**深 圳 大 学 实 验 报 告**

**课程名称： 编译原理**

**实验项目名称： 词法分析程序设计**

**学院： 计算机与软件学院**

**专业： 软件工程**

**指导教师： 尹剑飞**

**报告人： 钟靖扬 学号：2020111051 班级： 01**

**实验时间： 2023年3月1号-2023年3月20号**

**实验报告提交时间： 2023.3.25**

**教务处制**

|  |
| --- |
| **实验目的与要求：**  **目的：**针对状态矩阵、NFA、DFA、正规式、确定化算法、最小化算法、Thompson算法等词法分析应用问题，设计并实现相应的解决方案，通过设计FA相关类族，以及实现前述几个重要的算法，既加深对抽象的词法、自动机、正规式等形式语言理论基础概念的理解与掌握，也加强对面向对象程序编写能力和计算思维的培养。  **要求:**  **第一部分： 无符号数的识别及DFA的应用**  这部分又分为2个小实验   1. 输入：一个文本文件（源代码文件）   输出：将源代码中的无符号数识别出来并输出到另一个文件中  示例：如果输入是“123\*abc+def/99.2+9.9E+c”，那么输出是：（数字, 123），（其它，\*abc+def/），（数字，99.2），（其它，+），（异常，9.9E+c）  说明：其它是非数字打头的字符串；异常是数字打头，但最后却是不符合定义的无符号数。   1. 假设：用字符“ABCDEFGHIJK”（大写）分别表示数字0..9和.、E、+、-，那么，字符串“BCLD”表示数字“12E3”=12000；   输入：一个文本文件  输出：将隐藏在文本文件中的有效无符号数识别出来。  示例：如果输入是“BCD\*abc+def/JJKC+JKJL+c”，那么输出是：（数字, 123）、（数字，99.2），无效（异常）的无符号数不输出  **第二部分：DFA/NFA的读写及确定化、最小化算法 （选做）**  这部分又分为3个小实验   1. 输入：一个文本文件（格式自定义）   输出：读入文件中的DFA/NFA，创建对应的DFA/NFA对象，再写回另一文件中。  示例：如果输入是“f(S,a)=A, f(A,b)=B, {B}” //假设默认开始状态是S，  那么输出是：K={S，A，B}；Σ={a,b}；f(S,a)=A, f(A,b)=B；S；Z={B}   1. 输入：上一实验输出的NFA文件   输出：将读入的NFA进行确定化后，输出结果，需要考虑有/无ε的情况。   1. 输入：上一实验输出的DFA文件   输出：将读入的DFA进行最小化后，输出结果。  **第三部分：正规式及其应用**  这部分又分为3个小实验   1. Thompson方法的实现   输入：一个正规式  处理：创建与正规式对应的NFA对象，再将NFA对象写回文件，格式与前述实验相同，使得能够读回；  示例：如果输入是“a\*b”，那么输出是：K={A , B , C , D , E , F}；Σ={a, b}；f(A,ε)=B, f(B,a)=C, f(C, ε)=B, f(C, ε)=D, f(A, ε)=D, f(D, ε)=E, f(E, b)=F；A；Z={F}  说明：优先级处理可用栈或递归，思路见算符优先分析法。概要来说，判断当前运算符与下一个运算符间的优先级，当前≥下一个则计算（产生NFA），否则入栈或递归。   1. 从正规式到DFA   将上述实验与前述实验连接起来，使得读入一个正规式文件，能输出一个DFA。正规式文件格式自定，通常每行一个正规式。  示例：如果正规式文件包括2行，每行分别是“a”和“b”，那么，输出的DFA实际上是“a | b”对应的DFA。   1. 字符串的识别   输入：一个正规式文件和一个字符串文件  输出：判断字符串文件中的每个字符串，能否被正规式对应的DFA所识别  其次，再给每个正规式增加一个类别，识别到给定字符串符合某个特定正规式时，输出该类别。  示例：如果输入  a\* 类A  b 类B  那么对字符串aaa输出：aaa，类A |
