编译原理实验一

实验报告

151250100

刘宇翔

**实验目的：**

编写一个类似lex的程序，可以自己自定义正则表达式，用来分析其他的文本。由此来深入理解从正则表达式到NFA再到DFA再到最简DFA的转化以及在分析文本中如何处理正则表达式冲突的情形。重点在于并不是预定义好的正则表达式，而是可以任意输入在lex文件中，并且可以及时生效的程序。

**内容描述：**

此程序用c++实现，输入的内容有两个部分，第一个部分是自己写的类似lex的正则表达式定义文件，文件具体内容在Lexical Analyzer\lex\mylex.lex中。其中除了基础定义的用来分析java或c++的关键字和操作符之外，还包括了两个自己定义的在ch3中出现的正则表达式aa\*(bab\*a)\*(a|b)b\*和aa\*((bab\*a)\*(a|b)b\*)\*，这两个的优先级高于一切，并且在测试文件中有关于这两个正则表达式想匹配的词缀。

我自己模拟的lex文件分为两个部分，第一个部分为基础正则表达式，第二个部分为用来分析其他文件的词缀和相应的名字，如{mytest1} my\_regex\_1对应于匹配符合aa\*(bab\*a)\*(a|b)b\*的前缀并且获取这段内容，他的类型名字为my\_regex\_1。

第二个输入为需要分析的文本，在Lexical Analyzer\test\_output中，我提供了两份文本，一份为c++语言，一份为java。由于我提供的自己定义的lex文件中支持分析的文法比较少，所以这两份文件都是很简短的代码。

**方法：**

首先是对于单个正则表达式进行从RE->NFA->DFA->最简DFA的转换，其中的各个结构体或类的结构在Regex.h,special\_symbol.h,dfa.h中声明。为了更加便捷的分析正则表达式，采用函数指针与哈希表的形式代替switch语句，相关的函数操作在state\_operate.h中声明，其中有部分函数为Special\_Symbol类的友元函数。

在Lex.h与Lex.cpp中是对于输入的模仿lex文件的输入进行处理的文件。同时对于lex冲突的情况模仿书上对于冲突的处理，采取从同一个点开始取最长的匹配串，如果有多个最长的前缀相同，则按照lex文件中的先后顺序来排列

如lex文件中为:

Public 关键字

Letter 变量

则Public将会匹配到名字为关键字而不是变量。

**假设：**

假定所有当做lex的输入文件语法上都是正确的。同时正则表达式只支持^,$,+,[,],\*,|,?,(,)，-(不支持[^])

用来分析的文件可以为c++或者java

**Related FA descriptions：**

下面是我自己定义的类lex文件

ws {delim}+

letter [A-Za-z]

digit [0-9]

id {letter}({letter}|{digit})\*

number {digit}+(.{digit}+)?(E([+-])?{digit}+)?

punctuation \(|\)|\{|\}|\[|\]|\.|\,|\:|\;

basicop \+|\-|\\*|\/|%|=

compareop <|>|<=|>=|<>|==

sumop {basicop}+=

incrementop \+\+|\-\-

logicop \|\||!|&&|\||&|\^

operator <<|>>

op {punctuation}|{basicop}|{compareop}|{sumop}|{incrementop}|{logicop}|{operator}

mytest1 aa\*(bab\*a)\*(a|b)b\*

mytest2 aa\*((bab\*a)\*(a|b)b\*)\*

output System.out.println

reserved System|out|println|String|byte|short|int|long|float|double|char|boolean|bool|for|do|while|if|else|switch|case|break|default|continue|return|void|static|class|public|private|protected|new|import|printf|scanf|main|delete|struct|cout

$$

{ws} DEFAULT

{output} output

{reserved} reserved

{mytest1} my\_Regex\_1

{mytest2} my\_Regex\_2

{punctuation} punctuation

{basicop} basic\_op

{compareop} compare\_op

{sumop} sum\_op

{incrementop} increment\_op

{logicop} logic\_op

{operator} operator

\"([ -~])\*\" string

\[\] array

{id} var

{number} num

**相关重要的数据结构：**

由于文件中没有注释，所以部分结构在这边进行解释。

主要的类有Regex，Specific\_symbol，Lex，DFA，DFA\_State，State，下面将作出描述

class Regex

{

private:

