词法分析器生成器子模块

软件学院

陈睿 141250013

141250013@smail.nju.edu.cn

2016.10.27

目录

1. 目标 3
2. 内容描述 3
3. 构思与方法 3
4. 假设 4
    4.1. 实验环境 4
    4.2. 输入及符号假设 4
5. 有关自动机描述 4
6. 数据结构描述 4
7. 核心算法描述 6
    7.1. RE->NFA 6
    7.2. NFA->DFA 8
8. 测试用例 9
    8.1. Test 1 9
    8.2. Test 2 11
9. 问题与解决 12
10. 总结与感受 13

1. 目标

主要目标是尝试进行词法分析器生成器的构造，使我可以更深入理解龙书上相关算法，并自己通过代码进行实现，同时也锻炼一下自己的能力。

2. 内容描述

本报告主要描述了词法分析器生成器中一部分子模块的构造，主要包括:从正则表达式推出 NFA，再从 NFA 推出 DFA 这 2 个过程，以及以上 2 个过程的构思方法、具体实现过程。

3. 构思与方法

从整体思路来看，考虑到之前构造简易词法分析器的过程，最终核心还是要产生确定的有限状态自动机，即 DFA，之后应该就可以生成相应的模板代码去用于分析程序。因此，主要分为由 RE 构造出 DFA 和由 DFA 生成模板代码 2 部分，在这里主要实现了由 RE 构造出 DFA 的过程，借鉴龙书上相关算法，结合一部分自己所学知识即可。

具体来看，代码部分有如下步骤：

1. 输入正则表达式
2. 对输入的正则表达式进行符号化补全，加入相关连接符号
3. 将中缀的正则表达式转换为后缀形式
4. 对确定的后缀表达式进行 RE->NFA 的过程
5. NFA->DFA
6. 输出所有 DFA 状态

4. 假设

4.1. 实验环境

1. Mac OS
2. Clion + MinGW (gcc)

4.2. 输入及符号假设

1. 输入正则表达式仅包含连接、并、闭包运算
2. 仅包含 a-z 的符号
3. 假设 e 边用`&`记号表示

5. 有关自动机描述

有限状态机由代码构造。根据输入的正则表达式逐步先构造 NFA，再利用 NFA 构造 DFA。

分别采用龙书上 bottom-up 方法和子集构造法。

6. 数据结构描述

1. 状态点集以及边集，采用数组模拟链表实现

```
struct Edge{
    int des;
    char path;
    int next_edge;
};
struct Sta{
    int first_edge;
    bool is_start;
    bool is_end;
    std::set<int> rep;
};

void newAstate(Sta state[], int& state_nums);//新建一个状态
void newAedge(int start, int end, char path, Sta state[], Edge e[], int& e_nums);//新建一条边
```

2. NFA 类

```
class RE2NFA {
public:
    Sta NFA_states[max_states];
    Edge NFA_e[max_edges];
    int state_nums;
    int e_nums;
    void addDotToRE(string& re);
    void RE2POST(string& re);
    void m_RE2NFA(string re);
private:
    stack<string> stack_char;
    stack<char> stack_op;
    string combine();
    void newFromChar(char c);
    void joinTwoState(int& in1, int& out1, int& in2, int& out2);
    void orTwoState(int& in1, int& out1, int& in2, int& out2);
    void loopOneState(int& in, int& out);
};
```

3. DFA 类

```
class NFA2DFA {
public:
    Sta DFA_states[max_states];
    Edge DFA_e[max_edges];
    int DFA_state_nums;
    int e_nums;
    void m_NFA2DFA();
    NFA2DFA(Sta nfa[], Edge e[], set<char> path, int num){
        for (int i = 0; i < max_states; ++i) {
            this->NFA_states[i] = nfa[i];
        }
        for (int i = 0; i < max_edges; ++i) {
            this->NFA_e[i] = e[i];
        }
        this->path_set = path;
        this->state_nums = num;
    }
private:
    Sta NFA_states[max_states];
    Edge NFA_e[max_edges];
    int state_nums;
    set<char> path_set;
    bool EPclosure(set<int>& s);
};
```