| **方法、步骤：**  要完成本实验，依据实验要求进行分解，需要从数据结构设计和面向对象设计角度，思考并回答下述问题：   1. 如何识别无符号数？   如何判断一个输入的字符串中，包括无符号数字？如何判断无符号数的开始？中间字符及结束？  根据状态矩阵画出相应的DFA和状态转换表来获取数据。      得到的表如上所示，-2表示进入的是异常的状态，也就是开头数字合法，结尾却不能形成合法的数字，-1则是进入其他的状态，也就是无符号数之外的内容。  而我们需要的结果分别有三种，数字，其他和异常，因此根据我们的需要画出对应的状态表   1. 如何实现简单的字符映射加解密算法？   例如ABCD在ASCII表中是按顺序排列的，因此我们可以比较简单的直接用A～J来表示0～9用加法就可以完成一个简单的映射。  剩下的例如用K代替‘.’等等，可以用Map来实现key和value的映射，通过HashMap可以通过Key很快搜索到。   1. 如何定义DFA/NFA对象，特别的，对自动机里面的映射函数f如何定义、存储？（注意这里的“对象”，指用面向对象的程序设计思维方式（内部数据结构、消息）来定义一个DFA类、一个NFA类。）   请在这里，补充完善你的设计思路，完成后，请删去这句话。  这里用一个Automate的名称来定义自动机的类，里面首先要存的就是它的状态表，也就是上面第一问给出来的表，并且用二维int类型矩阵存起来，    如上图所示，通过数组索引的方式就很容易的可以是实现状态的转换。  而映射的关系，对于不同的输入，我们自然就有不同的映射，因此用Map可以很好地实现函数的映射.    而在判断的时候，我们会根据不同的输入，返回不同的字符串，例如输入符号什么的，就返回sign，输入数字，就返回number，那么它们就会去对应自动机的状态。   1. 如何实现确定化算法？（选做）   确定化算法需要把一个旧的状态集合作为一个新的状态，这种情况如何处理？又如何求某个状态的ε闭包？  首先要将空转移去掉，然后再去检查是否还是有不确定的状态转移。  如果有，则需要定义一个二维表格，列头表示的是各个不同的输入，例如a和b  行头表示的是不同的状态，当在已有的状态当中，发现某个输入生成了新的状态，则需要添加一行关于这个新的状态的数据  如这个NFA自动机    B状态下输入b会到达两个不同的状态，所以需要进行确定化  生成的表格如下：    将BN视为一个新的状态，即可完成确定化算法，将NFA转为DFA   1. 如何实现最小化算法？（选做）   最小化算法需要对原状态集进行等价类划分，这个划分要如何处理？  请在这里，补充完善你的设计思路，完成后，请删去这句话。   1. 汤普森方法需要处理算符的优先级，该如何处理？如何进行递归调用？   算符是给括号最高优先级，然后到或操作，到连接操作再到闭包操作，使用栈来实现运算，不需要用递归操作，当前的算符的优先级比栈顶的算符优先级高的时候，就将栈顶元素弹出并且进行运算，否则就入栈，不过这里的右括号需要稍微做一下特殊处理，因为右括号的优先级只比左括号优先级低   1. 如何根据给定DFA，判断并输出给定输入串的类别？   对于给定的DFA，有着对应的一个状态转换矩阵，可以看成一个二维数组，列就是输入的转移信息，行就是此刻处于的不同的状态。因此我们可以逐个读入输入串的字符，从起始状态开始，根据当前读入字符寻找下一个状态。 |
| **实验过程及内容：**  实验过程及内容，处理代码设计说明、代码及其注释外，特别关注编程过程。  **Part01**  第一问  我们定义identifyNumber类来实现判断  Main函数将类实例化后，直接输入路径进行判断输出就可以了。    先大致介绍一下该类下面的成员变量的作用  **成员变量**：  如下图所示，一共有七个私有的成员变量    Line就是用于存放每一行的数据，writer是文件写入流使用的类。  Numbers和Others是用于存放要输出的数字或者其他的内容，异常的内容实际上也是用Numbers来代替，因为异常最开始识别到的首先是数字。  IsException作为异常标志位，Automate是实现判断自动机状态的功能  State用来存放当前自动机识别的状态  **成员方法**：  这里只有两个成员方法。一个是用来初始化文件读写，获取文件数据的函数identify（）  这个方法很简单，功能只是将文件里的内容提取出来，并且读取每一行，然后去识别每一行的内容    最核心的功能在parseToken里面。  **ParseToken（）方法**：  在parseToken当中，用for循环去识别每行的每个字符。  根据**当前字符**以及已经构建好的**自动机**来判断下一个状态是什么。    整体的逻辑如上流程图，用了三个if-else来进行判断，主要区分的只有三种状态，也就是**异常**，**数字**和**其他**。  这里着重讲一下中间这个通过创建好的自动机获取状态迁移的方法。  代码接口调用如下    看看这个方法具体是怎么实现的。    看到只有一行的代码，只是单纯的去索引一个二维数组里面的值而已。  那么为什么用一个简单的二维数组就可以实现自动机呢。  我们要先画出无符号数识别的自动机出来，就很好理解了。    这个自动机实际上也并不复杂，绿色的是可以做为终结状态，而如果在红色的状态里面停止了识别，则可以识别为异常或者其他。  通过状态表可以更清晰地分辨异常，正常和其他的状态    如果是正数，则为正常的状态，而如果是-2，则是异常的状态，如果是-1，则是“其他”的状态  这样就完成了所有的识别流程了，最后我们看看识别出来的效果。    左边是我们的测试样例，右边是则是识别出来的最后的效果。  **第二部分**：  该题目的第二部分也是比较简单的，只是将输入的一些特殊字符和数字转换成了英文，类似于一种加密，而我们只需要去做相应的解密，将字母重新转换成第一部分的正常的数字和特殊字符即可。  来看看main函数是怎么实现的。    下面那个部分其实就是调用第一部分写的类文件的代码，因此其实区别只是在上面多了一个解密的类方法而已。    为了方便对应特殊字符解密，这里用了HashMap来做一个字符的映射，而0-9直接映射到A-J即可，不需要哈希表辅助映射了。    转换代码如上图所示，如果是J前面的字符，就直接映射为数字，如果是J后面的，就用hashMap的get方法来获取对应的字符。    最终效果如上图，Decode文件是对Test02文件解密之后的结果，然后直接用第一部分的代码来识别Decode文件即可。  **Part03**  这题比较复杂，需要分成很多步骤去完成  由于代码比较多，文件数量也较多，因此先介绍一下不同的文件的功能：    其他文件是一些结构代码，方便进行数据的处理和面向对象，  **第一部分：根据正规式生成NFA**  **程序代码入口**：  生成对应的NFA需要运行createNFA的Java类，在该类会调用REFIle类来对读入的文件进行处理。  这里给出一些正规式的**输入示例**    输出代码以及格式如下：  image-20230411172859719  主要的运算逻辑代码都在loadFromRegularExp（）里面。 生成后缀形式的正则式 说到算符优先级， 那么就不得不提到逆波兰式，机器是如何去计算那些四则运算的呢，较为常用的就是将中缀表达式转换成**后缀表达式**。  为什么要转换成后缀表达式呢，这样是为了让机器更容易去识别运算的优先级，像中缀表达式，由于括号，加法乘法的优先级不同，很容易导致运算顺序的不一致。  如：5 + 4 \* (3 - 2)  放在后面的式子往往会先被运算，为了更好地让电脑知道怎么去计算，我们就手动将其转化成后缀表达式  结果为：5432-\*+  这样就可以很轻松地进行运算了，因为运算的顺序**已经排列好了**。  这里就不展开阐述逆波兰式的思路了，接下来按照它的思路去给正规式转换成后缀形式的正规式。  这里举个正规式例子  正规式：a(b|aa)\*b  增加连接符：a-(b|a-a)\*-b  转换后：abaa-|\*b— 1.1增加连接符 看到上面转换的过程中，中间比四则运算多了一步“增加连接符”，是因为正规式当中的连接符号是不存在的，例如aa，实际上是两个字母对应的NFA的**连接**。  所以为了方便编写代码，我们要先将正规式处理一下  判断是否需要加连接符也很简单，源码如下：  image-20230409143141169  也是逐个遍历当前的正则式的字符，然后如果**当前字符nowWord**是’\*‘或’）‘或Unicode字符（数字字母和一些特殊字母），  则判断**下一个字符nextWord**是否是’（‘或者UniCode字符，是则说明需要连接。) 