

东南大学

编译原理课程设计

设计报告

组长： 09019204 曹邹颖

成员： 09019231 许志豪

09019104 陈逸彤

东南大学计算机科学与工程学院

二〇22年4月

设计任务名称		SeuLex	
完成时间	2022-04-09	验收时间	2022-05-22
本组成员情况			
学 号	姓 名	承 担 的 任 务	成 绩
09019204	曹邹颖	1. NFA的确定化、DFA的最小化 2. DFA转数组（跳转表） 3. 整体架构设计和优化 4. SeuLex应用 5. 报告的前三部分、第六部分	
09019231	许志豪	1. 正则表达式的规范化 2. 一个正规表达式到NFA的转换算法实现 3. 整合NFA 4. 报告第三、四、六部分	
09019104	陈逸彤	1. Lex输入文件的解析 2. Lex.cpp文件输出 3. 报告的第五、六部分	

1 编译对象与编译功能

1.1 编译对象

以c99.l文件为标准，可以实现C语言全集（在1999年推出的c99是C编程语言标准的过去版本）作为编译对象。将c99.l文件的格式进行了调整，如下：

```
{辅助定义部分}
%%
{识别规则部分}
%%
{用户子程序部分}
```

其中，辅助定义部分改为先以%{开始,%}结束声明用户自定义变量、常量和头文件，再声明正规表达式定义。同时，在词法定义部分，重写了一些不够严谨的正则表达式（如将 $L(\{L\}|\{D\})^*$ 重写为 $\{L\}(\{L\}|\{D\})^*$ ），其余部分没有删改。

具体请参考 .../SEULex/c99.l。

1.2 编译功能

项目整体功能包括：

- ① c99.l输入文件的解析，主要在 main()函数中完成；
- ② 正则表达式的规范化，主要在 StandardRE()函数中完成；
- ③ 由正则表达式构造 NFA，合并多个NFA，主要在 makeNFA()函数中完成；
- ④ 由 NFA 构造 DFA、DFA 最小化，主要在 NFA2DFA()、minDFA()函数中完成；
- ⑤ 词法分析器C++代码的生成，主要在 constructLexAnalysis()函数中完成。

项目还有一些特色功能，如NFA确定化时跳转表的可视化、C++代码的美化、C++代码的自动编译等。

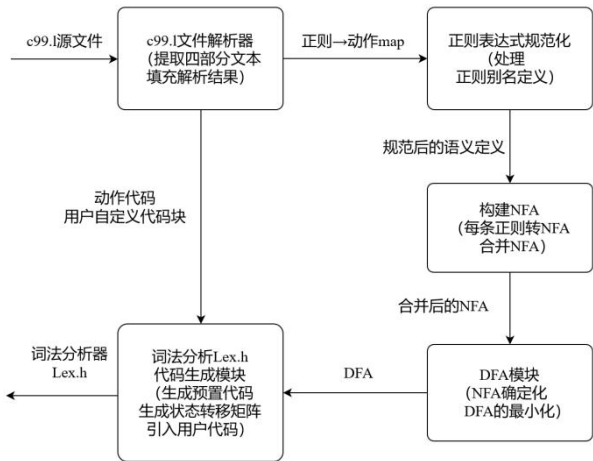
2 主要特色

- ① 实现了最长匹配原则
- ② 支持yywrap多输入文件处理、yytext 获取当前匹配字符串、ECHO输出yytext
- ③ 广泛使用接口与面向对象编程，代码各部分独立，从而易于调用
- ④ 代码具备一定的错误处理能力
- ⑤ 支持正则表达式四大元符号：?+*|（0 或 1 次，1 次或以上，0 次或以上，或）
- ⑥ 支持正则表达式范围与范围补（ASCII 为全集）：[A-Za-z0-9_]、[^]
- ⑦ 支持正则别名定义
- ⑧ 可以使用范围型转义字符\d([0-9])和\s([\t\r\n])
- ⑨ 运用C++的 regex 库以提供用于表示正则表达式和匹配结果的基本类型
- ⑩ 一些增强功能：NFA确定化跳转表的可视化、美化最终生成的 C++代码

3 概要设计与详细设计

3.1 概要设计

SeuLex分为c99.l文件解析器、正则表达式规范化模块、NFA 模块、DFA 模块、词法分析Lex.cpp代码生成模块。模块之间的关系和职责可以用如下的工作流程图来表示：



3.2 详细设计

3.2.1 C99.l文件解析的详细设计

Lex 文件解析器需要解析的c99.l文件主要分为四个部分：一，用户自定义变量、常量和头文件（以%{开头，以}%结尾）；二，正规表达式定义（在第一个%%之前）；三，保留字、正则表达式与相应的动作（起始于第一个%%，终止于第二个%%）；四，用户自定义子例程段（第二个%%之后）。它们可以通过对文件内容逐行扫描，对前面提到的定界符进行判别得到。注意：解析的同时要判断c99.l文件的结构是否符合要求，如果结构不正确需要报错处理。

