苏 州 大 学 实 验 报 告

院、系	计算机科学与技术学院	年级专业	22 计科	姓名	姜涛	学号	2227405073
课程名称	编译原理实践					成绩	
指导教师	段湘煜	同组实验者	无		实验日期	2024/9/29	

实 验 名 称 基于自动机的词法分析器

一. 实验题目

深入研究词法分析器，用有限状态自动机实现正则表达式，以能够识别出符合正则表达式的词法单元；

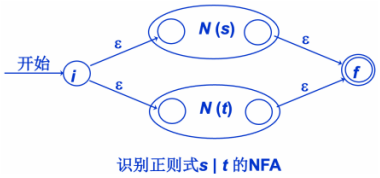
分别测试以正则表达式(a|b)*abb 以及 10|(0|11)0*1 为输入，输出各自的最简 DFA，输出形式为 DFA 的状态转移表。

根据 TEST 语言的如下词法规则，实现基于自动机的词法分析器。

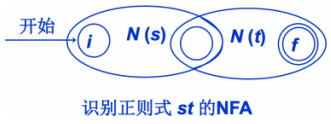
二. 实验原理及流程框图

1. 正则表达式到 NFA

继而构造“或”关系的 NFA:



构造“与”关系的 NFA:



构造闭包表达式的 NFA:

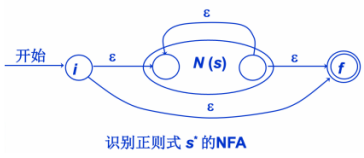


Figure 1 识别不同正则式的 NFA

2. 建立正则表达式的结构树：

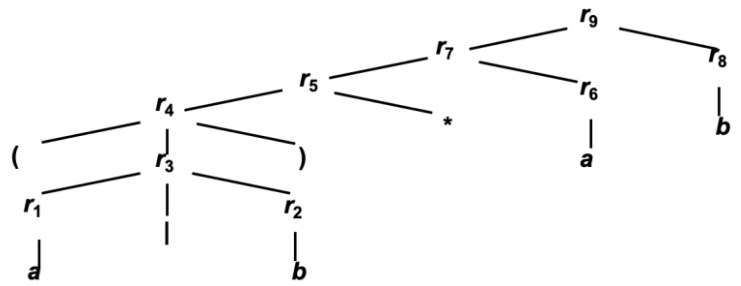


Figure 2 例：(a|b)*ab 的结构树

3. 深度遍历结构树，构建 NFA

深度优先遍历该树，每个 r 节点代表一个自动机，逐步由小的自动机，构建出完整的自动机。

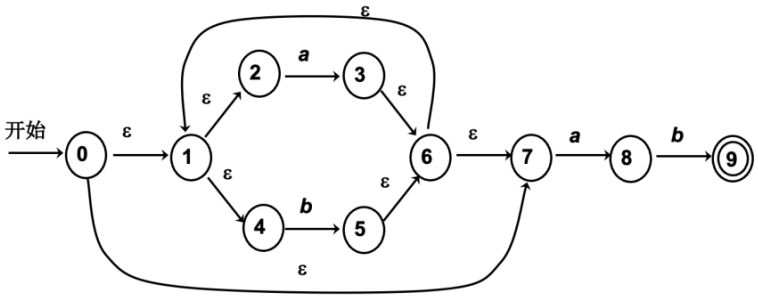


Figure 3 生成 NFA

4. NFA 到 DFA：

运用最小子集构造法将 NFA 转化为 DFA，如以上例子的 NFA 将被转化为以下具有 A，B，C，D 四个状态的 DFA。

教务处制

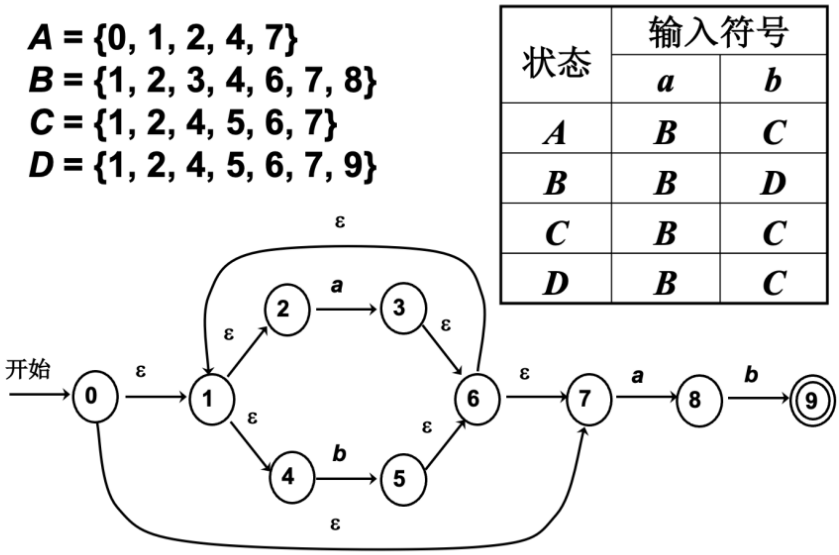


Figure 4 **构造** DFA

5. DFA 化简最简 DFA

依据可区别状态进行化简，对于不可区别的多个状态，可将此多个状态化简为一个状态。化简过程实例如下图，最下方的 DFA 有 4 个状态，经过多个步骤之后，发现状态 A 和 C 不可区分，则将 A、C 化简为一个状态，最终自动机为图中最上方的自动机。

