OT1 实现词法分析器构造算法

2112079 朱奕翔

Thompson算法代码实现部分

步骤

1. 编写C++代码实现Thompson算法，将正则表达式转换为NFA。
2. 实现了NFA的构建过程，包括处理连接（.）、闭包（*）和选择（|）等操作。
3. 将中缀正则表达式转换为后缀表达式，并进一步将后缀表达式转换为NFA。
4. 输出NFA的节点、边、状态数等信息到文件。

实现部分

1. Edge类中存了边的上一个节点和下一个节点的信息，以及边上的字符，#表示空字符，另外 ' '表示该边不是一个原子边，是N（）。

```
class Edge {
public:
    Edge() {
        this->begin = "";
        this->thro = ' ';
        this->end = "";
    };

    Edge(string begin, char thro, string end) {
        this->begin = begin;
        this->thro = thro;
        this->end = end;
    }

    void setThro(char ch) {
        this->thro = ch;
    }

    char getThro() {
        return this->thro;
    }

    void setBegin(string ch) {
        this->begin = ch;
    }

    string getBegin() {
        return this->begin;
    }
```

```
    void setEnd(string ch) {
        this->end = ch;
    }

    string getEnd() {
        return this->end;
    }


private:
    char thro;
    string begin;
    string end;

};
```

2. 封装了一个类，此处先介绍RTN的成员变量，zf是字符数，edgenum是边数，Begin是用来记录开始节点的，zfj是字符集。其实此处的成员变量都是为后续的NFATODFA服务的，只讨论Thompson算法时并不需要用到。

```
class RTN {

public:
    int zf = 0;
    int edgenum = 0;
    string Begin;
    char zfj[101];
    RTN() {};//构造函数不做任何处理
}
```

3. 判断是否是字母和是否合法的函数不过多介绍，是RTN类中的成员函数

```
bool isletter(char ch) {//判断是否为字母
        if (ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z') {
            return true;
        }
        else {
            return false;
        }
    }

    bool islegal(string s) {//判断字符串是否符合正规式的语法
        int Lbracket = 0;//记录左括号的个数
        int Rbracket = 0;//记录右括号的个数
        RTN r;
        for (int i = 0; i < s.length(); i++) {
            char ch = s[i];
            if (ch == '*' || ch == '|' || ch == '.') {
```

```
                continue;
            }
            //数字不考虑判断
            else if (ch >= 0 && ch <= 9) {
                continue;
            }//字母不考虑判断
            else if (r.isletter(ch)) {
                continue;
            }//如果输入左括号，左括号个数+1
            else if (ch == '(') {
                Lbracket++;
            }//如果输入右括号，右括号个数+1
            else if (ch == ')') {
                Rbracket++;
            }//如果输入的符号不符合以上情况，直接返回不合法
            else {
                return false;
            }
        }
        //如果左括号个数不等于右括号个数，返回不合法，反之返回合法
        if (Lbracket != Rbracket) {
            return false;
        }
        else {
            return true;
        }
    }
```

4. 由于正规式不包含连接字符，为了机器方便处理，将输入的表达式中的连接符号用"."补全。

```
//将输入的表达式中的连接符号用'.'补全
    string turnToConnect(string s) {
        string ns = s.substr(0, 1);
        for (int i = 1; i < s.length(); i++) {
            char prech = s[i - 1];
            char ch = s[i];
            //连续的字母之间需要添加上连接符号'.'
            if (isletter(prech) && isletter(ch)) {
                ns = ns + '.' + ch;
                //cout << endl << ns;
                continue;
            }
            //判断当前是字母，并且前面是右括号')'的情况
            else if (isletter(ch) && prech == ')') {
                ns = ns + '.' + ch;
                continue;
            }
            //判断当前是左括号'('，并且前面是右括号')'或者前面是字母的情况
            else if (ch == '(' && prech == ')' || ch == '(' && isletter(prech)) {
                ns = ns + '.' + ch;
                continue;
            }
```

```
            else {
                ns += ch;
                //cout << endl << ns;
                continue;
            }
        }
        return ns;
    }
```

