



南開大學  
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

---

## 实现词法分析器构造算法

---

艾明旭 2111033

年级：2021 级

指导教师：李忠伟

2023 年 10 月 25 日

# 目录

<b>一、 上下文无关文法</b>	<b>1</b>
(一) SysY 语言特性	1
(二) CFG 描述 SysY 语言特性	1
1. 关键字	1
2. 变量	2
3. 常量	3
4. 运算符和表达式	3
5. 语句	4
6. 函数	5
(三) 形式化定义	5
1. 变量声明	6
2. 常量声明	6
3. 表达式	6
4. 赋值表达式	7
5. 逻辑表达式	7
6. 关系表达式	7
7. 算数表达式	7
8. 函数	8
9. 系统操作	8
10. 循环语句	8
11. 分支语句	8
12. 跳转语句	8
<b>二、 正则表达式 <math>\Rightarrow</math> NFA</b>	<b>8</b>
(一) 正则表达式构造 CFG	8
(二) 构造数据结构	9
1. 状态:	9
2. 转换:	9
3. NFA 结构	9
4. 构造操作	9
5. 解析和构建	10
6. 扩展性	10
7. 内存管理	10
(三) Thompson 算法的详细思路	10
1. 基本 NFA 片段的构造	10
2. 连接操作	11
3. 选择操作	11
4. 闭包操作	11
5. 算法过程	11
6. 算法的优缺点	11
(四) 编程实现	11
1. 添加转换:	14
2. 构造 NFA	14

3.	NFA 的操作 . . . . .	14
4.	Thompson 算法的构造 . . . . .	14
5.	主函数: . . . . .	14
<b>三、 NFA 到 DFA 的转换</b>		<b>14</b>
(一)	从 NFA 到 DFA 的子集构造法的算法思路 . . . . .	14
1.	计算 e-closure(T) 的算法 . . . . .	14
2.	NFA 到 DFA 的转换 . . . . .	15
(二)	子集构造法的算法实现 . . . . .	16
1.	使用的数据结构 . . . . .	21
2.	功能 . . . . .	22
<b>四、 DFA 最小化</b>		<b>22</b>
(一)	DFA 最小化的实现思路 . . . . .	22
(二)	代码实现 . . . . .	23
(三)	所用到的结构 . . . . .	26
1.	DFA 结构 . . . . .	26
2.	split 函数 . . . . .	26
3.	minimize 函数 . . . . .	26
<b>五、 正则表达式转换到 DFA</b>		<b>27</b>
(一)	直接进行正则表达式到 DFA 的转换 . . . . .	27
(二)	解析组合子的构建 . . . . .	28
(三)	代码实现 . . . . .	28
(四)	算法的效率分析 . . . . .	32
1.	算法的准确性 . . . . .	32
2.	Thompson 算法的效率 . . . . .	32
3.	子集构造法的效率 . . . . .	32
4.	DFA 最小化的效率 . . . . .	33
5.	整体性能考量 . . . . .	33
<b>六、 总结</b>		<b>34</b>

item	system	Gcc version	Target	LLVM
content	ubuntu 22.04.3	11.4.0	x86_64.gnu	14.0.0

表 1: 实验环境

## 一、 上下文无关文法

### (一) SysY 语言特性

- 数据类型: int
- 变量声明、常量声明, 常量、变量的初始化
- 语句: 赋值 (=)、表达式语句、语句块、if、while、return
- 表达式: 算术运算 (+、-、\*、/、%, 其中 +、- 都可以是单目运算符)、关系运算 (==, >, <, >=, <=, !=) 和逻辑运算 (&& (与)、|| (或)、! (非))
- 注释
- 输入输出
- 数组
- 变量、常量作用域——在语句块中包含变量、常量声明, break、continue 语句
- 函数

### (二) CFG 描述 SysY 语言特性

- CFG 是一个四元组  $G = (N, T, P, S)$ , 其中
- (1)  $N$  是非终结符 (Nonterminals) 的有限集合;
  - (2)  $T$  是终结符 (Terminals) 的有限集合, 且  $N \cap T = \emptyset$ ;
  - (3)  $P$  是产生式 (Productions) 的有限集合,  $A \rightarrow a$ , 其中  $A \in N$  (左部),  $a \in (N \cup T)^*$  (右部), 若  $a = \epsilon$ , 则称  $A \rightarrow \epsilon$  为空产生式 (也可以记为  $A \rightarrow$ );
  - (4)  $S$  是文法的开始符号 (Start symbol),  $S \in N$

CFG 产生语言的基本方法: 推导

推导的定义

推导的定义将产生式左部的非终结符替换为右部的文法符号序列 (展开产生式, 用标记  $\Rightarrow$  表示), 直到得到一个终结符序列。

推导的符号:  $\Rightarrow$

推导的输入: 产生式左部

推导的输出: 一个终结符序列

#### 1. 关键字

C++ 常用关键字如表2所示, 每一个关键字在上下文无关文法中都会看作一个终结符, 即语法树的叶结点。本实验将选取其中的一部分作为子集, 构造 SysY 语言。

类型	关键字
数据类型相关	<i>int, bool, true, false, char, wchar_t, int, double, float, short, long, signed, unsigned</i>
控制语句相关	<i>switch, case, default, do, for, while, if, else, break, continue, goto</i>
定义、初始化相关	<i>const, volatile, enum, export, extern, public, protected, private, template, static, struct, class, union, mutable, virtual</i>
系统操作相关	<i>catch, throw, try, new, delete, friend, inline, operator, reinterpret_cast, typename</i>
命名相关	<i>using, namespace, typeof</i>
函数和返回值相关	<i>void, return, sizeof, typedef</i>
其他	<i>this, asm, _cast</i>

表 2: C++ 关键字

## 2. 变量

C 语言中规定，将一些程序运行中可变的值称之为变量，与常量相对。在程序运行期间，随时可能产生一些临时数据，应用程序会将这些数据保存在一些内存单元中，每个内存单元都用一个标识符来标识。这些内存单元我们称之为变量，定义的标识符就是变量名，内存单元中存储的数据就是变量的值。变量可以作左值，常量则只能作为右值。变量除了与常量相同的整型类型、实型类型、字符类型这三个基本类型之外，还有构造类型、指针类型、空类型。

### 三种基本类型

**整型变量** 整型常量为整数类型 *int* 的数据。可分别如下表示为八进制、十进制、十六进制

- 十进制整型变量：0, 123, -1
- 八进制整型变量：0123, -01
- 十六进制整型变量：0x123, -0x88

**实型变量** 实型变量是实际中的小数，又称为浮点型变量。按照精度可以分为单精度浮点数（float）和双精度浮点数（double）。浮点数的表示有三种方式如下：

- 指明精度的表示：以 f 结尾为单精度浮点数，如：2.3f；以 d 结尾为双精度浮点数，如：3.6d
- 不加任何后缀的表示：11.1, 5.5
- 指数形式的表示：5.022e+23f, 0f

**字符变量** *char* 用于表示一个字符，表示形式为 '需要表示的字符常量'。其中，所表示的内容可以是英文字母、数字、标点符号以及由转义序列来表示的特殊字符。如 'a' '3' ' ' '\n'。

**变量的定义** 用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。变量有三个基本要素：变量名，代表变量的符号；变量的数据类型，每一个变量都应具有一种数据类型且内存中占据一定的储存空间；变量的值，变量对应的存储空间中所存放的内容。变量的定义可以以如下的形式：

```
1 type variable_list
```

在我们定义的 SysY 语言中，将支持整型变量（十进制）、字符型变量以及行主存储的整型一维数组类型。

### 3. 常量

C 语言中规定，将一些不可变的值称之为常量。常量可以分为整型常量、实型常量、字符常量这三种常量，其形式与整型变量、实型变量、字符变量基本相同，只不过在声明时必须初始化且在程序中不可以改变其值。在我们定义的 SysY 语言中，将定义整型常量（十进制）和字符型常量。

### 4. 运算符和表达式

在 C 语言中，运算符分为算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、杂项运算符六大类。其中，我们所设计的 SysY 语言将定义以下运算符：