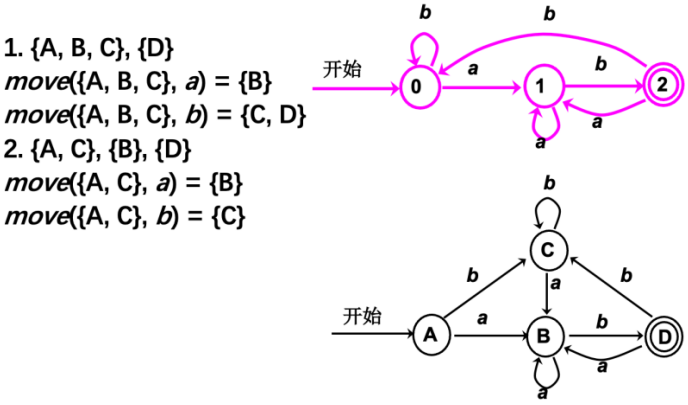


Figure 5DFA **化简最简** DFA

三. 实验步骤

1. 预处理正则表达式：
在正则表达式中显式插入连接符，从而方便构造正则表达式的结构树。

```cpp
string insertConcat(const string& regex) {
    string result;
    char prev = '\0';
    for (char c : regex) {
        if (prev != '\0') {
            // 如果前一个字符是字母数字、'*'、')'
            bool prevIsChar = isalnum(prev) || prev == RE_STAR_CHAR || prev == RE_rightb_char;
            // 如果当前字符是字母数字、'('
            bool currIsChar = isalnum(c) || c == RE_left_char;
            if (prevIsChar && currIsChar) {
                result += '.'; // 插入连接符
            }
        }
        result += c;
        prev = c;
    }
    return result;
}
```

Figure 6 在正则表达式中显式插入连接符.

2. 定义正则表达式的结构树：

```cpp
/*
定义正则表达式的节点类型，包括字符、连接、或运算和闭包。
*/
enum class RegexType {···

/*
定义正则表达式的节点结构。
*/
struct RegexNode {···
```

Figure 7 定义结构树节点类型，构造结构树（部分代码）

3. 由正则表达式构建结构树：

首先将正则表达式预处理后转化为后缀表达式，然后构造结构树。

```cpp
// 获取运算符优先级
int getPrecedence(char op) {···

// 转换为后缀表达式
string toPostfix(const string& regex) {···

/*
根据后缀表达式构建语法树。
*/
shared_ptr<RegexNode> buildSyntaxTree(const string& postfix) {···
```

Figure 8 构造结构树

4. 基于结构树，设计并构造 NFA

```
/*
定义 Token 类型的枚举。
*/
enum class TokenType {…
/*
NFA 状态结构，包含状态 ID、转换、是否为接受状态、Token 类型。
*/
struct NFAState {…
/*
NFA 结构，包含开始状态、结束状态、状态计数。
*/
struct NFA {…
/*
根据语法树构建 NFA。
*/
NFA buildNFA(const shared_ptr<RegexNode>& root) {…
```

Figure 9 **构造** NFA

5. 根据上边的 NFA 构造 DFA

```
/*
DFA 状态结构，包含状态 ID、转换、是否为接受状态、对应的 NFA 状态集合、Token 类型。
*/
struct DFAState {…
/*
DFA 结构，包含开始状态、所有状态列表、状态计数。
*/
struct DFA {…
// 计算 e-闭包
set<NFAState*> epsilonClosure(const set<NFAState*>& states) {…
// 移动函数
set<NFAState*> move(const set<NFAState*>& states, char symbol) {…
// 构建 DFA
DFA buildDFA(const NFA& nfa) {…
```

Figure 10 **构建** DFA

6. 将构建的 DFA 输出：

```
// 打印DFA
void printDFA(const DFA& dfa) {…
```

Figure 11 **输出** DFA **状态转移表**

7. 完成 TEST 的词法规则下每种词法单元的 DFA 构造
   最终合并成完成则给予自动机的词法分析器

```
402
403   // 词法分析函数
      ⚙ ∨
404 > vector<Token> lex(const DFA& dfa, const string& input) {···
485
```

Figure 12 进行词法分析

```
564     // 单字符符号
565     auto tree_singleword = parseRegex(singleword);
566     NFA nfa_singleword = buildNFA(tree_singleword);
567     nfa_singleword.end->tokenType = TokenType::SINGLEWORD;
568     nfaList.push_back(nfa_singleword);
569
570     // 除号
571     auto tree_division = parseRegex(division);
572     NFA nfa_division = buildNFA(tree_division);
573     nfa_division.end->tokenType = TokenType::DIVISION;
574     nfaList.push_back(nfa_division);
575
576     // 单字符比较符号
577     auto tree_comp_single = parseRegex(comp_single);
578     NFA nfa_comp_single = buildNFA(tree_comp_single);
579     nfa_comp_single.end->tokenType = TokenType::COMP_SINGLE;
580     nfaList.push_back(nfa_comp_single);
581
```

