词法分析器核心算法实现

RE转为NFA

其中数据结构 `struct edge` 来表示 NFA 的边。每个边有三个属性：`beginID`（开始状态的标识）、`trans`（边上的字符转移符号，等于 '=' 表示 ε-转移）、`endID`（结束状态的标识）。

```
struct edge
{
    int beginID = -1;
    char trans;
    int endID = -1;
};
```

`isLegal` 函数用于检查输入的正则表达式是否合法。

```
bool isLegal(string s)
{
    stack<char> bracket; // 用于检查括号是否匹配
    for (int i = 0; i < s.length(); i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z') // 如果是字母（操作数）
            continue;
        else if (s[i] == '|' || s[i] == '*' || s[i] == '.') // 如果是运算符（'|', '*', '.'）
            continue;
        else if (s[i] == '(') // 检查括号是否匹配
            bracket.push('(');
        else if (s[i] == ')')
        {
            if (bracket.empty())
                return false; // 括号不匹配，返回false
            else
                bracket.pop(); // 匹配成功，弹出左括号
        }
        else
            return false; // 不是合法字符，返回false
    }
    if (!bracket.empty())
        return false; // 括号未完全匹配，返回false

    for (int i = 0; i < s.length(); i++)
    {
        if (s[i] == '|') // '|' 不能作为正则表达式的第一个或最后一个字符，也不能紧跟在 '|', '*', ')', 或 '.' 之后
        {
            if (i == 0 || i == s.length() - 1)
                return false;
            else if (s[i + 1] == '|' || s[i + 1] == '*' || s[i + 1] == '.' || s[i + 1] == ')')
                return false;
        }
```

```
        else if (s[i] == '.') // '.' 不能作为正则表达式的第一个或最后一个字符，也不能紧跟在 '|', '*', ')', 或 '.' 之后
        {
            if (i == 0 || i == s.length() - 1)
                return false;
            else if (s[i + 1] == '|' || s[i + 1] == '*' || s[i + 1] == '.' || s[i + 1] == ')')
                return false;
        }
        else if (s[i] == '*') // '*' 不能作为正则表达式的第一个字符，也不能连续出现
        {
            if (i == 0)
                return false;
            else if (s[i + 1] == '*')
                return false;
        }
    }

    return true; // 正则表达式合法
}
```

`precedence` 函数用于确定运算符的优先级。它返回一个整数，其中 `*` 的优先级最高，`.` 的次之，`|` 的优先级最低。

```
int precedence(char c)
{
    if (c == '*')
    {
        return 3;
    }
    else if (c == '.')
    {
        return 2;
    }
    else if (c == '|')
    {
        return 1;
    }
    else
    {
        return -1;
    }
}
```

`toinfix` 函数将输入的正则表达式转换为中缀表达式，并插入 `.` 运算符以便于后续处理。

```
string toinfix(string expression)
{
    string infix;
    for (int i = 0; i < expression.length(); i++)
    {
        char tmp = expression[i];
        char next;
```

```
        if (i == expression.length() - 1)
        {
            next = '\0'; //结束
        }
        else
        {
            next = expression[i + 1];
        }
        if (((tmp != '(' && tmp != '.' && tmp != '|') || tmp == ')' || tmp ==
'*') && (next != ')' && next != '*' && next != '|' && next != '.' && next !=
'\0'))
        {
            infix = infix + tmp + '.';
        }
        else
        {
            infix = infix + tmp;
        }
    }
    return infix;
}
```