5. 转为后缀表达式可以更方便的处理操作优先级的问题，因此此处有一个将中缀表达式转为后缀表达式的成员函数。

```
//转为后缀表达式
    string ReversePolishType(string s) {
        stack<char> st;//栈用来将中缀表达式转为后缀表达式
        string ns = "";//结果字符串

        for (int i = 0; i < s.length(); i++) {//循环遍历字符串中的每个字符
            char ch = s[i];
            if (isletter(ch)) {//如果是字母，直接加到结果字符串中
                ns = ns + ch;
            }
            else if (ch == '(') {//如果是左括号直接入栈
                st.push(ch);
            }
            else if (ch == ')') {//如果是右括号将栈中内容弹出加入到结果字符串中，直到碰到左括号
                while (st.top() != '(') {
                    ns = ns + st.top();
                    st.pop();
                }
                st.pop();//将栈顶剩余的左括号弹出
            }
            else if (ch == '*') {
                //栈空或者优先级高于栈顶操作符时直接入栈
                if (st.empty() || st.top() == '.' || st.top() == '|' || st.top() == '(') {
                    st.push(ch);
                }
                else {//出现其他情况只可能是栈顶是*，所以循环将栈顶的*弹出，直到栈空或者优先级高于栈顶操作符时入栈
                    while (st.top() == '*') {
                        ns = ns + st.top();
                        st.pop();
                        if (st.empty()) break;
                    }
                    if (st.empty() || st.top() == '.' || st.top() == '|' || st.top() == '(') {
                        st.push(ch);
                    }
                }
```

```
            }
            else if (ch == '.') {//除了优先级有区别其他同理
                if (st.empty() || st.top() == '|' || st.top() == '(') {
                    st.push(ch);
                }
                else {
                    while (st.top() == '*' || st.top() == '.') {
                        ns = ns + st.top();
                        st.pop();
                        if (st.empty()) break;
                    }
                    if (st.empty() || st.top() == '|' || st.top() == '(') {
                        st.push(ch);
                    }
                }
            }
            else if (ch == '|') {//除了优先级有区别其他同理
                if (st.empty() || st.top() == '(') {
                    st.push(ch);
                }
                else {
                    while (st.top() == '*' || st.top() == '.' || st.top() == '|')
{
                        ns = ns + st.top();
                        st.pop();
                        if (st.empty()) break;
                    }
                    if (st.empty() || st.top() == '(') {
                        st.push(ch);
                    }
                }
            }
        }
        while (!st.empty()) {//如果读取完所有字符后栈中还有操作符，全部弹出
            ns = ns + st.top();
            st.pop();
        }

        return ns;
    }
```

