第一组（本组）听课笔记

——DFA 最小化

本组汇报选题是第四题：DFA 最小化。汇报主要从 DFA 最小化概念介绍、Hopcroft DFA 算法以及 Brzozowski 算法三个方面入手。

一、DFA 最小化概念介绍

1. 最小化的定义：

- 没有多余的状态（死状态）；
- 没有两个状态是相互等价的；

2. 两个状态等价的含义：

- 兼容性（一致性）——同是终态或同是非终态；
- 传播性（蔓延性）——如果 s 和 t 等价，那么从 s 出发读入某个 a 和从 t 出发读入某个 a 到达的状态等价。

3. 最小化 DFA 的好处

- 它占用的计算资源（内存、cache）会更少；
- 可以提高算法的运行效率和速度。

二、Hopcroft DFA 算法

Hopsroft 算法就是先根据非终结状态与终结状态将所有的节点分为 N 和 A 两大类。N 为非终结状态，A 为终结状态，之后再对每一组运用基于等价类实现的切割算法。

```
//基于等价类的思想
split(S)
    foreach(character c)
        if(c can split s)
            split s into T1, ..., Tk

hopcroft()
    split all nodes into N, A
    while(set is still changes)
        split(s)
```

算法抽象：

1: Q/θ ← {F, Q − F}

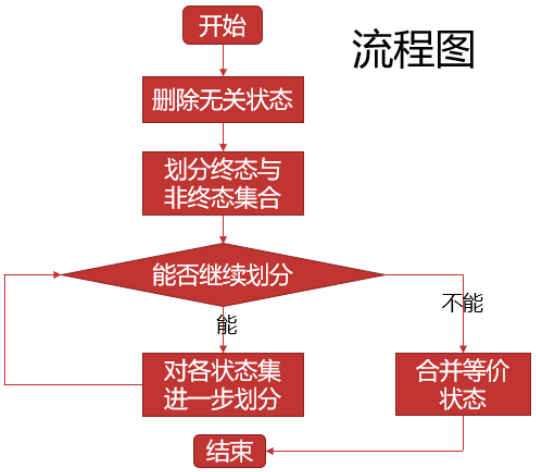
2: while (∃U, V ∈ Q/θ, a ∈ Σ)

s.t. Equation 1 holds do

3: Q/θ ← (Q/θ − {U})

∪ {U ∩ δ^-1(V, a), U − U ∩ δ^-1(V, a)}

4: end while



三、Brzozowski 算法

1. 算法思想

- 将 DFA 的边反转，使初始状态为接受状态，而接受状态为初始状态（将产生一个原语言反序的非确定有限状态自动机（NFA））；
- 将这个 NFA 用标准的幂集构造法（只构造转换后 DFA 的可达状态）转换为 DFA；此时的 DFA 就将会变小一点；
- 重复反转操作，此时就完成了 DFA 的最小化。

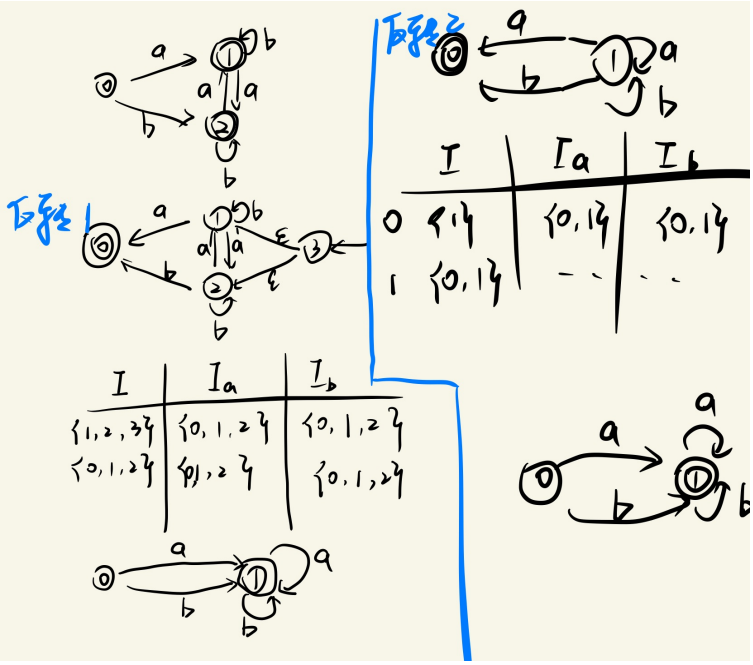
2. 浅显的分析

首先，DFA 转化为 NFA 的过程正是在消除重复前缀的过程。那么 B 算法第一步做得反转 DFA 再转换成 NFA 就是在消除重复后缀的过程。

那么，之所以消除重复前缀和后缀就可以得到最小化的 DFA 呢？

原因在于，合并相同状态主要在于看前一个状态和转换条件（读入的字符）是否重复了。而对于消除了前缀后缀的状态机之后，只剩下了初始态和终态，因此得到了最小化的 DFA。