`tosuffix` 函数将中缀表达式转换为后缀表达式（逆波兰式）。它使用栈来处理运算符的优先级，确保正确的运算符顺序。

```
string tosuffix(string infix)
{
    stack<char> op; // 用于存储运算符（'(', '.', '*')
    string suffix;   // 存储后缀表示的正则表达式

    for (int i = 0; i < infix.length(); i++)
    {
        char tmp = infix[i];
        if (tmp == '(')
        {
            op.push(tmp); // 如果是左括号，入栈
        }
        else if (tmp == ')')
        {
            // 如果是右括号，将栈顶的运算符出栈，直到遇到左括号
            while (op.top() != '(')
            {
                suffix = suffix + op.top(); // 将运算符加入后缀表示
                op.pop();
            }
            op.pop(); // 移除左括号
        }
        else if (tmp == '*' || tmp == '.' || tmp == '|')
        {
            // 如果是运算符（'*', '.', '|'）
            // 弹出栈中优先级高于等于当前运算符的运算符
            while (!op.empty() && op.top() != '(' && precedence(tmp) <=
precedence(op.top()))
            {
```

```
                suffix = suffix + op.top(); // 将运算符加入后缀表示
                op.pop();
            }
            op.push(tmp); // 当前运算符入栈
        }
        else
        {
            suffix = suffix + tmp; // 如果是字母（操作数），直接加入后缀表示
        }
    }

    // 将栈中剩余的运算符依次加入后缀表示
    while (!op.empty())
    {
        suffix = suffix + op.top();
        op.pop();
    }

    return suffix; // 返回后缀表示的正则表达式
}
```

`Tonfa` 函数接受后缀表达式，将其转换为 NFA。它遍历后缀表达式中的字符和运算符，根据每个字符和运算符的类型构建 NFA 边。具体处理如下：

- 对于小写字母字符，创建一个新的 NFA 边，标识为一个状态，具有一个转移字符，并将其起始状态和结束状态压入栈中。
- 对于 `|` 运算符，创建四个 ε-转移的边，将起始状态和结束状态入栈。
- 对于 `.` 运算符，合并前两个状态的结束状态和第三个状态的起始状态，将起始状态和结束状态入栈。
- 对于 `*` 运算符，创建四个 ε-转移的边，将起始状态和结束状态入栈。

```
void Tonfa(string suffix)
{
    vector<edge>().swap(nfa); // 初始化NFA
    int ID = -1;
    stack<int> beginst; // 用于存储边的起始ID
    stack<int> endst;   // 用于存储边的结束ID

    for (int i = 0; i < suffix.length(); i++)
    {
        char tmp = suffix[i];
        if (tmp >= 97 && tmp <= 122) // 如果当前字符是字母，创建一条新边
        {
            ID++;
            edge e;
            e.trans = tmp;
            e.beginID = ID;
            beginst.push(ID); // 存储边的起始ID
            ID++;
            e.endID = ID;
            nfa.push_back(e);
            endst.push(ID); // 存储边的结束ID
```

```
        }
        else if (tmp == '|')
        {
            ID++;
            edge e1;
            edge e2;
            edge e3;
            edge e4;
            e1.trans = e2.trans = e3.trans = e4.trans = '='; // ε（空转移）
            e1.beginID = e2.beginID = ID;
            e2.endID = beginst.top();
            beginst.pop();
            e1.endID = beginst.top();
            beginst.pop();
            e4.beginID = endst.top();
            endst.pop();
            e3.beginID = endst.top();
            endst.pop();
            beginst.push(ID);
            ID++;
            endst.push(ID);
            e3.endID = e4.endID = ID;
            nfa.push_back(e1);
            nfa.push_back(e2);
            nfa.push_back(e3);
            nfa.push_back(e4);
        }
        else if (tmp == '.')
        {
            int temp = endst.top();
            endst.pop();
            for (int i = 0; i < nfa.size(); ++i)
            {
                if (nfa[i].beginID == beginst.top())
                    nfa[i].beginID = endst.top();
            }
            beginst.pop();
            endst.pop();
            endst.push(temp);
        }
        else if (tmp == '*')
        {
            ID++;
            edge e1;
            edge e2;
            edge e3;
            edge e4;
            e1.trans = e2.trans = e3.trans = e4.trans = '='; // ε（空转移）
            e1.beginID = ID;
            e3.beginID = ID;
            e1.endID = beginst.top();
            e4.endID = beginst.top();
            beginst.pop();
            e2.beginID = endst.top();
            e4.beginID = endst.top();
            endst.pop();
```

```
            beginst.push(ID); // 存储当前NFA的起始状态
            ID++;
            endst.push(ID); // 存储当前NFA的结束状态
            e2.endID = ID;
            e3.endID = ID;
            nfa.push_back(e1);
            nfa.push_back(e2);
            nfa.push_back(e3);
            nfa.push_back(e4);
        }
        else
        {
            cout << "error" << endl;
        }
    }
    sort(nfa.begin(), nfa.end(), [](edge a, edge b)
         { return a.beginID < b.beginID; });
    cout << "beginID\ttrans\tendID" << endl;
    for (int i = 0; i < nfa.size(); i++)
    {
        cout << nfa[i].beginID << "\t" << nfa[i].trans << "\t" << nfa[i].endID <<
endl;
    }
}
```