6. 此处是核心代码，要对或，闭包，连接三种操作进行NFA的构建，从后缀表达式读取到字符时要创建一条边，同时根据Aflag指明前后节点，对于连接操作要将两个Edge对象通过一个新的Edge对象关联起来，或操作通过创建4个Edge对象按照Thompson构造法构造一个新的Edge，闭包同理。

```
//处理 |
    Edge Unite(Edge Left, Edge Right) {
        Edge ed1(to_string(Aflag), '#', Left.getBegin());
        Edge ed2(to_string(Aflag), '#', Right.getBegin());
        Edge ed3(Left.getEnd(), '#', to_string(Aflag + 1));
        Edge ed4(Right.getEnd(), '#', to_string(Aflag + 1));
```

```
        q.push(ed1);
        q.push(ed2);
        q.push(ed3);
        q.push(ed4);
        //thro不是空格说明是原子边，需要将边入队列
        if (Left.getThro() != ' ') {
            this->q.push(Left);
        }
        if (Right.getThro() != ' ') {
            this->q.push(Right);
        }
        //得到的新的边将thro置为空格以和原子边进行区分
        Edge ed(to_string(Aflag), ' ', to_string(Aflag + 1));
        Aflag = Aflag + 2;
        return ed;
    }

    //处理 连接
    Edge Join(Edge Left, Edge Right) {

        Edge ed1(Left.getEnd(), '#', Right.getBegin());
        q.push(ed1);
        if (Left.getThro() != ' ') {
            q.push(Left);
        }
        if (Right.getThro() != ' ') {
            q.push(Right);
        }


        Edge ed(Left.getBegin(), ' ', Right.getEnd());
        return ed;
    }

    //处理 闭包
    Edge Self(Edge edge) {

        Edge ed1(to_string(Aflag), '#', edge.getBegin());
        Edge ed2(edge.getEnd(), '#', to_string(Aflag + 1));
        Edge ed3(edge.getEnd(), '#', edge.getBegin());
        Edge ed4(to_string(Aflag), '#', to_string(Aflag+1));
        this->q.push(ed1);
        this->q.push(ed2);
        this->q.push(ed3);
        this->q.push(ed4);

        if (edge.getThro() != ' ') {
            this->q.push(edge);
        }

        Aflag = Aflag + 2;
        Edge ed(to_string(Aflag), ' ', to_string(Aflag + 1));
        return ed;
    }
```

```
    //传入的参数为后缀表达式，此处为Thompson算法的核心部分
    void PolishTypeToNFA(string s) {
        stack<Edge> st;
        Aflag = 0;
        for (int i = 0; i < s.length(); i++) {
            char ch = s[i];

            if (this->isletter(ch)) {//当字符为字母时，创建一个Edge对象，根据AFlag创
建不同状态的节点，并将thro设置为该字符
                Edge ed(to_string(Aflag), ch, to_string(Aflag + 1));
                Aflag = Aflag + 2;
                st.push(ed);//加入栈中，因为此处相当于将后缀表达式的计算，需要用栈来进
行
            }
            else if (ch == '*') {
                zf++;
                Edge ed = st.top();//单目运算符弹出栈顶一个元素
                st.pop();
                st.push(this->Self(ed));//压入新构建的闭包正规式
                Begin = to_string(Aflag - 2);
            }
            else if (ch == '.') {
                zf++;
                Edge ed1 = st.top();
                st.pop();
                Edge ed2 = st.top();
                st.pop();//连接是双目运算符，所以要弹出两个元素
                Edge ed = Join(ed2, ed1);
                st.push(ed);//将进行完连接运算的元素
                Begin = ed2.getBegin();
            }
            else if (ch == '|') {
                zf++;
                Edge ed1 = st.top();
                st.pop();
                Edge ed2 = st.top();
                st.pop();
                st.push(this->Unite(ed1, ed2));//与连接同理
                Begin = to_string(Aflag - 2);
            }
        }

        while (!this->q.empty())
        {
            Edge e = q.front();
            p.push(e);
            zf++;
            edgenum++;
            q.pop();
        }
        outfile << 53 << endl; //分别为字符数，节点数，开始节点状态，结束终止节点状态，
边数
        for (char letter = 'a'; letter <= 'z'; letter++) {
            outfile << letter << ' ';
```

```
        }
        for (char letter = 'A'; letter <= 'Z'; letter++) {
            outfile << letter << ' ';
        }
        outfile << '#' << ' ';
        outfile << Aflag << ' ' << Begin << ' ' << Aflag - 1 << ' ' << edgenum <<
endl;
        while (!this->p.empty())
        {
            Edge e = p.front();
            outfile << e.getBegin()  << ' ' << e.getThro() << ' ' << e.getEnd()
<< endl;

            cout << endl << "开始结点为: " << e.getBegin() << "     转换过程为: " <<
e.getThro() << "     结束结点为: " << e.getEnd();
            p.pop();//弹出队列元素并输出
        }

        outfile.close();
    }
```

子集构造实现部分

1. 初始化各种数据结构： 在类 NTD 中初始化了一些数据结构，如字符数 zf，字符集 zfj，状态数 zt，起始和结束态 Begin 和 End，存储边信息的二维数组 G，以及一些辅助数据结构。

```
class NTD {
public:
    struct Node {
        //DFA节点
        int s;//对s进行位操作可以管理包含了哪些NFA节点
        bool flag;
        //标记一个DFA节点集合是否包含NFA结束态,即表示DFA中是否是一个结束态
        Node(int ss, bool f) {//构造函数初始化
            s = ss;
            flag = f;
        }
    };

    struct Edge2 {
        //DFA边
        int from, to;//前一个节点和下一个节点
        char c;//转移边上的字符
        Edge2(int x, int y, char z) {//构造函数
            from = x;
            to = y;
            c = z;
        }
    };
```

```
    //字符数
    int zf;
    //字符集
    char zfj[MAX];
    //状态数，0开始
    int zt;
    //起始和结束态
    int Begin, End;
    //存边
    char G[MAX][MAX];
    //标记NFA状态是否被访问，求闭包用到
    int vis[MAX];
    //DFA节点集
    vector<Node> V;
    //DFA边集
    vector<Edge2> edge;
    //求过的闭包保存以后用
    int eps_clos[MAX];
    queue<int> eq;
}
```

