東南大學

编译原理课程设计

SeuLex 报告

东南大学计算机科学与工程学院

二〇一九年六月

1 编译对象与编译功能

1.1 编译对象

我们参考了1999年ANSI标准C语法，删除了部分有关十六进制数字的匹配部分，其他未做任何删减。

1. 词法中的基本元素

- 十进制整数： digit [0-9]
- 字母： letter [a-zA-Z_]
- 标识符： identifier {letter}({letter}|{digit})*
- 浮点数： number [-+]?{digit}*.?{digit}+

2. 词法正规表达式

- 多行注释："/*"
- 单行注释："//"[^\n]*
- auto 关键字："auto"
- _Bool 关键字："_Bool"
- break 关键字："break"
- case 关键字："case"
- char 关键字："char"
- _Complex 关键字："_Complex"
- const 关键字："const"
- continue 关键字："continue"
- default 关键字："default"
- do 关键字："do"
- double 关键字："double"
- else 关键字："else"
- enum 关键字："enum"
- extern 关键字："extern
- float 关键字："float"

- for 关键字："for"
- goto 关键字："goto"
- if 关键字："if"
- _Imaginary 关键字："_Imaginary"
- inline 关键字："inline"
- int 关键字："int"
- long 关键字："long"
- register 关键字："register"
- restrict 关键字："restrict"
- return 关键字："return"
- short 关键字："short"
- signed 关键字："signed"
- sizeof 关键字："sizeof"
- static 关键字："static"
- struct 关键字："struct"
- switch 关键字："switch"
- typedef 关键字："typedef"
- union 关键字："union"
- unsigned 关键字："unsigned"
- void 关键字："void"
- volatile 关键字："volatile"
- while 关键字："while"
- 识别符：{L}({L}|{D})*
- 十进制整型：[1-9]{D}*{IS}?
- 字符字面值：L?'(\\.|[^\\'\n])+'
- 浮点型：{D}+{E}{FS}?
- 字符串字面值：L?\"(\\.|[^\\"\n])*\"
- 多字符操作符...："..."

- 多字符操作符>>=："">>=""
- 多字符操作符<<=："<<=""
- 多字符操作符+=："+=""
- 多字符操作符-=："-=""
- 多字符操作符*=："*=""
- 多字符操作符/=："/=""
- 多字符操作符%=："%=""
- 多字符操作符&=："&=""
- 多字符操作符^=："^=""
- 多字符操作符|=："|=""
- 多字符操作符>>："">>""
- 多字符操作符<<："<<""
- 多字符操作符++："++""
- 多字符操作符--："--""
- 多字符操作符->："->""
- 多字符操作符&&："&&""
- 多字符操作符||："||""
- 多字符操作符<=："<=""
- 多字符操作符>=："">=""
- 多字符操作符==："==""
- 多字符操作符!=："!=""
- 单字符操作符 ;："";""
- 单字符操作符{：("{"|"<%")
- 单字符操作符}：("}"|"%>")
- 单字符操作符 ,："","
- 单字符操作符:："":""
- 单字符操作符=："="
- 单字符操作符(："("

- 单字符操作符)："")"
- 单字符操作符[：("["|"<:")
- 单字符操作符]：("]"|":>")
- 单字符操作符.："."
- 单字符操作符&："&"
- 单字符操作符!："!"
- 单字符操作符~："~"
- 单字符操作符-："-"
- 单字符操作符+："+"
- 单字符操作符*："*"
- 单字符操作符/："/"
- 单字符操作符%："%"
- 单字符操作符<："<"
- 单字符操作符>：">"
- 单字符操作符^："^"
- 单字符操作符|："|"
- 单字符操作符?："?"
- 无含义的格式符号：[\t\v\n\f]
- 除换行符外任意字符：.

