南開大學
Nankai University

计算机学院

编译原理 **opentopic** 实验

Thompson 构造法、子集构造法和 Hopcroft 算法的实现

姓名：张惠程

学号：2112241

专业：计算机科学与技术

2023 年 11 月 2 日

目录

1 实验要求

实现词法分析器核心构造算法: 正则表达式->NFA 的 Thompson 构造法、NFA->DFA 的子集构造法、DFA 的最小化算法。

我们在这次实验实现了全部的三种算法，串连成词法分析器的完整构造流程。

2 Thompson 构造法

我们首先实现由正规式转换为 NFA 的 Thompson 构造算法。

2.1 算法原理

Thompson 构造算法是一种将正则表达式转换为非确定性有限自动机（NFA）的算法。

2.1.1 核心思想

Thompson 构造算法的主要思想是将复杂的正则表达式拆分为简单的部分，然后为每个简单部分构造 NFA。接着，这些 NFA 可以组合成一个完整的 NFA，代表整个正则表达式。

2.1.2 算法步骤

1. 基础情况：对于正则表达式 ε（表示空字符串），其 NFA 有一个起始状态和一个接受状态，之间没有边。
2. 组合规则：给定两个 NFA，我们可以使用以下规则将它们组合起来：
   - 连接：如果有两个正则表达式 R 和 S，其 NFA 分别为 N(R) 和 N(S)，则它们的串联 RS 可由以下方式得到：将 N(R) 的接受状态与 N(S) 的起始状态连接。新的接受状态是 N(S) 的接受状态。
   - 选择：对于正则表达式 R S ，构造一个新的起始状态，并对 R 和 S 的每个起始状态添加一个 ε（无输入符号的转移）。然后，添加一个新的接受状态，并从 R 和 S 的接受状态到这个新状态添加 ε。
   - 闭包：对于正则表达式 R*，创建一个新的起始和接受状态。添加从新的起始状态到 R 的起始状态和新的接受状态的 ε，以及从 R 的接受状态到 R 的起始状态和新的接受状态的 ε
3. 递归构造：对于一个复杂的正则表达式，首先将其拆分为更简单的部分，并为每部分递归地构建 NFA。然后，使用上述组合规则将这些 NFA 组合成一个完整的 NFA。

2.2 代码实现

首先我们介绍一下大体的实现流程：我们将正则表达式从中缀形式转换为后缀形式，然后为后缀正则表达式构建 NFA。最后使用 generateDotFile 函数生成.dot 文件，用于可视化 NFA。

下面我们首先介绍我们在这里使用的数据结构，在这里我们定义了三个结构体 Transition，State 和 NFA:

```
 struct Transition
{
    char symbol;
    class State *target;

    Transition(char _symbol, class State *_target) : symbol(_symbol), target(_target)
        {}
};

class State
{
public:
    int id;
    bool isFinal;
    std::vector<Transition> transitions;

    State(int _id, bool _isFinal = false) : id(_id), isFinal(_isFinal) {}
};

class NFA
{
public:
    class State *start;
    class State *accept;

    NFA(class State *_start, class State *_accept) : start(_start), accept(_accept) {}
};
  }
```

Transition 结构体:

它代表从一个状态到另一个状态的转换，它有两个数据成员 char symbol 和 class State *targe 和一个构造函数 Transition(char _symbol, class State *_target) 。

- char symbol: 表示该转换的触发符号。换句话说，当读取到这个符号时，我们可以从一个状态移动到 target 指向的状态。
- class State *target: 这是指向表示此转换将要达到的状态的指针。
- Transition(char _symbol, class State *_target) : 用于创建一个新的转换。构造函数接受一个符号和一个目标状态，并通过初始化列表来初始化相应的数据成员。

State 类:

State 类代表 NFA 中的一个状态，它有两个数据成员 int id 和 bool isFinal，以及它的构造函数 State(int _id, bool _isFinal = false)。

- int id: 每个状态都有一个唯一的 ID，用于在处理和可视化 NFA 时区分不同的状态。
- bool isFinal: 表示该状态是否为 NFA 的终止状态。

- State(int _id, bool _isFinal = false)：用于创建一个新的状态。这个构造函数通过初始化列表来初始化 id 和 isFinal 数据成员。它还为 isFinal 成员提供了一个默认值 false，即默认为非终态。

NFA 类：

NFA 代表一个非确定性有限自动机，它有两个数据成员 class State *start 和 class State *accept，以及它的构造函数 NFA(class State *_start, class State *_accept)。

- class State *start: 指向 NFA 初态的指针。
- class State *accept: 指向 NFA 终态的指针。
- NFA(class State *_start, class State *_accept): 用于创建一个新的 NFA。这个构造函数通过初始化列表来初始化 start 和 accept 数据成员。

下面这一部分为创建状态和添加状态间连接的函数，是下面 thompsonConstruction 的辅助函数：

```
class State *createState(bool isFinal = false)
{
    return new class State(stateCount++, isFinal);
}

void addTransition(class State *from, class State *to, char symbol)
{
    from->transitions.push_back(Transition(symbol, to));
}
```

下面的四个函数是 Thompson 构造算法的核心部分，我们将依次进行介绍：

```
class NFA *thompsonConstruction(char inputChar)
{
    class State *startState = createState();
    class State *acceptState = createState(true);

    if (inputChar == ' ')
    {
        addTransition(startState, acceptState, '\0'); // 使用 '\0' 代表空转换
    }
    else
    {
        addTransition(startState, acceptState, inputChar);
    }

    return new class NFA(startState, acceptState);
}

class NFA *concatenate(class NFA *nfa1, class NFA *nfa2)
{
    // 将nfa2的开始状态的所有转换添加到nfa1的接受状态
    for (auto &transition : nfa2->start->transitions)
    {
        nfa1->accept->transitions.push_back(transition);
```

```
    }
    // 清除nfa2的开始状态的所有转换
    nfa2->start->transitions.clear();

    // 设置nfa1的接受状态为非终止状态
    nfa1->accept->isFinal = false;

    // 返回新的串联NFA
    return new class NFA(nfa1->start, nfa2->accept);
}

class NFA *alternate(class NFA *nfa1, class NFA *nfa2)
{
    class State *startState = createState();
    class State *acceptState = createState(true);

    addTransition(startState, nfa1->start, '\0');
    addTransition(startState, nfa2->start, '\0');
    addTransition(nfa1->accept, acceptState, '\0');
    addTransition(nfa2->accept, acceptState, '\0');
    nfa1->accept->isFinal = false;
    nfa2->accept->isFinal = false;

    return new class NFA(startState, acceptState);
}

class NFA *kleeneStar(class NFA *nfa)
{
    class State *startState = createState();
    class State *acceptState = createState(true);

    addTransition(startState, acceptState, '\0');
    addTransition(startState, nfa->start, '\0');
    addTransition(nfa->accept, acceptState, '\0');
    addTransition(nfa->accept, nfa->start, '\0');
    nfa->accept->isFinal = false;

    return new class NFA(startState, acceptState);
}
```