运算符	描述
+	把两个操作数相加
-	从第一个操作数中减去第二个操作数
*	把两个操作数相乘
/	分子除以分母
%	取模运算符，整除后的余数

表 3: 算术运算符

运算符	描述
==	检查两个操作数的值是否相等，如果相等则条件为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

表 4: 关系运算符

**表达式** 由运算分量和运算符按一定规则组成。运算分量是运算符操作的对象，通常是各种类型的数据。运算符指明表达式的类型；表达式的运算结果是一个值——表达式的值。出现在赋值运算符左边的分量为左值，代表着一个可以存放数据的存储空间；左值只能是变量，不能是常量或表达式，因为只有变量才可以带表存放数据的存储空间。出现在赋值运算符右边的分量为右值，右值没有特殊要求。

运算符	描述
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。
	称为逻辑或运算符。如果两个操作数中有任何一个非零，则条件为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。

表 5: 逻辑运算符

运算符	描述
=	简单的赋值运算符，把右边操作数的值赋给左边操作数。

表 6: 赋值运算符

运算符	描述
&	返回变量的地址。
*	指向一个变量。

表 7: 复杂运算符

**运算符优先级** 运算符中优先级确定了表达式中项的组合，这会极大地影响表达式的计算过程以及结果。运算的优先顺序为：括号优先运算 → 优先级高的运算符优先运算 → 优先级相同的运算参照运算符结合性依次进行。当表达式包含多个同级运算符时，运算的先后次序分为左结合规则和右结合规则。其中左结合规则是从左向右依次计算，包括的运算符有双目的算术运算符、关系运算符、逻辑运算符、位运算符、逗号运算符；右结合规则是从右向左依次计算，包括的运算符有可以连续运算的单目运算符、赋值运算符、条件运算符。运算符优先级由高到低排列：后缀 → 一元 → 乘除 → 加减 → 移位 → 关系 → 相等 → 位与 → 位异或 → 位或 → 逻辑与 → 逻辑或 → 条件 → 赋值 → 逗号

## 5. 语句

在 C 语言中，语句分为说明语句、表达式语句、控制语句、标签语句、复合语句和块语句。在我们所定义的 SysY 语言中，我们将定义表达式语句和控制语句，其中控制语句分为分支语句、循环语句和转向语句。

**表达式语句** 任意有效表达式都可以作为表达式语句，其形式为表达式后面加上“;”。

**分支语句** if 语句和 if……else 语句，由关键字 if 和 else 组成。其基本形式如下

```

1  if(expr){
2      stmts
3  }
4  else{
5      stmts
6  }
```

**循环语句** 又称重复语句，用于重复执行某些语句。本实验的循环语句由 while 语句实现，基本形式如下

```
1 while{  
2     stmts  
3 }
```

**转向语句** 用于从循环体跳出的 break 语句；用于立即结束本次循环而去继续下一次循环的 continue 语句；用于立即从某个函数中返回到调用该函数位置的 return 语句。

## 6. 函数

**函数的定义** 函数是一组一起执行一个任务的语句。程序中功能相同，结构相似的代码段可以用函数进行描述。函数的功能相对独立，用来解决某个问题，具有明显的入口和出口。函数也可以称为方法、子例程或程序等等。

**函数说明** C 语言中，函数必须先说明后调用。函数的说明方式有两种，一种是函数原型，相当于“说明语句”，必须出现在调用函数之前；一种是函数定义，相当于“说明语句 + 初始化”，可以出现在程序的任何合适的地方。在函数声明中，参数的名称并不重要，只有参数的类型是必需的。函数的声明形式如下所示：

```
1 return_type function_name(parameter list);
```

**形式化定义** C 语言中，函数的形式化定义如下所示：

```
1 return_type function_name(parameter list)  
2 {  
3     body of the function  
4 }
```

本实验定义的 SysY 语言的函数定义完全与此相同。

**函数的参数** 函数可以分为有参函数和无参函数。如果函数要使用参数，则必须声明接受参数值的变量，这些变量称为函数的形式参数。形式参数和函数中的局部变量一样，在函数创建时被赋予地址，在函数退出是被销毁。函数参数的调用分为传值调用和引用调用两种，传值调用是将实际的变量的值复制给形式参数，形式参数在函数体中的改变不会影响实际变量；引用调用是将形式参数作为指针调用指向实际变量的地址，当对在函数体中对形式参数的指向操作时，就相当于对实际参数本身进行的操作。

除此之外，函数还可以分为内联函数、外部函数等等，并还可以进行重载等操作。**本次实验的 SysY 语言，只对函数最基本的功能进行实现。**

## (三) 形式化定义

接下来，我们将采用 CFG 即上下文无关文法对 SysY 语言进行形式化定义。上下文无关文由一个终结符号集合  $V_T$ 、一个非终结符号集合  $V_N$ 、一个产生式集合  $P$  和一个开始符号  $S$  四个元素组成。在接下来的定义中，数位、符号和黑体字符串将被看作终结符号，斜体字符串将被看



作非终结符号。若多个产生式以一个非终结符号为头部，则这些产生式的右部可以放在一起，并用 | 分割。

名称	符号	名称	符号
声明语句	<i>decl</i>	标识符	<i>id</i>
标识符列表	<i>idlist</i>	数据类型	<i>type</i>
表达式	<i>expr</i>	一元表达式	<i>unary_expr</i>
赋值表达式	<i>assign_expr</i>	逻辑表达式	<i>logical_expr</i>
算数表达式	<i>math_expr</i>	关系表达式	<i>relation_expr</i>
数字	<i>digit</i>	整数	<i>decimal</i>
符号和字母	<i>character</i>	常量定义	<i>const_init</i>
分配内存	<i>allocate</i>	回收内存	<i>recovery</i>
语句	<i>stmt</i>	循环语句	<i>loop_stmt</i>
分支语句	<i>selection_stmt</i>	跳转语句	<i>jmp_stmt</i>
函数定义	<i>funcdef</i>	函数参数	<i>para</i>
函数参数列表	<i>paralist</i>	函数名称	<i>funcname</i>
函数返回值	<i>re_type</i>		

表 8: 下文中各符号含义

## 1. 变量声明

变量可以声明分为仅声明变量和声明变量且赋初值。(整型变量只支持十进制) 数组由指针实现。

$$\begin{aligned}
 idlist &\rightarrow idlist, id \mid id \\
 type &\rightarrow int \mid char \\
 decl &\rightarrow type \ idlist \mid \\
 &\quad type \ id = logical\_expr \mid \\
 &\quad type \ id = unary\_expr \mid \\
 &\quad type \ *id = \&id
 \end{aligned}$$

## 2. 常量声明

常量包括整型常量（十进制）和字符型常量。

$$const\_init \rightarrow const \ type \ id = unary\_expr$$

## 3. 表达式

表达式可以分为一元表达式、赋值表达式、逻辑表达式、算数表达式、关系表达式。

$$expr \rightarrow unary\_expr \mid assign\_expr \mid logical\_expr \mid math\_expr \mid relation\_expr$$

#### 4. 赋值表达式

赋值表达式不能对常量进行赋值。

$$\begin{aligned} \text{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{decimal} &\rightarrow \text{digit} \mid \text{decimal digit} \\ \text{character} &\rightarrow \_ \mid \text{a} - \text{z} \mid \text{A} - \text{Z} \\ \text{unary\_expr} &\rightarrow \text{decimal} \mid \text{'character'} \mid \text{id} \\ \text{assign\_expr} &\rightarrow \text{id} = \text{unary\_expr} \mid \\ &\quad \text{id} = \text{logical\_expr} \mid \\ &\quad \text{id} = \text{funcnmae}(\text{paralist}) \end{aligned}$$

#### 5. 逻辑表达式

逻辑表达式包括逻辑与、逻辑或、逻辑非运算。

$$\begin{aligned} \text{logical\_expr} &\rightarrow \text{unary\_expr} \mid \\ &\quad \text{!(logical\_expr)} \mid \\ &\quad \text{logical\_expr} \parallel \text{logical\_expr} \mid \\ &\quad \text{logical\_expr} \& \& \text{logical\_expr} \end{aligned}$$