对于第一和第四部分，处理方式简单：直接读入并存入Lex.cpp即可；对于第二部分，逐行读取每个正规表达式的表示符号与定义，并用结构体变量存储它们，并建立起这两者的对应关系；至于第三部分，因为每行的词法规则构成是保留字或者正则表达式 + 动作，中间用若干数量不一的制表符分隔开，所以我们先按行读取，然后按制表符将保留字或者正则表达式与动作分开，分别存入对应的变量中，此外，我们还要将第三部分使用的表示符号替换为正规式（即使用第二部分的对应关系进行替换）。

使用的变量
<pre>string ID; // 表示符号ID string RE; // 正则表达式RE Pair* newRule(string first, string second); // 正规表达式的表示符号与定义</pre>

3.2.2 正则表达式规范化的详细设计，即StandardRE()函数

首先StandardRE函数出现在识别词法规则的部分，主要的作用是识别正规表达式，将正规表达式的表示符号替换为正规式。例如，在c99.1中出现的

```
{L}({L}|{D})*      { count(); return(check_type()); }
```

其中{L}会被替换成[a-zA-Z]、{D}会被替换成[0-9]。而c99.1文件中，语句的格式较为统一，总结可以分为“”纯关键字”型、“”关键字”与{RE标识符}与RE连接符混合”型、“”符号”型。所以区分操作较为方便，关键在于找“{”，也就是找到所有的RE标识符。考虑到可能有多个“{”，函数内部的操作需要不断循环、处理遇到的所有的“{”。这里首先就出现了第一种return情景，即函数处理语句时未遇到“{”，那么函数就可以直接返回输入的语句参数，因为原语句并不包含RE标识符；而在处理遇到“{”的情况时，又有两个分支，即“{”有对应“}”和无对应“}”的情况，接下来先处理有对应的情况，从“{”的位置出发，开始往后遍历，如果找到了下一个“}”，就跳出遍历，而如果没找到下一个“}”，那么就直到遍历完整个语句，再跳出遍历。（注：这里可能会出现多层“{}”嵌套，所以考虑到这种情况，会使用一个count记录遇到的“{”的个数和“}”的个数，遇到了“{”就count+1，遇到了“}”就-1，最终只要count==0，就证明“{}”是一一对应的，准许提前跳出遍历），这里我们先不继续说明，而是回到未对应情况，“{”未对应情况，我们就可以同样利用count来判断，只要结束遍历之后，count!=0我们就直接返回传入函数的语句参数，因为若“{}”不成对出现，说明非RE的表示符号，而是单括号本身。再回到“{”对应的情况，此时“{”中的就是RE标识符，通过和一开始建立的“RE标识符-RE”map里去搜索，就可以将RE标识符一一转换，然后再和“{”左侧的部分与“}”右侧的部分重新拼接，并且函数return这拼接后的语句，就成功将正则表达式规范化了。

3.2.3 构建NFA的详细设计，即makeNFA()函数

补充：

NFA是允许两个节点之间有多条重边的，所以在一开始对边的结构体变量定义中，使用到看指针数组类型，具体如下

```
struct Edge
{
    string symbol; // 边上符号
    Edge* next = NULL;
    Node* nextNode = NULL;
};
```

可以看到每一个Edge的定义中使用到自身指针类型Edge*作为成员变量，表示了两点之间可以有多个重边。

makeNFA()函数部分主要靠内部不断循环使用边处理函数processEdge(Node* node, Edge* line)来实现，函数部分如下：

```
void makeNFA()
{
    for (Edge* i = NFA->edge; i != NULL; i = i->next)
        processEdge(NFA, i);
}
```

通过遍历Edge数组里的所有的边，并且对于每一个边（即边）进行处理，实现NFA的构造。

接下来主要对于processEdge函数进行分析。首先获得输入参数Edge上的symbol参数。第一步判断symbol的长度，如果symbol长度<=1，那就说明边上的字符为空字符或单字符，不存在复合字符的情况，不需要对与边进行处理。实际情况如下：

```
string str = line->symbol;
if (str.size() <= 1) // 空串边、单字符边不用处理
    return;
```

然后处理其他情况，第一种情况，检测到str（边上的字符，后同）第一个字符为“[”符号，此时可能就出现了字符为[1-9]{0-9}*{((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))}这样的情况当然也有另外的情况如“[1-9]” “[a-z A-Z]”，经有一段表达式... 首先来处理最简单的情况，也就是仅有一段正则表达式的情况。