Figure 13 构建词法单元的 DFA（部分代码）

```
594     // 合并所有 NFA
595     NFA combinedNFA;
596     combinedNFA.start = combinedNFA.newState();
597     int stateOffset = 1;
598
599
600
601 >   for (auto& nfa : nfaList) {···
638
639     combinedNFA.stateCount = stateOffset;
640
641     // 构建 DFA
642     DFA dfa = buildDFA(combinedNFA);
643
```

Figure 14 合并所有词法单元的 DFA 构建完整的词法分析自动机

四. 实验结果及分析

1. 测试以正则表达式(a|b)*abb 以及 10|(0|11)0*1 为输入，输出各自的最简 DFA，输出形式为 DFA 的状态转移表

```
671  void wk1()
672  {
673      // 示例1: (a|b)*abb
674
675      string regex1 =(string)""+ RE_left_char+ "a|b" + RE_rightb_char + RE_STAR_CHAR + "abb";
676
677      auto tree1 = parseRegex(regex1);
678      NFA nfa1 = buildNFA(tree1);
679      DFA dfa1 = buildDFA(nfa1);
680      cout << "正则表达式: ";
681      printregex(regex1);
682      printDFA(dfa1);
683
684      cout << "\n----------------------\n";
685
686      // 示例2: 10|(0|11)0*1
687      string regex2 = (string)"10|"+RE_left_char+"0|11"+RE_rightb_char+RE_STAR_CHAR+"1";
688      auto tree2 = parseRegex(regex2);
689      NFA nfa2 = buildNFA(tree2);
690      DFA dfa2 = buildDFA(nfa2);
691      cout << "正则表达式: "<< "\n";
692      printregex(regex2);
693      printDFA(dfa2);
694  }
```

Figure 15 测试输出 DFA 状态转移表

```
正则表达式: (a|b)*abb
DFA状态转移表:
状态    是否接受    转移
S0  否      a->S2 b->S1
S1  否      a->S2 b->S1
S2  否      b->S3 a->S2
S3  否      b->S4 a->S2
S4  是      a->S2 b->S1

----------------------
正则表达式:
10|(0|11)*1
DFA状态转移表:
状态    是否接受    转移
S0  否      1->S2 0->S1
S1  否      1->S3 0->S1
S2  是      1->S5 0->S4
S3  是      1->S5
S4  是
S5  否      1->S3 0->S1
```

