编译原理课程实验

实验报告

SEULEX 实验报告

2021 年 1 月 20 日

71118313 张晓铮

东南大学软件学院

Contents

1 实验目标

本次实验的目标为开发一款简易的 Lex 程序，可以接受用户输入的正规表达式，并生成基于用户输入的正规表达的词法分析程序

2 实验环境

- 本次实验采用的实验环境：
  1 操作系统：Ubuntu 18.04
  2 语言：C++ 11 标准
  3 编译工具：clang++ 6.0 CMake 3.10.2

3 实验内容

- 主要内容
  1 定义用户输入的正规表达式格式并读入
  2 用户输入正规表达式的规范化
  3 基于正规表达式生成 NFA
  4 将 NFA 转换为 DFA
  5 DFA 最小化
  6 生成程序代码文件

4 实验实现方法

在本次实验中，首先对于用户定义正规表达式的输入采用了 xml 文件的方式，并借助第三方 xml 解析库来解析 xml 文件，同时对输入正规表达式的规则做出了限制，具体规则见下文。对于正规表达式转换为 NFA 的算法，采用首先将正规表达式中的特殊符号去除，再将其转换为后缀形式，再利用 Thompson 算法将正规表达式转换为 NFA。利用 epsilon 闭包和子集构造法进行 NFA 的确定化，再利用等价类划分的方法进行 DFA 的最小化，最后根据生成的 DFA 生成最最终程序代码的生成。同时采用了配置文件的方法，可供用户自定义配置选项，提高了软件的通用性。

4.1 关键数据结构描述

4.1.1 config.h

该类用于读取用户的配置文件，配置文件采用.properties 格式，该类采用单例模式，读取后供后续程序使用。

```
#pragma once
#include <string>
using namespace std;

#define UTF8 1
#define ASCII 1
#define UNICODE 2

class Config
{
public:
    static Config *get_instance();
    int get_buffer_size();
    static void delete_instance();
    const string &get_source_file_loaction();
    int get_encoding();
    int get_charset();

private:
    Config();
    int buffer_size;
    int encoding;
    int charset;
    static Config *config;
    string source_file_location;
};
```

4.1.2 rule.h

rule 类用于存储用户定义的正规表达式以及每个正规表达式对应的要执行的动作

```
#pragma once

#include <string>
#include <vector>
using namespace std;

class rule
{
public:
    rule(const rule &);
    rule(const string &, const vector<string> &);
    const string &get_pattern();
    const vector<string> &get_actions();

private:
    string pattern = "";
    vector<string> actions;
};
```

4.1.3 NFAState.h

NFAState 用于存储 NFA 中的状态，states_map 主要作用是存储这个状态能通过哪些符号到达哪些状态。其次，states_map 的 key 使用 string 而不是 char 的原因是我们希望可以更加通用，因为对于不是英文的字符，一个字符的长度可能不是一个 char，而是两个或者更多字节，不使用 w_string 或 w_char 的原因是这个数据类型的大小在不同的操作系统中大小不一样，而且是一个定长的，如果大部分都是使用一个字节的 char 而使用 w_char 就会有些浪费空间，所以使用可变的 string 类型，后面会有专门的方法来解析长度变动的字符，如 UTF8 编码的文件。

```
#pragma once
#include <unordered_map>
```

```
#include <unordered_set>
using namespace std;

class NFAState
{
public:
    NFAState(int);
    NFAState(const NFAState &);
    void set_states_map(const string &, int);
    const unordered_multimap<string, int> &get_states_map()
        const;
    void find_epsilon_edge(unordered_set<int> &) const;
    void find_state_by_edge(const string &, unordered_set<int
        > &) const;
    int get_id() const;

private:
    int id;
    unordered_multimap<string, int> states_map;
};
```

4.1.4 NFA.h

该类用于存储 NFA 图，states 保存了所有生成的 NFAState，end_state 用来存储哪些状态是终态，以及它们对应的由用户定义的 rule，由于我们希望谓词优先，所以采用了有序的 map，以备 NFA 转换为 DFA 时可以查找哪个 rule 更靠前。

```
#pragma once
#include <unordered_map>
#include <map>
#include "source_file.h"
#include "NFAState.h"
#include "rule.h"
using namespace std;
```

```
8
9  class NFA
10 {
11 public:
12     NFA();
13     int get_start_state_id() const;
14     int is_contains_end(const unordered_set<int> &) const;
15     void can_reach(const unordered_set<int> &, unordered_map<
           string, unordered_set<int>> &);
16     void find_epsilon_closure(const unordered_set<int> &,
           unordered_set<int> &);
17     const rule &get_rule_by_id(int) const;
18     const NFAState &find_state_by_id(int) const;
19     const unordered_map<int, NFAState> &get_all_states()
           const;
20
21 private:
22     void construct_NFA();
23     source_file *source;
24     map<int, rule> end_state;
25     int start_state_id;
26     unordered_map<int, NFAState> states;
27 };
```