首先，仍然是处理“[”的另一半问题，如果没找“]”，就说明是普通“[”，直接结束总的第一种情况，进入到另一种情况下处理。

接下来，如果找到了另一半，那就是正常的正则表达式，对于正则表达式的处理，这里采用取巧的方法，由于1-9加上大小写的所有字母符号，基本已经囊括了所有的键盘字符了，就采用ASCII码的方式，从32-126遍历全部的键盘输入符号，然后每一个输入符号和这里的正则表达式进行正则匹配，如果匹配上了，那么就利用这个输入符号在processEdge输入参数Node的原Node和下一个Node两点之间新增加一条边，这样就将单边上的单个正则表达式完全解析为多条边的一个DFA。

```
bool b = true;
for (int j = 32; j < 128; j++) // 32-126分配了能在键盘上找到的字符，127是DELETE命令
{
    string st = "";
    st += (char)j; // ASCII码转对应字符，再改为字符串
    if (regex_match(st, regex(str))) // 调库，是否为正则表达式str能表示的字符
    {
        if (b) {
            line->symbol = st;
            b = false;
        }
        else
            newEdge(st, line->nextNode, node); // 若匹配成功，则加一条边
    }
}
```

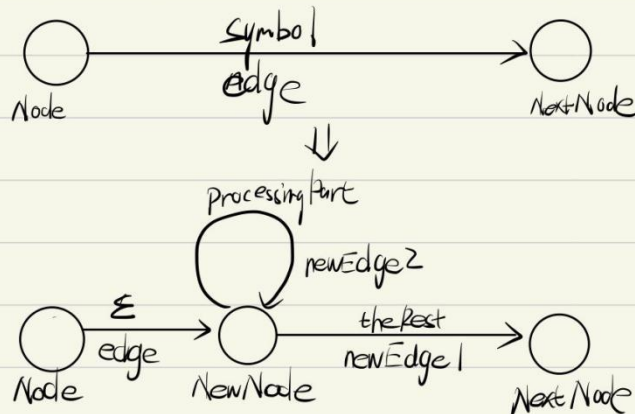
for循环从32到128，ASCII码转字符，然后进行regex匹配，匹配成功加一条边，在进入下一个循环。最简单的情况结束。

然后，基于最简单的情况，考虑一个更为复杂的情景，“[]”后面有剩余字符，以及单个“[”的处理。（补充，若“[]”后有剩余字符，都是以““[]”连接符“剩余字符”形式出现的）

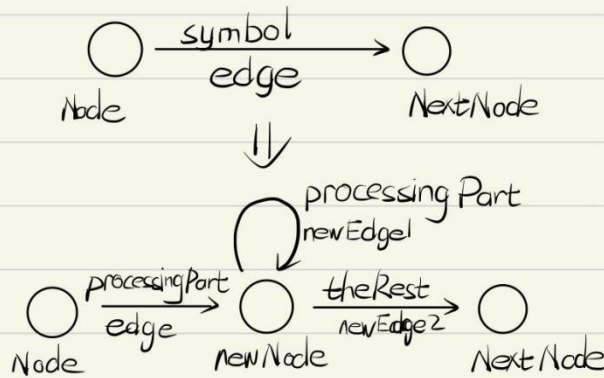
针对“[]”后面有剩余字符，首先提取原有的“[]”内的字符，作为processingPart以及提取连接符后面的部分作为theRest，连接符则有“*” “+” “?” “[”四种。

接下来对于四种连接符分别进行解释如何处理，当遇到“*”时，首先，新增一个节点用于插在原有边连接的两点之间，同时将原有边的起点与新增节点用一个新的边连接（原有边起点→新增节点），并且将边的symbol置为ε，将原有边的终点与新增节点用一个新的边连接（新增节点→原有边终点），并且将边的symbol置为theRest，而新增节点与其自身再增加一个边（即指向自己的边），并且将symbol置为processingPart。图示如下：