2. 计算 eps 闭包： E_closure 函数用于计算ε（空字符）闭包。通过广度优先搜索（BFS）来寻找从某个状态出发可以通过ε转移到的所有状态，并用位运算来表示这些状态的集合。

```
//求eps闭包
    int E_closure(int x) {
        if (eps_clos[x] != -1) {
            return eps_clos[x];//如果已经计算过闭包的，直接返回即可
        }
        queue<int> q;
        memset(vis, 0, sizeof(vis));//重置vis数组用来计算闭包
        int S = 0;//初始化为0，后续需要进行位操作
        S = S | (1 << x);//NFA中的状态数是几就移动几位，后续只要判断这个位上是不是1就好
了
        q.push(x);//为BFS做准备
        while (!q.empty()) {
            //BFS求闭包
            int v = q.front();
            q.pop();//弹出队列列首
            for (int w = 0; w < zt; w++) {//循环遍历NFA
                if (G[v][w] == '#' && !vis[w]) {//如果两个节点之间是空字符且该节点未
被访问
                    vis[w] = 1;//标记该节点被访问了
                    S |= (1 << w);//并且通过移位标记
                    q.push(w);//放入队列中继续BFS
                }
            }
        }
        eps_clos[x] = S;//BFS结束后
        return S;
    }
```

3. 计算状态集经过字符的转移： set_edge 函数用于计算一个状态集经过某个字符 c 转移到的状态集。它遍历状态集中的每个状态，并查找从该状态出发通过字符 c 可以到达的所有状态，再通过 ε 闭包找到这些状态的集合。

```
    int set_edge(int s, char c) {//定义一个函数，用于求一个状态集经过字符 c 转移到的状态集。
        //求一个状态集吃字符到达的状态集

        int i, j;
        int S = 0;
        for (i = 0; i < zt; i++) {//遍历整个NFA的节点
            if ((s >> i) & 1) {//如果匹配到了输入状态集的NFA节点
                for (j = 0; j < zt; j++) {//再遍历整个NFA

                    if (G[i][j] == c)//如果该节点到另一个节点有转移边上是字符c的
                        S |= E_closure(j);//将该节点的闭包加入字符集
                }
            }
        }
        return S;//返回字符集
    }
```

4. 检查 DFA 节点是否已存在： check 函数用于检查 DFA 节点集中是否已经包含了某个状态集。

```
bool check(int s) {
        //检查DFA节点集是否出现过
        for (int i = 0; i < V.size(); i++) {
            if (V[i].s == s) return true;
        }
        return false;
    }
```

5. 判断状态集是否包含结束态： is_end 函数用于判断一个状态集是否包含 NFA 的结束态。

```
bool is_end(int s) {
        //状态集是否包含终结点
        return (s >> End) & 1;
    }
```