#### 6. 关系表达式

关系表达式包括判断两个值是否相等或比较两值的大小。

$$\begin{aligned} \text{relation\_expr} &\rightarrow \text{unary\_expr} == \text{unary\_expr} \mid \\ &\quad \text{unary\_expr} != \text{unary\_expr} \mid \\ &\quad \text{unary\_expr} > \text{unary\_expr} \mid \\ &\quad \text{unary\_expr} < \text{unary\_expr} \mid \\ &\quad \text{unary\_expr} >= \text{unary\_expr} \mid \\ &\quad \text{unary\_expr} <= \text{unary\_expr} \end{aligned}$$

#### 7. 算数表达式

算数表达式包括加、减、乘、除、取模、取负六种运算。

$$\begin{aligned} \text{math\_expr} &\rightarrow \text{unary\_expr} \mid \\ &\quad - \text{unary\_expr} \mid \\ &\quad \text{math\_expr} + \text{math\_expr} \mid \\ &\quad \text{math\_expr} - \text{math\_expr} \mid \\ &\quad \text{math\_expr} * \text{math\_expr} \mid \\ &\quad \text{math\_expr} / \text{math\_expr} \mid \\ &\quad \text{math\_expr} \% \text{math\_expr} \end{aligned}$$

## 8. 函数

函数的返回值有整型、字符型、指针型三种，参数有整型、字符型、指针型、引用四种。（数组由指针实现）

$$\begin{aligned} funcdef &\rightarrow re\_type\ funcname(paralist)\ stmt \\ paralist &\rightarrow para, paralist \mid para \\ para &\rightarrow type\ id \mid \\ &\quad type * id \mid \\ &\quad type\& id \\ re\_type &\rightarrow type \mid type * \mid void \end{aligned}$$

## 9. 系统操作

系统操作包括分配内存和回收内存。

$$\begin{aligned} allocate &\rightarrow type *id = new\ type\ [id] \mid \\ &\quad type *id = new\ type\ [decimal] \\ recovery &\rightarrow delete\ []\ id \end{aligned}$$

## 10. 循环语句

循环语句利用 while 实现。

$$loop\_stmt \rightarrow while(expr)\ \{stmt\}$$

## 11. 分支语句

分支语句利用 if else 实现。

$$\begin{aligned} selection\_stmt &\rightarrow if(expr)\ \{stmt\} \mid \\ &\quad if(expr)\ \{stmt\}\ else\ \{stmt\} \end{aligned}$$

## 12. 跳转语句

跳转语句包括继续执行循环、退出循环和返回值。

$$\begin{aligned} jmp\_stmt &\rightarrow continue \mid \\ &\quad break \mid \\ &\quad return \mid \\ &\quad return\ expr \end{aligned}$$

# 二、 正则表达式 =>NFA

## (一) 正则表达式构造 CFG

首先利用需要转化成 NFA 的正则表达式手动的构造相应的 CFG，为后续的递推转换函数的编写做准备。

## (二) 构造数据结构

### 1. 状态:

NFA 的基本单元, 状态需要有是否终止以及转换函数两个参数

```
1     typedef struct State {  
2         int isFinal; // 是否为终止状态: 1表示是, 0表示否  
3         struct Transition* trans; // Pointing to transitions list  
4     } State;
```

### 2. 转换:

NFA 中, 从一个状态到另一个状态的移动是由转换驱动的。转换可以是基于符号 (例如 'a') 或不需要任何输入符号就可以进行的转换。

```
1     typedef struct Transition {  
2         State* from; // 起始状态  
3         State* to; // 终止状态  
4         char symbol; // 在这两个状态之间转移时要读取的字符  
5         struct Transition* next; // 链接到下一个转换的指针  
6     } Transition;
```

每个状态都有一个指向其转换列表的指针。

### 3. NFA 结构

NFA 的主体结构包括起始状态和终止状态。这两个状态是创建和操作 NFA 时最关键的参考点。

```
1     typedef struct Transition {  
2         State* from; // 起始状态  
3         State* to; // 终止状态  
4         char symbol; // 在这两个状态之间转移时要读取的字符  
5         struct Transition* next; // 链接到下一个转换的指针  
6     } Transition;
```

### 4. 构造操作

Thompson 构造法的美妙之处在于其简单性。它定义了一系列基本操作来处理正则表达式的基本元素:

- 联合 (Union): 通过创建一个新的起始状态和一个新的终止状态来处理两个 NFA, 然后将新的起始状态与这两个 NFA 的起始状态连接, 将这两个 NFA 的终止状态连接到新的终止状态。
- 串联 (Concatenation): 通过简单地将第一个 NFA 的终止状态连接到第二个 NFA 的起始状态来处理两个 NFA。
- 闭包 (Closure): 为一个 NFA 创建一个新的起始和终止状态, 然后添加必要的转换以实现 Kleene 星号的行为。

每个操作都会返回一个新的 NFA, 这个 NFA 可以与其他 NFA 通过上述操作进行组合。

## 5. 解析和构建

从正则表达式创建 NFA，我们还需要一个解析器。这个解析器会递归地读取正则表达式的每一部分，并为每一部分调用相应的构造操作。这通常涉及到诸如 `expression`, `term`, `factor` 和 `base` 之类的函数，它们对应于正则表达式语法的不同级别。

- 表达式: 表达式处理的是正则表达式中的顶级结构。它寻找 “[” 符号，这是正则表达式中的联合操作。每次遇到这个符号，解析器都会分裂，将正则表达式分为左右两部分，然后递归地对每部分进行解析。最终，两个解析的部分通过联合操作进行组合。
- 项: 项处理串联。它不断地寻找可以串联的元素，如字母、数字或子表达式。一旦找到这些元素，它会递归地调用更低级别的解析函数来处理它们，并通过串联操作将得到的 NFA 连接起来。
- 基本函数: 基本函数处理正则表达式中的最基本元素。这包括单个字符、字符类或括号中的子表达式。对于单个字符，解析器简单地为该字符创建一个新的 NFA。对于子表达式，解析器递归地调用 `expression` 函数来处理括号内的内容。
- 因子: 因子处理闭包操作，即 Kleene 星号 \*。当解析器在一个元素后面看到一个星号时，它知道该元素可以重复 0 次或多次。为了处理这个，它会首先解析星号前的元素，然后对得到的 NFA 应用闭包操作。

解析器处理完整个正则表达式，就会返回一个完整的 NFA，该 NFA 描述了整个正则表达式的匹配行为。

## 6. 扩展性

一旦我们有了基本的 NFA 构建块，就可以轻松地扩展它们以支持更复杂的正则表达式功能，如字符类、预定义字符集、向前和向后的断言等。

## 7. 内存管理

在构建 NFA 时，我们需要频繁地创建和删除状态和转换。因此，正确的内存管理至关重要。我们可以使用简单的内存池来分配和释放对象，以确保我们不会泄露任何内存。

当我们不再需要一个 NFA 时，我们应该释放其所有相关的状态和转换。最简单的方法是从起始状态开始，递归地遍历整个 NFA，并释放每个访问的状态和转换。

## (三) Thompson 算法的详细思路

Thompson 算法，也称为 Thompson 构造法，是一个将正则表达式转换为其等价的非确定性有限自动机 (NFA) 的算法。其核心思想是为正则表达式的每一个构造（例如字母、连接、选择和闭包）构建一个小的 NFA 片段，然后通过组合这些小的片段来形成描述整个正则表达式的 NFA。

### 1. 基本 NFA 片段的构造

每一个正则表达式的基本元素，如字符或，都可以被转换为一个非常简单的 NFA。例如，为字符 ‘a’ 构建的 NFA 只包含两个状态和一个转换，这个转换在 ‘a’ 上从起始状态指向终止状态。

## 2. 连接操作

连接操作的基本思想是将两个 NFA 片段首尾相接。考虑两个 NFA, NFA1 和 NFA2。为了构建描述正则表达式  $e_1e_2$  的 NFA, 我们可以简单地将 NFA1 的终止状态与 NFA2 的起始状态通过一个转换连接起来。

## 3. 选择操作

选择操作需要为两个 NFA 片段构建一个新的起始状态和一个新的终止状态。考虑两个 NFA, NFA1 和 NFA2。新的起始状态将通过转换与 NFA1 和 NFA2 的起始状态相连, 而 NFA1 和 NFA2 的终止状态都将通过转换与新的终止状态相连。这样, 新构建的 NFA 可以选择“走” NFA1 或 NFA2。