4.1.5 DFAState.h

该类和 NFAState 类似，用于表示 DFA 的状态，由于一个 DFA 状态中可能含有多个 NFA 状态，所以 NFAStates 用于存储该 DFAStates 中包含了哪些 NFAState，edges 用于保存该状态可以从哪些符号到哪些状态，key 采用 string 的原因和 NFAState 中的原因相同。

```
1 #pragma once
2 #include <unordered_set>
3 #include <unordered_map>
4 using namespace std;
```

```
5
6  class DFAState
7  {
8  public:
9      DFAState(int);
10     DFAState(const DFAState &);
11     void set_edges(const string &, int);
12     void set_NFAStates(int);
13     void set_NFAStates(const unordered_set<int> &);
14     int get_id() const;
15     unordered_set<int> &get_NFA_states();
16     const unordered_map<string, int> &get_all_edges() const;
17     int find_DFAState_by_edge(const string &);
18
19 private:
20     int id;
21     unordered_set<int> NFAStates;
22     unordered_map<string, int> edges;
23 };
```

4.1.6 DFA.h

该类用于存储 DFA，其中 states 用来存储生成的 DFAState，由于 DFA 的状态是按顺序生成的，所以直接利用 vector 存储即可，可以提高访问效率，同时 end_states 保存结束状态和它们对应的用户对应的 rule，由于这里不用确认 rule 的先后顺序，所以采用 unordered_map

```
1  #pragma once
2  #include <vector>
3  #include <unordered_map>
4  #include "DFAState.h"
5  #include "rule.h"
6  #include "NFA.h"
7  using namespace std;
8
```

```
9  class DFA
10 {
11 public:
12     DFA();
13     const vector<DFAState> &get_all_states() const;
14     const unordered_map<int, rule> &get_all_end_states()
           const;
15     int get_start_id() const;
16
17 private:
18     void construct_DFA();
19     int is_same_core(const unordered_multimap<int,
           unordered_set<int>> &, const unordered_set<int> &);
20     int is_same_state(const unordered_set<int> &);
21     int start;
22     unordered_map<int, rule> end_states;
23     vector<DFAState> states;
24 };
```

4.1.7 string_iterator.h

该类是一个接口类，用于用户配置的不同编码方式提供不同的实现类，来实现对不同编码的字符串进行遍历，可以实现对变长编码的字符串的遍历，如 UTF8 编码。目前只提供了 UTF8 编码的遍历实现类，实现方法为每次读取一个字节，对其按位与判断这个字符的字节数，然后读取相应数量的字节，实现对一个变长字符的读取。

```
1 #pragma once
2 #include <string>
3 using namespace std;
4
5 class string_iterator
6 {
7 public:
8     virtual const string next() = 0;
```

```
9       virtual bool has_next() = 0;
10      virtual void set_string(const string &) = 0;
11      virtual void set_current_pos(int) = 0;
12      virtual int get_current_pos() = 0;
13      virtual ~string_iterator(){};
14  };
```

4.1.8 charset.h

该类也是一个接口类，可以针对不同字符集构造全集和补集，用于处理中括号中的运^ 算和. 运算，可以根据用户配置的不同字符集提供不同实现类，目前只提供了 ASCII 字符集的实现类。

```
1  #pragma once
2  #include <unordered_set>
3  #include <string>
4  using namespace std;
5
6  class charset
7  {
8  public:
9      virtual void complementary_set(unordered_set<string> &) =
           0;
10     virtual void universe_set(unordered_set<string> &) = 0;
11     virtual ~charset(){}
12 };
```

4.2 关键算法描述

4.2.1 正规表达式的规范化

正规表达式的规范化即去除用户输入的表达式中的?,[],{},+ 等的运算符变为 *，() 等规范的运算符。为了实现这一目的，需要对用户输入的正规表达式做出一些限制，以便更好的规范化，对正规表达式的要求如下：

1 不支持 ^ 和 $（即行首和行尾）

2 关于转义

2.1 取消了"" 转义（个人认为存在两个转义方式会存在歧义）

2.2 \ 转义符号:

2.2.1 t, n, f, v, r 转义为特殊字符

2.2.2 (,), [, {, +, ?, *, ., |,
转义为普通字符

2.2.3], } 仅在 [, {后才需要转义，如 \{ddd}，}adb}] 则处理成普通字符

3 [] 处理:

3.1 \t,\n,\r,\v,\f,],{会被转义 (对于 [] 中的 {} 认为是字符，而不是引用其他表达式，所以请将 {也进行转义，否则会将此 {后的字符认为是引用其他正规表达式)

3.2 ^ 处于行首且未被 \ 转义则为非, 其他情况则认为是普通字符

3.3 -两侧都有字符则认为是特殊字符，只有数字，字母且按照顺序才合法，-只有一侧有字符则认为是普通字符

3.4 [] 中的 \ 都会被处理，比如 \9，这个字符并不需要转义，则保留 9 去除 \

3.5 其余认为是普通字符

4 {} 处理

4.1 {} 用于分层定义，引用其他定义好的正规表达式,{} 中为其他正规表达式名字

4.2 {} 中除了 \} 会被转义，其余均按普通字符处理。e.g. {\\b\t\}} 则对应引用名称为 \\b\t} 的正规表达式

5 ?, + 处理

5.1 (?, |?, (+, |+ 均不合法

5.2 如 a+?, a?+, a*? 等均不合法, 请输入括号: e.g. (a+)?

6 加. 运算符

6.1 (后,) 前不加点 ('(.', ').')

6.2 | 前后不加点

6.3 加点情况： char . (; char . char;).(;).char; *.(; *.char;

处理正规表达式的一般过程为首先去除大括号，将大括号中引用的表达式替换，再去除中括号，对于其中的-,.,^ 进行处理，全部替换为 | 运算符; 然后去除 + 和?，将其替换为等价的用 * 和 epsilon 表示的表达式; 最后对于转义全部替换为转义过后的字符, 再对表达式进行加 · 处理

4.2.2 正规表达式转 NFA

首先需要对正规表达式做中缀转后缀的操作：使用操作符栈和后缀表达式队列扫描中缀形式的正规表达式；若遇到字符就直接压到队列中（队列中保存的是当前情况下后缀形式的正规表达式）；若遇到的是 “(”、“)”、“.”、“|”、“*” 这些操作符，则根据当前栈顶元素的优先级和读入操作符的优先级作比较，如果栈顶优先级高则弹栈，否则把自身压入栈中。如果栈空也直接压入栈中。若定义了转义字符，则当读入字符是转义字符时，要判断后面的字符是否需要转义，如果是要转义的，那就不是操作符，作为字符处理

其次利用 Thompson 算法将后缀形式的正规表达式自下而上组合构建 NFA，使用栈来记录生成的小 NFA 的开始状态和结束状态，遇到操作符后根据相应规则弹出栈顶的小 NFA 进行转换合并，生成新的 NFA 并将新的 NFA 的开始状态和结束状态压入栈中，最后即可获得一个完整的 NFA。

4.2.3 NFA 确定化

- NFA 确定化
- 确定化过程
  - 对 NFA 的开始状态进行 epsilon 闭包，生成新的 DFA 初态，加入队列
  - 当队列不为空时
    - 取出队列头的未处理 DFA 状态 s
    - 对于字符（边）全集中每个字符 c 进行子集构造
      - 如果有对应的发出边
      - 对生成的集合进行 epsilon 闭包
      - 判断生成的集合是否与已经存在的 DFA 状态相同
      - 如果存在，记录相同的状态 s’
      - 如果不存在，则新建状态 s’，判断 s’ 是否为终态，如果为终态则决定终态对应的动作
      - 连边 s->s’

4.2.4 DFA 最小化

- DFA 最小化
- 使用弱等价，回头看采用一步一回头
- 遍历原 DFA，将每个终态单独存入一个集合中，其余非终态放入一个集合
- 直至没有新建集合为止，遍历所有集合：
  - 从集合 k 中任意取一个状态 s，遍历其余状态 s’
    - 取出队列头的未处理 DFA 状态 s
    - 对于字符（边）全集中每个字符 c 进行子集构造
      - 遍历 s 所有发出边 e
      - 如果 s’ 中也有 e 且与 s 到同一个集合，跳至下一个
      - 否则新建集合 k’，将 s’ 并入 k’ 中
    - 如果新建集合了，就终止循环
    - 将 k’ 中元素从 k 中删除

4.2.5 代码生成

对于最后生成的最小化 DFA 表，利用 map 的数据结构将其存储，根据最终的 DFA 在生成的文件中写 map 的插入语句，将 DFA 插入到生成代码中的 DFA 中，再利用用户配置的要处理文件的编码格式，写入读取相应编码的代码，最后再加入用户自定义的代码，由 yylex() 函数进行驱动进行解析。