4. <stack>，用于中缀转后缀的时候记录操作符和操作数

```
stack<string> stack_char;
stack<char> stack_op;
```

5. <queue>、<vector>、<set>，用于集合元素，以及 BFS 所需要的队列

```
queue<set<int>> q;
queue<int> q_state_num; //记录访问过的集合的state number
vector<set<int>> vis;
vector<int> vis_state_num; //记录访问过的集合的state number
set<int> first;
set<char>::iterator sit;
```

7. 核心算法描述

7.1. RE->NFA

1. 对输入的中缀正则表达式补全`.`（连接符号），用于进行后缀表达式的转换。只需要判断当前如果是字符，那么如果前一个符号为`)`,`*`,`字符`中的一种就加入`.`

2. 进行中缀表达式转后缀表达式

2.1 通过符号栈和操作数栈来实现，由于 3 个操作符无优先级，且*为单目操作符。

2.2 先判断是否为`*`，如果是，则直接对操作数栈中栈顶元素+`*`在压入栈中

2.3 如果是`)`，则一直进行出栈操作直到匹配到`)`

2.4 如果是`.`或`|`则在栈不空的情况下进行出栈操作

2.5 其他情况进栈

2.6 注意符号栈进行出栈操作实际上需要对操作数栈进行一个

`combine`操作，再 push 回去

3. 后缀表达式转 NFA

3.1 利用`bottom-up`的思想，构建分析树，再根据不同操作符创建状态，具体如下。

3.2 连接操作

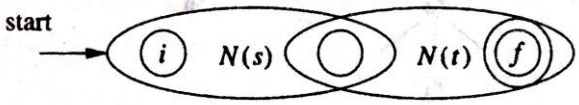


图 3-41 两个正则表达式的连接的 NFA

3.3 并操作

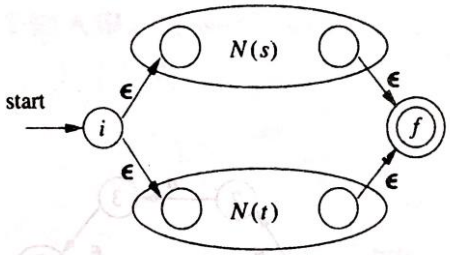


图 3-40 两个正则表达式的并的 NFA

3.4 闭包操作

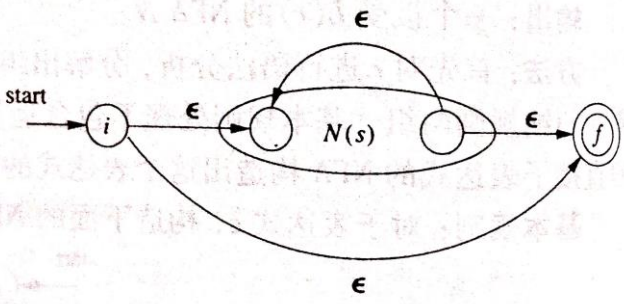