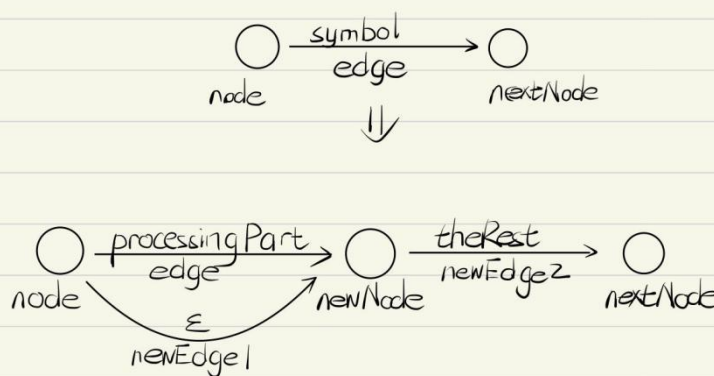
“*”的过程



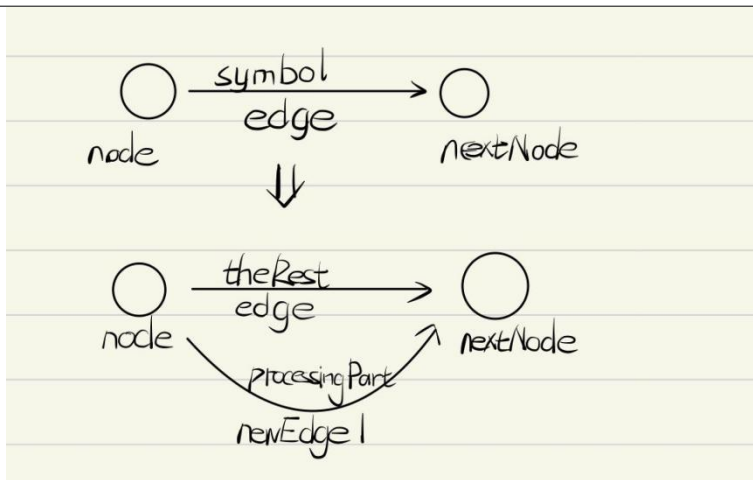
当遇到“+”时，与“*”的情况大致一样，唯一区别在于原edge的symbol不在置为“ε”，而是processingPart。如下图：



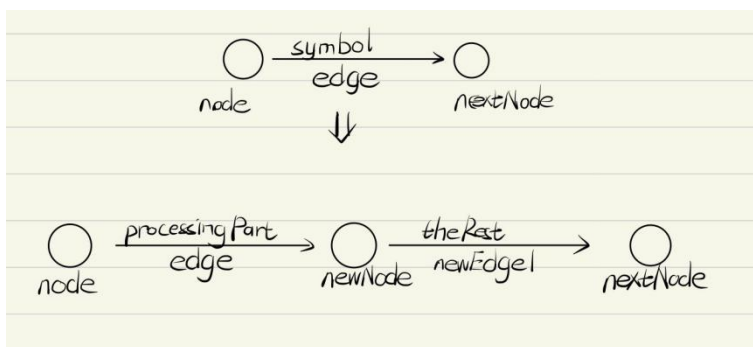
当遇到“?”时，大致流程仍与“*”一致，但是不同点在于，newNode没有指向自己的边，newEdge1成为一条空边，由node指向newNode。如下图：



当遇到“|”时，大致流程相对来说较为简单，只需增加一条newEdge1，作为原edge的平行边，指向仍然是node指向nextNode，newEdge1的symbol改为processingPart，而原edge的symbol改为theRest。如下图：

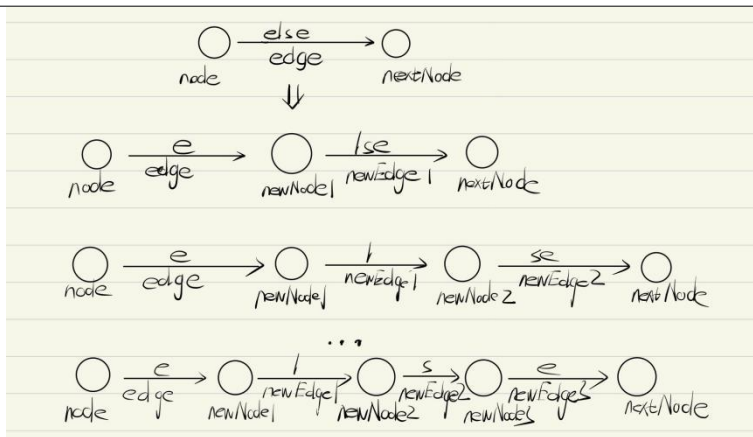


在这里，我们额外补充遇到单个“[”的情况，这时需要将“[”与后续的部分拆分开来，大致流程为，增加一个新节点newNode，同时增加一条新边newEdge1（newNode→nextNode），其symbol为theRest；原edge修改为node→newNode，并且symbol为processingPart。如下图所示：



以上，就处理完了“[”之后可能遇到的所有情况。接下来，处理其他情况，比如说第二种情况，遇到了“（”时的处理，类似于“[”，首先判断是否成对出现，当“（”成对出现时，并且后续无其他字符时，将“（”“）”中间夹的部分提取出来，并且去重新做processingEdge。而如果遇到了单“（”，“（”后包含连接符的情况，处理方式与“[”一致。