Specific\_symbol \* symbol;//正则表达式中特殊符号操作的类

State \* nfa\_root;//nfa节点的根节点

DFA \* dfa\_state;//dfa节点的集合

const string name;//该正则表达式的名字

const string pattern;// 该正则表达式的内容

public:

Regex(const string & re, const string & name);

~Regex() {};

bool match(const string & ma) const;//匹配全部字符串判断是否匹配

int greedy\_match(const string & ma) const;

//从起始开始匹配字符串最大的前缀，返回最大前缀的位置，如{[ ~]\*}匹配{digit}abc则会返回a的位置。

friend void output(Regex r);//输出正则表达式中的各种结构

const string & getName() const { return name; };

const string & getPattern() const { return pattern; };

vector<pair<int, int>> group(const string & ma) const;

//匹配字符串中的所有匹配项，如{a}ab{bc}cjdk{10jsl}，正则表达式为{[ ~]\*},则会返回{a},{bc},{10js1}。

vector<pair<int, int>> greedy\_group(const string & ma) const;

//贪婪模式的匹配组，贪婪模式即为{abc}db{ade}则会匹配为整个字符串而不是两个{abc}，{ade}

void output()const;

};

typedef unordered\_map<int, State\*> nfaset;

struct DFA\_State

{

nfaset \_unmap;//该dfa节点包含的nfa节点

set<char> direction;//该节点的后继的方向

int \_number;//节点号

bool end;//是否可以结束

bool isAnalysis;//是否被分析过

unordered\_map<char, DFA\_State\*> \_next;//dfa的后继dfa及诶单

DFA\_State(int num) :\_unmap(), \_number(num), \_next(), end(false), direction(), isAnalysis(false){};

void insert(State\*);

bool operator== (const DFA\_State & a);//重载==号用于判断\_unmap是否相等

bool operator== (const DFA\_State \* const a);//重载==号用于判断\_next是否相等

~DFA\_State() {};

};

class DFA

{

private:

vector<DFA\_State\*> list;//dfa列表

vector<DFA\_State\*> s\_list;//最简dfa列表

DFA\_State \* dfa\_root;

int dfa\_num;

bool find(const string & ma, const DFA\_State \* const now, int index = 0) const;

int group\_str(const string & ma, const DFA\_State \* const now, int index) const;

int greedy\_group\_str(const string & ma, const DFA\_State \* const now, int index) const;

public:

DFA() :list(),s\_list(),dfa\_root(nullptr), dfa\_num(0) {};

void trans(State\* root);

void simplify();

void printDFA();

void printSDFA();

bool match(const string & ma);//全匹配

vector<pair<int, int>> group(const string & ma);//非贪婪匹配组

vector<pair<int, int>> greedy\_group(const string & ma);//贪婪匹配组

int greedy\_match(const string & m); //贪婪匹配

};

class Lex

{

private:

vector<Regex> list;//输入的lex文件中的第一部分的正则表达式

vector<pair<Regex, string>> checklist;

// 输入的lex文件中的第二部分的正则表达式，用来分析文本

Regex replace;

//对于lex文件中形如{.\*?}的内容将要替换成之前的已有的正则表达式的形式，这个正则表达式是用来匹配一串字符串中所有{.\*?}出现的字符串的位置。

Regex\* default\_regex;

//在第二部分汇总设定的不用输出结果的正则表达式，如我自己的定义中为{ws} DEFAULT

void handle\_str(string & str);

//用来替换str中所有的{.\*?} {}中的字符串名字与先前出现的相匹配的正则表达式的内容。

public:

Lex(const string & path);//将要输入的自己定义lex文件，测试用的路径为lex/mylex.lex

void analysis(const string & path, vector<Token> & result) const;

//分析路径名为path的文件，同时输入的result是用来存放Token的容器，在这个函数中会在命令行中输出result的所有内容，并且在path中会输出一个后缀名为.output的结果文件。

};

typedef State\* (\*func)(Specific\_symbol\*,State \*, char val);

//对于正则表达式中函数指针的声明,全部的声明在state\_operate.h中，如对于\*的处理等

class Specific\_symbol

{

private:

struct State\_Stack//对于nfa状态进行处理的自定义的栈。具体功能为协助func函数进行操作

{

stack<State\*> start;//

stack<State\*> end;

vector<char> range;

bool connect;

State\* second;

State\* first;

