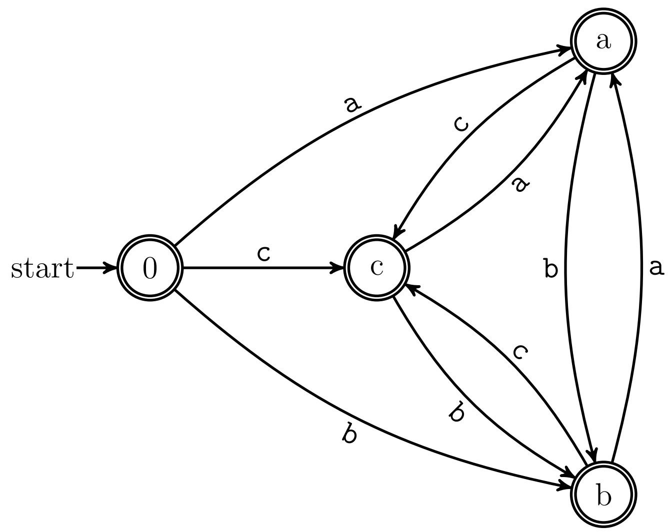
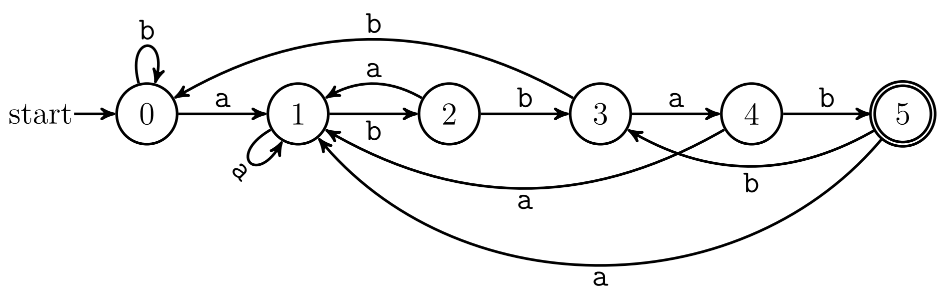
2019 级弘毅班《编译原理》第一次练习答案

一、 为下面的语言集合设计 DFA：

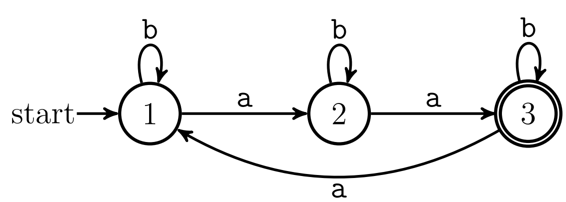
(1) 所有的以 a, b 和 c 组成的字符串，其中，a, b 和 c 均不连续出现。



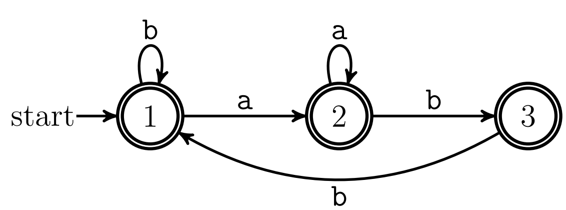
(2) 所有的以 a 和 b 组成的字符串，以 $abbab$ 结尾。



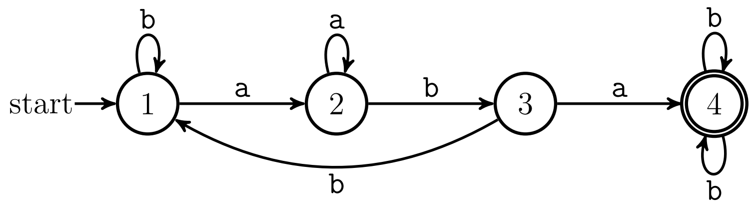
(3) 所有的以 a 和 b 组成的字符串，其中，a 出现的次数除 3 后的余数为 2。