第三种情况，遇到了“\””的处理，本质上就是对于“””的处理，首先仍然是判断是否成对出现“””，如果成对出现就说明，出现了关键字，例如“else”，如果同时后续无连接符，那么处理方式如下，检测关键字的第一个字符（例如else的e），并且将edge的symbol改为该字符，同时检测关键字有无后续字符（例如else就是l，然后时s，然后是e，最后结束），如果有一个后续字符，就加一个newNode和一个newEdge，原edge改为（原edge起点→newNode），newEdge的symbol置为后续字符，指向为newNode→nextNode，如下图所示：



而如果遇到了“/”单符号或者后续有连接符的情况，处理同“[”。

第四种情况，遇到了“\\”，本质是遇到了转义符“\”的情况，c99.1中可能遇到转义符的情况如下，“\\”“\”“\”“\t”“\v”“\n”“\f”“\0”“\?”“\r”“\a”“\b”，由于转义符之后总是有别的字符跟随，所以就将转义符部分提取出来作为processingPart（比如遇到“\t”，就将“\t”作为processingPart），而转义符后面的部分用作theRest，然后仍然依照处理“[”后有连接符或者单边“[”的处理方式，处理processingPart与theRest。

第五种情况，即不属于以上四种任何一种情况时，处理方法如下，将第一个字符单独取出来作为processingPart，剩下部分作为theRest。然后依照处理“[”后有连接符或者单边“[”的处理方式，处理processingPart与theRest。本质上相当于将字符串拆分并且逐个加边或者作其他处理。

以上，完成对于makeNFA()函数的分析。

3.2.4 构建DFA的详细设计，即NFA2DFA()、minDFA()函数

确定有限状态自动机类构造时，接收一个NFA，进行确定化和可选的最小化。

NFA2DFA()函数中的详细设计

使用子集法(ϵ -闭包)进行NFA的确定化。由NFA $M = \{S, \Sigma, f, S_0, Z\}$ 构造一个等价的DFA $M' = \{Q, \Sigma, \delta, I_0, F\}$ 的算法如下：

① 取 $I_0 = \epsilon\text{-Closure}(S_0)$ ，加入状态集Q，这里定义状态集合I的 $\epsilon\text{-Closure}(I)$ ，为I中的任何状态s经任意条 ϵ 边能到达的状态的集合；

② 接着定义状态集合I的a弧转换 $f(I, a)$ ，表示可从I中的某一状态经过一条a弧到达的状态的全体。则当状态集Q中有状态 $I_i = \{s_1, s_2, \dots, s_j\}$ ， $s_k \in S$ ， $0 \leq k \leq j$ ，而且M机中有 $f(\{s_1, s_2, \dots, s_j\}, a) = \bigcup_{k=0}^j f(s_k, a) = \{s_1, s_2, \dots, s_t\} = I_t$ ，便对于每个输入符号a，计算 $U = \epsilon\text{-Closure}(f(I_i, a))$ ，即 $\delta(I_i, a)$ （DFA的状态转移）= U。并且，若U不在状态集Q中，则将U加入Q中；

③ 重复第②步，直至Q中不再有新的状态加入为止；

④ 取 $F = \{I | I \in Q, \text{且 } I \cap Z \neq \emptyset\}$ 。

在上面这个NFA确定化的算法中，我们借助Node struct结构体中定义的set<int> NFANodeSet保存NFA状态编号，将一个DFA状态对应于NFA状态子集，从而在构建DFA节点时通过set数据结构的=相等判断，知晓是否新建过对应该 ϵ -闭包的DFA节点，若无则通过newDFANode()函数新建节点。

同时，对于每一个新建的DFA节点（即新的 ϵ -闭包），都需通过makeDFATable()函数构建对应的跳转表，此处跳转表采用map数据结构map<string, set<int>>Symbols记录所有非终结符与对应的NFA节点编号集合，便可完成跳转表的可视化。

另外，我们知道，子集构造法的每个DFA状态都对应于若干个NFA状态，那么在一个DFA接收态中，有没有可能包括多种动作代码不同的NFA接收态？我们发现，答案是肯定的。这种动作代码的冲突问题需要借助动作的优先级，即借助Lex的正则-动作部分“先定义的优先高”的规则进行解决。

minDFA()函数中的详细设计

DFA的最小化算法[又称划分法(Partition)]的原则是将DFA中的状态划分成不相交的子集，在每个子集内部其状态等价，而在不同子集间状态均不等价（即可划分的）。最后，从每个子集中任选一状态作为代表，消去其他的等价状态。将那些原来射入其他等价状态的弧改射入相应的代表状态。