State\_Stack() :start(), end(), second(nullptr), first(nullptr),range(),connect(false) {}

void push(State\* \_s) { second = first; first = \_s; };

void push(State\* \_s, State\* \_e) { start.push(\_s); end.push(\_e); };

pair<State\*, State\*> get() { return make\_pair(first, second); };

~State\_Stack()

{ delete second;

delete first;

while (!start.empty())

{

delete start.top();

delete end.top();

start.pop();

end.pop();

}

}

};

int number;

static Specific\_symbol\* singleton;//单例模式

bool \_trans;//转义符号

bool \_collection;//[]符号

unordered\_map<char, func> symbol;//自定义可以添加的符号

State\_Stack\* \_stack;

Specific\_symbol();

Specific\_symbol(const Specific\_symbol&) = delete;//不允许拷贝和赋值

Specific\_symbol & operator = (const Specific\_symbol&) = delete;

void clear();

public:

static Specific\_symbol\* getInstance();

~Specific\_symbol();

void insert(char, func);

func find(char);//找到对应字符对应的函数

State\* getStart();

//初始化函数

friend State\* initfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//普通函数

friend State\* normal(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//\*函数

friend State\* asteriskfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//|函数

friend State\* orfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//（函数

friend State\* lparenthesis(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//）函数

friend State\* rparenthesis(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//￥结束函数

friend State\* finfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//？函数

friend State\* switchfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//收集函数(在[]中)

friend State\* collectionfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//-函数

friend State\* connectfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

// \转义函数

friend State\* transfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//[函数

friend State\* leftcollectfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//]函数

friend State\* rightcollectfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

//+函数

friend State\* plusfunc(Specific\_symbol\* \_sp, State \* \_state, char \_val);

};

struct State

{

public:

bool start;//是否是起点

bool end;//是否是终点

int num;//状态号

multimap<char, State\*> next;//后继节点

State(int \_n, bool \_s, bool \_e) :num(\_n), start(\_s), end(\_e), next() {};

~State() {};

};

**核心算法：**

Regex中

for (auto i : re)

{

func op = symbol->find(i);

p = (\*op)(symbol,p, i);

}

为分析正则表达式的核心算法。Symbol为Special\_symbol的实例，re为字符串。Symbol->find返回值为函数指针，声明为typedef State\* (\*func)(Specific\_symbol\*,State \*, char val);

通过symbol中的一个自定义的栈的结构对于()进行处理，同时存储的当前节点和前继节点帮助了\*，？，+等函数的实现。

经过这个操作形成nfa的结构。具体的func种类和实现可以自己在state\_operate.h中添加并在state\_operate.cpp中实现，目前实现了^,$,+,[,],\*,|,?,(,)，-类型的函数

Nfa转dfa的核心部分为寻找该节点所有等价的可以通过e到达的节点。函数为

void e\_set(DFA\_State \* set)

void e\_find(State\* state, DFA\_State \* set)

通过递归寻找所有等价的节点。从而由一个nfa节点生成一个dfa节点

Nfa转dfa的实现是先生成根节点的dfa。放入数组中。

接下来循环每次都将一个新的节点进行所有方向的分析。生成的新的节点放入数组中。倘若数组个数没有增加则代表结束。

同时重载了DFA\_State的==号，有两种形式

bool operator== (const DFA\_State & a);

bool operator== (const DFA\_State \* const a);

第一种是用来在nfa生成dfa时排除掉重复的dfa状态，使得最后生成dfa的过程结束。

第二种是用来在dfa转最简dfa中判断两个节点是否等价时使用。

Dfa转最简dfa中就是使用第二种==号的重载，对于每个节点和已经确认不重复的节点比较。相等则排除，不等则使用。

在分析lex中。运用了非贪婪匹配组。算法为

int DFA::group\_str(const string & ma, const DFA\_State \* const now, int index) const

{

if (now->end)

{

return 0;

}

else

{

if (std::size\_t(index) >= ma.size())

{

return -1;

}

auto it = now->\_next.find(ma[index]);

if (it != now->\_next.end())

{

int num = group\_str(ma, it->second, index + 1);

return num == -1 ? -1 : num + 1;

}

return -1;

}

}

通过递归寻找最短的相匹配的串。用来替换lex中如id {letter}({letter}|{digit})\*中{letter}的字符串为letter代表的正则表达式。

在分析文本中，采用贪婪匹配的方式，同时解决了lex冲突

具体算法为

using std::size\_t;

ifstream file;

file.open(path);

string str;

while (!file.eof())