输出展示

RE1

```
Regular expression: (a|b)*abb
Suffix expression: ab|*a.b.b.
Begin state: 6
End state: 13
```

NFA:

```
flowchart LR
0((0))--a-->1((1))
1((1))--=-->5((5))
2((2))--b-->3((3))
3((3))--=-->5((5))
4((4))--=-->0((0))
4((4))--=-->2((2))
5((5))--=-->7((7))
5((5))--=-->4((4))
6((6:Start))--=-->4((4))
6((6:Start))--=-->7((7))
7((7))--a-->9((9))
9((9))--b-->11((11))
11((11))--b-->13(13:End)
```

RE转为DFA，DFA最小化

在上面代码的基础上，添加了 `struct state` 用于表示DFA。下面是 `struct state` 的定义和成员变量说明：

```
cppCopy codestruct state
{
    int lastID = -1;      // 上一个状态的标识
    char trans;           // 转移字符
    int nowID = -1;       // 当前状态的标识
    vector<int> closure; // 存储状态的ε-closure
    bool terminal = false; // 标识是否是终止状态
};
```

这个 `struct state` 数据结构的实例会存储整个 DFA，每个实例代表一个 DFA 状态。状态之间的转移关系通过 `lastID`、`trans` 和 `nowID` 这些成员变量建立起来。整个 DFA 存储在 `vector<state> dfa` 中，其中每个元素都代表一个状态。

`Todfa` 函数用于将NFA 转换为 DFA。首先构建一个DFA的初始状态，然后通过遍历NFA的边，从初始状态出发，利用ε-closure操作（包括闭包和移动操作）来构建DFA的状态集合。

```
void Todfa(vector<edge> nfa, string s1)
{
    for (int i = 0; i < nfa.size(); i++)
    {
        cout << "clusure(" << nfa[i].beginID << ")={";
        vector<int> v1;
        v1 = clusure(nfa[i].beginID); // 获取NFA中起始状态的ε闭包
        for (int j = 0; j < v1.size() - 1; j++)
        {
            cout << v1[j] << ",";
        }
        cout << v1[v1.size() - 1] << "}" << endl;
    }

    vector<char> alphabet = getalphabet(s1); // 获取正则表达式中的字母表
    state d1;
    int beginID = findbegin(); // 找到NFA的起始状态
    d1.lastID = 0;
    d1.nowID = 1;
    d1.trans = 'N'; // N表示新状态
    d1.closure.push_back(beginID);
    DFadd(d1.closure, '=', beginID); // 计算DFA中的新状态
    std::sort(d1.closure.begin(), d1.closure.end());
    dfa.push_back(d1);
    queue<state> q;
    q.push(d1);
    vector<state> dumplicate;

    while (!q.empty())
    {
        state s = q.front();
        state temp;
        for (int i = 0; i < alphabet.size(); i++)
        {
            vector<int> v1 = move(s.closure, alphabet[i]); // 根据输入字符进行状态迁
移
            vector<int> v2 = clusure_move(v1); // 计算新状态的ε闭包
            if (!index(v2))
```

```
            {
                temp.closure = v2;
                temp.lastID = s.nowID;
                temp.nowID = dfa.size() + 1;
                temp.trans = alphabet[i];
                dfa.push_back(temp); // 将新状态添加到DFA中
                q.push(temp);
            }
            else
            {
                temp.closure = v2;
                temp.lastID = s.nowID;
                temp.nowID = index(v2) + 1;
                temp.trans = alphabet[i];
                dumplicate.push_back(temp); // 处理状态迁移后的状态
            }
        }
        q.pop();
    }
    int endID = findend(); // 找到NFA的终止状态
    for (int i = 0; i < dfa.size(); i++)
    {
        if (count(dfa[i].closure.begin(), dfa[i].closure.end(), endID))
        {
            dfa[i].terminal = true; // 标记DFA中的终止状态
        }
    }

    dfa.insert(dfa.end(), dumplicate.begin(), dumplicate.end()); // 插入处理后的状态
    sort(dfa.begin(), dfa.end(), [](state a, state b)
         { return a.nowID < b.nowID; }); // 对DFA状态进行排序
    for (int i = 0; i < dfa.size(); i++)
    {
        if (count(dfa[i].closure.begin(), dfa[i].closure.end(), endID))
        {
            dfa[i].terminal = true;
        }
    }
}
```