1.2 设置算符优先级 接着是设置算符的优先级方便运算，这里经过判断后，直接得出优先级顺序应当如下：  image-20230409231056648  使用map，这样方便 if 语句用containsKey（）判断是否是操作符  **代码大概思路**如下：  image-20230409233138800  除了正常根据操作符优先级判断由于括号运算比较特殊，所以需要多加一个判断，  否则出现右括号的时候，右括号前面的操作符都会被优先输出（因为右括号优先级很高，保证右括号后面的算符**不会被先弹出**，只有等到左括号的时候才能弹出，但是这样右括号前面的就会因为优先级较低而**被先弹出**，这样就矛盾了） 2. 使用栈实现运算 首先来讲如何利用**栈结构**来实现运算  没错这一个部分也需要用到栈，足以见得数据结构的重要性  流程图如下：  image-20230409142614061  对正则式加上连接符号处理后，就可以开始识别了。  主要分为两种情况：   * 当前字符是数字或者字母，Java当中可以使用Character类的isLetterOrDigit（）方法来判断 * 如果不是上述情况，视为操作符，然后进行switch判断   在switch方法当中对不同的操作符进行不同的运算。  image-20230410233452996  如图，思路很清晰也很简单，具体的操作符运算细节这里暂不展开 3. 构造NFA类3.1 如何存放和表示状态转换表    * RegularExpression：存储输入的正则式 * statrStae、endState：表示起始和终止状态，分别对应的是输出的最后一行的两个输出，Z={终止状态} * stateCode：主要用于生成新的状态   注意：这里的state都是用Integer来表示，这样方便存储，  在需要输出的时候，再调用函数按自定的规则映射成相应的字母。   * transferMat：存放所有的状态迁移，即输出的 f（A，a）={B} * stateList：对应输出当中的K里面的内容 * msgList：转移条件，对应输出的 Σ={a, b} * Pair：用于存放单个状态转移内容，由起始状态和转移条件组成   image-20230411001312815  有了以上的内容之后，就可以开始进行数据存储和运算了。 3.2 如何进行闭包、连接等运算 刚开始去思考这些算法的时候，会觉得很抽象，思路都很难理清，代码难以下手，因此要先理清思路。  **画图**是个很好的方法，很形象。  这里一共有四个操作，除了前面提到的连接，闭包，或运算外，还有一个用于创建新的状态。  image-20230411002912864  给出具体的实现方法，并稍微进行讲解。          主要的操作和逻辑都已经介绍了，剩下一些涉及到的方法，根据方法名大致理解了操作思路即可。  **第一部分：根据正规式生成NFA**  先说明一下文件结构    **程序入口**：  运行createDFA的类，会去到REFIle类当中处理文件，并且在该类当中生成NFA，再继续对NFA进行处理，从而转换成DFA  因此要看懂代码的话，建议先去看之前正则式转换成NFA的文章，阅读本篇文章就会更加容易  已获得的其中一个正规式和NFA如下    该部分的代码就是基于这些数据进行操作。  看着很多状态很复杂是吧，我们将对应的状态机画出来，看着就简单很多了  image-20230411093305066  **输出样例**：  对应的DFA输出如下：    image-20230411172107224  可以看到初始状态仍然是A，但是结束状态变成了O，而且N已经不见了，  这里是因为O是一个新的状态，同时可以表示到达**B和F的状态**，后面会讲到如何生成 实现和输出代码主要用到的函数： image-20230411173312164  在讲解**正规式生成NFA**那篇文章当中已经介绍过loadFromRegularExp（）的代码  因此这里只讲removeEpsilon（）和NFAToDFA（）的思路以及代码 1. NFA去除空字符（仍可能为NFA） 为了让更好地转换成DFA，我们需要将NFA当中的 ε 转移去掉，让状态图清晰简洁一点。  注意，有的NFA去除空字符后，可能就成为DFA了，但也有的还是NFA。  例如上面给出的例子，会发现B状态输入b的时候，会回到B本身或者到达F终态。 