5 实验结果

可执行代码见源码，使用方法见源码中的 README，源码可从 github 上查看下载，地址 https://github.com/zhang-x-z/CompilerPrincipal-Lab

测试选取了解析 JSON 文件中的 Token，包括 string，{，}，number，[，]，:，逗号，true，null，false 以及空格（包括制表符换行），输入的 xml 文件如下

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Lex>
3     <userDefinitions></userDefinitions>
```

```
4       <reDefinitions>
5           <digit>
6               [0-9]
7           </digit>
8       </reDefinitions>
9       <rules>
10          <rule>
11              <re>
12                  "(\\.|[^\\"\n])*"
13  ␣␣␣␣␣␣␣␣␣␣␣␣␣</re>
14  ␣␣␣␣␣␣␣␣␣␣␣␣␣<action>
15  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"STRING"␣&lt;&lt;␣endl;
16  ␣␣␣␣␣␣␣␣␣␣␣␣␣</action>
17  ␣␣␣␣␣␣␣␣</rule>
18  ␣␣␣␣␣␣␣␣<rule>
19  ␣␣␣␣␣␣␣␣␣␣␣␣␣<re>
20  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\{
21  ␣␣␣␣␣␣␣␣␣␣␣␣␣</re>
22  ␣␣␣␣␣␣␣␣␣␣␣␣␣<action>
23  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"LEFT BRACE"␣&lt;&lt;␣endl;
24  ␣␣␣␣␣␣␣␣␣␣␣␣␣</action>
25  ␣␣␣␣␣␣␣␣</rule>
26  ␣␣␣␣␣␣␣␣<rule>
27  ␣␣␣␣␣␣␣␣␣␣␣␣␣<re>
28  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
29  ␣␣␣␣␣␣␣␣␣␣␣␣␣</re>
30  ␣␣␣␣␣␣␣␣␣␣␣␣␣<action>
31  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"RIGHT BRACE"␣&lt;&lt;␣endl;
32  ␣␣␣␣␣␣␣␣␣␣␣␣␣</action>
33  ␣␣␣␣␣␣␣␣</rule>
34  ␣␣␣␣␣␣␣␣<rule>
35  ␣␣␣␣␣␣␣␣␣␣␣␣␣<re>
36  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣,
37  ␣␣␣␣␣␣␣␣␣␣␣␣␣</re>
```

```
38 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
39 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"COMMA"␣&lt;&lt;␣endl;
40 ␣␣␣␣␣␣␣␣␣␣␣␣</action>
41 ␣␣␣␣␣␣␣␣</rule>
42 ␣␣␣␣␣␣␣␣<rule>
43 ␣␣␣␣␣␣␣␣␣␣␣␣<re>
44 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:
45 ␣␣␣␣␣␣␣␣␣␣␣␣</re>
46 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
47 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"COLON"␣&lt;&lt;␣endl;
48 ␣␣␣␣␣␣␣␣␣␣␣␣</action>
49 ␣␣␣␣␣␣␣␣</rule>
50 ␣␣␣␣␣␣␣␣<rule>
51 ␣␣␣␣␣␣␣␣␣␣␣␣<re>
52 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\[
53 ␣␣␣␣␣␣␣␣␣␣␣␣</re>
54 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
55 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"LEFT BRACKET"␣&lt;&lt;␣endl;
56 ␣␣␣␣␣␣␣␣␣␣␣␣</action>
57 ␣␣␣␣␣␣␣␣</rule>
58 ␣␣␣␣␣␣␣␣<rule>
59 ␣␣␣␣␣␣␣␣␣␣␣␣<re>
60 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣]
61 ␣␣␣␣␣␣␣␣␣␣␣␣</re>
62 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
63 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"RIGHT BRACKET"␣&lt;&lt;␣endl;
64 ␣␣␣␣␣␣␣␣␣␣␣␣</action>
65 ␣␣␣␣␣␣␣␣</rule>
66 ␣␣␣␣␣␣␣␣<rule>
67 ␣␣␣␣␣␣␣␣␣␣␣␣<re>
68 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣[\r\t␣\n]
69 ␣␣␣␣␣␣␣␣␣␣␣␣</re>
70 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
71 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"WIGHTSPACE"␣&lt;&lt;␣endl;
```