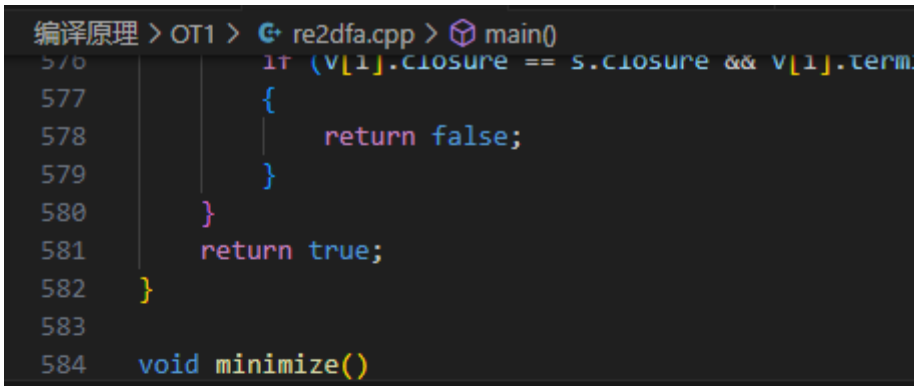
最小化DFA的过程如下。

```
void minimize()
{
    vector<state> minidfa; // 存储最小化后的DFA状态
    for (int i = 0; i < dfa.size(); i++)
    {
        vector<int>().swap(dfa[i].closure); // 清空DFA状态的闭包信息
    }
    bool endend = true; // 用于判断最小化过程是否结束
    while (endend)
    {
        vector<vector<int>> v; // 存储DFA状态的等价类
        vector<int> v; // 用于构建等价类
```

```
        vector<int> tomove; // 存储需要移动的等价类索引
        endend = false;

        // 构建DFA状态的等价类
        for (int j = 0; j < dfa.size(); j++)
        {
            vector<int>().swap(v);
            for (int i = 0; i < dfa.size(); i++)
            {
                if (dfa[i].lastID == j)
                {
                    int a = int(dfa[i].trans) + dfa[i].nowID * 100;
                    if (count(v.begin(), v.end(), a) == 0)
                    {
                        v.push_back(a);
                    }
                }
            }
            sort(v.begin(), v.end());
            V.push_back(v);
        }

        // 移除空等价类
        for (int i = 0; i < V.size(); i++)
        {
            if (V.back().empty())
            {
                V.pop_back();
            }
        }

        // 检测和移除相同的等价类
        for (int i = 0; i < V.size(); i++)
        {
            for (int j = i + 1; j < V.size(); j++)
            {
                if (V[i] == V[j] && !V[j].empty())
                {
                    endend = true;
                    tomove.push_back(j);
                    deletedumplicate(i, j); // 删除相同等价类
                }
            }
        }
    }

    // 构建最小化后的DFA
    for (int i = 0; i < dfa.size(); i++)
    {
        if (i == 0)
        {
            minidfa.push_back(dfa[i]); // 添加初始状态
        }
        else
        {
            if (unique(minidfa, dfa[i]))
```

```
            {
                minidfa.push_back(dfa[i]); // 添加非重复的状态
            }
        }
    }

    // 输出最小化后的DFA
    cout << "============MinimizedDfa============" << endl;
    cout << "lastID\ttrans\tnowID\tisterminal" << endl;
    for (int i = 0; i < minidfa.size(); i++)
    {
        cout << minidfa[i].lastID << "\t" << minidfa[i].trans << "\t" <<
minidfa[i].nowID << "\t" << minidfa[i].terminal << endl;
    }
}
```