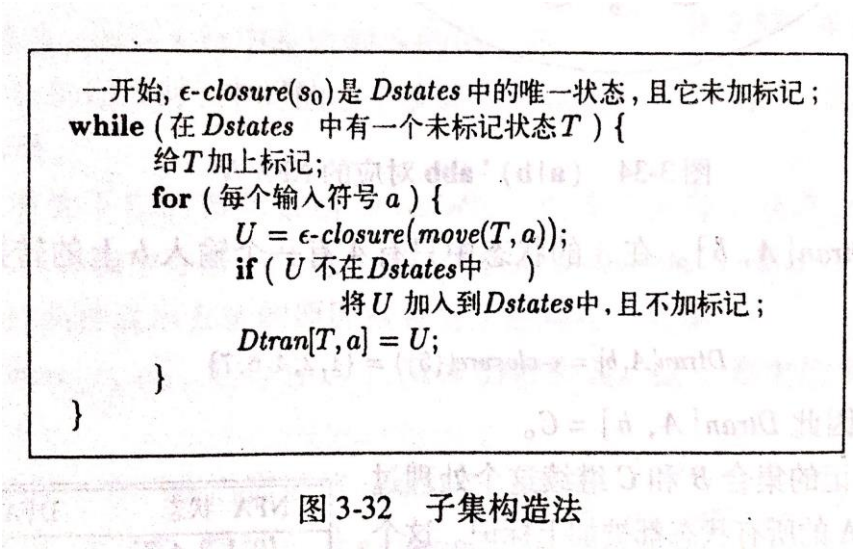
图 3-42 一个正则表达式的闭包的 NFA

3.5 注意，这里直接把状态作为点，转换作为边进行有向图的构建，加入相应的 e 边进行构造。

7.2. NFA->DFA

利用子集构造法

1. 首先状态为 NFA 中初始状态的 e-closure，然后逐步求出各个状态，具体如书上伪代码，见下图：



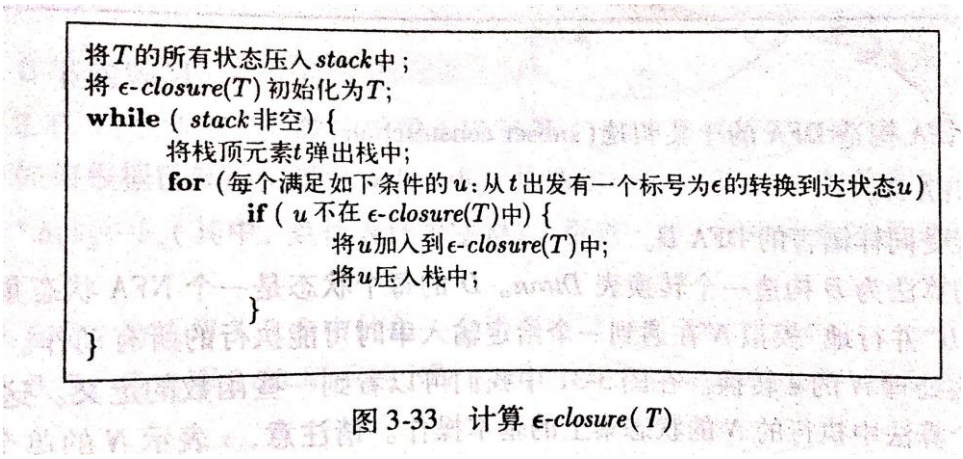
一开始，ϵ-closure(s_0)是 $Dstates$ 中的唯一状态，且它未加标记；
while (在 $Dstates$ 中有一个未标记状态 T) {
    给 T 加上标记；
    for (每个输入符号 a) {
        $U = \epsilon$-closure($move(T, a)$)；
        if (U 不在 $Dstates$ 中)
            将 U 加入到 $Dstates$ 中，且不加标记；
        $Dtran[T, a] = U$；
    }
}

图 3-32　子集构造法

2. 注意，这边采用 STL<queue>作为数据结构，并采用广度优先遍历进行状态生成。

3. 并采用 STL<set>作为记录状态集的数据结构，防止重复状态。

4. 其中计算 e-closure 的方法也如书上所示伪代码，见下图：



将 T 的所有状态压入 $stack$ 中；
将 ϵ-closure(T) 初始化为 T；
while ($stack$ 非空) {
    将栈顶元素 t 弹出栈中；
    for (每个满足如下条件的 u：从 t 出发有一个标号为 ϵ 的转换到达状态 u)
        if (u 不在 ϵ-closure(T) 中) {
            将 u 加入到 ϵ-closure(T) 中；
            将 u 压入栈中；
        }
}

图 3-33　计算 ϵ-closure(T)