Figure 16 测试结果

从测试结果中来看，能够成功并正确的构造出 NFA 并转化为 DFA，最终输出状态转移表。

2. 根据 TEST 语言的如下词法规则，实现基于自动机的词法分析器。

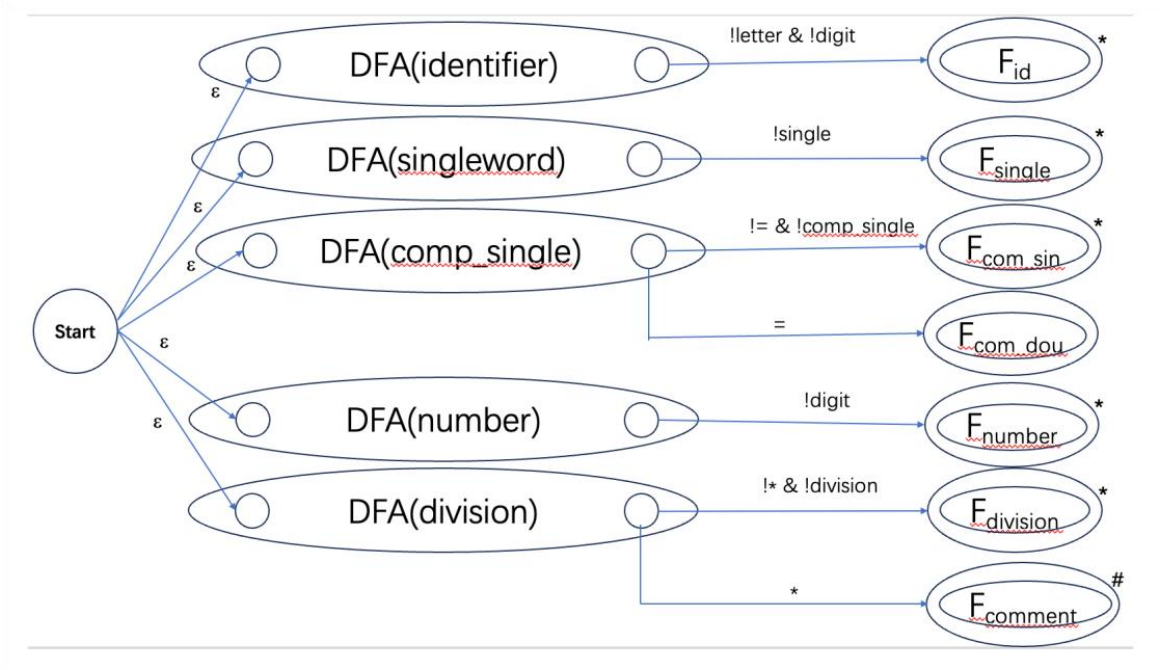


Figure 17 整体自动机

```
// 定义字母和数字
string letter = "a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|"
                "A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z";
string digit = "0|1|2|3|4|5|6|7|8|9";
;
// 定义各个 Token 的正则表达式
string number = RE_left_char + digit + RE_rightb_char+RE_left_char + digit + RE_rightb_char+RE_STAR_CHAR;
//cout<<number<<endl;
string identifier = RE_left_char + letter + RE_rightb_char+RE_left_char + letter + "|" + digit + RE_rightb_char+RE_STAR_CHAR;
string singleword = "+|-|*|(|)|{|}|:|,|;";
string division = "/";
string comp_single = "<|>|!|=";
string comp_double = ">=|<=|!=|==";
string comment_left = "/*";

// 构建各个 NFA
vector<NFA> nfaList;
```

Figure 18 词法单元正则表达式

```
// 测试词法分析器
string input = "{int a = (10);} /* this is comment */ b >= a3 + 100 ; a==3*3; ";
auto tokens = lex(dfa, input);

// 输出 Token 序列
for (const auto& token : tokens) {
    cout << "TokenType: " << tokenTypeToString(token.type) << ", \tValue: " << token.value << endl;
}
```

Figure 19 测试输入并输出词法分析结果

```
TokenType: SINGLEWORD,  Value: {
TokenType: IDENTIFIER,  Value: int
TokenType: IDENTIFIER,  Value: a
TokenType: COMP_SINGLE,     Value: =
TokenType: SINGLEWORD,  Value: (
TokenType: NUMBER,  Value: 10
TokenType: SINGLEWORD,  Value: )
TokenType: SINGLEWORD,  Value: ;
TokenType: SINGLEWORD,  Value: }
TokenType: COMMENT,     Value: /* this is comment */
TokenType: IDENTIFIER,  Value: b
TokenType: COMP_DOUBLE,     Value: >=
TokenType: IDENTIFIER,  Value: a3
TokenType: SINGLEWORD,  Value: +
TokenType: NUMBER,  Value: 100
TokenType: SINGLEWORD,  Value: ;
TokenType: IDENTIFIER,  Value: a
TokenType: COMP_DOUBLE,     Value: ==
TokenType: NUMBER,  Value: 3
TokenType: SINGLEWORD,  Value: *
TokenType: NUMBER,  Value: 3
TokenType: SINGLEWORD,  Value: ;
```

Figure 20 **词法分析测试结果**

从测试结果中可以看到，程序能够成功正常进行词法分析，覆盖测试了所有词法单元。

五. 实验总结

在本次实验中，成功实现了基于有限状态自动机的词法分析器，探索了从正则表达式到 NFA、再到 DFA 的构建过程。通过递归构建结构树，并利用子集构造法将 NFA 转化为 DFA，测试了正则表达式(a|b)*abb 和 10|(0|11)0*1，验证了程序能够正确地生成最简 DFA 并输出其状态转移表。同时，通过整合 TEST 语言的词法规则，实现了对词法单元的完整分析。实验结果表明，程序能够正确覆盖所有词法单元，达到预期效果。这次实验加深了对词法分析器和编译器前端实现的理解，进一步巩固了自动机和正则表达式的应用技能。

六. 代码

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
#include <stack>
#include <unordered_map>
```

```cpp
#include <unordered_set>
#include <set>
#include <queue>
#include <functional>
using namespace std;

#define EPSILON 52917
#define Re_star 127
#define leftb 128
#define rightb 129
const char RE_STAR_CHAR =Re_star;
const char RE_left_char = leftb;
const char RE_rightb_char = rightb;

/*
定义正则表达式的节点类型，包括字符、连接、或运算和闭包。
*/
enum class RegexType {
    CHAR,       // 单个字符
    CONCAT,     // 连接
    UNION,      // 或
    STAR        // 闭包 restar
};

/*
定义正则表达式的节点结构。
*/
struct RegexNode {
    RegexType type;
    char value; // 仅在 type == CHAR 时使用
    shared_ptr<RegexNode> left;
    shared_ptr<RegexNode> right;

    // 构造函数（char）
    RegexNode(RegexType t, char val = '\0') : type(t), value(val), left(nullptr),
right(nullptr) {}

    // 构造函数用于二元操作符（连接、或）
    RegexNode(RegexType t, shared_ptr<RegexNode> l, shared_ptr<RegexNode> r)
        : type(t), value('\0'), left(l), right(r) {}

