实验文档

软件 13 杨楠

编程语言与环境

操作系统：Windows 10
平台：VS Code
语言：C++11
编译：cmake MinGW-w64 g++

第一次实验

第一次实验，是根据已给定的 NFA，执行输入串，如果输入串被该 NFA 接受，还需要返回一条接受路径。

实现思路

对路径进行回溯，DFS，这里使用了栈来实现。参考了“第一次实验讲解”课件第 47 页里的内容，也相应地实现了 match()函数和 backtrace()函数。

- **回溯法：基于DFS，所以使用栈**
- **伪代码**

```
stack=[(0, input, 0)]//(状态，剩余输入串，当前是第几步)
while !stack.empty():
  q, str, step = stack.pop()
  path[step] = (q, str)//path中记录的是第x步的状态和剩余串
  if q是终态 && str是空串:
    return backtrace(path)//找到了，返回路径
  for rule in rules[q].reverse():
    if rule.by == ε:
      stack.push((rule.dest, input, step+1))
    else if rule.match(input[0])://即规则与当前输入字符匹配
      //去掉input的首字母
      stack.push((rule.dest, input[1:], step+1))
```

上述课件中的伪代码有些瑕疵，比如 stack 要 push 的不是 input 或 input[1:]，而是 str 或 str[1:]，等等。
在实验一的代码中，对于 stack 中的元素和 path 中的元素，参考了上述课件，用“剩余输入串”来记录。而到了实验三，我修改为了记录当前在字符串的位置 index。

重难点问题

对于ε转移的死循环的处理（即，可能存在完全由ε转移构成的环）。我在程序中是使用一个 set 集合来存储某个状态曾经经历过的情况。由于 set 里元素的互异性，如果发现在该状态下，当前剩余输入串的情况已经经历过了，那么就不将其入栈。我在实验一中是记录当前状态经历过的所有“剩余输入串”，到了实验三修改为了 index。

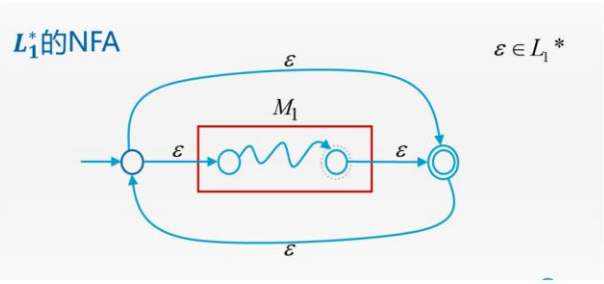
第二次实验

第二次实验，是根据正则表达式（所生成的语法分析树）来构造 NFA。

实现思路

从语法分析树的根节点开始，往下访问，递归构造自动机。编号从 0 开始，构造的时候编号是总体往后递增的，以最后一个状态作为终态。
一个 regex 是若干个 expression 的并。那么从当前的节点往后接这几个自动机的并。最后这些自动机又汇总到一个状态。（这就也使得最后一个状态为唯一的终态）
每个 expression 是若干个 expressionItem 的连接，那么相应地构建自动机的连接。
每个 expressionItem 是由一个 normalItem，以及一个可有可无的 quantifier 组成。根据 quantifier 的情况，对这部分自动机进行相应的处理。
参考了“第二次实验讲解”课件里的思路，通过添加ε转移实现相应的 quantifier，通过修改这些ε转移在 rules 中的先后顺序，实现贪婪和非贪婪匹配。

- 非贪婪匹配的处理



- L*?如何处理?
  - 贪婪匹配和非贪婪匹配，本质是先走什么、后走什么。
  - 恰当的处理走每条规则的顺序
    - 例如，恰当的调整规则在rules数组中出现的顺序

每个 normalItem 是由一个 single 或者一个 group 组成。对于 single，在自动机添加相应的转移。对于 group，则递归处理其中的 regex。
由上，设置了 compileRegex()函数，compileExpression()函数，compileExpressionItem()函数，compileSingle()函数，参数分别是相应的节点（指针）。在 compile()函数中，编译的入口是

compileRegex(tree)。之后就是递归调用相应的函数。
关于 match()函数，由于只需要返回一次匹配结果，则从头开始依次取 text 文本的子串，调用 exec()，直到返回非空路径 path。

重难点问题

对于是否贪婪匹配的处理。主要是修改相应规则的出现顺序。由于我 DFS 是用栈处理，越靠后的规则反而更靠近栈顶，导致越先访问。在实验二，我是在 compileExpression()函数中处理的，在实验三，则修改为在 compileExpressionItem()中处理。

第三次实验

第三次实验是实现更多功能。

实现思路

对于捕获分组，我记录了相应的左右括号所对应的状态 pair，在匹配时进行处理。涉及区间限定符时，将多次重复的归为一类，匹配时尽量匹配靠后的。
对于多行文本的处理，样例 txt 的行尾序列是 LF，即\n，我是基于此进行的处理。
对于 flag，根据 s 的有无，设置 match()函数返回值。根据 m 的有无，在 exec()函数进行处理。
对于 anchor，将其作为特殊的ε转移来处理，在 exec()函数中特判。
对于 rangeQuantifier，参考了实验文档的思路，比如 a{2,5}处理成 aaa?a?a?，并根据是否贪婪匹配进行处理。
对于 matchAll 和 replaceAll，使用基于当前字符串位置 index 的处理方式。matchAll 时，从前往后扫描，replaceAll 时，从后往前替换文本。
修改了对于正则表达式的转义字符的处理。
优化了自动机生成相关的函数，使得状态编号是严格递增，这样就不需要记录当前最晚生成的状态编号，而只需记录当前状态总数即可。
优化了 exec()函数。将 stack 和 set 改为 nfa 的成员变量，而非函数局部变量。将 stack 和 set 的元素中，关于“剩余输入串”的记录方式，改为记录 index。
优化了 backtrace()函数，将 path 元素记录“剩余输入串”的方式，改为记录 index。

重难点问题

从“剩余输入串”到 index 的转变。这很有利于处理很多事情，比如 anchor，flag，matchAll 和 replaceAll 等等，因为这些是前后文相关的，而不单单是剩余输入串的事情。
对于区间限定符的处理。处理起来比较复杂，需要分不同的情况讨论。