## 4. 闭包操作

为了构建描述正则表达式  $e^*$  的 NFA, 我们需要为给定的 NFA 片段添加两个新状态: 新的起始状态和新的终止状态。新的起始状态通过转换连接到给定 NFA 的起始状态和新的终止状态。此外, 给定 NFA 的终止状态通过转换连接到其起始状态和新的终止状态。

## 5. 算法过程

- 初始化一个空堆栈。遍历正则表达式的每一个字符。
- 对于正则表达式中的每一个基本元素, 如字符或, 创建一个新的 NFA 片段, 并将其压入堆栈。
- 对于正则表达式中的每一个操作符 (如连接、选择或闭包), 从堆栈中弹出所需数量的 NFA 片段, 应用操作, 然后将结果压入堆栈。
- 遍历正则表达式结束后, 堆栈顶部应该只有一个 NFA 片段, 这个片段描述了整个正则表达式。

## 6. 算法的优缺点

Thompson 算法的主要优点是它总是生成一个大小最小的 NFA, 这对于后续的处理, 如 NFA 到 DFA 的转换, 是非常有利的。

然而, Thompson 算法生成的 NFA 是非确定性的, 这意味着对于某些输入字符, 可能有多于一个可能的下一个状态。这使得 NFA 的运行需要更多的计算, 因为必须跟踪所有可能的当前状态。为了解决这个问题, 可以使用子集构造法将 NFA 转换为等价的确定性有限自动机 (DFA)。

## (四) 编程实现

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct State {
5      int isFinal; // 是否为终止状态: 1表示是, 0表示否
6      struct Transition* trans; // Pointing to transitions list
7  } State;
8
```

```
9 typedef struct Transition {
10     State* from;    // 起始状态
11     State* to;      // 终止状态
12     char symbol;    // 在这两个状态之间转移时要读取的字符
13     struct Transition* next; // 链接到下一个转换的指针
14 } Transition;
15
16 typedef struct NFA {
17     State* start;
18     State* end;
19     Transition* transitions; // 转换的列表
20 } NFA;
21
22 void addTransition(Transition** transitions, State* from, State* to, char
    symbol) {
23     Transition* t = malloc(sizeof(Transition));
24     t->from = from;
25     t->to = to;
26     t->symbol = symbol;
27     t->next = *transitions;
28     *transitions = t;
29 }
30
31 NFA createNFAWithSymbol(char symbol) {
32     NFA newNFA;
33     State* start = malloc(sizeof(State));
34     State* end = malloc(sizeof(State));
35
36     start->isFinal = 0;
37     start->trans = NULL;
38     end->isFinal = 1;
39     end->trans = NULL;
40     newNFA.start = start;
41     newNFA.end = end;
42     addTransition(&(newNFA.transitions), start, end, symbol);
43     return newNFA;
44 }
45
46 NFA unionOp(NFA nfa1, NFA nfa2) {
47     NFA newNFA;
48     State* newStart = malloc(sizeof(State));
49     State* newEnd = malloc(sizeof(State));
50
51     newStart->isFinal = 0;
52     newStart->trans = NULL;
53     newEnd->isFinal = 1;
54     newEnd->trans = NULL;
55     newNFA.start = newStart;
```

```
56     newNFA.end = newEnd;
57     addTransition(&(newNFA.transitions), newStart, nfa1.start, 0);
58     addTransition(&(newNFA.transitions), newStart, nfa2.start, 0);
59     addTransition(&(newNFA.transitions), nfa1.end, newEnd, 0);
60     addTransition(&(newNFA.transitions), nfa2.end, newEnd, 0);
61
62
63     return newNFA;
64 }
65
66
67 NFA thompsonConstruct(char* regex) {
68     return expression(&regex);
69 }
70 NFA concatOp(NFA nfa1, NFA nfa2) {
71     addTransition(&(nfa1.transitions), nfa1.end, nfa2.start, 0);
72     nfa1.end = nfa2.end;
73     return nfa1;
74 }
75 NFA closureOp(NFA nfa) {
76     NFA newNFA;
77     State* newStart = malloc(sizeof(State));
78     State* newEnd = malloc(sizeof(State));
79
80     newStart->isFinal = 0;
81     newStart->trans = NULL;
82     newEnd->isFinal = 1;
83     newEnd->trans = NULL;
84
85     addTransition(&(newNFA.transitions), newStart, nfa.start, 0);
86     addTransition(&(newNFA.transitions), newStart, newEnd, 0);
87     addTransition(&(newNFA.transitions), nfa.end, nfa.start, 0);
88     addTransition(&(newNFA.transitions), nfa.end, newEnd, 0);
89
90     newNFA.start = newStart;
91     newNFA.end = newEnd;
92     return newNFA;
93 }
94
95 int main() {
96     char* regex = "a|b*";
97     NFA result = thompsonConstruct(regex);
98     printf("Constructed NFA for regex: %s\n", regex);
99
100     return 0;
101 }
```

### 函数的说明



### 1. 添加转换:

addTransition 函数用于在转换列表的前面添加一个新的转换。

### 2. 构造 NFA

createNFAWithSymbol 函数接受一个符号, 并为它创建一个简单的 NFA。这个 NFA 从一个非终止状态开始, 读取该符号后转到终止状态。

### 3. NFA 的操作

- unionOp: 实现 NFA 的并集操作。该操作将两个给定的 NFA nfa1 和 nfa2 合并为一个新的 NFA, 其中新的起始状态可以转换到 nfa1 和 nfa2 的起始状态, 而 nfa1 和 nfa2 的结束状态都可以转换到新的结束状态。
- concatOp: 实现 NFA 的串联操作。将 nfa1 的结束状态与 nfa2 的起始状态连接, 并返回串联后的 NFA。
- 实现 NFA 的闭包操作。闭包操作创建一个新的 NFA, 其中新的起始状态可以转换到给定 NFA 的起始状态或新的结束状态。同时, 给定 NFA 的结束状态可以转换回其起始状态或新的结束状态。

### 4. Thompson 算法的构造

在后续的 expression 当中展现

### 5. 主函数:

为字符串 "a|b\*" 创建一个 NFA, 并输出消息表示已为给定的正则表达式构建了 NFA。

## 三、 NFA 到 DFA 的转换

### (一) 从 NFA 到 DFA 的子集构造法的算法思路

非确定性有限自动机 (NFA) 与确定性有限自动机 (DFA) 都是有限自动机的两种形式。NFA 在其状态转换中允许存在模糊性或非确定性, 而 DFA 在每个状态和输入符号组合下都只有一个明确的后继状态。为了将正则表达式转换为 DFA, 我们首先将其转换为 NFA (使用 Thompson 构造法或其他方法), 然后使用子集构造法将 NFA 转换为 DFA。

#### 1. 计算 e-closure(T) 的算法

算法说明:

- 创建一个空栈 stack、空集合 visited 和空集合 result, 用于保存计算结果。
- 将起始状态集合 T 中的每个状态依次入栈, 并标记为已访问。
- 将入栈的状态逐个添加到结果集合 result 中。
- 循环遍历栈中的状态, 直到栈为空。
- 对于当前弹出的状态 currentState, 获取其 转移集合 epsilonTransitions。

- 遍历 转移集中的每个状态，如果该状态尚未访问过，则将其入栈、标记为已访问，并添加到结果集中。
- 当栈为空时，算法结束，返回结果集合 result 作为  $\epsilon\text{-closure}(T)$  的计算结果。

代码示例如下：