- thompsonConstruction(char inputChar):

  函数为单个字符创建一个 NFA。基本思路是为输入字符建立一个简单的 NFA，其中只有一个起始状态和一个接受状态，并在它们之间有一个转换。

  如果 inputChar 是空格，这意味着我们要添加一个 ε 转换（用\0 表示）。这里我们不能直接使用希腊字母 ε 作为空串，主要是因为它并不是一个字符，我们无法正确地读取它。所以我们在这里通过使用空格来代替希腊字母 ε 表示空串。比如我们输入字符串”a* b| ” 来代表正规式”a*εb|ε”。

  否则，它为给定的 inputChar 添加一个 Transition。

- concatenate(class NFA *nfa1, class NFA *nfa2):

  该函数用于连接两个 NFA，nfa1 和 nfa2。

  1. 函数首先将 nfa2 的起始状态的所有转换添加到 nfa1 的接受状态。
  2. 然后，清除 nfa2 的起始状态的所有转换。
  3. 将 nfa1 的接受状态设置为非接受状态，因为在连接后，接受状态应该是 nfa2 的接受状态。
  4. 最后，返回串联的 NFA，其起始状态为 nfa1 的起始状态，而接受状态为 nfa2 的接受状态。

  但是这个函数的连接会导致我前面代码中设置的 ID 可能缺失，因为在这里只是简单地舍弃了状态，而我的 ID 会因此缺失。举个例子，s4 和 s5 使用函数做连接，我们会舍弃 s4 状态，在最后的 NFA 中没有状态 s4 但是有状态 s5，导致编号出现问题。但是这并不影响我们生成 NFA、NFA->DFA 和 DFA 最小化的正确性，以后我们可能会需要在这种情况下对 ID 进行重新分配。

- alternate(class NFA *nfa1, class NFA *nfa2)

  函数实现了 NFA 中 nfa1 和 nfa2 的选择操作。

  1. 我们首先会创建新的起始状态和接受状态。
  2. 将新的起始状态与两个 NFA 的起始状态之间用 ε 连接。
  3. 两个 NFA 的接受状态都连接到新的接受状态，同样使用 ε 连接。
  4. 最后，将原始 NFA 的接受状态都被设置为非接受状态。

- kleeneStar(class NFA *nfa) 函数为实现了 NFA 中 Kleene 闭包（*）的功能。

  1. 首先我们会创建新的起始状态和接受状态。
  2. 添加从新的起始状态到新的接受状态的 ε 转换（即\0。
  3. 添加从新的起始状态到旧的 NFA 的起始状态的 ε 转换。
  4. 添加从原始 NFA 的接受状态到新的接受状态的 ε 转换。
  5. 添加从原始 NFA 的接受状态到其起始状态的 ε 转换。
  6. 最后，将设置原始 NFA 的接受状态为非终止状态。

  上面的三个函数 concatenate、alternate、kleeneStar 实际上就是我们 NFA 中连接、选择和闭包操作的具体 C++ 代码实现。

下面是我们将中缀表达式转换为后缀表达式的函数 infixToPostfix 和使用后缀表达式生成 NFA 的函数 generateThompsonNFAFromPostfix：

```
1 std::string infixToPostfix(const std::string &regex)
2 {
3     std::stack<char> stack;
4     std::string postfix = "";
5     std::string modifiedRegex = "";
6 
7     // 插入串联操作符
8     for (size_t i = 0; i < regex.size() - 1; ++i)
9     {
```

```
        modifiedRegex += regex[i];
        if ((std::isalpha(regex[i]) || regex[i] == '*' || regex[i] == ')') &&
            (std::isalpha(regex[i + 1]) || regex[i + 1] == '('))
        {
            modifiedRegex += '.';
        }
    }
    modifiedRegex += regex.back();

    for (char c : modifiedRegex)
    {
        if (std::isalpha(c) || c == ' ') // 判断字符是否为字母
        {
            postfix += c;
        }
        else if (c == '(')
        {
            stack.push(c);
        }
        else if (c == ')')
        {
            while (stack.top() != '(')
            {
                postfix += stack.top();
                stack.pop();
            }
            stack.pop();
        }
        else if (c == '*' || c == '|' || c == '.')
        {
            // 调整操作符的优先级
            while (!stack.empty() && stack.top() != '(' &&
                   ((c == '*' && stack.top() == '*') ||
                    (c == '.' && (stack.top() == '.' || stack.top() == '*')) ||
                    (c == '|' && (stack.top() == '.' || stack.top() == '*' ||
                        stack.top() == '|'))))
            {
                postfix += stack.top();
                stack.pop();
            }
            stack.push(c);
        }
    }

    while (!stack.empty())
    {
        postfix += stack.top();
        stack.pop();
    }
```

```
57
58      return postfix;
59  }
60  NFA *generateThompsonNFAFromPostfix(const std::string &postfix)
61  {
62      std::stack<NFA *> nfaStack;
63
64      for (char c : postfix)
65      {
66          if (isalpha(c) || c == ' ') // 使用C++中的isalpha函数检查字符是否为字母
67          {
68              nfaStack.push(thompsonConstruction(c));
69          }
70          else if (c == '|')
71          {
72              NFA *nfa2 = nfaStack.top();
73              nfaStack.pop();
74              NFA *nfa1 = nfaStack.top();
75              nfaStack.pop();
76              nfaStack.push(alternate(nfa1, nfa2));
77          }
78          else if (c == '*')
79          {
80              NFA *nfa = nfaStack.top();
81              nfaStack.pop();
82              nfaStack.push(kleeneStar(nfa));
83          }
84          else if (c == '.')
85          {
86              NFA *nfa2 = nfaStack.top();
87              nfaStack.pop();
88              NFA *nfa1 = nfaStack.top();
89              nfaStack.pop();
90              nfaStack.push(concatenate(nfa1, nfa2));
91          }
92      }
93
94      return nfaStack.top();
95  }
```

infixToPostfix:

函数首先插入隐式的连接运算符.（因为正则表达式里连接操作为隐式，我们这里使用. 使其能够显式地表现出来），然后使用 Shunting Yard 算法将正则表达式从中缀表示法转换为后缀表示法，我们在这里对 Shunting Yard 算法流程做一个简要的介绍：

- 处理操作数: 如果当前字符是字母（操作数），则直接添加到后缀字符串中。
- 处理左括号: 如果当前字符是 (，则将其推入栈中。
- 处理右括号: 如果当前字符是)，则从栈中弹出操作符并添加到后缀字符串中，直到遇到左括号为

止。

- 处理操作符: 如果当前字符是操作符（如 |, *, .），则根据其优先级和栈顶操作符的优先级进行以下处理：
  - 当栈不为空且栈顶操作符的优先级大于或等于当前操作符的优先级时，从栈中弹出操作符并添加到后缀字符串中。
  - 将当前操作符推入栈中。
- 处理栈中剩余的操作符: 在处理完正则表达式的所有字符后，将栈中剩余的所有操作符弹出并添加到后缀字符串中。

值得注意的是，这里的 * 是单目运算符，在操作符中具有最高的优先级，比如 a*b 在转换为后缀的时候依旧为 a*b。操作符的优先级由高到低为 *（闭包），.（连接），|（选择）。generateThompsonNFAFromPostfix：

该函数的目的是根据给定的后缀表示法的正则表达式生成对应的非确定性有限自动机（NFA）。

- 首先，我们初始化了一个 NFA 栈 nfaStack 用于构建和存储 NFA 的子结构。
- 然后我们进入循环，遍历后缀表示法的正则表达式中的每个字符 c：
  - 如果 c 为操作数，即 c 是一个字母或空格，我们调用函数 thompsonConstruction(c) 为该字符创建一个基础 NFA，并将 NFA 推入栈中。
  - 如果 c 是'|' 符号，从 nfaStack 中弹出顶部的两个 NFA（nfa2 和 nfa1）。使用 alternate(nfa1, nfa2) 函数将这两个 NFA 进行选择操作，并将结果 NFA 压栈。
  - 如果 c 为'*'，从 nfaStack 中弹出栈顶的 nfa 并使用 kleeneStar(nfa) 函数得到新的 NFA 并将其压入栈中。
  - 如果 c 是'.' 符号，即 c 表示连接操作，从 nfaStack 中弹出顶部的两个 NFA（nfa2 和 nfa1），调用 concatenate(nfa1, nfa2) 得到连接后的新的 NFA 并将其压栈。
- 最后我们得到的栈顶元素即为整个后缀表达式对应的 NFA。

下面的函数将结果 NFA 转换为.dot，以便我们之后可以使用 Graphviz 工具进行可视化：

```
void generateDotFile(class NFA *nfa, const std::string &filename)
{
    std::ofstream outfile(filename);

    if (outfile.is_open())
    {
        outfile << "digraph NFA {\n";
        outfile << "  rankdir=LR;\n";
        outfile << "  node [shape = circle];\n";

        std::stack<class State *> stack;
        std::vector<int> visitedStates;
        stack.push(nfa->start);

        while (!stack.empty())
```

```
16          {
17              class State *currentState = stack.top();
18              stack.pop();
19
20              if (std::find(visitedStates.begin(), visitedStates.end(),
                    currentState->id) != visitedStates.end())
21                  continue;
22              visitedStates.push_back(currentState->id);
23
24              if (currentState->isFinal)
25              {
26                  outfile << "  \""
27                          << "S" << currentState->id << "\" [shape = doublecircle];\n";
28              }
29              else
30              {
31                  outfile << "  \""
32                          << "S" << currentState->id << "\" [shape = circle];\n";
33              }
34
35              for (const auto &transition : currentState->transitions)
36              {
37                  outfile << "  \"S" << currentState->id << "\" -> \"S" <<
                        transition.target->id << "\" [label=\"" << (transition.symbol ==
                        '\0' ? " " : std::string(1, transition.symbol)) << "\"];\n";
38
39                  stack.push(transition.target);
40              }
41          }
42
43          outfile << "}\n";
44          outfile.close();
45          std::cout << "NFA已生成到 " << filename << " 文件中\n";
46      }
47      else
48      {
49          std::cerr << "无法打开文件以写入输出\n";
50      }
51  }
```

3 子集构造法

我们现在已经得到了由正规式转换的 NFA，现在我们使用子集构造法将 NFA 转换为对应的 DFA。

3.1 算法原理

子集构造算法，也称为 powerset 构造算法，是一个用于将非确定性有限自动机（NFA）转换为确定性有限自动机（DFA）的经典算法。

3.1.1 核心思想

子集构造算法的核心思想是利用“状态的集合”来捕获 NFA 的非确定性。因为 NFA 可能会在一个给定的输入符号上有多个可能的转换，或者通过 ε 转换而无需任何输入，因此在某一时刻，NFA 可能会处于多个状态。而 DFA 是确定性的，它在每一个时刻只能处于一个状态。子集构造法要做的，就是将 NFA 的的非确定性状态转换为 DFA 确定性的状态集合，可以用来捕获 NFA 在某个输入符号上所有可能的状态转换。