1.1 算法思路： 思路是很简单的  两个由空字符连接起来的状态可以被简化成一个状态，如图  image-20230411094347218  对应我们的代码，也就是将 f（A，ε) = B 当中的B替换成A  伪代码如下：    该实现方法我们仍然放在类NFA当中，为**removeEpsilon（）**，因为去除epsilon后可能仍然是NFA，所以就放在该类中 1.2 实现效果： 成功将空字符去除后，状态机如下：  image-20230411153738592  看到图片当中仍有不确定的状态转移，说明还是NFA 1.3 注意事项：  * **起始**和**终止**状态可能发生变化：   在替代掉该带有空转移的目标状态之前，  image-20230411161409879   * 最后对替换后的目标状态列表进行**去重**   多个状态可能被替换成同一个状态 2.确定化算法2.1 根据NFA写出状态转移表 将其去重之后就可以进行确定化算法的实现，  主要的思路就是根据当前的NFA创建一个**状态表**：  还是以之前的那个正规式作为例子：a（b|aa）\*b  image-20230411153738592  当前状态机如上图，得到如下状态表，生成思路并不难：    这里看到B、F是一个叠加状态，我们可以用一个新的符号来代表这个状态。由此可以生成一个新的状态机  image-20230411172107224  如上图状态机，这样每个状态的每个输出都有唯一的对应的目标状态，成功地从NFA转换成了DFA。 2.2 DFA类的实现以及代码创建状态表（最难的部分） 为了更好地面向对象，提高代码复用性以及封装性，DFA会继承于NFA类，并且新增一些属于DFA的成员函数和成员数据。  同时这里的代码量比较大，变量也很多，有很多要注意的操作细节，个人认为由于能力问题写的太过复杂了，导致越写越难，如果有更好更简单地思路是非常正常的。  以下为DFA的**成员属性**：     * **stateFormat**：这里就是存放的二维表格，用Map是为了方便映射状态，key存放的实际就是状态表的行头。 * newStateMap：用于存放新的状态，例如{ B、F }生成的新状态存放在key当中，B、F一起存放在该map的Value当中. * newerStateMap：在生成新的状态的数据的时候，可能会进一步又产生一个新的状态，因此用多一个Map来存放，方便区分，这个概念可能没那么好懂，后面讲解具体实现的时候会再次展开来讲。 * endState：这里覆盖了NFA当中对应的endState，是因为DFA的结束状态往往不止一个。   生成**状态表流程图**如下：  image-20230411195905471  流程来说是比较简单的，但是具体的代码实现起来有较多需要注意的点。 2.2.1 generateStateFormat（）：增加现有状态迁移信息到表当中。 具体代码如下：  image-20230411200933063  **关注细节**：   * for循环的是transferMat，是NFA当中已经构建好的状态迁移信息 * saveAllStateMap的作用是用于存放所有的状态以及映射的旧状态 * 当size==2的时候，说明当前状态迁移产生的不止一个状态，因此要用   新的状态来表示这个多状态。   * 当需要增添新的行的时候，就进入到generateNewStateFormat（）  2.3 generateNewStateFormat（）生成新状态 代码流程如下：  由于代码较长，分几步来讲解该函数   1. 遍历所有的新状态，然后新增行   image-20230412002125427   1. 填写行数据内容   image-20230412002713154   1. 判断是否需要递归   最后判断newerStateMap里面的数据是否为空，否说明这次新增表格数据的时候又产生了新的目标状态组合，因此需要递归，再次增加表的数据。  image-20230412002915491  该函数与generateStateFormat还是会有比较大的区别，要注意的细节也更加多。  **注意事项**：   * 在该函数当中，产生新的状态时要存放到newerStateMap当中，而不是newStateMap，注意区分 * 该函数是可以进行递归调用的，因为可能不断产生新的状态。  