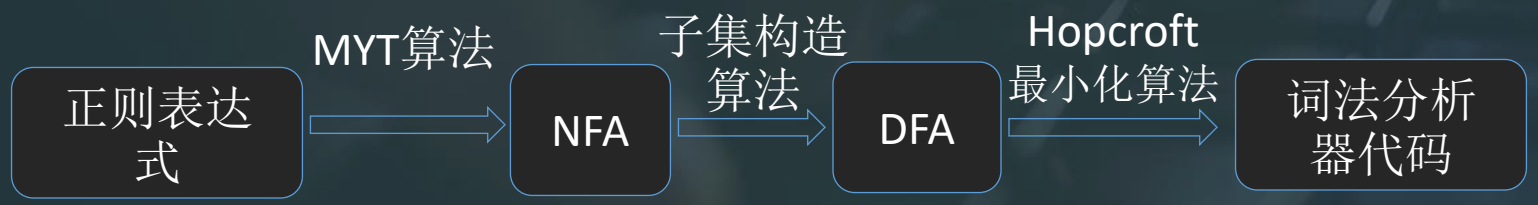
从正则表达式到DFA

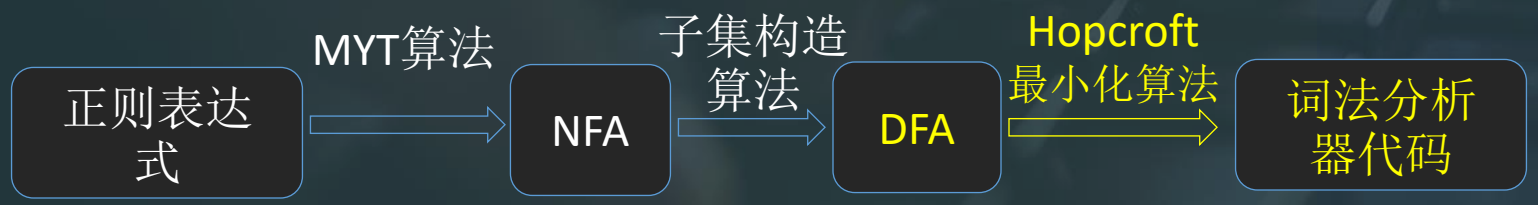
2.2.3DFA的化简


规范声明
生成器
词法分析器


正则表达式
MYT算法
NFA
子集构造算法
DFA
Hopcroft最小化算法
词法分析器代码

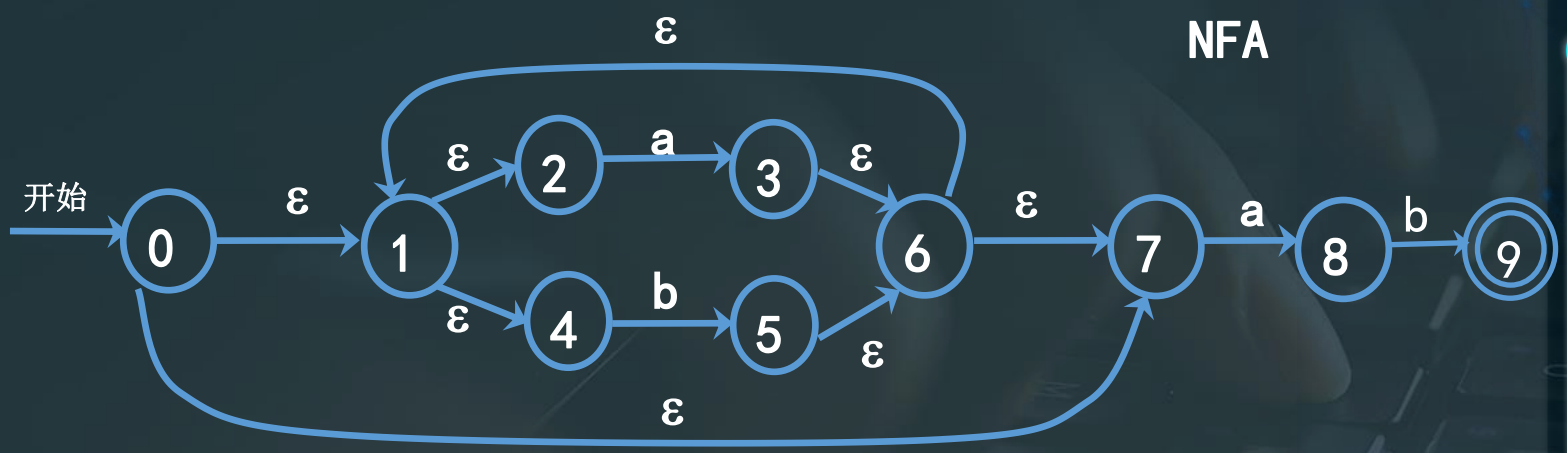
2.2.3DFA的化简

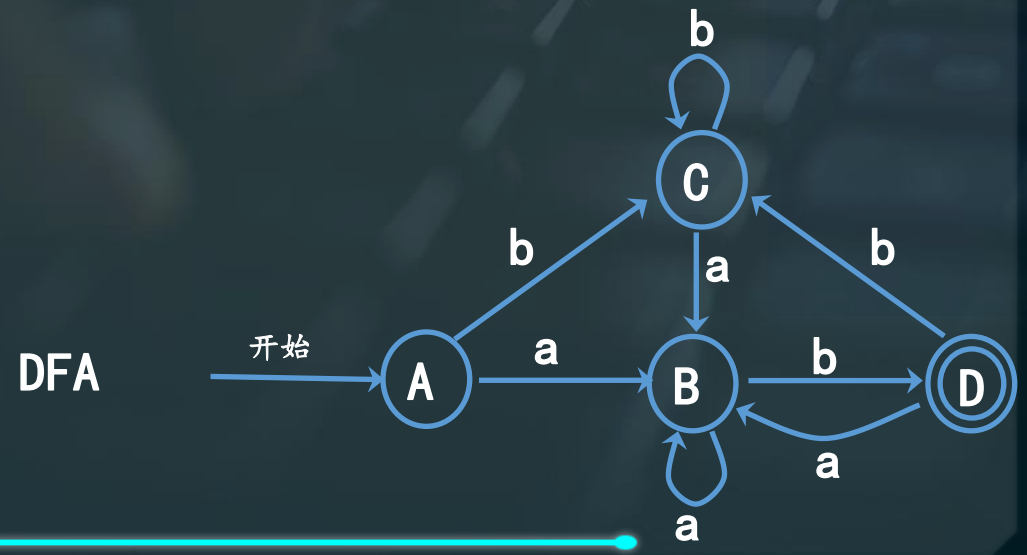

规范声明
生成器
词法分析器


正则表达式
MYT算法
NFA
子集构造算法
DFA
Hopcroft
最小化算法
词法分析器代码

2.2.3DFA的化简

将(a|b)*ab的NFA转换为DFA

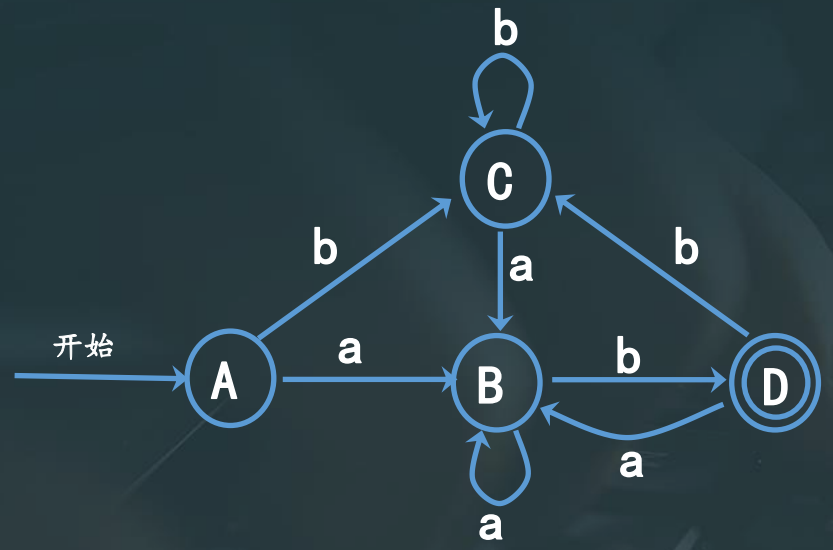


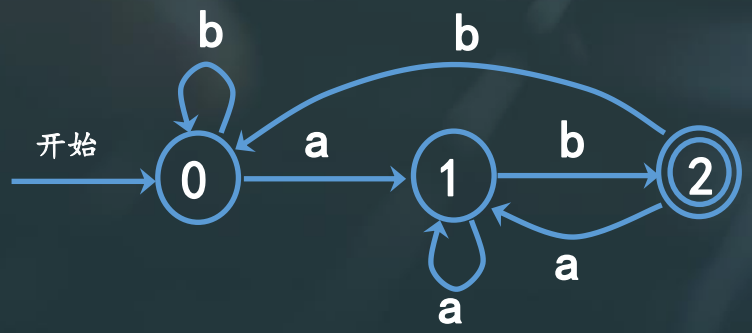


2.2.3DFA的化简

(a|b)*ab的DFA

DFA

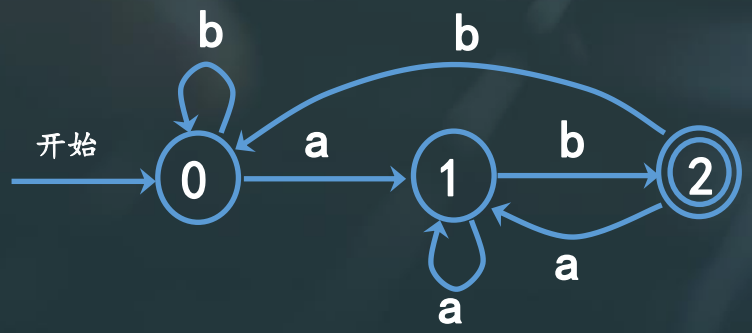




2.2.3DFA的化简

(a|b)*ab的DFA

- A和C都不是接受状态;