    // 构造函数用于单元操作符（闭包 restar)
    RegexNode(RegexType t, shared_ptr<RegexNode> l)
        : type(t), value('\0'), left(l), right(nullptr) {}
};
```

```
string insertConcat(const string& regex) {
    string result;
    char prev = '\0';
    for (char c : regex) {
        if (prev != '\0') {
            // 如果前一个字符是字母数字、'*'、')'
            bool prevIsChar = isalnum(prev) || prev == RE_STAR_CHAR || prev ==
RE_rightb_char;
            // 如果当前字符是字母数字、'('
            bool currIsChar = isalnum(c) || c == RE_left_char;
            if (prevIsChar && currIsChar) {
                result += '.'; // 插入连接符
            }
        }
        result += c;
        prev = c;
    }
    return result;
}



// 获取运算符优先级
int getPrecedence(char op) {
    switch(op) {
        case RE_STAR_CHAR:
            return 3;
        case '.':
            return 2;
        case '|':
            return 1;
        default:
            return 0;
    }
}

// 转换为后缀表达式
string toPostfix(const string& regex) {
    string postfix;
    stack<char> opStack;
    for (char c : regex) {
        if (c == RE_left_char) {
            opStack.push(c);
```

```
        }
        else if (c == RE_rightb_char) {
            while (!opStack.empty() && opStack.top() != RE_left_char) {
                postfix += opStack.top();
                opStack.pop();
            }
            if (!opStack.empty()) opStack.pop(); // 弹出 '('
        }
        else if (c == RE_STAR_CHAR || c == '.' || c == '|') {
            while (!opStack.empty() && getPrecedence(opStack.top()) >=
getPrecedence(c)) {
                postfix += opStack.top();
                opStack.pop();
            }
            opStack.push(c);
        }
        else {
            postfix += c; // 操作数直接添加到后缀表达式
        }
    }
    while (!opStack.empty()) {
        postfix += opStack.top();
        opStack.pop();
    }
    ;
    return postfix;
}

/*
根据后缀表达式构建语法树。
*/
shared_ptr<RegexNode> buildSyntaxTree(const string& postfix) {
    stack<shared_ptr<RegexNode>> stack;
    for (char c : postfix) {
        if (c == RE_STAR_CHAR) {
            auto child = stack.top(); stack.pop();
            auto node = make_shared<RegexNode>(RegexType::STAR, child);
            stack.push(node);
        }
        else if (c == '.') {
            auto right = stack.top(); stack.pop();
            auto left = stack.top(); stack.pop();
            auto node = make_shared<RegexNode>(RegexType::CONCAT, left, right);
            stack.push(node);
        }
```

```
        else if (c == '|') {
            auto right = stack.top(); stack.pop();
            auto left = stack.top(); stack.pop();
            auto node = make_shared<RegexNode>(RegexType::UNION, left, right);
            stack.push(node);
        }
        else {
            auto node = make_shared<RegexNode>(RegexType::CHAR, c);
            stack.push(node);
        }
    }
    return stack.empty() ? nullptr : stack.top();
}

/*
完整的解析函数，传入正则表达式，返回语法树根节点。
*/
// shared_ptr<RegexNode> parseRegex(const string& regex) {
//     string withConcat = insertConcat(regex);
//     string postfix = toPostfix(withConcat);
//     // cout << "Postfix: " << postfix << "\n"; // 调试用
//     return buildSyntaxTree(postfix);
// }

shared_ptr<RegexNode> parseRegex(const string& regex) {
    string withConcat = insertConcat(regex);
    //cout << "With Concat: " << withConcat << endl; // 调试输出
    string postfix = toPostfix(withConcat);
    //cout << "Postfix: " << postfix << endl; // 调试输出
    return buildSyntaxTree(postfix);
}


/*
定义 Token 类型的枚举。
*/
enum class TokenType {
    NONE,
    COMMENT,        // 注释
    COMP_DOUBLE,    // 双字符比较符号
    COMP_SINGLE,    // 单字符比较符号
    DIVISION,       // 除号
    SINGLEWORD,     // 单字符符号
    NUMBER,         // 数字
    IDENTIFIER      // 标识符
};
```

```
/*
NFA 状态结构，包含状态 ID、转换、是否为接受状态、Token 类型。
*/
struct NFAState {
    int id;
    unordered_map<unsigned, vector<NFAState*>> transitions;
    bool isFinal;
    TokenType tokenType; // 新增字段

    NFAState(int id_) : id(id_), isFinal(false), tokenType(TokenType::NONE) {}
};

/*
NFA 结构，包含开始状态、结束状态、状态计数。
*/
struct NFA {
    NFAState* start;
    NFAState* end;
    int stateCount;

    NFA() : start(nullptr), end(nullptr), stateCount(0) {}

    NFAState* newState() {
        return new NFAState(stateCount++);
    }
};

/*
根据语法树构建 NFA。
*/
NFA buildNFA(const shared_ptr<RegexNode>& root) {
    NFA nfa;