按照这个原则构造算法如下：

① 首先将状态集F划分为终态集和非终态集，即 $\Pi_0 = \{I_0^1, I_0^2\}$ ，设 I_0^1 为非终态集， I_0^2 为终态集；

② 假定k次划分后，已含有m个子集，记作 $\Pi_k = \{I_k^1, I_k^2, \dots, I_k^m\}$ 。这些子集到现在为止都是可区分的，然后继续考察这些子集是否还可以划分。设任取一子集 $I_k^i = \{s_1, s_2, \dots, s_t\}$ ，若存在一个输入字符a，使得 $f(I_k^i, a)$ 不全包含在现行 Π_k 的某子集中，这说明 Π_k 中存在不等价的状态，他应该还可一分为二，做一次划分；

③ 重复步骤②，直到所含的子集数不再增加为止。到此 Π 中的每个子集都是不可再分的了；

④ 对每个子集任取一个状态为代表，若该子集包含原有的初态，则此代表状态便为最小化后DFA的初态；若该子集包含原有的终态，则此代表状态便为最小化后DFA的终态。

以上过程可以在有限步内结束，因为状态数是有限的。

需要注意的是，对于终态，不能把它们放在同一个划分，而要在一开始就全部构建独立的划分（即一个新状态不能包含多个终态），这是因为终态上绑定着“动作代码”属性，因此它们之间不是真正等价的。

3.2.5 词法分析代码生成的详细设计，即constructLexAnalysis() 函数

在完成了上述所有步骤后，即可生成词法分析程序的 C 代码。具体分为如下几步：

- ① 直接复制c99.l文件文件解析后得到的第一部分，即：用户自定义变量、常量和头文件；
- ② C/C++程序必须包含的一些预申明，如：using namespace std;;
- ③ 词法分析函数定义string analysis(char *yytext,int n);
- ④ 状态标识符state初始化，状态数N;
- ⑤ 生成接收状态到动作代码的 switch case 的映射表，注意：若到了最后一个状态就跳出循环，若发什么错误就返回ERROR，还要特别处理所有的转义运算符。

yylex在匹配词法单元时，要遵循最长匹配原则。因此，每当到了接收状态，则暂时记录下来，然后继续进行状态转移，尝试接触下一个接收状态，进行更长的匹配。这样，记录的永远是当前看到的最长的接收状态；同长度情况下永远是最早出现接收状态（后出现规定不覆盖），符合 Lex 的优先级要求。直到无法进行更长的匹配，则把失败 的多余匹配全部回退，然后采取最近记录的接收状态进行接收，填充 yytext。

- ⑥ 直接复制c99.l文件文件解析后得到的第四部分，即：用户自定义子例程段。

增强功能的详细设计

- ① NFA确定化时跳转表的可视化：在makeDFATable()函数构建对应的跳转表时，此处跳转表采用map数据结构map<string, set<int>>Symbols记录所有非终结符与对应的NFA节点编号集合，便可完成跳转表的可视化。下面是一个示例：

I	a	b
$I_0=\{x,5,1\}$	$I_1=\{5,3,1\}$	$I_2=\{5,4,1\}$
$I_1=\{5,3,1\}$	$I_3=\{5,3,2,1,6,y\}$	$I_2=\{5,4,1\}$
$I_2=\{5,4,1\}$	$I_1=\{5,3,1\}$	$I_4=\{5,4,1,2,6,y\}$
$I_3=\{5,3,2,1,6,y\}$	$I_3=\{5,3,2,1,6,y\}$	$I_5=\{5,1,4,6,y\}$
$I_4=\{5,4,1,2,6,y\}$	$I_6=\{5,3,1,6,y\}$	$I_4=\{5,4,1,2,6,y\}$
$I_5=\{5,1,4,6,y\}$	$I_6=\{5,3,1,6,y\}$	$I_4=\{5,4,1,2,6,y\}$
$I_6=\{5,3,1,6,y\}$	$I_3=\{5,3,2,1,6,y\}$	$I_4=\{5,1,4,6,y\}$

- ② C++代码美化：生成的C++代码格式可能不够美观，我们使用一些简单的方法，如制表符、分析大括号层数等，对代码进行了一定的格式化。下面是一个示例：

美化前	美化后
<pre>int main() { if (...) { cout<<"Error"<<endl; } }</pre>	<pre>int main() { if (...) { cout<<"Error"<<endl; } }</pre>

