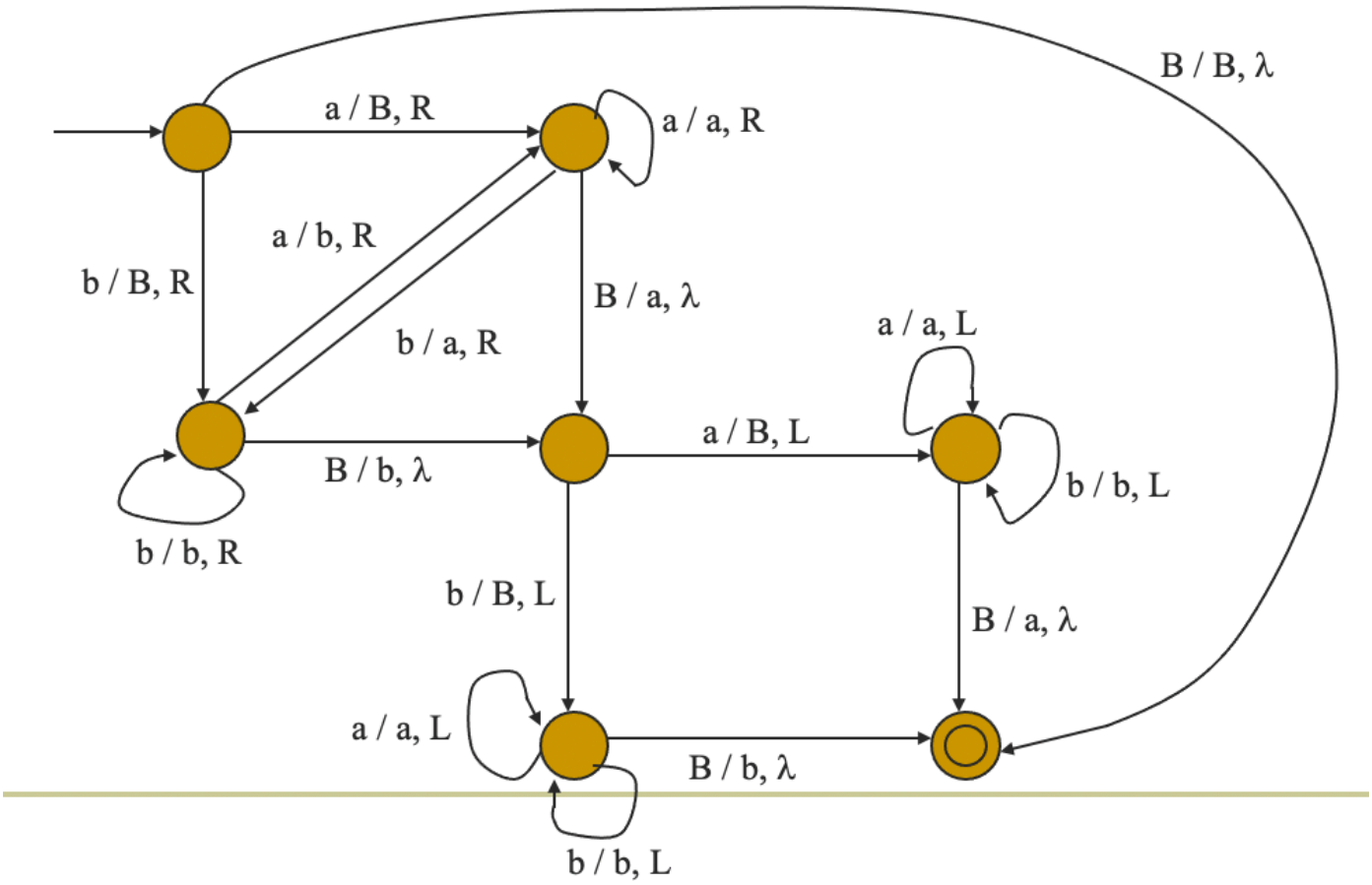
描述2：设计一个程序，让一个输入的字符串集体右移一位，需要在原来的字符串上修改



栗子3

描述1：构造一个图灵机来让一个字符串集体循环右移一位，也就是`abbaBB->aabbBB`，字符串中只会出现a和b

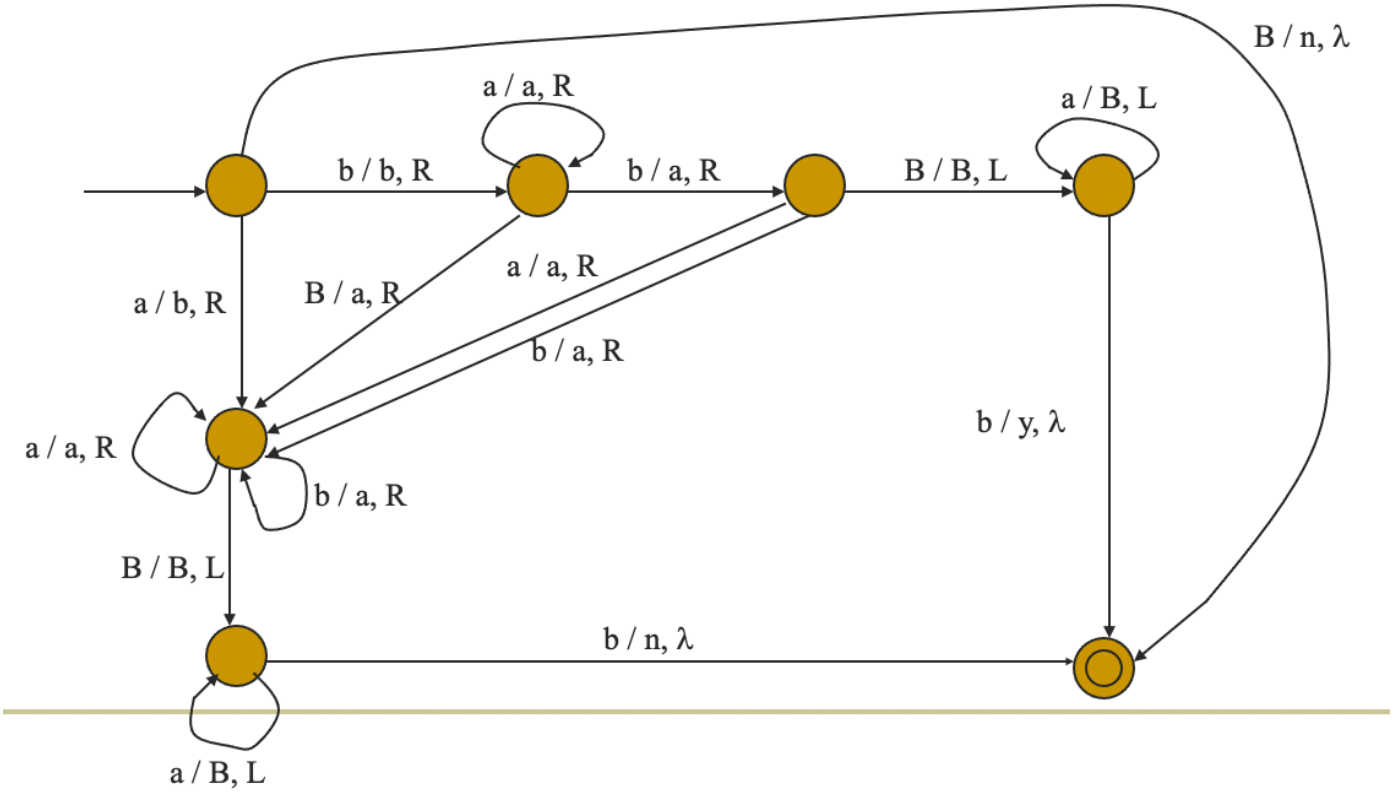
描述2：设计一个程序，让一个输入的字符串集体循环右移一位，需要在原来的字符串上修改



栗子4

描述1：构造一个图灵机来接受这个语言 $L = \{ba^ib|i \geq 0\}$，也就是判定一个字符串是否符合这个形式，符合的话在磁带上放下一个y，不符合放下一个n，其他字符均为Blank

描述2：设计一个程序，来识别输入的字符串是否符合 $L = \{ba^ib|i \geq 0\}$，的形式，符合的话输出y，否则输出n



为什么设计图灵机这么难?