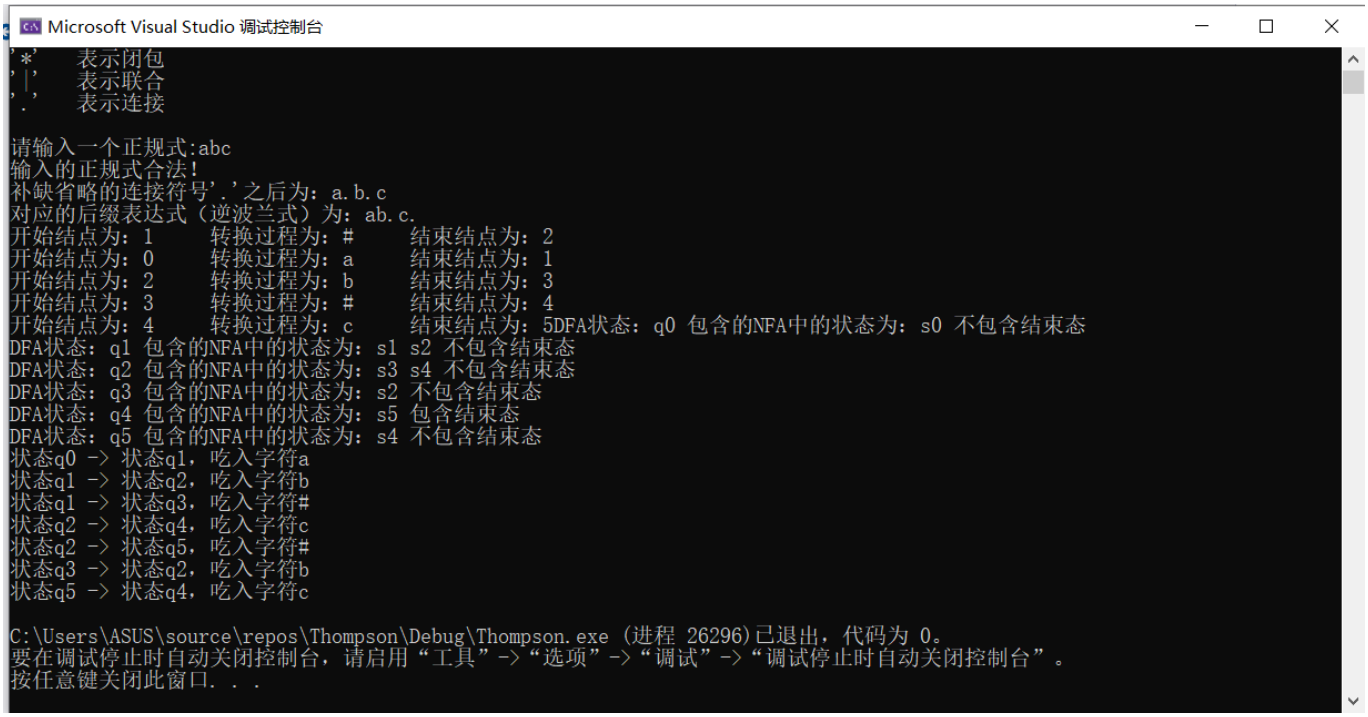
6. 子集构造算法： ZJGZ 函数是子集构造算法的核心。它初始化一个队列 work，将初始状态的 ε 闭包加入 DFA 节点集，并标记是否包含结束态。然后循环处理队列 work 中的状态集，对每个字符计算状态集经过字符的转移，生成 DFA 边并将新的状态集加入 DFA 节点集。

```
void ZJGZ() {
        //子集构造算法
        int i;
        queue<int> work;

        work.push(E_closure(Begin));//定义一个队列 work，用于存储待处理的状态集。
        //加入DFA节点集
        V.push_back(Node(E_closure(Begin), is_end(E_closure(Begin))));//将初始状态的闭包加入DFA节点集，并标记是否包含终结点。
        while (!work.empty()) {//循环处理未处理的状态集

            int v = work.front();//取出队头的状态集

            work.pop();
            for (i = 0; i < zf; i++) {
                //遍历字符集
                //生成NFA吃完该字符所能到达的所有状态
                int s = set_edge(v, zfj[i]);
                if (s != 0) {
                    edge.push_back(Edge2(v, s, zfj[i]));////如果能到达的状态集不为空，说明有边，将这条边加入边集
                    if (!check(s)) {//如果该节点没在DFA节点集合里出现过
                        V.push_back(Node(s, is_end(s)));//将该节点加入DFA节点集合
                        work.push(s);//并且加入未处理的状态集合中
                    }
                }
            }
        }
    }
```

7. 输出 DFA 节点信息： out_put 函数用于输出 DFA 节点的信息，包括节点编号、包含的 NFA 状态，以及是否包含结束态。

```
void out_put(int i, int s, bool f) {
        printf("DFA状态: q%d 包含的NFA中的状态为: ", i);
        for (int j = 0; j < zt; j++)
            if ((s >> j) & 1) printf("s%d ", j);
        if (f) printf("包含结束态\n");
        else printf("不包含结束态\n");
    }
```

结果演示



前半部分展示了如何将正规式转换为NFA，后半部分根据NFA再构造出DFA。