3. 举例



4. 弊端

NFA 转 DFA 的时间复杂度是远超多项式时间的，但是，往往在经历 DFA 最小化这一步之前肯定会有 NFA 转化到 DFA 这一步，因此，做了两遍与做一遍的时间复杂度其实是一样的。

因为存在这样的正则语言，其反序的最小 DFA 的规模是原语言 DFA 规模的指数大，所以 B 算法在最坏情况下的复杂度是指数的。

第二组听课笔记

——词法分析器

1. **词法分析**：主要任务是读入源程序的输入字符，将它们组成词素，生成并输出一个词法单元序列，这个词法单元序列被输出到语法分析器进行语法分析。

   设计方法：单词符号分类、词法分析输出形式、状态转换图、正规表达式、有限自动机。

2. **C 语言的词法分析器**：处理 c 语言源程序，过滤掉无用符号，判断源程序中单词的合法性，并分解出正确的单词，以二元组形式存放在文件中。

识别单词符号:

```
第一类：标识符  letter(letter | digit)* 无穷集
第二类：常数  (digit)+ 无穷集
第三类：保留字(32)
auto    break   case    char    const   continue
default do      double  else    enum    extern
float   for     goto    if      int     long
register return short   signed  sizeof  static
struct  switch  typedef union   unsigned void
volatile while
第四类：界符 '/*'、'//'、 (){}[]" " ' 等
第五类：运算符 <、<=、>、>=、=、+、-、*、/、^、等
```

将各类单词符号的识别封装为函数来进行判断。根据返回值 true 或者 false 进一步输出符号。

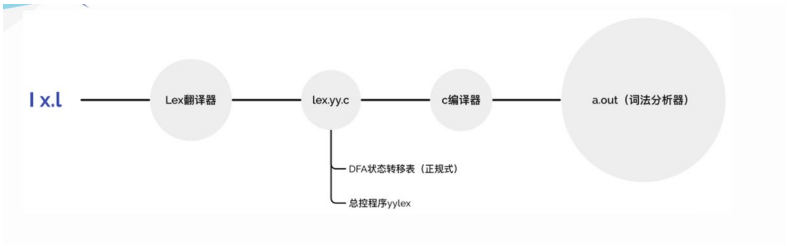
第三组听课笔记

——LEX

1. **LEX 是什么?**：主要功能是生成一个词法分析器的 C 源码，描述规则采用正则表达式。

   核心部分：Lex 编译器将输入的模式转换成一个状态转换图，并生成相应的代码存放到文件 lex . y y . c 中，这些代码模拟了状态转换图。

2. **LEX 的基本原理**:



3. **LEX 的使用：**

   将.l 文件生成可执行文件的命令:

   $flex test.l

   $gcc lex.yy.c –lfl

$./a.out

.l 文件定义段：以符号%{和%}包裹，里面为以 C 语法写的一些定义和声明。

.l 文件词法规则段：列出的是词法分析器需要匹配的正规式，以及匹配该正规式后需要进行的相关动作。

.l 文件辅助函数段：辅助函数一般是在词法规则段中用到的函数。

4. LEX 中的冲突解决：

当输入的多个前缀与一个或多个模式匹配时，Lex 将按照以下两个规则选择词素：

- 总是选择最长的前缀；
- 如果最长的可能前缀与多个模式匹配，总是选择在 Lex 程序中先被列出的模式。

5. 向前看运算符：

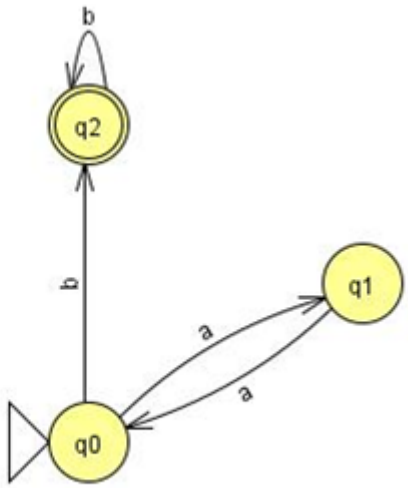
某些时候我们希望仅当词素后面跟随特定的字符时，这个词素才可以与某个模式进行匹配。 r1/r2，r1 表示词素，r2 表示附加模式，只有附加模式与输入匹配时，我们才可以确定词素与某个模式匹配。