{

getline(file, str);

size\_t start = 0;

string word;

size\_t index = 0;

while (start < str.size())

{

string p = str.substr(start);

//cout << p <<endl;

pair<int, string> temp(default\_regex->greedy\_match(p), "DEFAULT");

for (auto it : checklist)

{

int i = it.first.greedy\_match(p);

if (i > temp.first)

{

temp = make\_pair(i, it.second);

}

}

if (temp.first <= 0)

{

break;

}

if (temp.second != "DEFAULT")

{

result.push\_back(Token(temp.second, p.substr(0, temp.first)));

}

start = start + temp.first;

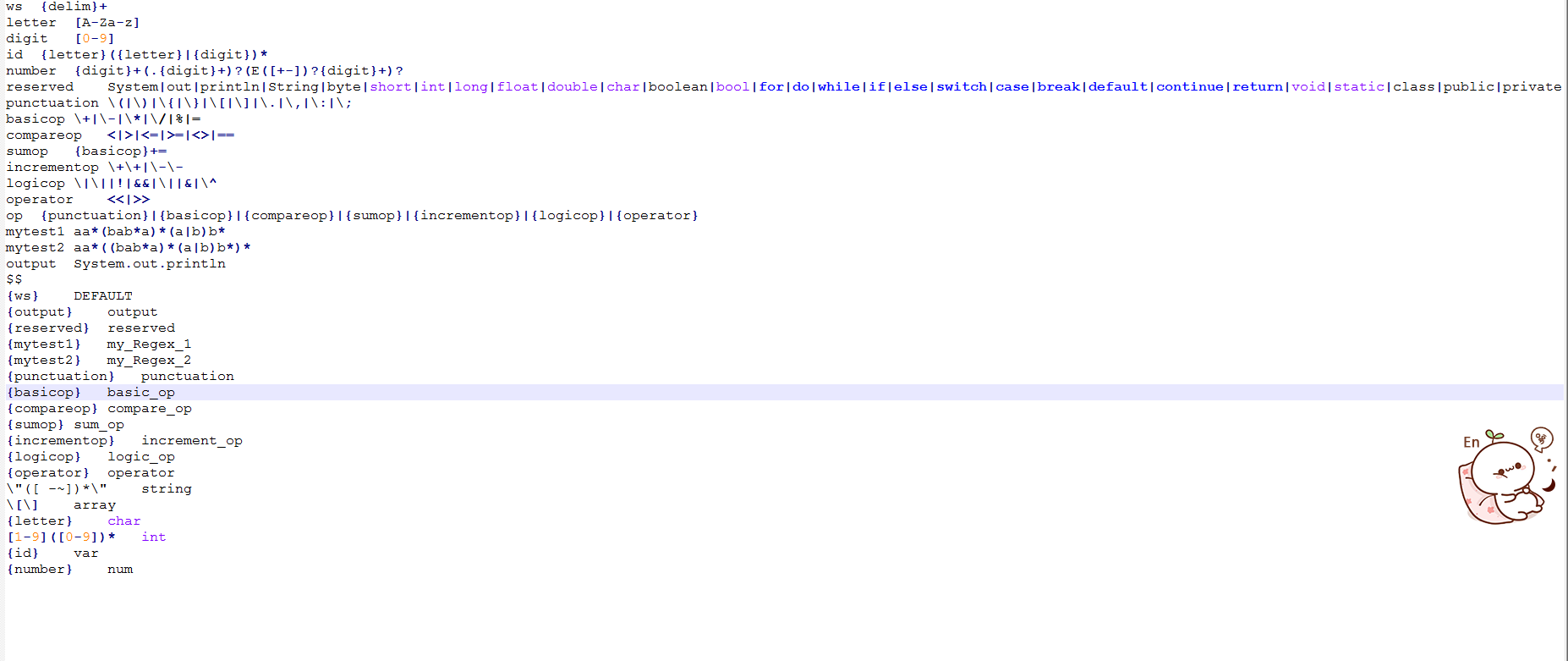
}

}

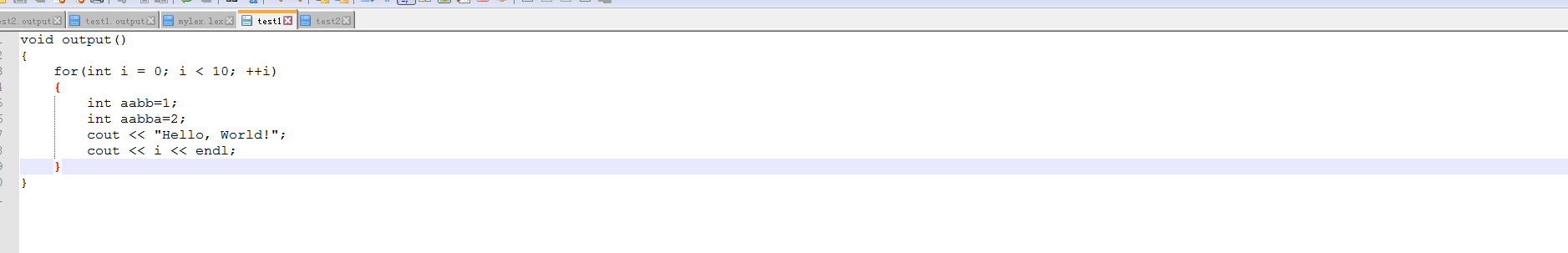
对于字符串每次都贪婪匹配最适应最长最先出现的正则表达式，匹配之后下一次从该位置往后再进行匹配。贪婪模式的匹配类似于非贪婪模式，略有一点小变化。这里就不做展示，在文件dfa\_to\_sdfa.cpp中实现