3.1.2 算法步骤

下面我们说明这个算法的步骤：

1. 初始化：创建一个新的 DFA 状态，它表示 NFA 的起始状态的 闭包（通过零个或多个 转换可以到达的所有状态），并将此状态标记为未处理，加入到未处理的集合中。
2. 处理未处理的状态：
   - 程序的主体是在迭代地处理未处理的集合，当未处理的集合为空时，程序结束。
   - 对于每个输入符号，计算 NFA 状态集合的转移，并找到这些转移的 ε 闭包。这个闭包定义了一个新的 DFA 状态。
   - 如果这个新的 DFA 状态之前未出现过，将其添加到 DFA 中并标记为未处理。
   - 在 DFA 中添加从当前状态到新状态的转移，使用当前处理的输入符号。
3. 终态：在 DFA 中，含有 NFA 终态的状态我们将其视作 DFA 中的终态，并会被标记为终态。

3.2 代码实现

下面我们具体地给出实现子集构造法的代码，并对其进行介绍。我们在这里会根据 Thompson 构造算法得到的 NFA 进一步来构造 DFA，最后使用自定义的 generateDotFileForDFA 函数生成.dot 文件，以便最后我们使用 graphviz 来可视化 DFA 结构。

我们首先介绍在 DFA 中使用的数据结构结构体 DFAState：

```
struct DFAState
{
    int id;
    bool isFinal;
    std::set<State *> nfaStates;
    std::map<char, DFAState *> transitions;
    DFAState(int _id) : id(_id), isFinal(false) {}
};
std::vector<DFAState *> dfaStates;
std::map<std::set<State *>, int> stateMap;
```

DFAState 表示了在确定性有限自动机（DFA）中的一个状态。在子集构造法中，一个 DFA 的状态实际上是由一个或多个 NFA 的状态组成的。下面我们介绍结构体中的成员：

- int id：这是每个 DFA 状态的唯一标识符（例如，0、1、2 等）。设定 ID 是为了确保每个 DFA 状态都是唯一的，我们使用这个 ID 进行区分。这与 NFA 中我们设定 ID 的目的是一致的。
- bool isFinal：isFinal 是一个表示是否为终态的布尔值变量，在 DFA 的状态中，含有一个或多个 NFA 终态，我们将其设置为终态。
- std::set<State *> nfaStates：用来表示构成 DFA 状态的所有 NFA 状态，在子集构造法中，DFA 状态用 NFA 状态集合表示。
- std::map<char, DFAState *> transitions：这是一个映射，描述了从当前 DFA 状态在给定输入字符下的转换。键是输入字符（例如，'a'、'b' 等），值是转换后到达的 DFA 状态的指针。映射提供了一个查找表，帮助我们快速找到给定输入字符下的转换。

我们定义了全局变量 std::vector<DFAState *> dfaStates 用于存储所有的 DFA 状态。std::map<std::set<State *>, int> stateMap 用于映射 NFA 状态集到对应 DFA 状态的 ID。

下面我们需要首先介绍 std::set<State *> collectStatesFromNFA(NFA *nfa) 这个函数，我们通过这个函数来收集 NFA 中所有的状态：

```
std::set<State *> collectStatesFromNFA(NFA *nfa)
{
    std::set<State *> states;
    std::stack<State *> stack;
    stack.push(nfa->start);

    while (!stack.empty())
    {
        State *curr = stack.top();
        stack.pop();

        std::cout << "Processing state: S" << curr->id << std::endl; // 输出当前处理的状态

        if (states.find(curr) == states.end())
        {
            states.insert(curr);
            std::cout << "Inserted state: S" << curr->id << " to states set. Total states: " << states.size() << std::endl; // 输出状态集合大小

            for (const auto &trans : curr->transitions)
            {
                std::cout << "Transition from S" << curr->id << " to S" << trans.target->id << " with label: " << (trans.symbol == '\0' ? "ε" : std::string(1, trans.symbol)) << std::endl; // 输出转移信息
                stack.push(trans.target);
            }
        }
```

```
    }
    return states;
}
```

函数基于深度优先搜索（DFS）策略遍历整个 NFA。为了进行 DFS，我们使用栈来进行辅助。首先我们将起始状态推入栈。然后，我们不断从栈顶取出状态，检查它是否已经被处理过。如果状态是新的，我们将其添加到已访问的状态集中，然后将所有从该状态出发的转换的目标状态推入栈。通过这种方式，我们确保处理所有可以从起始状态到达的状态。同时，在函数中我加入了一些调试输出可以看到每一步的处理结果。

下面我们正式进入到子集构造法的实现部分，我们在这里先介绍函数 getOrCreateDFAState：

```
int getOrCreateDFAState(const std::set<State *> &nfaStateSet)
{
    // 输出正在处理的NFA状态集
    std::cout << "Checking or creating DFA state for NFA states: ";
    for (State *s : nfaStateSet)
    {
        std::cout << s->id << " ";
    }
    std::cout << std::endl;

    if (stateMap.find(nfaStateSet) == stateMap.end())
    {
        DFAState *newState = new DFAState(dfaStates.size());

        for (State *s : nfaStateSet)
        {
            if (s->isFinal)
            {
                newState->isFinal = true;
                break;
            }
        }
        newState->nfaStates = nfaStateSet;
        dfaStates.push_back(newState);
        stateMap[nfaStateSet] = newState->id;

        // 输出已经为NFA状态集创建了新的DFA状态的信息
        std::cout << "Created new DFA state " << newState->id << " for NFA states: ";
        for (State *s : nfaStateSet)
        {
            std::cout << s->id << " ";
        }
        std::cout << std::endl;
    }
    else
    {
        // 输出已经为NFA状态集找到了现有的DFA状态的信息
```

```
        std::cout << "Found existing DFA state " << stateMap[nfaStateSet] << " for
            NFA states: ";
        for (State *s : nfaStateSet)
        {
            std::cout << s->id << " ";
        }
        std::cout << std::endl;
    }

    return stateMap[nfaStateSet];
}
```

这个函数的主要思路是当给定一个 NFA 状态集时，该函数首先检查是否已经为这个状态集创建了一个 DFA 状态（即是否在 stateMap 中）。

如果没有，则创建一个新的 DFA 状态，并为其分配一个 ID。新状态的接受状态属性基于 NFA 状态集中是否有任何接受状态来设置。然后函数将新的 DFA 状态添加到 dfaStates 向量中，并将 NFA 状态集与新 DFA 状态的 ID 关联在 stateMap 中。