第四组听课笔记

——DFA代码化

具体实例代码

以(aa)*b+为例子，根据其DFA的三个状态，设计了三个状态跳转函数。并且设计了检查是否到达终态的函数。




```
void state0(char c)
{
    if (c == 'a') {
        curstate = 1;
    }
    else if (c == 'b') {
        curstate = 2;
    }

    // -1 is used to check for any invalid symbol
    else {
        curstate = -1;
    }
}
```

增加判断正则表达式是否合法的函数：

```
bool isAccepted(char str[])
{
    // store length of string
    int i, len = strlen(str);
```

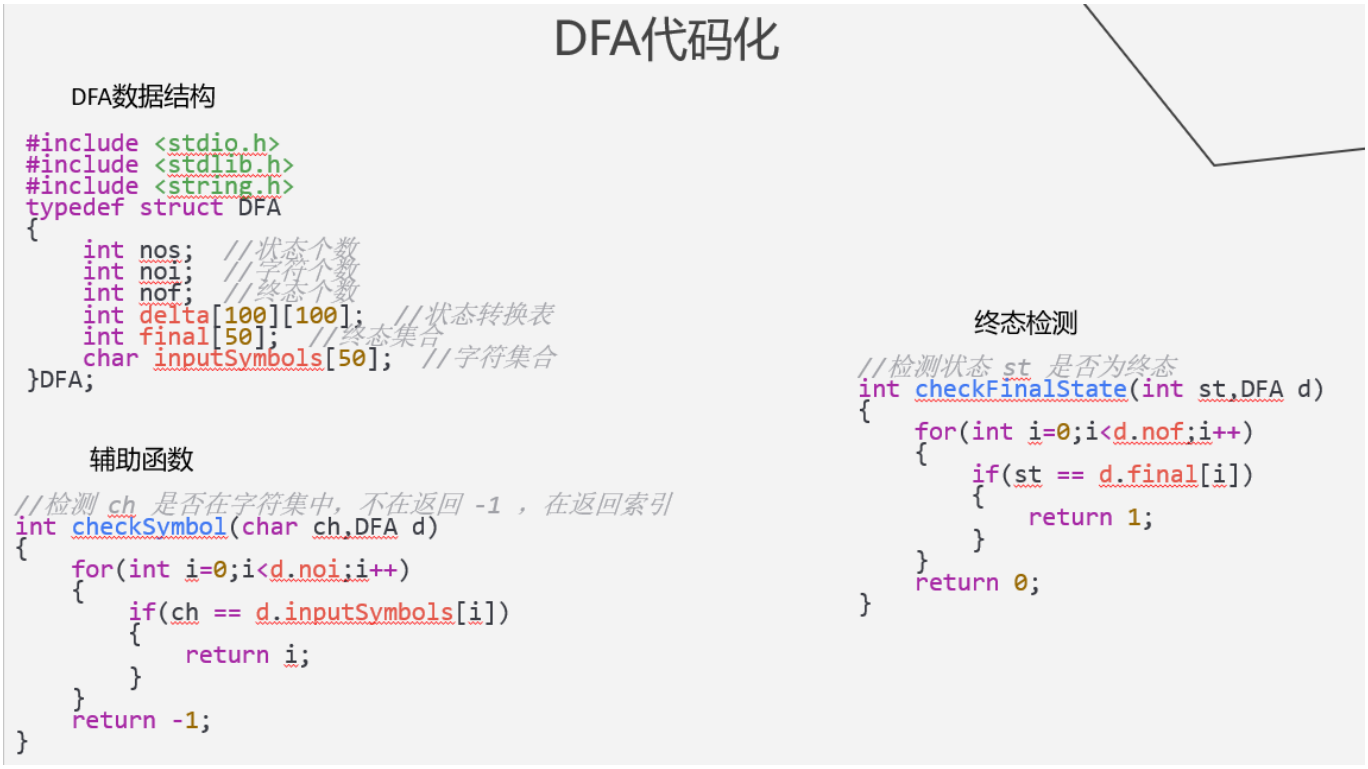
```
        for (i = 0; i <
         len; i++) {
            if (curstate == 0)
                state0(str[i]);

            else if (curstate == 1)
                state1(str[i]);

            else if (curstate == 2)
                state2(str[i]);

            else
                return false;
        }
        if (curstate == 2)
            return true;
        else
            return false;
    }
```