此外我们，编写了处理多行注释的正则表达式，这里我们对其进行一番阐释：

"/*+[^*]**+(?:[^/*][^*]**+)*/"

1. /*+ 匹配注释开始以及任意个数*

2. [^*]**+ 匹配除*以外的 0+个字符，后跟 1+个*

3. (?:[^/*][^*]**+)* 0+个非/且非*的字符后跟 1+个*

1.2 编译功能

- 辅助函数和相关数据结构：helper.h
- Lex 输入文件的解析：RE_Extraction.h
- 正规表达式的解析：RE_Standardization.h
- 一个正规表达式到 NFA 的转换算法实现：Re2NFA.h
- 多个 NFA 的合并：Re2NFA.h
- NFA 的确定化：NFAToDFA.h
- DFA 的最小化：DFAMinimization.h
- SeuLex 应用，生成 Lexer 代码： GenerateCCode.h
- 总控程序：SEU_Lex.cpp

2. 主要特色

1. 基本囊括了 1999 年 ANSI 标准 C 语法对 Lexer 的所要支持的全部 Token 种类要求。
2. 支持了丰富的非规范正规表达式操作符，同时支持字符串识别、多行注释等高级正则表达式的解析功能。
3. 生成的 lex.c 程序使用了数组来存储 DFA 表的信息，提高了运行效率，同时也通过动态的方式，将 lex.l 文件解析并生成成 Lexer。
4. 加入了 DFA 的最小化算法，项目经过优化，运行效率较高，运行仅需不到 5 秒。

3 概要设计与详细设计

3.1 概要设计

- ParseLexFile.cpp
  - read_and_parse_lex(string, map<string,string> &, vector<Rules> &, vector<string> &, vector<string> &)

    读入并解析.l 文件。
  - trim(string &s)

    去除字符串的首尾空格。
  - split(const string& str, const string& delim)

    字符串分割。
  - remove_comment(string& s)

    去除注释。
- RE_Standardization.cpp
  - re_Standardize(vector<Rules>& Vec, map<string, string>& M)

    将.l 文件中非规范的正规表达式标准化。
  - static void reMap(string& exp, const map<string, string>& reMap);

    处理.l 中的分层定义，处理花括号{}
  - static void replace_square_brace(string& exp,bool print=1);

    处理表示集合的方括号[]
  - static void replace_question_and_add(string& exp);

    处理表示字符数量的？和+
  - static void replace_dot(string &exp);

    处理表示除去换行符外的任意字符的.
  - static void add_dot(string &exp,bool);

    加点
  - void replace_escape(string& exp)

    处理转义字符
- REToSuffixForm.cpp
  - re_to_suffix(vector<Rules> &)

    将规范化的正规表达式转换为后缀形式。
- Re2NFA.h

- `string StandardForm2SuffixForm(string standardForm)`

  规范化的正规表达式转换为后缀形式。

- `NFA SuffixForm2NFA(string suffixForm, int tokenType)`

  将每个后缀形式的表达式转换为对应的小 NFA

- `NFA CombineNFA(vector<NFA> allMiniNFA)`

  将所有小 NFA 按照优先级 NFA 合并。

- `NFA GenerateNFA(vector<pair<string, int> >& REset)`

  根据 l 文件定义的所有正则表达式生成一个大 NFA

- NFAToDFA.cpp
  - `NFAToDFA(const NFA&, DFA&)`

    将 NFA 确定化为 DFA。

  - `epsilon_closure(unordered_set<int>&, const unordered_map<int, NFAstate>&)`

    对给定的 NFA 状态集合，求 epsilon 闭包。

  - `chooseAction(const DFAstate&, const map<int, int>&, int&)`

    对给定的 DFA 状态判断是否为终态，若为终态则确定其对应的语义动作。

  - `subset_construction(const unordered_set<int>&, unordered_set<int>&, const char, const unordered_map<int, NFAstate>&)`