- 对任意的输入, 它们总是转到同一个状态.



Hopcroft算法

- Split (G)
  - G 是一个状态集合
  - 输出T_1, …, T_n. 其中 $G = T_1 \cup T_2 \cup \ldots \cup T_n$, 并且$T_i \cap T_j = \Phi$.

```
for （每个字母表中的符号 a ）{
    if (a可以分解 G) {
        将S分解为 T1, …, Tn
        return T1, …, Tn
    }
}
return G
```

Hopcroft算法

- Split (G)
  - G 是一个状态集合
  - 输出$T_1, \ldots, T_n$. 其中 $G = T_1 \cup T_2 \cup \ldots \cup T_n$, 并且$T_i \cap T_j = \Phi$.

```
for （每个字母表中的符号 a ）{
    if (a可以分解 G) {
        将G分解为 T1, …, Tn
        return T1, …, Tn
    }
}
return G
```

Hopcroft算法

- Split (G)
  - G 是一个状态集合
  - 输出T_1, …, T_n. 其中 $G = T_1 \cup T_2 \cup \ldots \cup T_n$, 并且$T_i \cap T_j = \Phi$.

```
for （每个字母表中的符号 a ）{
    if (a可以分解 G) {
        将G分解为 T1, …, Tn
        return T1, …, Tn
    }
}
return G
```

2.2.3DFA的化简

Hopcroft算法

- 输入: 一个DFA D, 其状态集合S, 字母表Σ, 开始状态s_0, 接受状态集合F.
- 输出: 一个DFA D’，它和D接受相同的语言, 且状态数最少.

```
首先构造包含两个状态集合 F 和 S-F 的划分, Π = {F, S-F}.
while (true) {
        Π’ = Φ
        for (Π 中的每一个状态集合G) {
            T1, …, Tn = Split(G)
            Π’ = Π’ ∪ {T1, …, Tn}
        }
        if (Π’ == Π) {
                return Π
        } else {
                Π = Π’
        }
}
```