利用了状态转换表的思想,确定要输入 DFA 的信息，包含状态个数，字符有哪些，终结状态有哪些，以及转化关系（转化表）。



DFA的应用

计算机操作系统中的进程状态与切换可以作为 DFA 算法的一种近似理解。

第五组听课笔记

——正则表达式

概念

正则表达式(Regular Expression)是一种文本模式，包括普通字符 （例如，a 到 z 之间的字母）和特殊字符（称为"元字符"）。

正则表达式使用单个字符串来描述，匹配一系列匹配某个句法规则的字符串。

在验证、查找、替换中应用很广。

贪婪与非贪婪

举个例子:

/<.*>/

`<h1>hello world!</h>`匹配到的是整个字符串0-19。

/<.*?>/

`<h1>hello world!</h>`匹配到的就是字符串0-3。

也就是说通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从"贪婪"表达式转换为"非贪婪"表达式或者最小匹配。

普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、一些标点符号和一些其他符号。

非打印字符

非打印字符也可以是正则表达式的组成部分。

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

特殊字符

所谓特殊字符，就是一些有特殊含义的字符。

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n之外的任何单字符。要匹配 .，请使用 \。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
|	指明两项之间的一个选择。要匹配 |，请使用 \\|。

第六组听课笔记

——DFA确定化

基本概念

M = (S，Σ，δ，s0，F)

S：有穷状态集
Σ：输入字母表，即输入符号集合。且ε不是 Σ中的元素
δ：将S×Σ映射到S的转换函数。□s∈S, a∈Σ, δ(s,a)表示从状态s出发，沿着标记为a的边所能到达的状态。
s0：开始状态 (或初始状态)，s0∈S
F：接收状态（或终止状态）集合，F⊆ S

子集构造法

ε-closure(s):能够从NFA的状态s开始通过ε转换到达的NFA状态集合。

ε-closure(T):能够从T中某个NFA状态s开始通过ε转换到达的NFA状态集合。

DFA和NFA对比

NFA：

1. 能比较直观地反推出正则表达式。
2. 状态数更多。

DFA：

1. 看起来比较抽象复杂不易反推出正则表达式。
2. 状态数更少。

转化通用步骤：RE→NFA→子集构造法→DFA状态转化表→DFA

第七组听课笔记

——TINY

介绍

不像C，C++等真实的语言那么庞大，也不是太小，足够反映真实编译器。

1. TINY的程序结构是一个由分号分隔开的语句序列。
2. 既无过程也无声明。
3. 所有的变量都是整型变量，通过赋值即可声明。
4. 只有两个控制语句： if语句和repeat语句，这两个控制语句本身也可包含语句序列。If语句有一个可选的else部分且必须由关键字end结束。
5. read语句和write语句完成输入/输出。
6. TINY的表达式只有布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式的比较组成，比较符号是<或=。
7. 算术表达式可以包括整型常数、变量、参数以及4个整型算符+、 - 、*、/，具有一般的数学属性。
8. 注释放在大括号内，注释不能嵌套。

语法

注释：放在一对大括号内，不能嵌套；
关键字：read write if end repeat until else；
类型：只支持整型和布尔型；
运算符：+ - * / ( ) < = :=，其中:=为赋值运算，=为判断。没有>和<=和>=。

词法单元

词法单元类型	词法单元	词素（例）
关键字（预定义符）	IF	if
	THEN	then
	ELSE	else
	END	end
	REPEAT	repeat
	UNTIL	until
	READ	read
	WRITE	write
自定义符	ID	myName
	NUM	123

运算符	ASSIGN	:=
	EQ	=
	LT	<
	PLUS	+
	MINUS	-
	TIMES	*
	OVER	/
	LPAREN	(
	RPAREN	)
	SEMI	;
错误	ERROR	>=

非终结符	含义	展开
program	程序	stmt_seq
stmt_seq	若干条语句	stmt_seq SEMI stmt | stmt
stmt	单条语句	if_stmt | repeat_stmt | assign_stmt | read_stmt | write_stmt | error
if_stmt	判断语句	IF exp THEN stmt_seq END | IF exp THEN stmt_seq ELSE stmt_seq END
repeat_stmt	循环语句	REPEAT stmt_seq UNTIL exp
assign_stmt	赋值语句	ID ASSIGN exp
read_stmt	输入语句	READ ID
write_stmt	输出语句	WRITE exp
exp	判断表达式	simple_exp LT simple_exp | simple_exp EQ simple_exp| simple_exp
simple_exp	加减表达式	simple_exp PLUS term | simple_exp MINUS term | term
term	乘除表达式	term TIMES factor | term OVER factor | factor
factor	括号表达式	LPAREN exp RPAREN | NUM | ID | error