最后输出DFA。效果如下。



```
编译原理 > OT1 > re2dfa.cpp > main()
576             if (V[i].closure == s.closure && V[i].term
577             {
578                 return false;
579             }
580         }
581         return true;
582     }
583
584     void minimize()
```

问题 6　输出　调试控制台　终端　端口　GITLENS

```
---------DFA---------
beginID trans   endID
0       a       1
1       =       5
2       b       3
3       =       5
4       =       0
4       =       2
5       =       7
5       =       4
6       =       4
6       =       7
7       =       8
8       a       9
9       =       10
10      b       11
11      =       12
12      b       13
clusure(0)={0}
clusure(1)={0,1,2,4,5,7,8}
clusure(2)={2}
clusure(3)={0,2,3,4,5,7,8}
clusure(4)={0,2,4}
clusure(4)={0,2,4}
clusure(5)={0,2,4,5,7,8}
clusure(5)={0,2,4,5,7,8}
clusure(6)={0,2,4,6,7,8}
clusure(6)={0,2,4,6,7,8}
clusure(7)={7,8}
clusure(8)={8}
clusure(9)={9,10}
clusure(10)={10}
clusure(11)={11,12}
clusure(12)={12}
alphabet:a b
move(0,2,4,6,7,8,a)={1,9}
clusure_move(1,9)={0,1,2,4,5,7,8,9,10}
move(0,2,4,6,7,8,b)={3}
clusure_move(3)={0,2,3,4,5,7,8}
move(0,1,2,4,5,7,8,9,10,a)={1,9}
clusure_move(1,9)={0,1,2,4,5,7,8,9,10}
move(0,1,2,4,5,7,8,9,10,b)={3,11}
clusure_move(3,11)={0,2,3,4,5,7,8,11,12}
move(0,2,3,4,5,7,8,a)={1,9}
clusure_move(1,9)={0,1,2,4,5,7,8,9,10}
move(0,2,3,4,5,7,8,b)={3}
clusure_move(3)={0,2,3,4,5,7,8}
move(0,2,3,4,5,7,8,11,12,a)={1,9}
```

```
clusure_move(1,9)={0,1,2,4,5,7,8,9,10}
move(0,2,3,4,5,7,8,11,12,b)={3,13}
clusure_move(3,13)={0,2,3,4,5,7,8,13}
move(0,2,3,4,5,7,8,13,a)={1,9}
clusure_move(1,9)={0,1,2,4,5,7,8,9,10}
move(0,2,3,4,5,7,8,13,b)={3}
clusure_move(3)={0,2,3,4,5,7,8}
============MinimizedDfa============
lastID  trans   nowID   isterminal
0       N       1       0
1       a       1       0
1       b       1       0
1       b       1       1
PS D:\文件\编译原理\OT1>
```