```
1  function epsilonClosure(T)
2  stack = empty stack
3  visited = empty set
4  result = empty set
5
6  // 将T中的状态入栈
7  for each state in T
8      push state to stack
9      add state to visited
10     add state to result
11
12 // 遍历栈中的状态
13 while stack is not empty
14     currentState = pop stack
15
16     // 获取当前状态的 转移集合
17     epsilonTransitions = getEpsilonTransitions(currentState)
18
19     // 遍历 转移集中的每个状态
20     for each state in epsilonTransitions
21         if state is not in visited
22             push state to stack
23             add state to visited
24             add state to result
25
26 return result
```

## 2. NFA 到 DFA 的转换

使用子集构造法将 NFA 转换为 DFA。这种方法的基本思想是为 NFA 中的每个状态集合定义一个 DFA 状态。

- 创建一个空的 DFA 'dfa'
- 使用 epsilonClosure 函数计算 NFA 初始状态的 闭包，并将其设为 DFA 的初始状态。
- 将初始状态加入到未标记状态列表 unmarkedStates 中。
- 循环遍历未标记状态列表，直到列表为空。
- 从未标记状态列表中弹出一个状态 currentState。
- 对于 DFA 的字母表中的每个符号，执行以下步骤：1. 使用 move 函数计算 currentState 在当前符号下的状态转移。2. 使用 epsilonClosure 函数计算新状态 newState 的 闭包。3. 如

果 newState 不在 DFA 的状态集合中，将其添加到 DFA 的状态集合和未标记状态列表中。

4. 将从 currentState 到 newState 的符号转移添加到 DFA 的转移表中。

- 使用某种方法计算 DFA 的最终状态集合。
- 返回转换后的 DFA。

```

1  function nfaToDfa(nfa)
2  dfa = empty DFA
3  initialState = epsilonClosure(nfa.initialState)
4  dfa.addState(initialState)
5
6  unmarkedStates = [initialState]
7
8  while unmarkedStates is not empty
9      currentState = unmarkedStates.pop()
10     for each symbol in alphabet
11         newState = epsilonClosure(move(currentState, symbol))
12         if newState not in dfa.states
13             dfa.addState(newState)
14             unmarkedStates.append(newState)
15             dfa.addTransition(currentState, symbol, newState)
16
17     dfa.finalStates = calculateFinalStates(dfa)
18
19 return dfa

```

## (二) 子集构造法的算法实现

其中我们使用了一些数据结构，包括：使用链表表示 NFA 和 DFA。使用链表存储状态集合以及转移关系。使用堆栈计算 e-closure。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // NFA的状态结构
5  typedef struct NFAStruct {
6      int id; // 状态ID
7      int isFinal; // 是否为终态
8      struct Transition* transitions; // 转移关系链表
9      struct NFAStruct* next; // 用于状态集合的链表
10 } NFAStruct;
11
12 // 转移关系结构
13 typedef struct Transition {
14     NFAStruct* to; // 目标状态
15     char symbol; // 转移符号, 0代表epsilon
16     struct Transition* next; // 下一个转移
17 } Transition;

```

```
18
19 // DFA的状态结构
20 typedef struct DFAState {
21     NFAStruct* nfaStates; // NFA状态集合
22     int marked; // 是否已标记
23     struct DFAState* next; // 下一个DFA状态
24     struct DFATransition* transitions; // DFA转移关系
25 } DFAState;
26
27 // DFA的转移关系结构
28 typedef struct DFATransition {
29     DFAState* to; // 目标状态
30     char symbol; // 转移符号
31     struct DFATransition* next; // 下一个转移
32 } DFATransition;
33
34 // 堆栈结构，用于计算e-closure
35 typedef struct Stack {
36     NFAStruct* state;
37     struct Stack* next;
38 } Stack;
39
40 // 堆栈操作：推入
41 void push(Stack** s, NFAStruct* state) {
42     Stack* newNode = malloc(sizeof(Stack));
43     newNode->state = state;
44     newNode->next = *s;
45     *s = newNode;
46 }
47
48 // 堆栈操作：弹出
49 NFAStruct* pop(Stack** s) {
50     if (!*s) return NULL;
51     NFAStruct* ret = (*s)->state;
52     Stack* temp = *s;
53     *s = (*s)->next;
54     free(temp);
55     return ret;
56 }
57
58 // 计算e-closure
59 NFAStruct* eClosure(NFAStruct* T) {
60     Stack* s = NULL;
61     NFAStruct* eClosureResult = NULL;
62     NFAStruct* current = T;
63
64     // 将T中的所有状态推入堆栈
65     while (current) {
```

```

66     push(&s, current);
67     current = current->next;
68 }
69
70 while (s) {
71     NFAStruct* t = pop(&s);
72     // 如果t不在eClosureResult中, 则添加
73     if (!inStateList(eClosureResult, t)) {
74         addToStateList(&eClosureResult, t);
75         Transition* trans = t->transitions;
76         while (trans) {
77             if (trans->symbol == 0 && !inStateList(eClosureResult, trans
78                 ->to)) { // epsilon 转移
79                 push(&s, trans->to);
80             }
81             trans = trans->next;
82         }
83     }
84     return eClosureResult;
85 }
86
87 // NFA到DFA的子集构造法
88 DFAState* subsetConstruction(NFAStruct* start) {
89     // 初始化
90     DFAState* Dstates = NULL;
91     addToDFAStateList(&Dstates, eClosure(start));
92
93     DFAState* currentDFA = Dstates;
94     while (currentDFA && !currentDFA->marked) {
95         currentDFA->marked = 1;
96
97         // 对于每个输入符号a
98         for (char a = 'a'; a <= 'z'; a++) { // 假设符号集为a-z
99             NFAStruct* U = move(currentDFA->nfaStates, a);
100             U = eClosure(U);
101
102             if (!inDFAStateList(Dstates, U)) {
103                 addToDFAStateList(&Dstates, U);
104             }
105             addDFATransition(currentDFA, U, a);
106         }
107         currentDFA = currentDFA->next;
108     }
109     return Dstates;
110 }
111
112 // 辅助函数部分

```

```
113 // 添加状态到状态链表
114 void addToStateList(NFAState** list, NFAState* state) {
115     NFAState* newNode = malloc(sizeof(NFAState));
116     *newNode = *state; // 浅复制
117     newNode->next = *list;
118     *list = newNode;
119 }
120
121 // 检查状态是否在状态链表中
122 int inStateList(NFAState* list, NFAState* state) {
123     while (list) {
124         if (list->id == state->id) return 1;
125         list = list->next;
126     }
127     return 0;
128 }
129
130 // 计算在给定符号下的转移状态
131 NFAState* move(NFAState* T, char symbol) {
132     NFAState* result = NULL;
133     while (T) {
134         Transition* trans = T->transitions;
135         while (trans) {
136             if (trans->symbol == symbol && !inStateList(result, trans->to)) {
137                 addToStateList(&result, trans->to);
138             }
139             trans = trans->next;
140         }
141         T = T->next;
142     }
143     return result;
144 }
145
146 // DFA状态管理函数
147 // DFA输出
148 void printDFA(DFAState* start) {
149     if (!start) return;
150
151     printf("DFA States and their transitions:\n");
152     DFAState* curr = start;
153     while (curr) {
154         NFAState* nfaState = curr->nfaStates;
155         printf("DFA State derived from NFA states: ");
156         while (nfaState) {
157             printf("%d ", nfaState->id);
158             nfaState = nfaState->next;
159         }
160     }
```

```
161     printf("\n");
162     DFATransition* trans = curr->transitions;
163     while (trans) {
164         NFAStruct* nfaToState = trans->to->nfaStates;
165         printf(" On symbol %c to DFA state from NFA states: ", trans->
            symbol);
166         while (nfaToState) {
167             printf("%d ", nfaToState->id);
168             nfaToState = nfaToState->next;
169         }
170         printf("\n");
171         trans = trans->next;
172     }
173     curr = curr->next;
174 }
175 }
176 // 检查DFA状态是否在状态链表中
177 int inDFAStateList(DFAState* list, NFAStruct* nfaStates) {
178     while (list) {
179         NFAStruct* stateList1 = list->nfaStates;
180         NFAStruct* stateList2 = nfaStates;
181         while (stateList1 && stateList2 && stateList1->id == stateList2->id)
182             {
183                 stateList1 = stateList1->next;
184                 stateList2 = stateList2->next;
185             }
186         if (!stateList1 && !stateList2) return 1; // 找到相同的DFA状态
187         list = list->next;
188     }
189     return 0;
190 }
191 // 添加DFA状态到状态链表
192 void addToDFAStateList(DFAState** list, NFAStruct* nfaStates) {
193     DFAState* newDFAState = malloc(sizeof(DFAState));
194     newDFAState->nfaStates = nfaStates;
195     newDFAState->marked = 0;
196     newDFAState->transitions = NULL;
197     newDFAState->next = *list;
198     *list = newDFAState;
199 }
200 // 添加DFA转移
201 void addDFATransition(DFAState* from, DFAState* to, char symbol) {
202     DFATransition* newTrans = malloc(sizeof(DFATransition));
203     newTrans->to = to;
204     newTrans->symbol = symbol;
205     newTrans->next = from->transitions;
206     from->transitions = newTrans;
207 }
```

```

207
208
209
210 int main() {
211     // 创建一个简单的NFA: a|b
212     NFAStruct* start = malloc(sizeof(NFAStruct));
213     NFAStruct* aState = malloc(sizeof(NFAStruct));
214     NFAStruct* bState = malloc(sizeof(NFAStruct));
215     NFAStruct* finalState = malloc(sizeof(NFAStruct));
216
217     start->id = 0; aState->id = 1; bState->id = 2; finalState->id = 3;
218     start->isFinal = 0; aState->isFinal = 0; bState->isFinal = 0; finalState
        ->isFinal = 1;
219     start->transitions = NULL; aState->transitions = NULL; bState->
        transitions = NULL; finalState->transitions = NULL;
220     start->next = aState; aState->next = bState; bState->next = finalState;
        finalState->next = NULL;
221
222     // 添加NFA的转移
223     addTransition(&start->transitions, start, aState, 'a');
224     addTransition(&start->transitions, start, bState, 'b');
225     addTransition(&aState->transitions, aState, finalState, 'e'); // epsilon
        transition
226     addTransition(&bState->transitions, bState, finalState, 'e'); // epsilon
        transition
227
228     // 调用subsetConstruction进行NFA到DFA的转换
229     DFAStruct* dfaStart = subsetConstruction(start);
230
231     // 打印DFA
232     printDFA(dfaStart);
233
234     return 0;
235 }