如果之前已经为这个状态集创建了一个 DFA 状态，该函数只返回现有 DFA 状态的 ID。

下面我们介绍一下实现闭包操作的 eClosure 函数和计算从 NFA 状态集中通过一个给定符号所有可达集合的函数 move：

```
std::set<State *> eClosure(State *state)
{
    std::set<State *> result;
    std::stack<State *> stack;
    stack.push(state);

    while (!stack.empty())
    {
        State *current = stack.top();
        stack.pop();

        if (result.find(current) != result.end())
            continue;
        result.insert(current);

        for (Transition t : current->transitions)
        {
            if (t.symbol == '\0' && result.find(t.target) == result.end())
            {
                stack.push(t.target);
            }
        }
    }
    return result;
}

std::set<State *> eClosure(const std::set<State *> &stateSet)
```

```
{
    std::set<State *> result;
    for (State *s : stateSet)
    {
        std::set<State *> temp = eClosure(s);
        result.insert(temp.begin(), temp.end());
    }

    // Debug output
    std::cout << "eClosure of states: ";
    for (State *s : stateSet)
    {
        std::cout << s->id << " ";
    }
    std::cout << "results in states: ";
    for (State *s : result)
    {
        std::cout << s->id << " ";
    }
    std::cout << std::endl;

    return result;
}

std::set<State *> move(const std::set<State *> &stateSet, char symbol)
{
    std::set<State *> result;
    for (State *s : stateSet)
    {
        for (Transition t : s->transitions)
        {
            if (t.symbol == symbol)
            {
                result.insert(t.target);
            }
        }
    }

    // Debug output
    std::cout << "Moving with symbol: " << symbol << " from states: ";
    for (State *s : stateSet)
    {
        std::cout << s->id << " ";
    }
    std::cout << "to states: ";
    for (State *s : result)
    {
        std::cout << s->id << " ";
    }
```

```
77      std::cout << std::endl;
78
79      return result;
80  }
```

eClosure(State state):

这个函数计算并返回一个 NFA 状态的 ε 闭包。ε 闭包是从一个状态开始，可以仅通过 ε 转换到达的所有状态的集合。

主要流程为:

- 使用深度优先搜索（通过栈）来遍历 NFA 并找到所有可通过 ε 转换到达的状态。
- 如果当前状态已经在结果集中，则跳过。
- 否则，将其添加到结果集中并查看其所有 ε 转换。

eClosure(const std::set<State *> &stateSet):

这个函数计算并返回一个 NFA 状态集合的 闭包。它为状态集中的每个状态调用上面的 eClosure(State *state) 函数，最后将结果合并。move(const std::set<State > &stateSet, char symbol):

这个函数计算并返回从 NFA 状态集中通过一个给定符号的转换可以到达的所有状态的集合。对于状态集中的每个状态，查看其所有转换。如果转换的符号与给定的符号匹配，则添加转换的目标状态到结果集。

下面是子集构造法的核心函数 constructDFAFromNFA，用于将 NFA 转换为 DFA:

```
1   void constructDFAFromNFA(NFA *nfa, const std::set<State *> &nfaStates)
2   {
3       std::set<State *> startStateSet = eClosure(nfa->start);
4       std::queue<std::set<State *>> processQueue;
5       processQueue.push(startStateSet);
6       getOrCreateDFAState(startStateSet);
7
8       std::set<char> inputSymbols;
9       for (State *s : nfaStates)
10      {
11          for (Transition t : s->transitions)
12          {
13              if (t.symbol != '\0')
14              {
15                  inputSymbols.insert(t.symbol);
16              }
17          }
18      }
19
20      while (!processQueue.empty())
21      {
22          std::set<State *> currentStateSet = processQueue.front();
23          processQueue.pop();
24          DFAState *currentDFAState = dfaStates[getOrCreateDFAState(currentStateSet)];
25
```

```
26          // 输出当前正在处理的DFA状态
27          std::cout << "Processing DFA state: ";
28          for (State *s : currentStateSet)
29          {
30              std::cout << s->id << " "; //
                    假设State有一个名为"name"的成员，表示状态的名称
31          }
32          std::cout << std::endl;
33
34          for (char symbol : inputSymbols)
35          {
36              std::set<State *> nextStateSet = eClosure(move(currentStateSet, symbol));
37
38              // 输出对应于给定符号的转移的结果状态集合
39              std::cout << "Moving with symbol " << symbol << " results in states: ";
40              for (State *s : nextStateSet)
41              {
42                  std::cout << s->id << " ";
43              }
44              std::cout << std::endl;
45
46              if (!nextStateSet.empty())
47              {
48
49                  if (stateMap.find(nextStateSet) == stateMap.end()) // Check if this
                        DFA state hasn't been processed yet
50                  {
51                      processQueue.push(nextStateSet);
52                  }
53                  int nextStateId = getOrCreateDFAState(nextStateSet); // Always get or
                        create DFA state
54                  currentDFAState->transitions[symbol] = dfaStates[nextStateId];
55              }
56          }
57      }
58  }
```

下面我们介绍一下函数的执行流程，或者说是使用子集构造法将 NFA 转换为 DFA 的过程。

- 首先，我们使用 eClosure 函数得到 NFA 的起始状态的 ε 闭包，即在不使用任何输入符号的情况下可以到达的状态集合。这个状态集将成为 DFA 的第一个状态。
- 此外，将起始状态的 ε 闭包放入 processQueue 中等待处理，并使用 getOrCreateDFAState 函数确保它在 DFA 状态集中有对应的状态。
- 接下来函数收集所有的输入符号，遍历所有的 NFA 状态和它们的转换，收集所有非 ε 转换的输入符号。这些符号将被用来计算从一个状态集到另一个状态集的转移。
- 接下来我们开始构造 DFA，使用宽度优先搜索（通过队列 processQueue）遍历待处理的状态集。对于每一个正在处理的状态集：