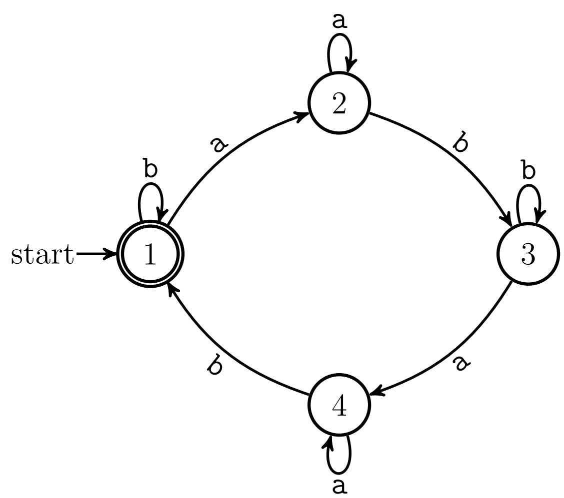
(4) 所有的以 a 和 b 组成的字符串，其中，没有 aba 子串。



(5) 所有的以 a 和 b 组成的字符串，其中，至少有一个 aba 子串。



(6) 所有的以 a 和 b 组成的字符串，其中子串 ab 出现的次数为偶数。



二、 为以下的语言集合编写正则表达式：

(1) 所有的以 a, b 和 c 组成的字符串，其中，a, b 和 c 均不连续出现。
$(a|\varepsilon)(((c|\varepsilon)(bc)^*ba))|((b|\varepsilon)(cb)^*ca))^*(c|\varepsilon)(bc)^*(b|\varepsilon)$

(2) 所有的以 a 和 b 组成的字符串，以 $abbab$ 结尾。
$(a|b)^*abbab$

(3) 所有的以 a 和 b 组成的字符串，其中，a 出现的次数除 3 后的余数为 2。
$(b^*ab^*ab^*a)^*b^*ab^*ab^*$

(4) 所有的以 a 和 b 组成的字符串，其中，没有 aba 子串。
$b^*(aa^*bbb^*)^*a^*b^*$

(5) 所有的以 a 和 b 组成的字符串，其中，至少有一个 aba 子串。
$(a|b)^*aba(a|b)^*$

(6) 所有的以 a 和 b 组成的字符串，其中子串 ab 出现的次数为偶数。
$b^*a^*(a^*abb^*a^*abb^*)^*a^*$

三、 现给下述正规表达式：

$$(ab|b)^*aa(ba|b)^*$$

试对上述正规表达式：

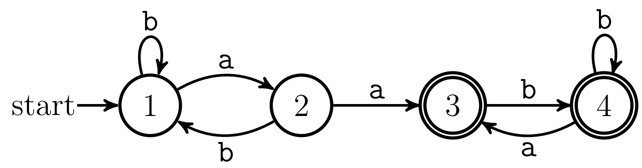
(1) 用 Thompson 构造法将上述正规表达式转化为 NFA；

(2) 用子集构造法将上述的 NFA 转化为 DFA；

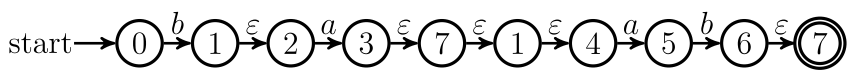
(3) 用状态的划分法将上述的 DFA 的状态最小化；

(4) 用自然语言描述正规表达式所表达的语言.

所描述的语言为以 a 和 b 组成的串，有并仅有唯一的 aa 子串，其最小自动机为：



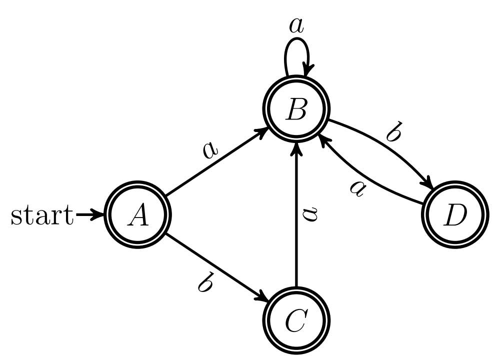
四、 (1)



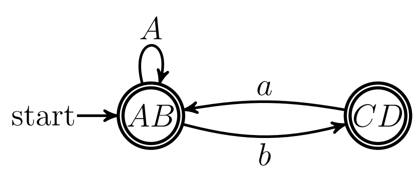
(2)

$A = \{0, 1, 2, 4, 7\}$, $B = \{1, 2, 3, 4, 5, 7\}$, $C = \{1, 2, 4, 7\}$, $D = \{1, 2, 4, 6, 7\}$.