```

### 1. 使用的数据结构

- NFAStruct: NFA 的状态结构, 其中包含了状态 ID、是否为终态的标记、该状态的转移关系链表, 以及一个指向下一个状态的指针 (用于创建一个状态列表)。
- Transition: 表示从一个 NFA 状态到另一个 NFA 状态的转移关系。这里的转移可以是基于一个字符或者是 epsilon(用 0 表示)。
- DFAStruct: DFA 的状态结构。一个 DFA 的状态是由多个 NFA 的状态组成的。这里使用 nfaStates 来保存这些 NFA 的状态, marked 标记来检查该状态是否已经处理过, transitions 表示 DFA 的转移关系。
- DFATransition: 表示从一个 DFA 状态到另一个 DFA 状态的转移关系。



- Stack: 辅助数据结构, 用于计算 epsilon 闭包 (e-closure)。

## 2. 功能

- eClosure: 计算给定 NFA 状态集合的 epsilon 闭包。这是通过不断跟随 epsilon 转移来完成的, 直到没有更多的 epsilon 转移为止。
- subsetConstruction: 核心功能, 将 NFA 转换为 DFA。这通过迭代方式完成, 对每一个新发现的 DFA 状态, 基于所有可能的输入符号计算其转移, 然后检查是否已经有对应的 DFA 状态存在。
- addToStateList & inStateList: 用于管理 NFA 状态的链表。
- addToDFAStateList & inDFAStateList: 用于管理 DFA 状态的链表。
- addDFATransition: 为 DFA 添加一个新的转移。
- move: 给定一个 NFA 状态集和一个符号, 计算在该符号下的所有可能的转移。
- printDFA: 打印结果的 DFA。

# 四、 DFA 最小化

## (一) DFA 最小化的实现思路

实现 DFA 最小化, 使用了利用 DFA 的一个初始状态逐步的进行划分的方法, 算法的核心逻辑如下:

- 将 DFA 的状态根据最终状态分割成初始分区集合 partitions, 其中每个分区包含一个状态。
- 初始化 createNewPartition 为 false, 用于标记是否需要创建新的分区。
- 进入循环, 直到 createNewPartition 为 false:  
将 createNewPartition 设为 false。创建一个空的列表 newPartitions, 用于存储新的分区。  
遍历每个分区 partition, 如果 partition 的大小大于 1, 执行以下步骤:  
根据分区中的状态的转移关系进行分组, 即根据相同的转移目标状态将状态进行分组。  
对于每个分组 group:  
如果 group 的大小大于 1, 将其添加到 newPartitions 中。设置 createNewPartition 为 true, 表示需要创建新的分区。
- 如果 createNewPartition 为 true, 则更新 partitions 为 newPartitions。
- 返回根据最终分区创建的新的 DFA。

```
1 function minimizeDFA(dfa)
2   partitions = splitByFinalStates(dfa)
3   createNewPartition = false
4
5   while createNewPartition is true
6     createNewPartition = false
```

```

7         newPartitions = empty list
8
9         for each partition in partitions
10             if partition.size > 1
11                 groupByTransitions(partition)
12
13         for each group in partition.groups
14             if group.size > 1
15                 newPartitions.append(group)
16                 createNewPartition = true
17
18         if createNewPartition is true
19             partitions = newPartitions
20
21     return createNewDFA(partitions)

```

## (二) 代码实现

```

1     #include <iostream>
2     #include <vector>
3     #include <map>
4     #include <set>
5     #include <algorithm>
6
7     using namespace std;
8
9     class DFA {
10    public:
11         set<int> states;           // All states.
12         map<pair<int, char>, int> transition; // Transition function.
13         int startState;
14         set<int> acceptStates;
15
16         DFA(set<int> states, map<pair<int, char>, int> transition, int startState
17             , set<int> acceptStates)
18             : states(states), transition(transition), startState(startState),
19               acceptStates(acceptStates) {}
20
21     };
22
23     set<set<int>> split(const set<int>& group, const DFA& dfa, const set<set<int>
24         >>& partitions, const vector<char>& alphabet) {
25         map<int, int> signature;
26
27         for (int state : group) {
28             int sig = 0;
29             for (char a : alphabet) {

```

```

27         int to = dfa.transition.at({ state, a });
28         int idx = 0;
29         for (const set<int>& part : partitions) {
30             if (part.find(to) != part.end()) {
31                 sig |= (1 << idx);
32                 break;
33             }
34             idx++;
35         }
36     }
37     signature[state] = sig;
38 }
39
40 set<set<int>> newGroups;
41 map<int, set<int>> groupBySig;
42 for (auto [state, sig] : signature) {
43     groupBySig[sig].insert(state);
44 }
45 for (auto [_, g] : groupBySig) {
46     newGroups.insert(g);
47 }
48
49 return newGroups;
50 }
51
52 DFA minimizeDFA(const DFA& dfa, const vector<char>& alphabet) {
53     set<set<int>> partitions;
54     partitions.insert(dfa.acceptStates);
55     set<int> nonAcceptStates;
56
57     // Initialize nonAcceptStates as states - acceptStates
58     std::set_difference(dfa.states.begin(), dfa.states.end(), dfa.
        acceptStates.begin(), dfa.acceptStates.end(), std::inserter(
        nonAcceptStates, nonAcceptStates.begin()));
59     partitions.insert(nonAcceptStates);
60
61     while (true) {
62         set<set<int>> newPartitions;
63         for (const set<int>& group : partitions) {
64             set<set<int>> splitted = split(group, dfa, partitions, alphabet);
65             newPartitions.insert(splitted.begin(), splitted.end());
66         }
67         if (newPartitions == partitions) break;
68         partitions = newPartitions;
69     }
70
71     map<int, int> oldStateToNewState;
72     int newState = 0;