  - 获取或创建其对应的 DFA 状态。
  - 输出当前正在处理的 DFA 状态的信息。
  - 对于每一个输入符号，计算由当前状态集通过该符号转移得到的状态集。这是通过先使用 move 函数，然后使用 eClosure 函数来完成的。
  - 如果结果状态集不为空，并且尚未被处理，则将其加入队列中。
  - 函数确保了为结果状态集获取或创建一个对应的 DFA 状态，并更新当前 DFA 状态的转移映射。

- 当 processQueue 为空，DFA 转换完成，程序结束。

自此，我们完成了子集构造法的代码部分，下面同样给出输出.dot 文件的函数 generateDotFileForDFA：

```
void generateDotFileForDFA(const std::string &filename)
{
    std::ofstream outfile(filename);

    if (outfile.is_open())
    {
        outfile << "digraph DFA {\n";
        outfile << "  rankdir=LR;\n";
        outfile << "  node [shape = circle];\n";

        for (DFAState *dfaState : dfaStates)
        {
            // 我们将使用NFA状态的集合作为DFA状态的名字
            std::string stateName = "{";
            for (State *nfaState : dfaState->nfaStates)
            {
                stateName += "S" + std::to_string(nfaState->id) + ",";
            }
            stateName.back() = '}'; // 替换最后的逗号

            if (dfaState->isFinal)
            {
                outfile << "  \"" << stateName << "\" [shape = doublecircle];\n";
            }
            else
            {
                outfile << "  \"" << stateName << "\" [shape = circle];\n";
            }

            for (const auto &transition : dfaState->transitions)
            {
                std::string targetName = "{";
                for (State *nfaState : transition.second->nfaStates)
                {
                    targetName += "S" + std::to_string(nfaState->id) + ",";
                }
```

```
37                  targetName.back() = '}';
38
39                  outfile << "  \"" << stateName << "\" -> \"" << targetName << "\"
                        [label=\"" << transition.first << "\"];\n";
40              }
41          }
42
43          outfile << "}\n";
44          outfile.close();
45          std::cout << "DFA已生成到 " << filename << " 文件中\n";
46      }
47      else
48      {
49          std::cerr << "无法打开文件以写入输出\n";
50      }
51  }
```

4 Hopcroft 的算法

Hopcroft 算法是一种用于最小化确定性有限自动机（DFA）的算法。这个算法旨在通过合并等价的状态来简化给定的 DFA，从而得到一个具有最少状态的等价 DFA。

4.1 算法原理

4.1.1 核心思想

算法的主要思想是基于“状态的等价性”。两个状态如果对于所有输入都有相同的行为，则它们是等价的。Hopcroft 算法通过不断地细化状态的划分来找到所有的等价状态，并最终将它们合并。

4.1.2 算法步骤

1. 我们为 DFA 中的终态和非终态分别初始化集合 A 和 B，并使用队列来维护。我们将 A 和 B 分别加入待处理的工作队列。
2. 接下来，选择并从工作队列中删除一个集合。
3. 对于 DFA 的每个输入符号 a：将所有状态分成两组：一组是在输入 a 时会转移到当前选定集合的状态；另一组是不会的。如果这样的划分导致其中一个集合被分成两个更小的子集，那么：
   - 如果其中一个子集的大小小于原始集合的大小，将两个子集都加入工作队列。
   - 否则，只将较小的子集加入工作队列。
4. 我们将重复上面步骤 2、3，直到队列为空。

最后，等价的状态集合会被合并成单一的状态，从而得到一个最小化的 DFA。

4.2 代码实现

我们下面将对 Hopcroft 算法给出实现的代码，实际上就是 miniDFA 函数：

```
void minimizeDFA()
{
    // 初始化
    std::vector<std::set<DFAState *>> partitions;
    std::set<DFAState *> accepting, nonAccepting;

    for (DFAState *state : dfaStates)
    {
        if (state->isFinal)
        {
            accepting.insert(state);
        }
        else
        {
            nonAccepting.insert(state);
        }
    }

    partitions.push_back(accepting);
    partitions.push_back(nonAccepting);

    std::vector<std::set<DFAState *>> newPartitions;
    bool partitioned = true;

    while (partitioned)
    {
        partitioned = false;
        newPartitions.clear();

        for (const auto &part : partitions)
        {
            std::map<std::string, std::set<DFAState *>> splitSets;

            for (DFAState *state : part)
            {
                std::string signature = "";
                for (const auto &[symbol, targetState] : state->transitions)
                {
                    for (size_t i = 0; i < partitions.size(); ++i)
                    {
                        if (partitions[i].find(targetState) != partitions[i].end())
                        {
                            signature += std::to_string(i) + symbol;
                            break;
                        }
                    }
```

```
47                  }
48                  splitSets[signature].insert(state);
49              }
50
51              for (const auto &[_, splitSet] : splitSets)
52              {
53                  newPartitions.push_back(splitSet);
54              }
55          }
56
57          if (newPartitions.size() != partitions.size())
58          {
59              partitioned = true;
60              partitions = newPartitions;
61          }
62      }
63
64      // 创建新的DFA状态
65      std::vector<DFAState *> newDFAStates;
66      for (const auto &part : partitions)
67      {
68          if (!part.empty())
69          {
70              DFAState *newState = new DFAState(newDFAStates.size());
71              newDFAStates.push_back(newState);
72              std::cout << "Creating new state with id: " << newState->id << std::endl;
                    // 调试输出
73          }
74      }
75
76      // 设置转换
77      for (size_t index = 0; index < partitions.size(); ++index)
78      {
79          const auto &part = partitions[index];
80          DFAState *newState = newDFAStates[index];
81          DFAState *representative = *part.begin();
82          newState->isFinal = representative->isFinal;
83
84          for (const auto &[symbol, targetState] : representative->transitions)
85          {
86              for (size_t i = 0; i < partitions.size(); ++i)
87              {
88                  if (partitions[i].find(targetState) != partitions[i].end())
89                  {
90                      newState->transitions[symbol] = newDFAStates[i];
91                      std::cout << "Setting transition: " << symbol << " -> State " <<
                            newDFAStates[i]->id << std::endl; // 调试输出
92                      break;
93                  }
```

```
            }
        }
    }

    // 释放原始DFA状态的内存
    for (DFAState *state : dfaStates)
    {
        delete state;
    }

    // 更新DFA状态列表
    dfaStates = newDFAStates;
}
```