在上面四个栗子中，如果大家使用C、Python这些编程语言可以非常容易解决，但是设计一个图灵机来解决却需要一些小技巧，如果让大家设计一个解决排序问题的图灵机，那恐怕要花上几个小时甚至几天的时间，最后设计出来的图灵机状态可能极其复杂。为什么会这样?

极少的变量

一个比较明显的原因就是你拥有的信息太少了，在写代码的时候你可以随意的声明很多变量，但是在图灵机中，你的迁移函数中只能收到一个字符，甚至你能用的所有存储空间，不过就是一个无限长的磁带而已，你能使用的变量只有一个（尽管它无限长）。

函数设计困难

在编程的时候，我们可以把一个复杂的问题分割成不同的小问题，然后编写不同的函数来处理它，但是在图灵机中，编写子函数是比较复杂的事情，因为我们没有显式的栈，如果你在一个状态 q 遇到两个不同的字符的时候需要前往两个不同的子函数再返回，然后根据返回值做一些操作，这样图灵机的状态可能会变得非常多，因为你可能需要连个状态来区分从不同的子函数返回，也有可能造成迁移函数非常复杂。

我们将图灵机能够接受的语言的范围，也就是设计图灵机能够解决的问题称为图灵机的表达能力。

图灵机的表达能力和现在的计算机比起来，哪个更强？我们通过上面的栗子已经看出来，在应付一些稍微麻烦的算法时，设计一个图灵机已经是一个相当困难的任务，那么图灵机的表达能力比计算机弱吗?

对图灵机表达能力的讨论

一个磁带多个磁道的图灵机

在之前的栗子中，我们构造的图灵机都只有一个磁带，那么从考虑解决变量少的问题出发，我们可以拓展出一个磁带上有两个磁道的图灵机，看看这样图灵机的表达能力是否增强。在单磁带多磁道图灵机中，磁带上所有的磁道都由一个共同的磁头定位，迁移函数被拓展为

$$\delta(q,Z)=(P,Y,D)$$
$$Z=10$$
$$Y=00$$
$$D=L$$

```
TapeA
Track1 ...BBB010101101BBB...
Track2 ...BBBBBB01010BBBB...
HeadA           _
```

假设一个图灵机 T 有一个磁带A和两个磁道，假设当前图灵机的状态为 q，磁带上的符号为 a_1,a_2，修改后的符号为 b_1b_2，状态转换成 p，需要注意，一个磁头是对应一个磁带的，也就是说一个磁头同时标记了两个磁道上的字符。那么我们可以构造一个单磁带单磁道的图灵机，他的状态空间和 T 完全相同，同时因为磁带只有一个，我们可以把的两条磁道按照先后顺序压缩，比如符号为a_1,a_2 的时候，我们可以直接把它看成一个符号 a_1a_2，这样我们就可以用**单磁带单磁道的图灵机模拟单磁带多磁道的图灵机**了。于此同时，单磁带单磁道的图灵机其实就是一个单磁带多磁道的图灵机只使用了一个磁道的结果，所以单磁带多磁道的图灵机也可以接受所有单磁带单磁道图灵机所接受的语言。

给图灵机的磁带增加多个磁道，对图灵机的表达能力没有任何改变，那如果我们增加多个磁带呢?

多个磁带的图灵机

在多个磁带的图灵机中，每个磁带我们都给他分配一个单独的磁头，迁移函数的形式拓展为

$$\delta(q,Z)=(P,Y,D)$$
$$Z=(b,f)$$
$$Y=(g,d)$$
$$D=(L,R)$$

```
TapeA
Track1 ...BBB010101101BBB...
HeadA       _
TapeB
Track1  ...BBB0101011BBBBB...
HeadB           _
```