```


```

73     for (const set<int>& group : partitions) {
74         for (int oldState : group) {
75             oldStateToNewState[oldState] = newState;
76         }
77         newState++;
78     }
79
80     map<pair<int, char>, int> newTransition;
81     for (auto [key, value] : dfa.transition) {
82         int oldSource = key.first;
83         char symbol = key.second;
84         int oldTarget = value;
85         newTransition[{oldStateToNewState[oldSource], symbol}] =
            oldStateToNewState[oldTarget];
86     }
87
88     set<int> newAcceptStates;
89     for (int oldAcceptState : dfa.acceptStates) {
90         newAcceptStates.insert(oldStateToNewState[oldAcceptState]);
91     }
92
93     return DFA({ 0, 1, 2, 3 }, newTransition, oldStateToNewState[dfa.
        startState], newAcceptStates);
94 }
95
96 int main() {
97     set<int> states = { 0, 1, 2, 3 };
98     map<pair<int, char>, int> transition = {
99         {{0, 'a'}, 1},
100         {{0, 'b'}, 2},
101         {{1, 'a'}, 3},
102         {{1, 'b'}, 2},
103         {{2, 'a'}, 1},
104         {{2, 'b'}, 3},
105         {{3, 'a'}, 3},
106         {{3, 'b'}, 3}
107     };
108     int startState = 0;
109     set<int> acceptStates = { 3 };
110
111     DFA dfa(states, transition, startState, acceptStates);
112
113     DFA minimizedDfa = minimizeDFA(dfa, { 'a', 'b' });
114
115     cout << "States: ";
116     for (int state : minimizedDfa.states) {
117         cout << state << " ";
118     }

```

```
119     cout << "\nAccept States: ";
120     for (int acceptState : minimizedDfa.acceptStates) {
121         cout << acceptState << " ";
122     }
123
124     return 0;
125 }
```

代码的输出样例如下：



```
D:\dasanshang\bianyiyuan\c X + v
States: 0 1 2 3
Accept States: 2
-----
Process exited after 0.5789 seconds with return value 0
请按任意键继续. . .
```

图 1: dfa 最小化

### (三) 所用到的结构

#### 1. DFA 结构

我们定义了一个 DFA 类来表示一个确定有限自动机。这个 DFA 类包含以下成员：

- states：表示 DFA 中所有状态的集合。
- transition：一个映射，代表 DFA 的转移函数。它将每个 (状态, 符号) 对映射到另一个状态。
- startState：表示 DFA 的起始状态。
- acceptStates：表示 DFA 中所有的接受状态的集合。

#### 2. split 函数

将给定的状态组基于转移函数进行分裂。这是最小化过程的核心，其中我们尝试找到是否存在两个状态在所有输入符号上都有相同的行为。如果有，它们将被视为等价的，否则它们将被分开。

#### 3. minimize 函数

是程序的主体函数，分为以下步骤：

- 初始化：我们首先将所有状态分为两个组：接受状态和非接受状态。这是最小化算法的起始划分，因为一个接受状态肯定不能等价于一个非接受状态。

- 循环划分：我们继续对当前的状态组进行分裂，直到不能再分裂为止。具体来说，我们检查每个状态组中的状态是否对所有输入符号都有相同的转移。如果不是，我们将这些状态分裂到不同的组中。
- 构造新的 DFA：一旦我们获得了最终的状态划分，我们就可以使用这些分组中的任意一个状态作为代表来构造新的 DFA。新 DFA 中的转移是根据旧 DFA 中的转移和最终的状态划分来定义的。
- 删除死状态和不可达状态：最后，我们删除所有死状态（即那些对所有输入都不进行任何转移的状态）和所有从初始状态开始不可达的状态。

## 五、 正则表达式转换到 DFA

### （一） 直接进行正则表达式到 DFA 的转换

这种方法涉及到正则表达式的句法和语义分析，以及使用查找表来直接生成 DFA 的状态和转移。简单的步骤如下：

- 将中缀正则表达式转换为后缀形式，得到 postfixRegex。
- 初始化一个空栈 stack。
- 对于 postfixRegex 中的每个字符：如果字符是操作数，即普通字符或字符集等，创建一个对应的片段 fragment，并将其压入栈中。  
如果是连接操作符，从栈中弹出两个片段 fragment2 和 fragment1，将它们进行连接，并将连接后的片段 concatenatedFragment 压入栈中。  
如果是并操作符，从栈中弹出两个片段 fragment2 和 fragment1，将它们进行并操作，并将合并后的片段 unionedFragment 压入栈中。  
如果是闭包操作符，从栈中弹出一个片段 fragment，将其进行闭包操作，并将闭包后的片段 kleenedFragment 压入栈中。
- 从栈中弹出最终的片段 finalFragment。
- 根据最终片段 finalFragment 创建一个 DFA。
- 返回生成的 DFA。

程序的伪代码如下：

```
1 function regexToDFA(regex)
2   postfixRegex = convertToPostfix(regex)
3   stack = empty stack
4
5   for each character in postfixRegex
6     if character is an operand
7       createFragment(character)
8       stack.push(fragment)
9     else if character is an operator
10      if character is concatenation operator
11        fragment2 = stack.pop()
```

```
12         fragment1 = stack.pop()
13         concatenateFragments(fragment1, fragment2)
14         stack.push(concatenatedFragment)
15     else if character is union operator
16         fragment2 = stack.pop()
17         fragment1 = stack.pop()
18         unionFragments(fragment1, fragment2)
19         stack.push(unionedFragment)
20     else if character is Kleene closure operator
21         fragment = stack.pop()
22         kleeneClosure(fragment)
23         stack.push(kleenedFragment)
24
25     finalFragment = stack.pop()
26     createDFA(finalFragment)
27
28     return DFA
```

算法中, 需要实现一些辅助函数, 例如将中缀正则表达式转换为后缀形式的函数 `convertToPostfix`, 以及根据操作符进行片段操作的函数, 如连接操作的 `concatenateFragments`, 并操作的 `unionFragments`, 闭包操作的 `kleeneClosure` 等。具体的实现可能涉及到对状态、转移、接受状态等的创建和操作。

## (二) 解析组合子的构建

解析组合子是一种函数式编程概念, 允许我们组合小的解析函数来构建更复杂的解析器。使用解析组合子, 我们可以直接构建一个从输入字符串到输出标记的解析器简单的操作步骤如下:

- 定义基本解析器: 为每种基本模式(如字符、字符串、数字等)定义一个解析器。
- 组合解析器: 定义函数来组合这些基本解析器, 以处理更复杂的模式和结构。
- 递归构造: 为复杂的正则表达式模式定义递归解析器。
- 运行解析器: 将组合好的解析器应用于输入字符串, 直接生成输出标记。