最终结果示例如下： **输入：**  a(b|aa)\*b  a\*b  0(1|00)  a\*b(b|(ab)\*c)ca  **输出：**  the 1 DFA:  K= {A, B, F, N, O}; Σ={a, b};  f(O, b)= {O}, f(A, a)= {B}, f(B, a)= {F}, f(B, b)= {O}, f(F, a)= {B}, f(O, a)= {F},  ;  A; Z={N, O, }  ----------------  the 2 DFA:  K= {C, F}; Σ={a, b};  f(C, a)= {C}, f(C, b)= {F}, ;  C; Z={F, }  ----------------  the 3 DFA:  K= {A, B, F, H}; Σ={0, 1};  f(A, 0)= {B}, f(B, 0)= {F}, f(B, 1)= {H}, f(F, 0)= {H}, ;  A; Z={H, }  ----------------  the 4 DFA:  K= {C, F, H, J, T, V}; Σ={a, b, c};  f(C, a)= {C}, f(C, b)= {F}, f(T, a)= {V}, f(F, a)= {J}, f(F, b)= {H}, f(F, c)= {H},  f(H, c)= {T}, f(J, b)= {F}, ;  C; Z={V, }  ----------------  **Part03 识别字符串**  在前面的实验当中，我们已经成功根据正规式生成了对应的DFA了，因此现在可以直接读入字符串进行识别。  在这里就不去讲述如何读写文件了，直接讲解需要的功能：根据DFA识别字符串  用到的思路是根据生成的状态转移表stateFormat，就可以很容易的完成字符串识别了。    又是这个例子，每次循环当中读入一个**字符**，根据当前字符可以获取下一个要到达的状态。  例如从A开始，读入aaab，第一个字符是a，则下一个状态为B  第二个字符还是a，则下一个状态是F，以此类推......  逻辑代码如下：  image-20230412010059943 最终结果示例如下： |
| **实验结论：**   1. 测试用例的设计与说明   **Part01**  对每部分的实验，你分别设计了什么测试数据（测试用例）进行测试，你设计测试数据的出发点是什么，有什么目的？    这里用了两个测试用例  对于该题目，个人认为主要的难点在于区分其他和异常的类型，根据我们的状态转移表，  是有几种情况会识别成异常，因此设计的测试用例主要是用于识别出这些异常。对应的输出如下，可以看到代码成功地识别了所有的数字，其他和异常。  **Part03**  对于正规式生成NFA的用例，这里列出了八个，后面的几个用例比较复杂，看到生成的NFA的迁移状态也非常的多，最后一个用例生成了三十多个新的状态。  而在NFA生成DFA的过程当中，看到有的用例在去除空字符之后，将成为了DFA，每个输入只有唯一一个确定的状态，而有的测试数据发现仍然是NFA，因此需要通过确定化算法生成对应的DFA    例如正规式：a（b｜aa）\*a，就是一个典型的NFA，对于状态B输入b字符，会有两个不同的状态的走向。  **Part03**  最后一个识别字符串的用例，设计了比较多多用例，可以用于校对生成的DFA是否正确，我们可以根据给出的正规式手动构造一个字符串，从而可以反向达到一个校对生成的DFA是否正确的效果。     1. 测试结果的说明与分析   对测试（实验）结果进行分析说明  **Part01**  输出结果如下    有了状态迁移表后，识别流程相对简单了很多，看到考研正确的识别无符号数  异常的情况一般都是在E后面出现的，因为E后面的数字或者符号往往有着相对严格的要求。  **Part02**  下图是a（b | aa）\*b的正规式生成的DFA和NFA结果    看到两个的差距还是有很明显的不同的，DFA的数据简化了非常多，并且去掉了空字符，  他们的起始状态都是A，而终止状态当中，DFA则与NFA有所不同；  在有的正规式当中，可能起始状态和终止状态都会有所差异。  例如最后一个正规式对于的DFA和NFA    **Part03**  最后是字符串的识别结果    可以看到有的字符串是有二义性的，就是可能会被不同的正规式识别，  通过字符串的识别，我们也可以反过来验证自动机的准确性。 |