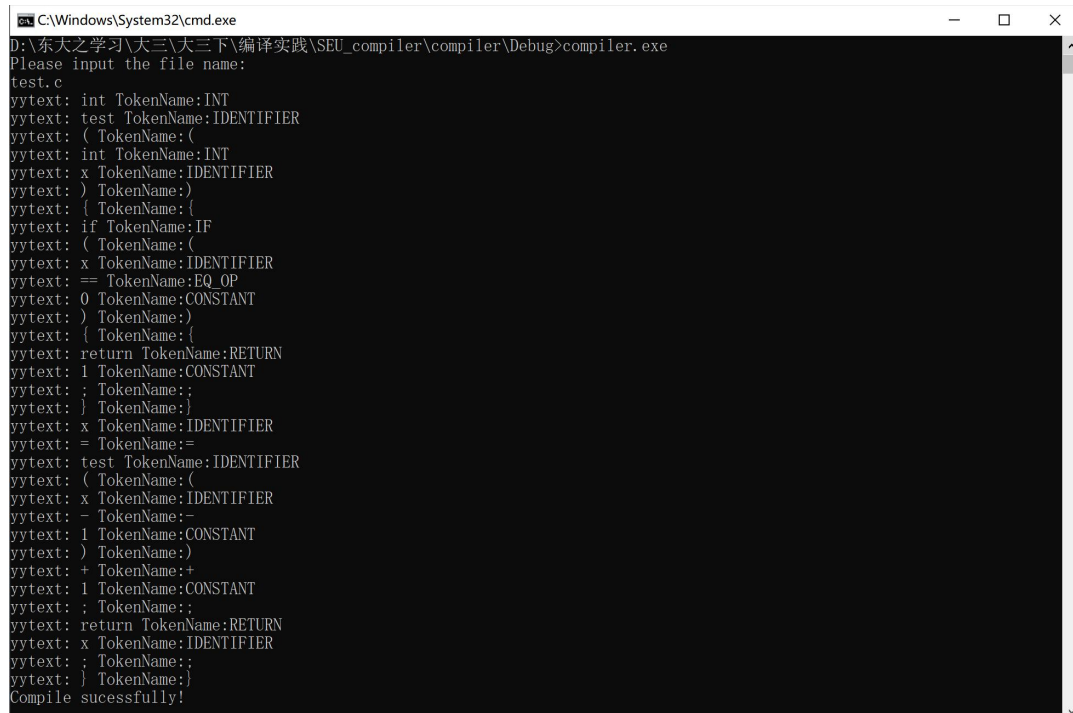
- ③ C++代码的自动编译：在调用SeuLex时，附加-c参数，则SeuLex会使用ChildProcess自动唤起编译器，传递合适的参数，编译生成Lex.cpp文件。

4 使用说明

测试 Lex.h 使用说明

我们通过VisualStudio在Debug模式下运行一遍我们的SeuLex项目文件，从而在项目文件夹下的Debug文件夹中便会自动生成对应的exe文件。

1. 找到exe文件，在对应的文件夹下打开cmd
2. 输入exe名称进行运行
3. 按照提示输入我们待解析的C99.1文件名
4. 便能得到结果



```
C:\Windows\System32\cmd.exe
D:\东大之学习\大三\大三下\编译实践\SEU_compiler\compiler\Debug>compiler.exe
Please input the file name:
test.c
yytext: int TokenName:INT
yytext: test TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: int TokenName:INT
yytext: x TokenName:IDENTIFIER
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: if TokenName:IF
yytext: ( TokenName:(
yytext: x TokenName:IDENTIFIER
yytext: == TokenName:EQ_OP
yytext: 0 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: return TokenName:RETURN
yytext: 1 TokenName:CONSTANT
yytext: ; TokenName;;
yytext: } TokenName;}
yytext: x TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: test TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: x TokenName:IDENTIFIER
yytext: - TokenName:-
yytext: 1 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: + TokenName:+
yytext: 1 TokenName:CONSTANT
yytext: ; TokenName;;
yytext: return TokenName:RETURN
yytext: x TokenName:IDENTIFIER
yytext: ; TokenName;;
yytext: } TokenName;}
Compile sucessfully!
```

5 测试用例与结果分析

使用我们的词法分析器进行测试。输入待分析的代码文件名，输出yytext与对应的TokenName。结果如下：

测试用例 1 (test_1.c)

源代码：

```
int main()
{
    double x;
    int n = 5;
    for (int i = 0; i < n; i++)
    {
        if (i > 3) x = 3.14;
        else x = 0;
    }
    return 0;
}
```

分析结果（按列从上到下阅读），经验证完全正确：

yytext	TokenName	yytext	TokenName
int	INT	i	IDENTIFIER
main	IDENTIFIER	++	INC_OP
(())
))	{	{
{	{	if	IF
double	DOUBLE	((
x	IDENTIFIER	i	IDENTIFIER
;	;	>	>
int	INT	3	CONSTANT
n	IDENTIFIER))
=	=	x	IDENTIFIER
5	CONSTANT	=	=
;	;	3.14	CONSTANT
for	FOR	;	;
((else	ELSE
int	INT	x	IDENTIFIER
i	IDENTIFIER	=	=
=	=	0	CONSTANT
0	CONSTANT	;	;
;	;	}	}
i	IDENTIFIER	return	RETURN
<	<	0	CONSTANT
n	IDENTIFIER	;	;
;	;	}	}