```
72  ␣␣␣␣␣␣␣␣␣␣␣␣</action>
73  ␣␣␣␣␣␣␣␣</rule>
74  ␣␣␣␣␣␣␣␣<rule>
75  ␣␣␣␣␣␣␣␣␣␣␣␣<re>
76  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣[1-9]{digit}*|0
77  ␣␣␣␣␣␣␣␣␣␣␣␣</re>
78  ␣␣␣␣␣␣␣␣␣␣␣␣<action>
79  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"INT"␣&lt;&lt;␣endl;
80  ␣␣␣␣␣␣␣␣␣␣␣␣</action>
81  ␣␣␣␣␣␣␣␣</rule>
82  ␣␣␣␣␣␣␣␣<rule>
83  ␣␣␣␣␣␣␣␣␣␣␣␣<re>
84  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣([1-9]{digit}*\.{digit}*)|(0\.{digit}*)
85  ␣␣␣␣␣␣␣␣␣␣␣␣</re>
86  ␣␣␣␣␣␣␣␣␣␣␣␣<action>
87  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"FLOAT"␣&lt;&lt;␣endl;
88  ␣␣␣␣␣␣␣␣␣␣␣␣</action>
89  ␣␣␣␣␣␣␣␣</rule>
90  ␣␣␣␣␣␣␣␣<rule>
91  ␣␣␣␣␣␣␣␣␣␣␣␣<re>
92  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣true
93  ␣␣␣␣␣␣␣␣␣␣␣␣</re>
94  ␣␣␣␣␣␣␣␣␣␣␣␣<action>
95  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"TRUE"␣&lt;&lt;␣endl;
96  ␣␣␣␣␣␣␣␣␣␣␣␣</action>
97  ␣␣␣␣␣␣␣␣</rule>
98  ␣␣␣␣␣␣␣␣<rule>
99  ␣␣␣␣␣␣␣␣␣␣␣␣<re>
100 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣false
101 ␣␣␣␣␣␣␣␣␣␣␣␣</re>
102 ␣␣␣␣␣␣␣␣␣␣␣␣<action>
103 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"FALSE"␣&lt;&lt;␣endl;
104 ␣␣␣␣␣␣␣␣␣␣␣␣</action>
105 ␣␣␣␣␣␣␣␣</rule>
```

```
106 ␣␣␣␣␣␣␣␣<rule>
107 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣<re>
108 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣null
109 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣</re>
110 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣<action>
111 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cout␣&lt;&lt;␣"NULL"␣&lt;&lt;␣endl;
112 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣</action>
113 ␣␣␣␣␣␣␣␣</rule>
114 ␣␣␣␣</rules>
115 ␣␣␣␣<userCode></userCode>
116 </Lex>
```

配置文件如下

```
1 buffer_size=60
2 source_file_location=./test.xml
3 encoding=utf8
4 charset=ascii
```

测试用 JSON 文件:

```
1  {
2      "name": "ZXZ",
3      "age": 21,
4      "GPA": 3.92,
5      "is_male": true,
6      "is_famale": false,
7      "courses": [
8          "Database", "CompilePrincipal", null
9      ]
10 }
```

最后输出结果如图:

```
zxz@LAPTOP-4OJS0263:~/Projects/testLex $ ./lex < t.json
LEFT BRACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
STRING
COMMA
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
INT
COMMA
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
FLOAT
COMMA
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
TRUE
COMMA
WIGHTSPACE
```

```
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
FALSE
COMMA
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COLON
WIGHTSPACE
LEFT BRACKET
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
```

```
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
STRING
COMMA
WIGHTSPACE
STRING
COMMA
WIGHTSPACE
NULL
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
WIGHTSPACE
RIGHT BRACKET
WIGHTSPACE
RIGHT BRACE
WIGHTSPACE
```

6 实验小结

通过本次实验，对编译原理中的词法分析部分有了较深的理解。在实验过程中，一开始遇到的阻力很大，比如最开始的.l 文件的读取处理，由于不想在这个部分花费太多时间，想把精力集中在后面的算法部分，就利用 xml 文件来组织用户输入的正规表达式，并利用第三方库

来解析。同时对于正规表达式规范化部分也遇到了很多问题，甚至去 github 上查看了 flex 的源代码，最后利用自己规定的正规表达式规则，并且用字符串的搜索匹配方法，对正规表达式不断扫描，每次扫描进行一种处理，最终将正规表达式转换为规范化的正规表达式。

对于中缀转后缀，后缀转 NFA 和 NFA 转 DFA 以及 DFA 最小化，有确定的算法，写起来比较轻松。

此外还遇到的一个问题就是对于字符的处理，因为我希望这个程序能处理多种字符，而不只是处理 ASCII 中的字符，所以提供了一个 charset 的接口，可以让用户自己配置自己词法分析器中的字符集，比如可以配置 UNICODE 字符集等，但由于时间精力的原因，目前只提供了 ASCII 的字符集实现，但由于使用了接口，使得程序易于扩展，只要添加相应的实现类以及配置选项即可。