2.2.3DFA的化简

Hopcroft算法

- 输入: 一个DFA D, 其状态集合S, 字母表Σ, 开始状态s_0, 接受状态集合F.
- 输出: 一个DFA D’，它和D接受相同的语言, 且状态数最少.

```
首先构造包含两个状态集合 F 和 S-F 的划分, Π = {F, S-F}.
while (true) {
        Π’ = Φ
        for (Π 中的每一个状态集合G) {
            T1, …, Tn = Split(G)
            Π’ = Π’ ∪ {T1, …, Tn}
        }
        if (Π’ == Π) {
                return Π
        } else {
                Π = Π’
        }
}
```

2.2.3DFA的化简

Hopcroft算法

- 输入: 一个DFA D, 其状态集合S, 字母表Σ, 开始状态s_0, 接受状态集合F.
- 输出: 一个DFA D’，它和D接受相同的语言, 且状态数最少.

```
首先构造包含两个状态集合 F 和 S-F 的划分, Π = {F, S-F}.
while (true) {
        Π’ = Φ
        for (Π 中的每一个状态集合G) {
            T1, …, Tn = Split(G)
            Π’ = Π’ ∪ {T1, …, Tn}
        }
        if (Π’ == Π) {
                return Π
        } else {
                Π = Π’
        }
}
```

2.2.3DFA的化简

Hopcroft算法

- 输入: 一个DFA D, 其状态集合S, 字母表Σ, 开始状态s_0, 接受状态集合F.
- 输出: 一个DFA D′，它和D接受相同的语言, 且状态数最少.

```
首先构造包含两个状态集合 F 和 S-F 的划分, Π = {F, S-F}.
while (true) {
        Π′ = Φ
        for (Π 中的每一个状态集合G) {
            T1, …, Tn = Split(G)
            Π′ = Π′ ∪ {T1, …, Tn}
        }
        if (Π′ == Π) {
                return Π
        } else {
                Π = Π′
        }
}
```

2.2.3DFA的化简

Hopcroft算法

- 输入: 一个DFA D, 其状态集合S, 字母表Σ, 开始状态s_0, 接受状态集合F.
- 输出: 一个DFA D’ ，它和D接受相同的语言, 且状态数最少.

```
首先构造包含两个状态集合 F 和 S-F 的划分, Π = {F, S-F}.
while (true) {
        Π’ = Φ
        for (Π 中的每一个状态集合G) {
            T1, …, Tn = Split(G)
            Π’ = Π’ ∪ {T1, …, Tn}
        }
        if (Π’ == Π) {
                return Π
        } else {
                Π = Π’
        }
}
```

对Π 中的每一个状态集合G, G中的每个状态都是等价的.

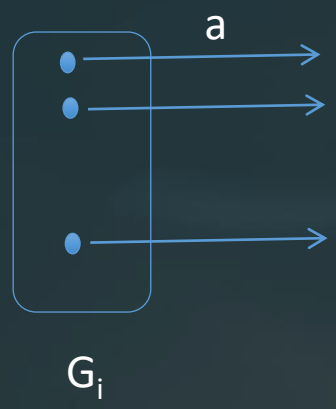
Hopcroft算法

- Split (G)
  - G 是一个状态集合
  - 输出$T_1, \ldots, T_n$. 其中 $G = T_1 \cup T_2 \cup \ldots \cup T_n$, 并且$T_i \cap T_j = \Phi$.

```
for （每个字母表中的符号 a ）{
    if (a可以分解 G) {
        将G分解为 T1, …, Tn
        return T1, …, Tn
    }
}
return S
```

2.2.3DFA的化简

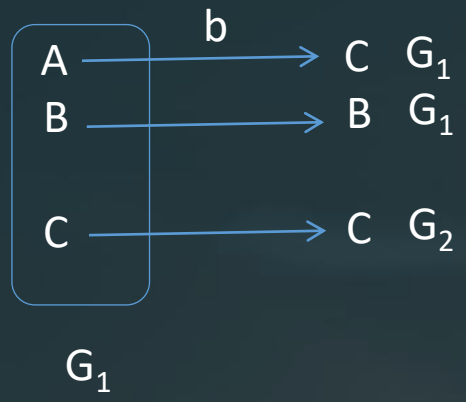
$\Pi = \{G_1, G_2, \ldots, G_n\}$，判断符号a是否可以分解$G_i$



对G_i中的每一个状态，经过符号a可以到达的状态属于哪个状态集合.