1. 首先根据是否为终止状态(或接受状态)将所有 DFA 状态分为两组:`finalStates` 和 `nonFinalStates`。同时，初始化一个 `partitions` 变量，开始时只有两个部分：终止状态和非终止状态。
2. 使用 `wasChanged` 标志来执行循环，直到分区不再改变为止。
   - 对于每个当前分区中的状态，查找其转移函数。转移函数是根据输入符号将状态映射到另一个状态分区的集合。
   - 使用 `transitionPartitions` 来收集具有相同转移函数的状态。
   - 更新 `newPartitions` 以包括由 `transitionPartitions` 确定的新分区。
   - 如果 `newPartitions` 的大小与 `partitions` 不同，说明发生了改变。将 `newPartitions` 赋值给 `partitions` 并继续循环。
3. 对于每个新的状态分区，创建一个新的 DFA 状态。新状态的 ID 是分区的索引，如果分区包含任何终止状态，则该状态为终止状态。同时，我们需要复制转移函数，但现在转移到新的最小化状态。
4. 删除所有旧的 DFA 状态并替换为新的最小化状态。

我们在这里同样给出实现输出 DFA 最小化后.dot 文件的函数 generateMinimizedDotFileForDFA

```
void generateMinimizedDotFileForDFA(const std::string &filename)
{
    std::ofstream outfile(filename);

    if (outfile.is_open())
    {
        outfile << "digraph MinimizedDFA {\n";
        outfile << "  rankdir=LR;\n";
        outfile << "  node [shape = circle];\n";

        for (DFAState *dfaState : dfaStates)
        {
            std::string stateName = "S" + std::to_string(dfaState->id);
```

```
            std::cout << "Processing state with id: " << dfaState->id << std::endl;
                // 调试输出
            if (dfaState->isFinal)
            {
                outfile << "  \"" << stateName << "\" [shape = doublecircle];\n";
            }
            else
            {
                outfile << "  \"" << stateName << "\" [shape = circle];\n";
            }

            for (const auto &transition : dfaState->transitions)
            {
                if (!transition.second)
                {
                    std::cerr << "Error: Invalid pointer for target DFA state." <<
                        std::endl;
                    continue; // Skip this transition
                }

                std::cout << "Transition: " << transition.first << " -> State " <<
                    transition.second->id << std::endl; // 调试输出

                std::string targetName = "S" + std::to_string(transition.second->id);
                outfile << "  \"" << stateName << "\" -> \"" << targetName << "\"
                    [label=\"" << transition.first << "\"];\n";
            }
        }

        outfile << "}\n";
        outfile.close();
        std::cout << "Minimized DFA has been generated to " << filename << "\n";
    }
    else
    {
        std::cerr << "Unable to open file for writing output\n";
    }
}
```

5 程序测试

在这一部分，我们将构造一些正规式和输入来测试我们程序的正确性，正如前面所提到的，我的编号可能会出现跳过的问题，但是我绘制的 NFA、DFA 以及最小化后的 DFA 正确性不受影响。

5.1 测试样例 1

我们首先测试我们第三章书面作业中的正则式：b*a((b|ε)(a|b|ε))。在我们的程序中 ε 用空格（\0）来表示，所以我们程序中输入的字符串为 b*a((b|)(a|b|))。

输入：b*a((b|)(a|b|))