测试用例 2 (test_2.c)

源代码：

```
int squre(int x)
{
    int temp = x * x;
    return temp;
}
void main()
{
    const double Pi = 3.141592;
    int r = 0;
    double c, s;
    while (1)
    {
        if (r < 7)
        {
            ++r;
            continue;
        }
        else if (r < 0)
        {
            c = 0;
            s = 0;
            break;
        }
        else {
            c = 2 * Pi * r;
            s = Pi * squre(r);
            break;
        }
    }
}
```

分析结果（按列从上到下阅读），经验证完全正确：

yytext	TokenName	yytext	TokenName
int	INT	{	{
squre	IDENTIFIER	++	INC_OP
((r	IDENTIFIER
int	INT	;	;
x	IDENTIFIER	continue	CONTINUE
))	;	;
{	{	}	}
int	INT	else	ELSE
temp	IDENTIFIER	if	IF

=	=	((
x	IDENTIFIER	r	IDENTIFIER
*	*	<	<
x	IDENTIFIER	0	CONSTANT
;	;))
return	RETURN	{	{
temp	IDENTIFIER	c	IDENTIFIER
;	;	=	=
}	}	0	CONSTANT
void	VOID	;	;
main	IDENTIFIER	s	IDENTIFIER
((=	=
))	0	CONSTANT
{	{	;	;
const	CONST	break	BREAK
double	DOUBLE	;	;
Pi	IDENTIFIER	}	}
=	=	else	ELSE
3.141592	CONSTANT	{	{
;	;	c	IDENTIFIER
int	INT	=	=
r	IDENTIFIER	2	CONSTANT
=	=	*	*
0	CONSTANT	Pi	IDENTIFIER
;	;	*	*
double	DOUBLE	r	IDENTIFIER
c	IDENTIFIER	;	;
,	,	s	IDENTIFIER
s	IDENTIFIER	=	=
;	;	Pi	IDENTIFIER
while	WHILE	*	*
((squre	IDENTIFIER
1	CONSTANT	((
))	r	IDENTIFIER
{	{))
if	IF	;	;
((break	BREAK
r	IDENTIFIER	;	;
<	<	}	}
7	CONSTANT	}	}
))	}	}

6 课程设计总结（包括设计的总结和需要改进的内容）

09019204 曹邹颖

SeuLex的设计过程使我对构造词法分析器的基本原理有更为深入的理解和掌握，将词法分析过程的理论运用到实际，更加熟练地构造与化简DFA。在设计SeuLex的过程中，我充分实践了将NFA确定化、最小化DFA这两部分工作，难点便是将RE-NFA-DFA的转换图、转换表映射到代码算法中，不止有NFA-DFA转换表的可视化还有一些便利的数据结构做辅助；最小化划分时，因为有多个终止状态对应着不一样的表达式，所以对终结符需要逐一划分。

SeuLex仍存在DFA最小化不够完善的问题，并且在基于最小化的DFA生成的相应词法分析程序阶段，有着冗余代码，虽然没有逻辑问题但是会存在生成代码过长的问題，有待改进。

09019223 许志豪

项目中的lex进行实现时使用了一定的模块化设计，将lex文件解析、正则式Regex、NFA、DFA与代码生成各自作为一个独立的模块，通过各接口进行相互之间的交互。这一设计结构极大地方便了具体功能的开发过程。本人在lex部分的开发中主要实现了生成NFA的功能以及对lex文件解析的部分功能，实现过程基本按照预先搭好的架构进行，因此较为顺利，切实感受到了一个好的架构对程序开发的重要性。

但lex部分仍然有待改进之处，其中最为明显的是运行速度，因为开发过程中并没有注重考虑运行效率，因此许多地方存在一定的优化问题有待改进。

09019104 陈逸彤

这次的SeuLex难度颇高且工程量巨大，不过我负责的部分相对比较简单——C99.1文件解析和词法分析代码生成的详细设计。前者是对C99.1文件中的数据进行简单的处理并存储下来，后者是将我们构建的DFA用代码的方式最终实现出来。感谢队友们的帮助，我顺利地完成了我的任务。

然而，我们的项目也存在需改进之处，如果考虑到效率的话，项目中存在的多重for循环确实是一个需要优化的部分。如果测试用的正则表达式极其庞大复杂，则会影响到程序运行的效率。

7 教师评语

签名：

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做PPT文件、本设计报告等一切可放入光盘的内容的光盘。