状态转换图为：



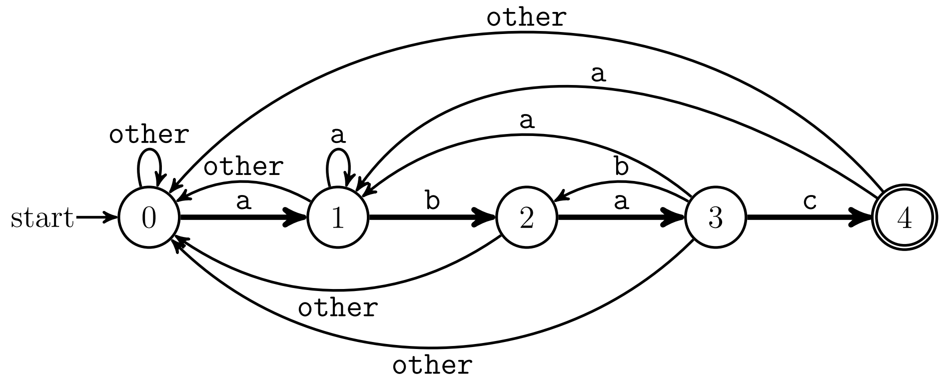
(3) 最小 DFA 如下所示：



(4) 由 a 和 b 组成无连续 b 的字符串.

(5) $r = (b \mid \varepsilon)(a \mid ab)^*$.

五、 KMP 算法将搜索关键字的时间转换为线性时间，设 `t` 是要搜索的关键字，该算法就是一个接受以 `t` 为后缀的自动机。现需要设计一个函数 `int match(char *s)`, 函数对字符串 `s` 扫描，如果发现有子串“`abac`”，则返回 1，否则返回 0，对应的接受 `abac` 为后缀的自动机状态图如下所示:



(1) 将上述状态图转换为流程图；

(2) 利用 C 语言的控制流结构实现函数 `int match(char *s)`;

```
int match (char *s)
{
  char *cp = s;

  while ( *cp != 0 ) {
      /* state 0 */
    if (*cp == 'a')  { /* state 1*/
      cp++;
      for ( ; ; ) {
        if (*cp == 'b') cp++; /* state 2 */
        else break;
        if (*cp == 'a') { /* state 3 */
          cp++;
          if (*cp == 'c')  /* state 4 */
          return 1;
        }
      }
    }
    else
      cp++;
  }
  return 0;
}
```

或

```
int match (char *s)
{
  char *cp = s;

  while ( *cp != 0 ) {
```

```
      /* state 0 */
    while (*cp++ == 'a')  { /* state 1 */
      for ( ; ; ) {
        if (*cp == 'b') cp++; /* state 2 */
        else break;
        if (*cp == 'a') { /* state 3 */
          cp++;
          if (*cp == 'c')  /* state 4 */
            return 1;
        }
      }
    }
  }
  return 0;
}
```

(3) 思考：为什么一般的正则表达式匹配不能用 KMP 算法。
KMP 算法的关键是 `failure(int p)`(p 是自动机的状态，见新版龙书 pp136) 的计算，设要匹配的字符串为 $\mathtt{s} = \mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_\mathtt{n}$，则识别以 s 为后缀的自动机有 $\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{n}$ 共 $n+1$ 个状态 (自动机理论可以证明), 在状态 $\mathtt{s}_\mathtt{p}$ 匹配失败转移的状态 $\mathtt{s}_\mathtt{q}$ 是使得 $\mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_\mathtt{q}$ 能成为 $\mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_\mathtt{p}$ 后缀的最大数 q, 如上例中的接受 “abac” 为结尾的自动机, 即 “a” 是同时为 “aba” 前缀和后缀最大子串，所以 `failure(3) = 1`，而一般自动机对应的状态转移函数不能用该方法计算.

附上 KMP 算法的 C 语言实现

```
    /* KMP algorithm, s is the searched string,
       k is keyword to search
     */
  int kmp_match(char *s, char * k)
  {
    int i,j;
    int *failure = (int *) malloc
                   ((strlen(k)+1)* sizeof(int));

      /* if keyword empty, it will match any thing */
    if (*k == 0) return 1;

      /* compute failure function */
    failure[0] = -1;       /* set up for loop */
    for ( i = 0; k[i] != 0; i++) {
      failure[i + 1] = failure[i] + 1;
      while (failure[i + 1] > 0 &&
             k[i] != k[failure[i + 1] - 1])
        failure[i + 1] = failure[failure[i+1]-1]+1;
```

```
  }

  /* suffix automata */
  j = 0;
  for (i = 0; s[i]!= 0; i++)
    for (;;) {      /* state 0 */
      if (s[i] == k[j]) {  /* state j */
        j++;        /* state j+1 */
        if (k[j] == 0)
          return i; /* accepting state */
        break;
      }
      else
       if (j == 0)
         break;  /* goto state 0 */
       else
         j = failure[j];  /* goto state failure[j] */
    }
  return 0;
}
```