同样，假设一个图灵机 T 有 K 个磁带，因为磁道的个数不改变表达能力，所以我们假设这 K 个磁带都只有一个磁道，现在我们构造一个一个磁带，拥有 $2K$ 个磁道的图灵机 T'，对于每个T中的磁带，我们都用T'中的一个磁道来表示，再用另一个磁道来表示这个磁带的磁头是在什么位置，对于这样的图灵机T'，我们可以证明T和T'他们的表达能力是等价的，这里留给大家证明(((o(゜▽゜)o)))

给图灵机加上很多磁带、磁道，其实都完全不改变图灵机本身的表达能力，也就是说，一个变量的图灵机和一堆变量的图灵机，他们能做的事情是一样的。图灵机和现代的计算机相比，哪个表达能力更强？答案是**一样的**。也就是说，一个最简单的图灵机，和我们手中复杂的计算机，他们能解决的问题是一样多的！图灵机无法解决的问题，计算机也无法解决，计算机能解决的，图灵机一定可以解决。

> The Church-Turing Thesis
>
> Everything we can compute on a physical computer can be computed on a Turing Machine

本次实验只需要大家能够实现一个执行图灵机的程序

2. 实验要求

本次实验中，同学们需要实现两个方法

```
public boolean execute() {} // Executor.java
public String snapShot() {} //Executor.java
```

execute()方法需要让读取的图灵机在事先给好的磁带上执行。我们在测试的时候，会先构造好一个图灵机对象，然后将对象和磁带作为参数传给Executor类，并且执行execute方法，execute方法相当于调试时的单步执行，这样是为了方便我们测试大家的实现，返回true表示可以继续执行，返回false表示不能再继续执行。snapShot方法需要能够支持快照功能，即按照指定的格式，返回以下信息

- 图灵机的状态
- 执行的步数
- 每一条磁带上的情况
    - 包括当前磁带的索引，索引从0开始
    - 包括每个磁道和磁头，磁头只需要给出一个数即可，最开始的时候，每个磁带上的磁头都初始化为第一个磁道上最左侧的非空格符号处
    - 每个磁带的第一个磁道打印出最左非空格符号道最右非空格符号的内容，其余的磁道打印该范围内磁道上所有的字符
    - 当磁头指向的位置为非空格符号外侧的空格符号时，也需要打印出空格符号，空格符号用磁带中的空格符号表示

具体的格式、假设等会在用例中指出

本次实验中，图灵机和磁带不合法的地方仅包含以下情况

1. 输入的磁带中出现了不属于输入符号集的字符
2. 图灵机中的磁带数和输入磁带的磁带数冲突
3. 终结状态集不属于状态集
4. 空格符号出现在了输入符号集
5. 空格符号没有出现在磁带符号集
6. 输入符号集并不是磁带符号集的子集
7. 迁移函数的新旧状态不属于状态集
8. 迁移函数读取的符号和新写入的符号不属于磁带符号集
9. 图灵机的迁移函数相互冲突，下面解释一下相互冲突的含义

    1. 同样的状态，同样的输入，却转移到了不同的状态，得到了不同的输出，磁头有不同的偏移
10. （思考，不要求报出）存在一个输入让图灵机无法停止
    1. 无法停止意味着图灵机永远不会停机，也永远不会进入到终止态

当输入出现了上述错误时，需要能够报告出来，假设出现了输入符号集并不是磁带符号集的子集的错误，那么需要在标准错误流中输出`Error: 6`，不能遇到一个错误后就直接退出，需要报告出全部的错误，不可以多报，除了7，8，9以外同一个错误不需要报多次。

参考用例

每个用例我们都会输入一个图灵机和一个磁带

```
; This example program checks if the input string is a a_nb_n.
; Input: a string of a's and b's, e.g. 'aaabbb'
; the finite set of states
#Q = {0, 1, 2, 3, 4}

; the finite set of input symbols
#S = {a, b}

; the complete set of tape symbols
#G = {a, b, _}

; the start state
#q0 = 0

; the set of final states
#F = {4}

#B = _

 #N = 1
; the transition functions

; State 0: start state
#D 0 a _ r 1
#D 0 _ _ r 4

; State 1:
#D 1 a a r 1
#D 1 b b r 1
#D 1 _ _ l 2

; State 2:
#D 2 b _ l 3

; State 3:
#D 3 a a l 3
#D 3 b b l 3
```

```
#D 3 _ _ r 0
```