我下面给出输出结果：

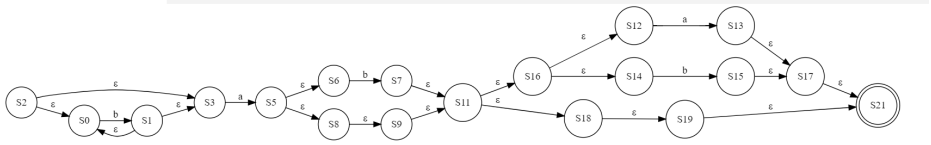

图 5.1: NFA

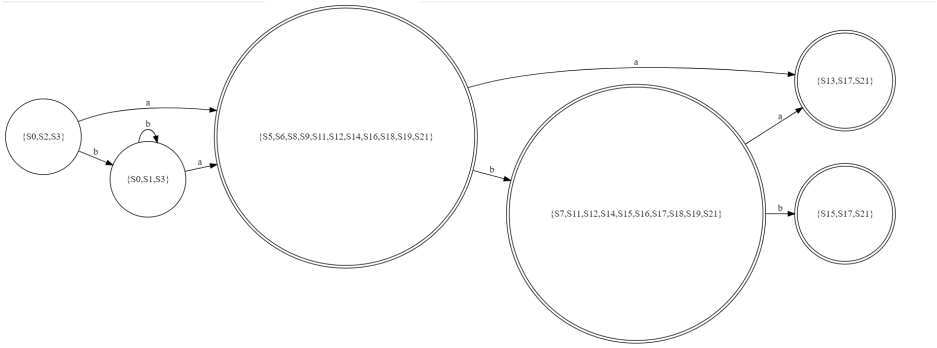

图 5.2: DFA

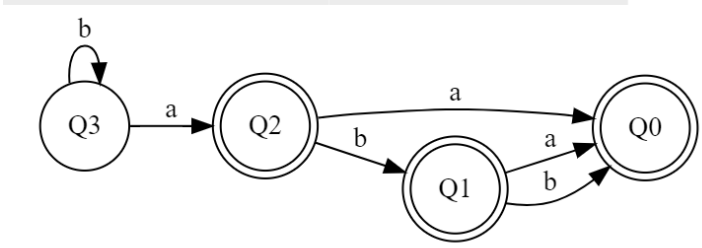

图 5.3: Minimized DFA

5.2 测试样例 2

正规式：d*(ab*)*|c

输入：d*(ab*)*|c

输出：

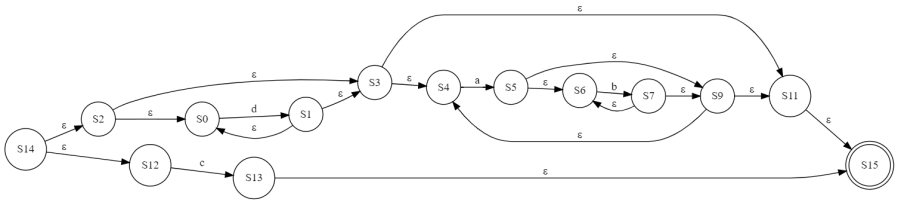

图 5.4: NFA

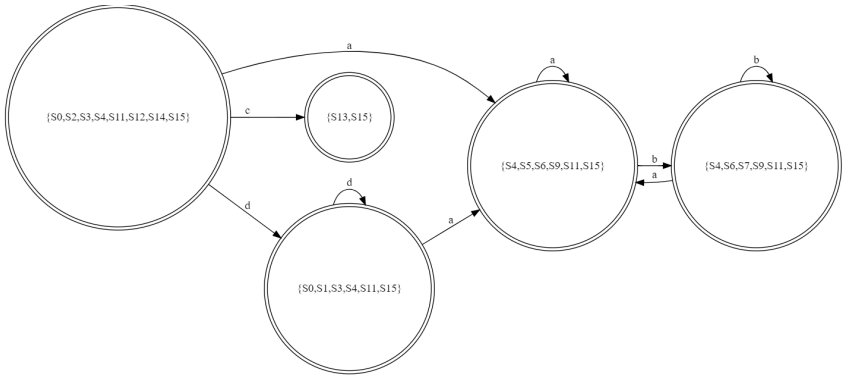

图 5.5: DFA

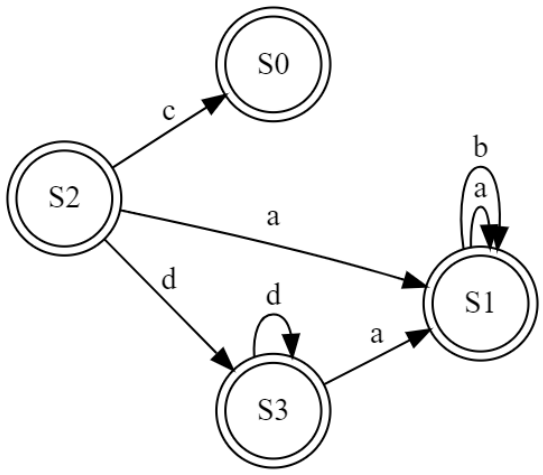

图 5.6: Minimized DFA

5.3 测试样例 3

正规式：a|(b|c|e)ε|ε|d*

输入:a|(b|c|e) | |d*

输出:

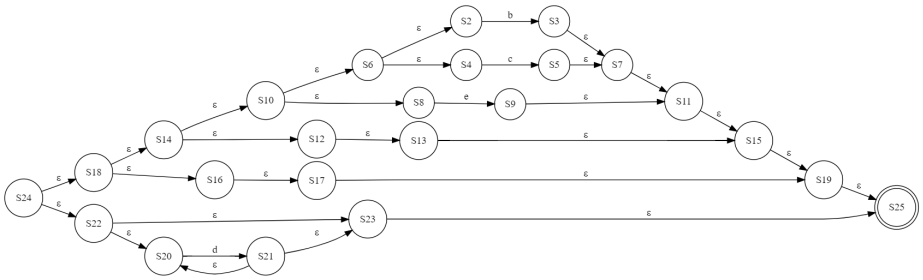

图 5.7: NFA

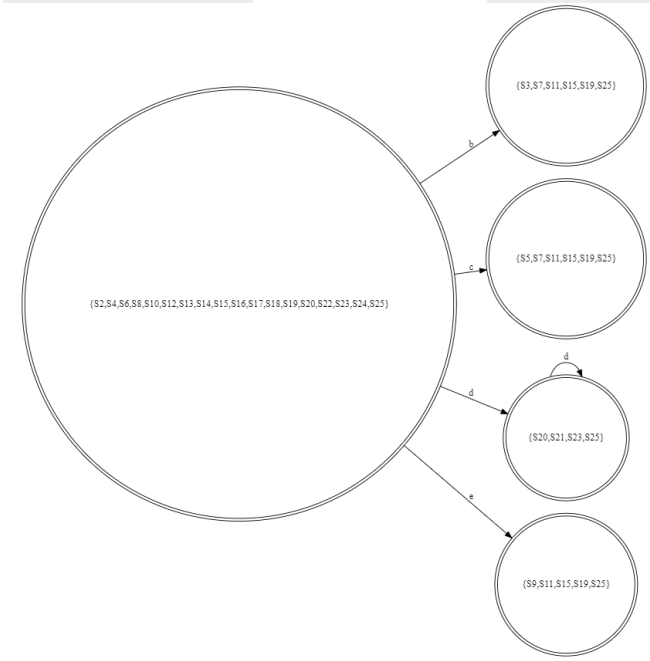

图 5.8: DFA

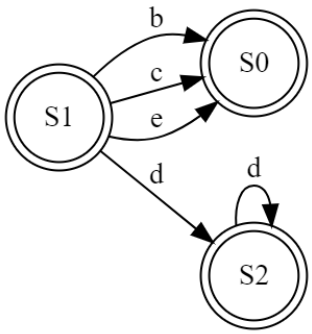

图 5.9: Minimized DFA

6 实验总结

在本次 opentopic 实验中，成功实现了 Thompson 构造法、子集构造法和 Hopcroft 算法的实现，串联出了完整的词法分析器构造流程。同时，我们通过生成.dot 文件的方式，使用 Graghviz 工具可视化地画出了生成的 NFA、DFA 和最小化后得到的 DFA。最后，我们构造了一些测试样例来验证程序的正确性。

这次实验的代码高达 600 多行，书写技术文档也消耗了大量时间。我们通过这次实验，实现了词法分析器的核心构造算法，对我的工程能力有了很大的提升。

7 代码链接

本次实验的代码已上传到编译原理仓库。