2.2.3DFA的化简

Π = {G_1={A, B, C}, G_2={D}}, 判断符号b是否可以分解G_1

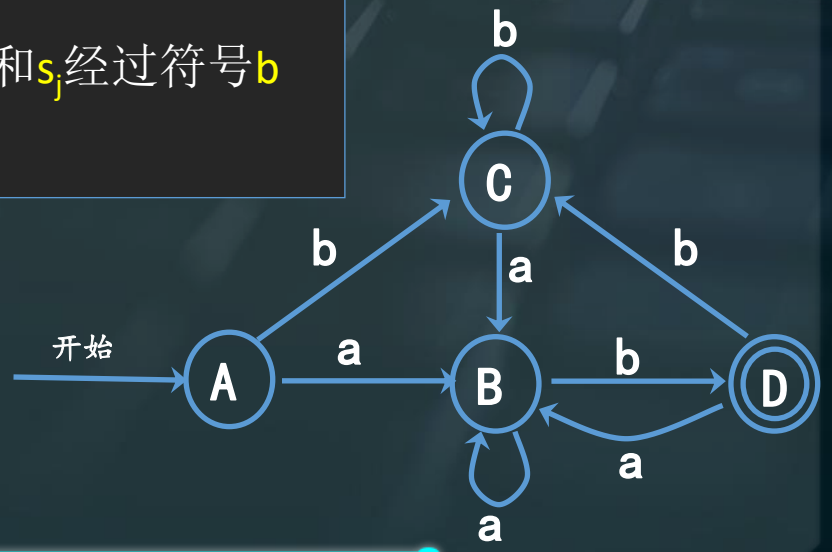


如果这些转换到达的状态落入当前划分的不同组, 我们就说b可以分解G_1.

将G_1划分为多个组, 使得同组的s_i和s_j经过符号b到达同一个组的状态.

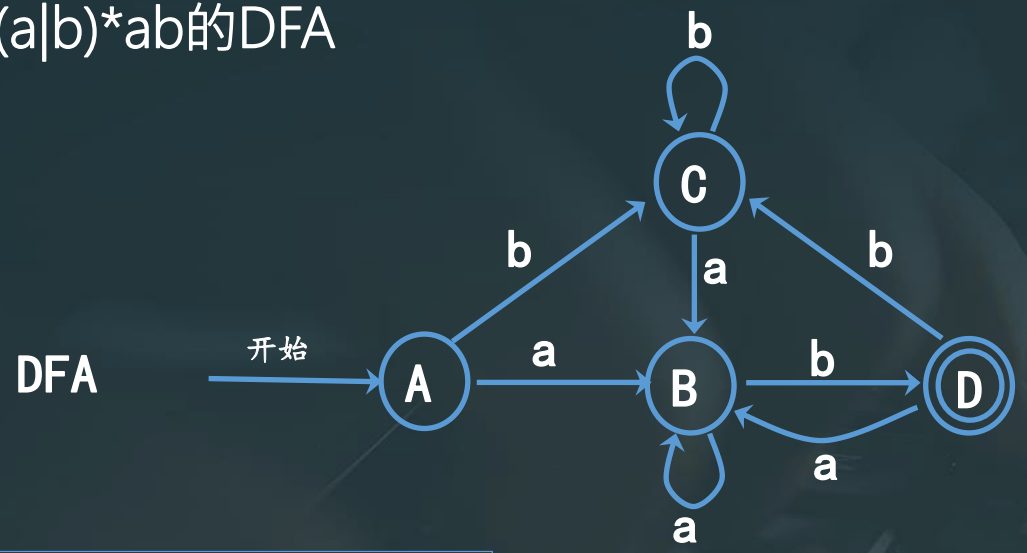
G_1 {A, B, C} 划分为 {A, B} 和 {C}.