如果给出的磁带是`___b____`，最后磁带会变成`___b____`，并且图灵机处于初始状态

如果给出的磁带是`___aaabb___`，最后磁带会变成`_____________`，并且图灵机处于状态2，因为无路可走而halts

如果给出的磁带是`___cccaaabbb___`，标准错误流中输出`Error: 1`，图灵机处于初始状态

如果给出的磁带是`___aaabbb___`，最后磁带会变成`_________`，并且图灵机处于终止态，如果在最开始调用Executor的snapShot方法，会返回如下信息

```
Step  : 0
Index0: 3 4 5 6 7 8
Tape0 :
Track0: a a a b b b
Head0 : 3
State : 0
```

再调用一次execute，接着调用snapShot会得到

```
Step  : 1
Index0: 4 5 6 7 8
Tape0 :
Track0: a a b b b
Head0 : 4
State : 0
```

同理，如果是两个磁道的话（本例与上一个例子的输入和图灵机无关），假如输入一个磁带，两个磁道，第一个磁道上的信息为`___aaabbb__`，第二个磁道上的信息为`_____abbb_______`，需要打印出如下信息

```
Step  : 0
Index0: 3 4 5 6 7 8
Tape0 :
Track0: a a a b b b
Track1: _ _ a b b b
Head0 : 3
State : 0
```

如果是多个磁带的话，需要打印出如下信息

```
Step  : 0
Index0: 3 4 5 6 7 8
Tape0 :
Track0: a a a b b b
Track1: _ _ a b b b
Head0 : 3
Index0: 1 2 3 4
Tape0 :
Track0: t r u e
Track1: _ _ _ _
Head0 : 2
State : 0
```

其余的情况以此类推。

需要注意的是冒号和前面的label之间的关系，**冒号总是紧接着最长的label后面**，冒号后面如果没有内容，那么任何字符都不要添加，冒号后如果有内容，一定会有一个空格在冒号和内容之间，如

```
Step   : 0
Index0 : 3 4 5 6 7 8
Tape0  :
Track0 : a a a b b b
Track1 : _ _ a b b b
Head0  : 7
Index0 : 1 2 3 4
Tape0  :
Track0 : t r u e
Track1 : _ _ _ _
...
Track99: _ _ _ _
Head0  : 2
State  : 0
```

我们假设一条磁带上每一个磁道的第一个元素的Index都是0，注意，这里需要考虑磁带和磁道变长的情况，因为图灵机磁带和磁道的长度都是无限的，在输入的两端都是无限的空格，但是我们显然无法给出一个无限长度的输入，因此，大家需要自己做好磁带磁道的变长工作，当磁带磁道变长的时候，Index和磁头的位置都需要重新计算。**但是大家也不能随意修改磁带磁道的长度**，下面给出一个增长的规则（适用于多磁带多磁道）

- 当一个磁带的磁头走到了这个磁带的某个磁道的最尾部时，需要给**所有**磁道增加一个空格符
- 当一个磁带的磁头走到了这个磁带的最左侧的时候（也就是向左移动一次后，head从0变成了-1），需要给所有的磁道的开头都增加一个空格符
- 其他的任何时候，都不需要特地去给磁道增加空格符，否则测试会失效