    // 递归构建 NFA
    function<pair<NFAState*, NFAState*>(shared_ptr<RegexNode>)> build =
[&](shared_ptr<RegexNode> node) -> pair<NFAState*, NFAState*> {
        if (node->type == RegexType::CHAR) {
            NFAState* start = nfa.newState();
            NFAState* end = nfa.newState();
            start->transitions[node->value].push_back(end);
            return {start, end};
        }
        else if (node->type == RegexType::CONCAT) {
            auto left = build(node->left);
```

```
            auto right = build(node->right);
            left.second->transitions[EPSILON].push_back(right.first);
            return {left.first, right.second};
        }
        else if (node->type == RegexType::UNION) {
            NFAState* start = nfa.newState();
            NFAState* end = nfa.newState();
            auto left = build(node->left);
            auto right = build(node->right);
            start->transitions[EPSILON].push_back(left.first);
            start->transitions[EPSILON].push_back(right.first);
            left.second->transitions[EPSILON].push_back(end);
            right.second->transitions[EPSILON].push_back(end);
            return {start, end};
        }
        else if (node->type == RegexType::STAR) {
            NFAState* start = nfa.newState();
            NFAState* end = nfa.newState();
            auto sub = build(node->left);
            start->transitions[EPSILON].push_back(sub.first);
            start->transitions[EPSILON].push_back(end);
            sub.second->transitions[EPSILON].push_back(sub.first);
            sub.second->transitions[EPSILON].push_back(end);
            return {start, end};
        }
        return {nullptr, nullptr};
    };

    auto result = build(root);
    nfa.start = result.first;
    nfa.end = result.second;
    nfa.end->isFinal = true;
    return nfa;
}

/*
DFA 状态结构，包含状态 ID、转换、是否为接受状态、对应的 NFA 状态集合、Token 类型。
*/
struct DFAState {
    int id;
    unordered_map<unsigned, DFAState*> transitions;
    bool isFinal;
    set<NFAState*> nfaStates;
    TokenType tokenType; // 新增字段
```

```
    DFAState(int id_) : id(id_), isFinal(false), tokenType(TokenType::NONE) {}
};

/*
DFA 结构，包含开始状态、所有状态列表、状态计数。
*/
struct DFA {
    DFAState* start;
    vector<DFAState*> states;
    int stateCount;

    DFA() : start(nullptr), stateCount(0) {}

    DFAState* newState(const set<NFAState*>& nfaStates_) {
        DFAState* state = new DFAState(stateCount++);
        state->nfaStates = nfaStates_;
        // 确定 tokenType
        for (auto s : nfaStates_) {
            if (s->isFinal) {
                if (state->tokenType == TokenType::NONE || s->tokenType <
state->tokenType) {
                    state->tokenType = s->tokenType;
                }
                state->isFinal = true;
            }
        }
        states.push_back(state);
        return state;
    }
};

// 计算 e-闭包
set<NFAState*> epsilonClosure(const set<NFAState*>& states) {
    set<NFAState*> closure = states;
    stack<NFAState*> stackStates;
    for (auto s : states) stackStates.push(s);

    while (!stackStates.empty()) {
        NFAState* state = stackStates.top(); stackStates.pop();
        if (state->transitions.find(EPSILON) != state->transitions.end()) {
            for (auto next : state->transitions.at(EPSILON)) {
                if (closure.find(next) == closure.end()) {
                    closure.insert(next);
                    stackStates.push(next);
                }
```

```
            }
        }
    }
    return closure;
}

// 移动函数
set<NFAState*> move(const set<NFAState*>& states, char symbol) {
    set<NFAState*> result;
    for (auto s : states) {
        if (s->transitions.find(symbol) != s->transitions.end()) {
            for (auto next : s->transitions.at(symbol)) {
                result.insert(next);
            }
        }
    }
    return result;
}

// 构建 DFA
DFA buildDFA(const NFA& nfa) {
    DFA dfa;
    queue<set<NFAState*>> q;

    // 初始状态
    set<NFAState*> startSet = epsilonClosure({nfa.start});
    dfa.start = dfa.newState(startSet);
    q.push(startSet);

    while (!q.empty()) {
        set<NFAState*> current = q.front(); q.pop();

        DFAState* currentDFA = nullptr;
        for (auto state : dfa.states) {
            if (state->nfaStates == current) {
                currentDFA = state;
                break;
            }
        }

        // 获取所有可能的输入符号（排除 e）
        unordered_set<char> symbols;
        for (auto s : current) {
            for (auto &[c, _] : s->transitions) {
                if (c != EPSILON) symbols.insert(c);
```

```
            }
        }

        for (char symbol : symbols) {
            set<NFAState*> moveSet = move(current, symbol);
            set<NFAState*> closureSet = epsilonClosure(moveSet);

            if (closureSet.empty()) continue;

            // 检查是否已经存在
            bool found = false;
            DFAState* existingState = nullptr;
            for (auto state : dfa.states) {
                if (state->nfaStates == closureSet) {
                    existingState = state;
                    found = true;
                    break;
                }
            }

            if (!found) {
                DFAState* newDFAState = dfa.newState(closureSet);
                q.push(closureSet);
                existingState = newDFAState;
            }

            currentDFA->transitions[symbol] = existingState;
        }
    }

    return dfa;
}

// 定义 Token
struct Token {
    TokenType type;
    string value;
};