## (三) 代码实现

```
1 #include <iostream>
2 #include <string>
3 #include <stack>
4 #include <map>
5 #include <set>
6 #include <vector>
7
8 using namespace std;
9
10 struct TreeNode {
11     char value;
12     TreeNode* left;
```

```
13     TreeNode* right;
14     int position;
15     set<int> firstpos, lastpos;
16     bool nullable;
17 };
18
19 class RegexParser {
20 public:
21     RegexParser(const string& regex) : regex(regex), pos(0) {}
22
23     TreeNode* parse() {
24         auto node = expression();
25         if (pos != regex.size()) {
26             throw runtime_error("Unexpected character");
27         }
28         return node;
29     }
30
31 private:
32     string regex;
33     int pos;
34     TreeNode* term() {
35         auto node = factor();
36
37         while (pos < regex.size() && (regex[pos] != ')') && regex[pos] != '|')
38             {
39                 auto rightNode = factor();
40                 auto concatNode = new TreeNode{ '.', node, rightNode }; // '.'表示连接操作
41                 node = concatNode;
42             }
43         return node;
44     }
45     TreeNode* expression() {
46         auto node = term();
47
48         while (pos < regex.size() && regex[pos] == '|') {
49             ++pos;
50             auto rightNode = term();
51             auto unionNode = new TreeNode{ '|', node, rightNode };
52             node = unionNode;
53         }
54
55         return node;
56     }
57     TreeNode* primary() {
58         if (pos >= regex.size()) {
59             throw runtime_error("Unexpected end of regex");
60         }
61     }
```



```
59     }
60
61     if (regex[pos] == '(') {
62         ++pos;
63         auto node = expression();
64         if (regex[pos] != ')') {
65             throw runtime_error("Expected ')'");
66         }
67         ++pos;
68         return node;
69     }
70     else {
71         auto leafNode = new TreeNode{ regex[pos] };
72         ++pos;
73         return leafNode;
74     }
75 }
76
77 TreeNode* factor() {
78     auto node = primary();
79
80     while (pos < regex.size() && regex[pos] == '*') {
81         ++pos;
82         auto starNode = new TreeNode{ '*', node };
83         node = starNode;
84     }
85
86     return node;
87 }
88
89 };
90
91 class DFABuilder {
92     int nextPosition = 1;
93     map<int, char> positionToChar;
94     map<int, set<int>> followpos;
95
96 public:
97     // 根据语法树计算 nullable、firstpos、lastpos 和 followpos
98     void calculate(TreeNode* node) {
99         if (!node) return;
100
101         if (node->value == '|') {
102             calculate(node->left);
103             calculate(node->right);
104             node->nullable = node->left->nullable || node->right->nullable;
105             node->firstpos.insert(node->left->firstpos.begin(), node->left->firstpos.end());
106             node->firstpos.insert(node->right->firstpos.begin(), node->right->firstpos.end());
107         }
108     }
109 }
```

```
105         node->lastpos.insert(node->left->lastpos.begin(), node->left->
106             lastpos.end());
107     node->lastpos.insert(node->right->lastpos.begin(), node->right->
108         lastpos.end());
109 }
110 else if (node->value == '*') {
111     calculate(node->left);
112     node->nullable = true;
113     node->firstpos = node->left->firstpos;
114     node->lastpos = node->left->lastpos;
115     for (int position : node->left->lastpos) {
116         followpos[position].insert(node->left->firstpos.begin(), node
117             ->left->firstpos.end());
118     }
119 }
120 else if (node->value == '.') {
121     calculate(node->left);
122     calculate(node->right);
123     node->nullable = node->left->nullable && node->right->nullable;
124     if (node->left->nullable) {
125         node->firstpos.insert(node->left->firstpos.begin(), node->
126             left->firstpos.end());
127         node->firstpos.insert(node->right->firstpos.begin(), node->
128             right->firstpos.end());
129     }
130     else {
131         node->firstpos = node->left->firstpos;
132     }
133     if (node->right->nullable) {
134         node->lastpos.insert(node->left->lastpos.begin(), node->left
135             ->lastpos.end());
136         node->lastpos.insert(node->right->lastpos.begin(), node->
137             right->lastpos.end());
138     }
139     else {
140         node->lastpos = node->right->lastpos;
141     }
142     for (int position : node->left->lastpos) {
143         followpos[position].insert(node->right->firstpos.begin(),
144             node->right->firstpos.end());
145     }
146 }
147 else { // 叶子节点
148     node->nullable = false;
149     node->position = nextPosition++;
150     positionToChar[node->position] = node->value;
151     node->firstpos.insert(node->position);
```

```
145         node->lastpos.insert(node->position);
146     }
147 }
148 };
149
150 int main() {
151     string regex;
152     cout << "Enter regex: ";
153     cin >> regex;
154     RegexParser parser(regex);
155     TreeNode* tree = parser.parse();
156     DFABuilder dfaBuilder;
157     dfaBuilder.calculate(tree);
158     catch (const runtime_error& e) {
159         cerr << "Error: " << e.what() << endl;
160     }
161
162     return 0;
163 }
```

#### (四) 算法的效率分析

##### 1. 算法的准确性

所有的转换方法都确保了结果 DFA 与原始正则表达式具有相同的匹配语义。说明我们的编程计算结果是正确的可靠有效的。

##### 2. Thompson 算法的效率

Thompson 算法的时间复杂度为  $O(n)$ ，其中  $n$  是正则表达式的长度。Thompson 算法的主要特点是在正则表达式的每个字符之间构建片段，然后通过组合这些片段来构造最终的确定有限状态机 (DFA)。该算法的关键优势在于它的局部性质，每个字符的处理仅与前一个字符相关，因此在构造过程中无需对整个正则表达式进行全局分析。由于该算法仅遍历正则表达式一次，并且只需要对每个字符进行一些局部操作（例如创建片段、连接、并等），因此它的运行时间与正则表达式的长度成线性关系，即  $O(n)$ 。

##### 3. 子集构造法的效率

子集构造法是一种将非确定有限状态自动机 (NFA) 转换为确定有限状态自动机 (DFA) 的常见方法之一。它的效率在处理简单的正则表达式时较高，但在处理复杂的正则表达式时可能会出现状态爆炸的问题，导致算法的效率下降。具体而言，子集构造法的时间复杂度取决于两个因素：

- NFA 的状态数目：对于给定的 NFA，子集构造法需要遍历所有可能的状态组合来构建相应的 DFA 状态。因此，NFA 中的状态数目越多，需要构造的 DFA 状态数目就越多，耗时也会相应增加。
- NFA 的转移函数：子集构造法需要根据 NFA 的转移函数来计算 DFA 的转移函数。如果 NFA 的转移函数较为复杂或包含  $\epsilon$ -转移（空转移），这可能导致生成的 DFA 状态数目增

加，并增加算法的时间复杂度。

在处理简单的正则表达式时，子集构造法的时间复杂度通常为  $O(2^n)$ ，其中  $n$  是 NFA 中状态的数量。这是因为对于每个 DFA 状态，存在  $2^n$  个可能的子集来表示从 NFA 状态集中的哪些状态可以到达该 DFA 状态。但是，对于复杂的正则表达式，状态爆炸的问题可能会导致指数级的状态增长，使得子集构造法变得低效。

为了解决状态爆炸问题，可以采用一些优化技术，如状态合并和最小化 DFA 的方法。这些技术可以减少生成的 DFA 状态数目并提高算法的效率。

总的来说，子集构造法在处理简单的正则表达式时具有较高的效率，但在处理复杂的正则表达式时可能会面临状态爆炸的问题。因此，在实际应用中，需要根据正则表达式的复杂度和要求权衡使用子集构造法还是其他算法。

#### 4. DFA 最小化的效率

DFA 最小化是一种将确定有限状态自动机 (DFA) 转换为最简形式的过程，即消除等价状态，以减少状态的数量并提高效率。DFA 最小化的效率取决于以下几个因素：

- DFA 的状态数目：对于给定的 DFA，状态数目越多，最小化的过程就需要处理更多的状态，导致算法的效率降低。
- DFA 的转移函数的复杂性：如果 DFA 的转移函数较为复杂，即存在大量的转移，那么最小化算法需要进行更多的计算来确定等价状态，从而增加了算法的时间复杂度。
- 初始分割的选择：DFA 最小化通常是通过初始分割 (initial partition) 和后续的状态合并 (state merging) 来实现的。初始分割的选择会影响到最小化算法的效果和效率。不同的初始分割策略可能导致不同的状态合并次数和最终状态数目，从而影响算法的效率。

总体而言，DFA 最小化的时间复杂度通常是线性的，即  $O(n)$ ，其中  $n$  是 DFA 的状态数目。然而，实际的最小化算法的效率可能受到所使用的具体算法、DFA 的结构和复杂性的影响。

常用的 DFA 最小化算法包括 Hopcroft 算法和 Brzozowski 算法等。这些算法在处理不同类型的 DFA 时可能表现出不同的效率。为了提高最小化效率，可以考虑使用已有的最小化算法及其相关优化技术，如延迟合并和状态表的压缩等。

#### 5. 整体性能考量

从正则表达式到最小化 DFA 的整个流程是有效的，但每个步骤都有其优点和缺点。Thompson 构造法为我们提供了直观和简单的方法来从正则表达式构建 NFA，但可能生成许多状态。子集构造法可以从 NFA 生成 DFA，但可能会产生过多的状态。最后，DFA 最小化可以显著减少状态数，从而提高 DFA 的效率。

## 六、 总结

本次实验整体性的描绘了从正则表达式到 NFA，从 NFA 到 DFA，DFA 最小化，以及直接正则转 DFA 的方法，并对其进行了编程实现，整体下来工作量较大，实现的功能较多，还需要我们进行更加深入的探究

实验当中尝试了许多特殊的函数构造方法等，我在编程的过程当中遇到了很多的困难，不过最终成功解决，完成如此高编程量的作业很开心。

希望我在后续的编译原理学习工作当中能够借鉴本次实验的经验，充分的将本次实验当中所运用到的知识贯彻下去，学习好后续的课程。

NIKU