更多测试用例见代码

测试代码中给出一个实现了3+6的图灵机的例子，对应的磁带是`_ _ _ _ 3 + 6 _ _ _`（空格为了看清楚，实际测试里没有空格），分割线测试里也是没有的，这里只是为了美观

```
Step  : 0
```

```
Tape0 :
Index0: 4 5 6
Track0: 3 + 6
Head0 : 4
State : init
--------
Step  : 1
Tape0 :
Index0: 4 5 6
Track0: 2 + 6
Head0 : 5
State : add
--------
Step  : 2
Tape0 :
Index0: 4 5 6
Track0: 2 + 6
Head0 : 6
State : add
--------
Step  : 3
Tape0 :
Index0: 4 5 6
Track0: 2 + 7
Head0 : 5
State : sub
--------
Step  : 4
Tape0 :
Index0: 4 5 6
Track0: 2 + 7
Head0 : 4
State : sub
--------
Step  : 5
Tape0 :
Index0: 4 5 6
Track0: 1 + 7
Head0 : 5
State : add
--------
Step  : 6
Tape0 :
Index0: 4 5 6
Track0: 1 + 7
Head0 : 6
State : add
--------
Step  : 7
```

```
Tape0 :
Index0: 4 5 6
Track0: 1 + 8
Head0 : 5
State : sub
--------
Step  : 8
Tape0 :
Index0: 4 5 6
Track0: 1 + 8
Head0 : 4
State : sub
--------
Step  : 9
Tape0 :
Index0: 4 5 6
Track0: 0 + 8
Head0 : 5
State : add
--------
Step  : 10
Tape0 :
Index0: 4 5 6
Track0: 0 + 8
Head0 : 6
State : add
--------
Step  : 11
Tape0 :
Index0: 4 5 6
Track0: 0 + 9
Head0 : 5
State : sub
--------
Step  : 12
Tape0 :
Index0: 4 5 6
Track0: 0 + 9
Head0 : 4
State : sub
--------
Step  : 13
Tape0 :
Index0: 4 5 6
Track0: _ + 9
Head0 : 4
State : final_state
```

代码指导

```
.
├── main
│   ├── java
│   │   └── edu
│   │       └── nju
│   │           ├── Executor.java
│   │           ├── State.java
│   │           ├── Tape.java
│   │           ├── TransitionFunction.java
│   │           ├── TuringMachine.java
│   │           └── Utils.java
│   └── resources
└── test
    └── java
        ├── ExecutorTest.java
        └── IOUtils.java
```

> 在开始实验之前，先将大家实验一的代码复制到TuringMachine.java中

为了降低难度，我们挖掉了实现好的代码，留下了一些方法，大家可以通过补全这些方法来完成本次作业，也可以把这些方法全都删掉，毕竟我们的测试只和实验要求里要求的必须实现的两个方法有关。这些方法的作用我们都标记在方法名上面了，我们会用一张调用图来描述这些方法之间的关系。当然，我们也十分鼓励大家删掉这些方法，自己思考应该如何设计，现在工业界有非常多的开发框架可以给大家使用，这些开发框架屏蔽了大量的细节，提供了大量非常优秀的接口和最佳实践，所以在系统设计这方面，我认为很多同学都是有欠缺的。

如果大家想要自己构思一个好的设计的话，建议大家遵守面向对象设计原则，把高内聚低耦合放在心上，做好数据和职责的一致性。当然，因为我们对图灵机的语法做了一些微小的改动，所以有一些实验一中写好的方法可能需要改动，不改动只在原方法里继续添加内容也是可以的，这取决于大家实验一的代码。

这里建议大家把磁头的位置和磁带的内容都放到Tape类中进行处理，Executor中只维护一个TM对象、当前状态和磁带信息，在每次执行后都调用一些更新的方法，这样模块性会好一些。一定要清楚哪些内容应该内聚，哪些内容应该解耦。

大部分需要检查的不合法的地方都是和集合与集合之间的关系有关的，幸好Java已经提供好了相关的库可以让我们调用，求两个Set的交集可以用`retainAll`这个方法，并集、差集大家可以自己去研究一下，当然自己造轮子也是一种乐趣。

错误处理相关的信息在加载图灵机的时候就可以检测出一部分了，如果大家实验一写的代码可拓展性比较好的话，很容易继续往里面添加内容。第一次错误处理只涉及到一些简单的语法错误，本次错误处理会涉及一些语义上的错误，得益于我们简单的图灵机语法，这件事情非常容易。大家可以想象一下，如果要分析Java这门语言的语法错误和语义错误该如何分析?

> 编译原理

实验二的代码在实验三中还会继续用到，希望大家保持一个良好的代码风格。实验二虽然步骤繁琐了些，但是实现的功能会非常有助于大家实验三调试代码。

预祝大家顺利完成实验二