DFA



2.2.3DFA的化简

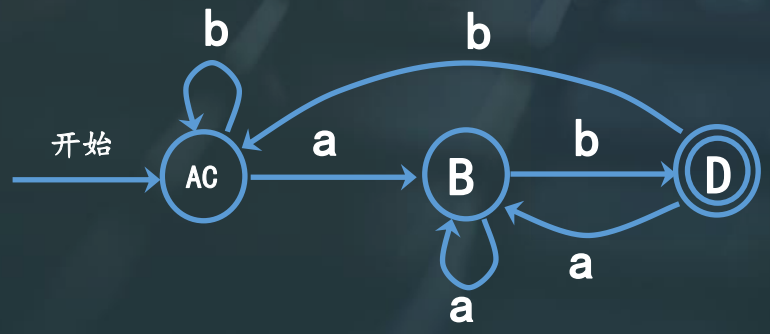
示例1:　(a|b)*ab的DFA



首先构造Π = { {A, B, C}, {D} }

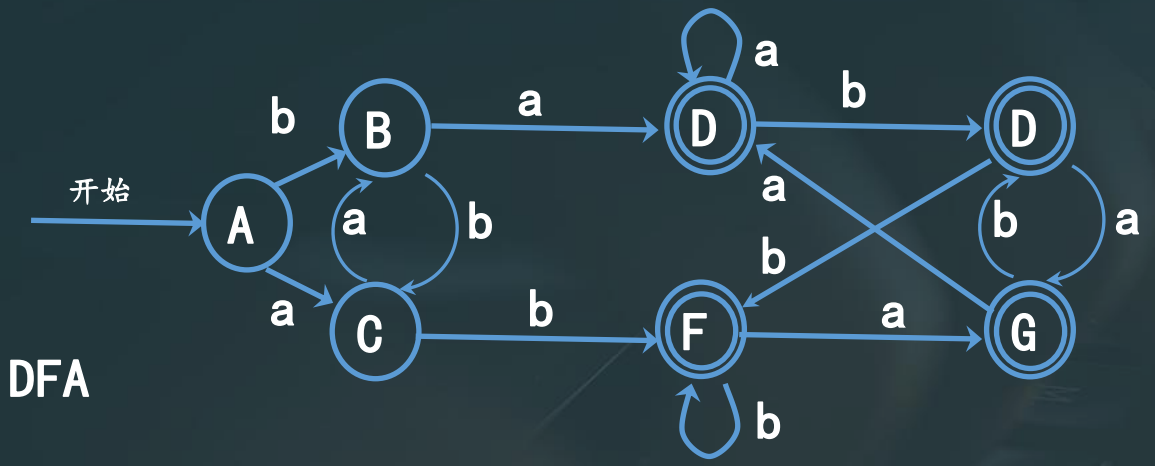
Split({A, B, C}) 得到 {A, C} {B};
因此 Π = { {A, C}, {B}, {D} }

Split({A, C}) 得到 {A, C};
因此 Π = { {A, C}, {B}, {D} }



2.2.3DFA的化简

示例1:　(a|b)*(aa|bb)(a|b)* 的DFA

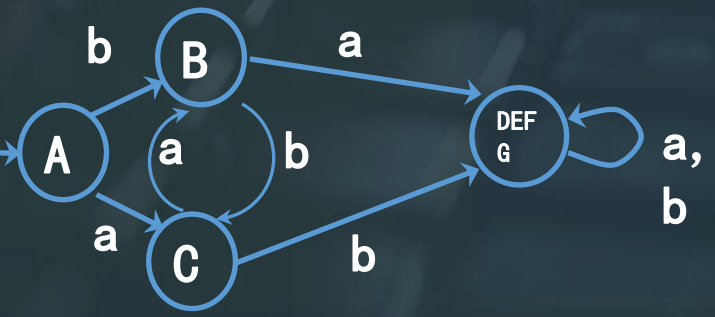


DFA

首先构造Π = { {A, B, C}, {D, E, F, G} }

Split({A, B, C}) 得到 {A, C} {B};
Split({D, E, F, G}) 得到 {D, E, F, G}
因此 Π = { {A, C}, {B}, {D, E, F, G} }

Split({A, C}) 得到 {A} {C};
因此 Π = { {A} {C}, {B}, {D, E, F, G} }



编译原理

苏州大学　李军辉