| **心得体会：**  除了实验的心得体会外，  还要额外增加说明你对文法、直接推导、推导、句型、句子、语言这些概念的理解。  **心得体会**：  首先来说说对第一题的体会，才刚做完第一个实验不久，因此肯定是对第一个实验的对C语言文法分词是记忆犹新的，因此当碰到第一题要识别无符号数的时候，第一感觉就是觉得非常的相似，感觉类型似乎是一样的，于是直接套用了大概的模板，但是后面意识到本节课的内容是学到了DFA自动机，题目也是写明了，需要用到自动机的内容，因此才意识到解题的思路其实是和第一次实验的大有不同。  尽管思路上是类似的，但是这种无符号数实际上需要考虑的情况有很多，如果像之前那样使用if语句来判断，就很容易会出现非常复杂的情况，代码也难以看懂，可能会有很多if-else的语句嵌套，而在参考了DFA的思路之后，发现使用状态表来判断下一个输入处于的状态是非常的方便，只需要很少的判断逻辑，一个简单的for循环加上一个二维矩阵的索引，就可以判断出这个单词是否是一个无符号数了，而如果是判断整句的话，稍微修改一下逻辑，也是可以很快地识别出句子中所有的无符号数，比较麻烦的只不过是将一些异常情况提取出来。而第一题的第二小问本质上只是需要我们做一下字符的转换，也就是做一个简单的解密算法而已，用Map就可以很容易地实现映射。  接下来第三题才是真正比较难的题目，这题花的时间个人感觉比以往加起来的时间都要多，首先是代码的设计思路比较难懂，一开始没有想到用逆波兰式去进行转换，因此觉得要生成NFA非常的困难，在听老师讲解后，发现使用算符优先级去转换成逆波兰式后，发现思路上就容易很多了，于是开始了构造NFA的类。  到了第二部分需要转换成DFA也花费了很多时间，不仅要把空转移去掉，还要生成二维表格来表示状态转移表，该过程的代码量非常地多，即使是采用面向对象的思想，也有很多的细节需要注意，因此不仅在前期思路构想和编写代码需要花很多时间，后期还经常出现得不到想要的结果的情况，例如DFA的终止状态会有多个，又或者在去除空字符的过程中，终止状态和起始状态都有可能改变，因此往往需要进行调试，检查错误，好在最后成功实现了所有的功能，不由得感慨这次实验的难度之高，以及代码量之多  **概念理解**：   1. 文法（Grammar）：文法是一种系统化的规则集合，用于描述和解释语言的结构和功能。文法包括语音（音素、音节和音位）、形态（词汇、词类、派生和屈折）和句法（短语和句子的结构）等方面。文法规则可以用于生成语言的合法句子，也可以用于解释语言的句子结构。 2. 直接推导（Immediate Constituent Analysis）：直接推导是一种语法分析方法，通过将句子分解为更小的成分（短语、词和词素），以确定它们之间的语法关系和句子结构。这种方法可以追溯到20世纪早期，是句法学的早期方法之一，但现在已被更高级的分析方法所取代。 3. 推导（Parsing）：推导是一种基于文法规则的语言分析方法，它用于确定句子的结构和含义。通过分析词汇和语法规则，推导可以将一个句子分解为短语、词和词素，并确定它们之间的关系和作用。推导可以用于理解自然语言的句子，也可以用于计算机语言的编译。 4. 句型（Sentence Pattern）：句型是一种抽象的语言结构，它用于描述句子的基本结构和语法规则。句型包括主语、谓语、宾语、定语、状语等成分，并指出它们在句子中的位置和语法作用。不同的语言具有不同的句型，但是在同一种语言中，句型可以根据不同的语境和语用目的而有所变化。 5. 句子（Sentence）：句子是一组有意义的单元，它由一个或多个单词组成，表达出完整的思想或概念。句子具有语法结构和含义，是语言交流和表达的基本单位。句子可以根据语气、语调、语法结构等方面进行分类和描述，用于不同的语言学研究和应用领域。 6. 语言（Language）：语言是一种人类交流和表达思想的系统化符号体系，它具有一定的语法规则、词汇和语用功能。语言可以用口头、书面、手势等多种方式进行表达，可以用于不同的社会和文化环境中。 |
| 指导教师批阅意见：  成绩评定：  指导教师签字：尹剑飞  年 月 日 |
| 备注： |