// 词法分析函数
vector<Token> lex(const DFA& dfa, const string& input) {
    bool isIncomment = false;
    vector<Token> tokens;
    size_t pos = 0;
    while (pos < input.length()) {
```

```
        DFAState* current = dfa.start;
        size_t lastAcceptPos = pos;
        TokenType lastAcceptType = TokenType::NONE;
        size_t i = pos;
        while (i < input.length()) {
            char c = input[i];
            if (current->transitions.find(c) != current->transitions.end()) {
                current = current->transitions[c];
                i++;
                if (current->isFinal) {
                    lastAcceptPos = i;
                    lastAcceptType = current->tokenType;
                }
            } else {
                break;
            }
        }
        if (lastAcceptType != TokenType::NONE) {

            if(lastAcceptType==TokenType::DIVISION and
lastAcceptPos<input.length() and  input[lastAcceptPos]=='*')
            {
                lastAcceptType=TokenType::COMMENT;
            }

            if(lastAcceptType==TokenType::COMMENT)
            {
                int endpos=i+1;
                while(endpos+1<input.length() and input[endpos]!='*' and
input[endpos+1]!='/')endpos++;
                endpos++;
                endpos++;
                lastAcceptPos=endpos;
            }
            else if(lastAcceptType==TokenType::COMP_SINGLE or
lastAcceptType==TokenType::COMP_DOUBLE or lastAcceptType==TokenType::SINGLEWORD)
            {
                int endpos=pos;
                if(input[pos]=='>' or input[pos]=='<' or input[pos]=='=' or
input[pos]=='!')
                {
                    if(pos+1<input.length() and input[pos+1]=='=')
                    {
                        endpos=pos+2;
                        lastAcceptPos=endpos;
```

```
                    lastAcceptType=TokenType::COMP_DOUBLE;
                }
            }

            if(lastAcceptType==TokenType::COMP_DOUBLE and
lastAcceptPos-1==pos)
            {
                lastAcceptType=TokenType::COMP_SINGLE;
            }
        }

        if(lastAcceptType==TokenType::COMMENT and input[pos]=='*')
        {
            lastAcceptType=TokenType::SINGLEWORD;
            lastAcceptPos=pos+1;
        }

        //cout<<input[pos]<<endl;
        string tokenValue = input.substr(pos, lastAcceptPos - pos);
        tokens.push_back({ lastAcceptType, tokenValue });
        pos = lastAcceptPos;

    } else {
        // 无法识别的字符，跳过或报错
        if(input[pos]==' ')
        {
            pos++;
            continue;
        }

        cout << "无法识别的字符: " << input[pos] << endl;
        pos++;
    }
}
    return tokens;
}

// 将 TokenType 转换为字符串，便于输出
string tokenTypeToString(TokenType type) {
    switch (type) {
        case TokenType::IDENTIFIER:
            return "IDENTIFIER";
        case TokenType::NUMBER:
            return "NUMBER";
        case TokenType::SINGLEWORD:
```

```
            return "SINGLEWORD";
        case TokenType::DIVISION:
            return "DIVISION";
        case TokenType::COMP_SINGLE:
            return "COMP_SINGLE";
        case TokenType::COMP_DOUBLE:
            return "COMP_DOUBLE";
        case TokenType::COMMENT:
            return "COMMENT";
        default:
            return "NONE";
    }
}


// 打印 DFA
void printDFA(const DFA& dfa) {
    cout << "DFA 状态转移表:\n";
    cout << "状态\t 是否接受\t 转移\n";
    for (auto state : dfa.states) {
        cout << "S" << state->id << "\t" << (state->isFinal ? "是" : "否") << "\t\t";
        for (auto &[c, next] : state->transitions) {
            cout << (char)c << "->S" << next->id << " ";
        }
        cout << "\n";
    }
}


// 测试函数
void wk2() {
    ;
    // 定义字母和数字
    string letter = "a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|"
                    "A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z";
    string digit = "0|1|2|3|4|5|6|7|8|9";
    ;
    // 定义各个 Token 的正则表达式
    string number = RE_left_char + digit + RE_rightb_char+RE_left_char + digit +
RE_rightb_char+RE_STAR_CHAR;
    //cout<<number<<endl;
    string identifier = RE_left_char + letter + RE_rightb_char+RE_left_char + letter
+ "|" + digit + RE_rightb_char+RE_STAR_CHAR;
    string singleword = "+|-|*|(|)|{|}|:|,|;";
    string division = "/";
    string comp_single = "<|>|!|=";
```

```
    string comp_double = ">=|<=|!=|==";
    string comment_left = "/*";

    // 构建各个 NFA
    vector<NFA> nfaList;

    // 标识符
    auto tree_identifier = parseRegex(identifier);
    NFA nfa_identifier = buildNFA(tree_identifier);
    nfa_identifier.end->tokenType = TokenType::IDENTIFIER;
    nfaList.push_back(nfa_identifier);