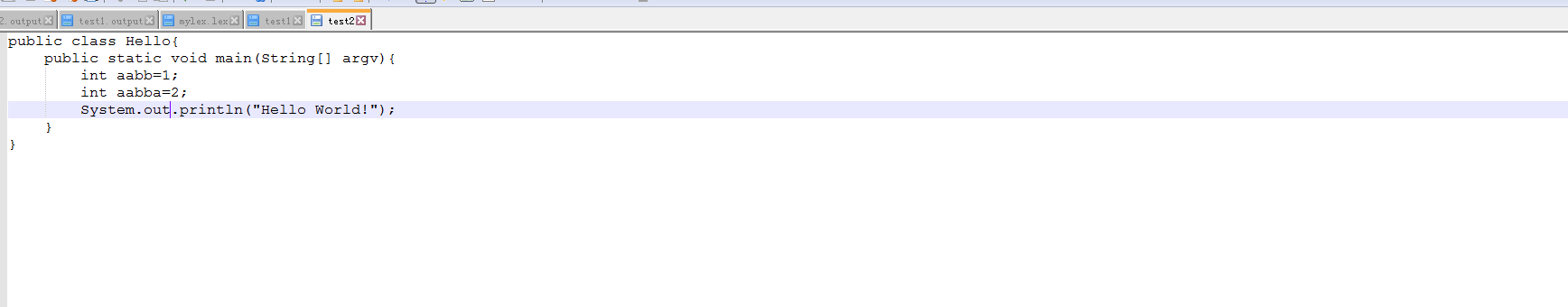
#### Use cases on running

输入的lex文件路径为lex/mylex.lex

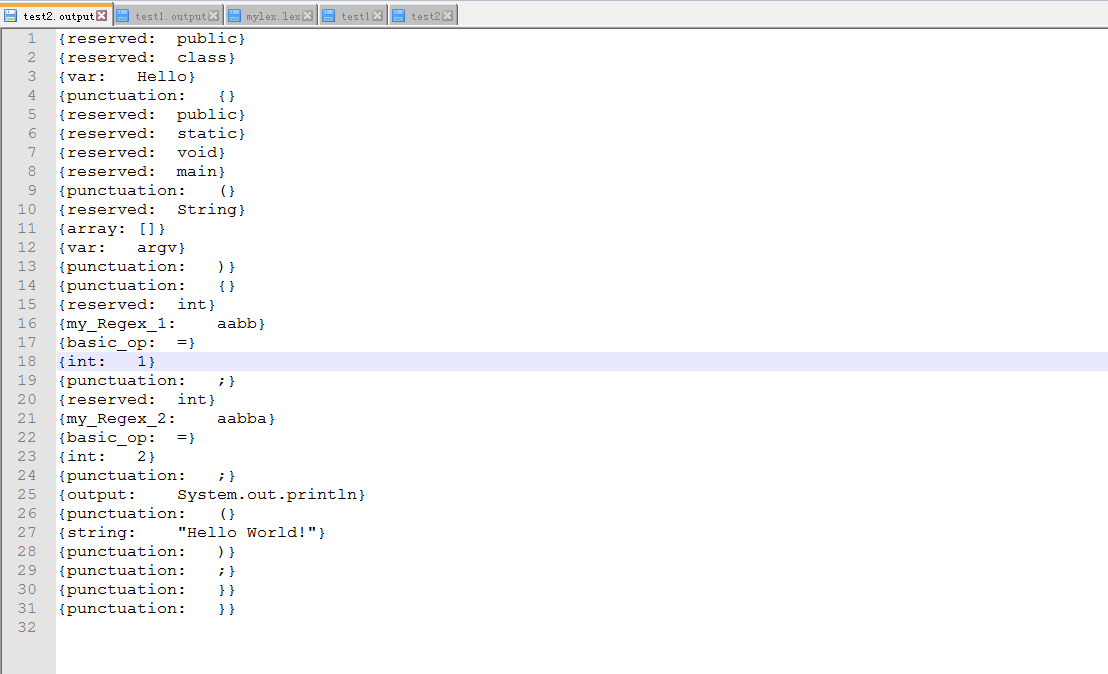


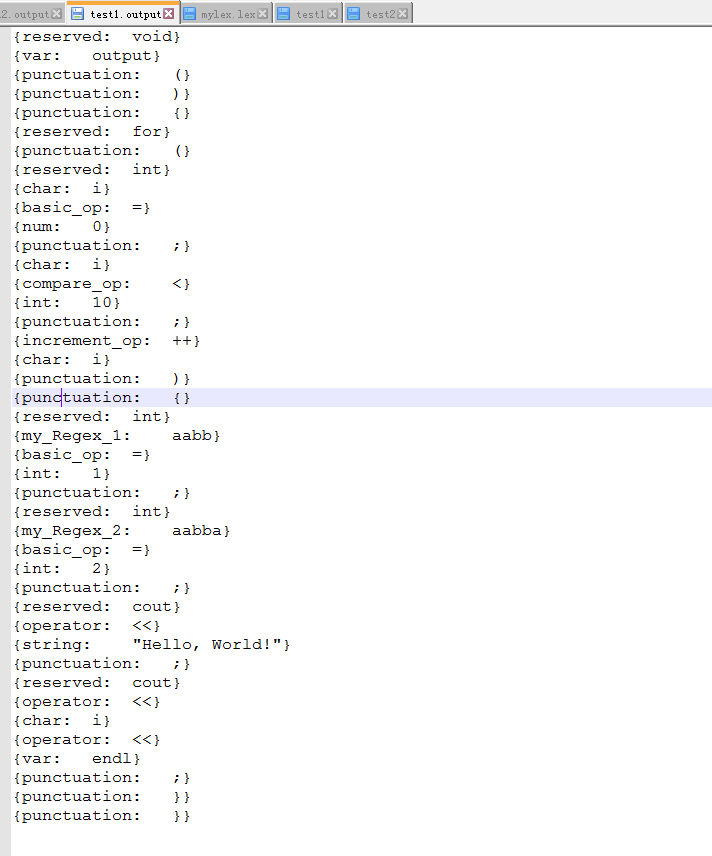
两个输入的分析文本的内容，路径为test\_output/下





输出为





1. **Problems occurred and related solutions**

问题1：对于正则表达式中\*的处理一开始想的比较简单。只是当前节点指向前节点并且吧当前节点转移到前一个节点。这样会造成一个问题，如(a|b)\*c\*时。

生成的nfa会使得从c返回后还能经过a或者b

解决方案：

在每次进行普通函数操作与分支操作时都生成一个空路径，之后由空路径开始继续发展。这会使得上例中c\*返回的节点并不是(a|b)\*返回的节点。而是(a|b)\*返回节点的后继。所以无法再次经过(a|b)\*。

问题2：

对于重复的dfa节点。本来想利用set<\_ty,less<\_ty>>中重载dfa\_state的<符号来方便的实现，后来发现对于dfa\_state中的nfaset(即nfa节点的集合)，并不方便明确的定义<号，因为set判断是否重复时用a<b||b<a来实现的。

解决：

最后只能重载==号并且使用vector来装载dfa\_state并在判断重复时遍历vector来判断

**j) Your feelings and comments**

这次实验让我更加了解正则表达式的分析和转换。同时也让我深刻的理解到书上对于|符号和\*符号在正则表达式中出现时状态图为什么需要那么多看似冗余的空路径。