5. 这里同样采用 STL<queue>，采用广度优先遍历进行一层一层 e 边的迭代。

8. 测试用例

8.1. Test 1

1. 测试输入

(a|b)a*b*abb

2. 测试输出

```
/Users/raychen/Library/Caches/CLion2016.1/cmake/
(a|b)a*b*abb
--------Formal RE--------
(a|b).a*.b*.a.b.b
--------Postfix RE--------
ab|a*.b*.a.b.b.
--------NFA--------
```

```
-------NFA--------
[start]state: 4
  4-->2[&]
  4-->0[&]
state: 2
  2-->3[b]
state: 0
  0-->1[a]
state: 3
  3-->5[&]
state: 1
  1-->5[&]
state: 5
  5-->9[&]
  5-->6[&]
state: 9
  9-->13[&]
  9-->10[&]
state: 6
  6-->7[a]
state: 13
  13-->15[a]
state: 10
  10-->11[b]
state: 7
  7-->9[&]
  7-->6[&]
state: 15
  15-->17[b]
state: 11
  11-->13[&]
  11-->10[&]
state: 17
  17-->19[b]
state: 19[end ]
-------DFA--------
```

```
-------DFA--------
[start]state: 0
  0-->2[b]
  0-->1[a]
state: 2
  2-->4[b]
  2-->3[a]
state: 1
  1-->4[b]
  1-->3[a]
state: 4
  4-->4[b]
  4-->6[a]
state: 3
  3-->5[b]
  3-->3[a]
state: 6
  6-->9[b]
state: 5
  5-->7[b]
  5-->6[a]
state: 9
  9-->10[b]
state: 7[end ]
  7-->4[b]
  7-->6[a]
state: 10[end ]

Process finished with exit code 0
```

8.2. Test 2

1. 测试输入

(abb)*a*

2. 测试输出

```
/Users/raychen/Library/Caches/CLion2016.1/cmake/genera
(abb)*a*
-------Formal RE--------
(a.b.b)*.a*
-------Postfix RE--------
ab.b.*a*.
-------NFA--------
```

```
-------NFA--------
[start]state: 6
  6-->7[&]
  6-->0[&]
state: 7
  7-->11[&]
  7-->8[&]
state: 0
  0-->1[a]
state: 11[end ]
state: 8
  8-->9[a]
state: 1
  1-->3[b]
state: 9
  9-->11[&]
  9-->8[&]
state: 3
  3-->5[b]
state: 5
  5-->7[&]
  5-->0[&]
-------DFA--------
```

```
-------DFA--------
[start]state: 0[end ]
  0-->1[a]
state: 1[end ]
  1-->4[b]
  1-->3[a]
state: 4
  4-->5[b]
state: 3[end ]
  3-->3[a]
state: 5[end ]
  5-->1[a]

Process finished with exit code 0
```

9. 问题与解决

1. 后缀表达式转 NFA 时，如何记录中间产生的各个状态。

解决：实际上，从语法分析树可以看到，最多只会有 2 个当前状态集，并且

当前状态集实际上只有 start 状态和 end 状态是会被后续转换所用到的，于是只需要用 4 个变量表示即可。如我代码中：

```
int in1 = -1; // start state 1
int out1  = -1; // end state 1
int in2 = -1; // start state 2
int out2 = -1; // end state 2
```

2. NFA 转 DFA 时，如何记录当前状态集所对应的状态号？因为没有建立对应状态集的数据结构，状态集实际上也是一个状态？

解决：采用多一个`STL<vector>`的方式，跟踪记录相应状态号。

```
queue<set<int>> q;
queue<int> q_state_num; //记录访问过的集合的state number
vector<set<int>> vis;
vector<int> vis_state_num; //记录访问过的集合的state number
```

10. 总结与感受

通过这次实验，我感觉复习了一些以前学过的算法，如 BFS，中缀转后缀等。同时感受到了词法分析器生成器的有趣之处，希望有空可以实现一个自己的简单分析生成器。

没有了一开始看到 lex 生成出一个分析代码的模板出来那种震惊了，感觉自己也可以实现，开心~