    // 数字
    auto tree_number = parseRegex(number);
    if (!tree_number) {
        cout << "Failed to parse the number regex." << endl;
        return;
    }
    NFA nfa_number = buildNFA(tree_number);
    if (!nfa_number.end) {
        cout << "Failed to build NFA for the number regex." << endl;
        return;
    }
    nfa_number.end->tokenType = TokenType::NUMBER;
    nfaList.push_back(nfa_number);

    // 单字符符号
    auto tree_singleword = parseRegex(singleword);
    NFA nfa_singleword = buildNFA(tree_singleword);
    nfa_singleword.end->tokenType = TokenType::SINGLEWORD;
    nfaList.push_back(nfa_singleword);

    // 除号
    auto tree_division = parseRegex(division);
    NFA nfa_division = buildNFA(tree_division);
    nfa_division.end->tokenType = TokenType::DIVISION;
    nfaList.push_back(nfa_division);

    // 单字符比较符号
    auto tree_comp_single = parseRegex(comp_single);
    NFA nfa_comp_single = buildNFA(tree_comp_single);
    nfa_comp_single.end->tokenType = TokenType::COMP_SINGLE;
    nfaList.push_back(nfa_comp_single);

    // 双字符比较符号
```

```
    auto tree_comp_double = parseRegex(comp_double);
    NFA nfa_comp_double = buildNFA(tree_comp_double);
    nfa_comp_double.end->tokenType = TokenType::COMP_DOUBLE;
    nfaList.push_back(nfa_comp_double);
    ;
    // 注释
    auto tree_comment = parseRegex(comment_left);
    NFA nfa_comment = buildNFA(tree_comment);
    nfa_comment.end->tokenType = TokenType::COMMENT;
    nfaList.push_back(nfa_comment);
    ;
    // 合并所有 NFA
    NFA combinedNFA;
    combinedNFA.start = combinedNFA.newState();
    int stateOffset = 1;



    for (auto& nfa : nfaList) {
        if (nfa.start == nullptr) {
            cout << "Error: NFA start state is null." << endl;
            continue;
        }

    unordered_set<NFAState*> visitedAdjust;

    function<void(NFAState*)> adjustStateID = [&](NFAState* state) {
        if (state == nullptr || visitedAdjust.count(state)) return;
        visitedAdjust.insert(state);
        state->id += stateOffset;
        for (auto& [c, vec] : state->transitions) {
            for (auto& nextState : vec) {
                adjustStateID(nextState);
            }
        }
    };
    adjustStateID(nfa.start);

    unordered_set<NFAState*> visitedRestore;

    function<void(NFAState*)> restoreStateID = [&](NFAState* state) {
        if (state == nullptr || visitedRestore.count(state)) return;
        visitedRestore.insert(state);
        state->id = -state->id;
        for (auto& [c, vec] : state->transitions) {
```

```
            for (auto& nextState : vec) {
                restoreStateID(nextState);
            }
        }
    };
    restoreStateID(nfa.start);

    combinedNFA.start->transitions[EPSILON].push_back(nfa.start);
    stateOffset += nfa.stateCount;
    }

    combinedNFA.stateCount = stateOffset;

    // 构建 DFA
    DFA dfa = buildDFA(combinedNFA);

    // 测试词法分析器
    string input = "{int a = (10);} /* this is comment */ b >= a3 + 100 ; a==3*3;
";
    auto tokens = lex(dfa, input);

    // 输出 Token 序列
    for (const auto& token : tokens) {
        cout << "TokenType: " << tokenTypeToString(token.type) << ", \tValue: " <<
token.value << endl;
    }

    //printDFA(dfa);

}



void printregex(string s)
{
    for(auto i:s)
    {
        if(i==RE_left_char)cout<<"(";
        else if(i==RE_rightb_char)cout<<")";
        else if(i==RE_STAR_CHAR)cout<<"*";
        else cout<<i;
    }
    cout<<endl;
}

void wk1()
```

```
{
    // 示例 1: (a|b)*abb

    string regex1 =(string)""+ RE_left_char+ "a|b" + RE_rightb_char + RE_STAR_CHAR
+ "abb";

    auto tree1 = parseRegex(regex1);
    NFA nfa1 = buildNFA(tree1);
    DFA dfa1 = buildDFA(nfa1);
    cout << "正则表达式: ";
    printregex(regex1);
    printDFA(dfa1);

    cout << "\n----------------------\n";

    // 示例 2: 10|(0|11)0*1
    string regex2 =
(string)"10|"+RE_left_char+"0|11"+RE_rightb_char+RE_STAR_CHAR+"1";
    auto tree2 = parseRegex(regex2);
    NFA nfa2 = buildNFA(tree2);
    DFA dfa2 = buildDFA(nfa2);
    cout << "正则表达式: "<< "\n";
    printregex(regex2);
    printDFA(dfa2);
}


int main() {
    wk1();
    wk2();
    return 0;
}
```