    对给定的 NFA 状态集合，做对应出边的子集构造。

- DFAMinimization.cpp
  - `DFAMinimization(DFA&)`

    对传入的 DFA 进行最小化。

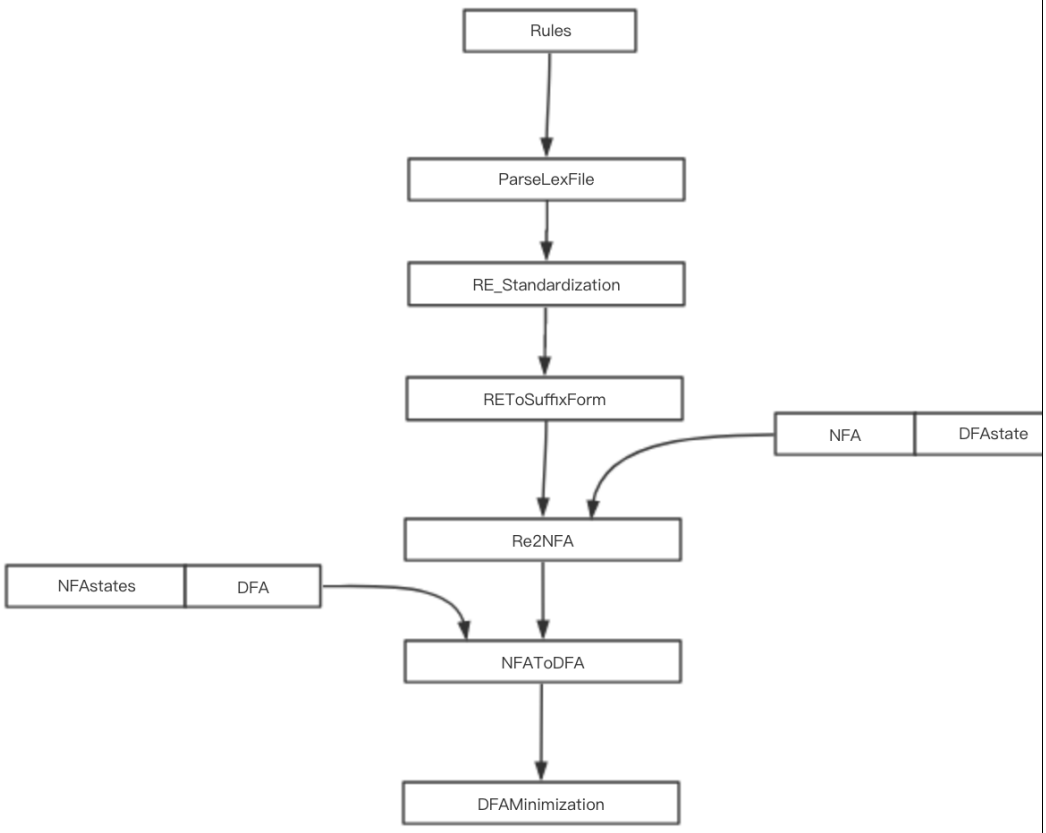
  - `splitDFASets(vector<unordered_set<int> > &, DFA&`

    对于给定的 DFA 状态集合，扫描所有出边判断映射状态是否在统一集合内，并将映射集合不同的状态分到不同的子集中。

- GenerateCCode.cpp
  - `generateCCode(DFA&dfa, vector<Rules>& lex_rules)`

    依据 DFA 状态和 lex 规则来生成 Lex.yy.cpp 文件，将 DFA 状态信息和 lex 规则信息转化为数组输出到数组中，放入词法分析程序中使用。

各个模块由 Main.cpp 中的 main()函数按顺序调用。
以下为 main()函数中调用顺序和数据流图：



3.2 **详细设计**

数据结构

- 存储正规表达式和对应动作的结构体

```
typedef struct {
    string pattern; //RE
    vector<string> actions; //对应动作（语句）
} Rules;
```

为了保证在随后的操作中能做到正规表达式和动作语句的一一对应关系，我们在一开始解析.l 文件时就使用 Rules 结构体记录这一关系。其中因为语句可以是不止一条的，所以使用 vector 来按行存储每条语句。

- 存储 NFA 状态的结构体

```
typedef struct NFAstate{
  int number = 0; //状态标号
  unordered_multimap<char, int> exEdgesMultiMap; //发出边,键为边上的值，值为下一个状态标号
} NFAstate;
```

每个 NFA 状态中使用 unorder_mulitmap 存储所有的发出边，一是考虑到使用 map 查询发出边对应的状态效率较高，同时又注意到 NFA 的状态可以有多个相同的发出边至不同状态，所以使用 unorder_mulitmap 来做存储。此外，我们在设计 NFA 状态的数据结构时，还充分地考虑了随后 NFA 转 DFA 算法的效率问题。之所以不将所有的发出边和状态分离集中存储，是考虑到随后的 epsilon 闭包和子集构造操作，需要查询每个状态的对应发出边，这样将发出边存储在状态内，减少了很多的查询操作。

- 存储 NFA 的结构体

```
typedef struct {
  int startState = 0;//开始状态标号
  map<int, Rules> endStatesMap;//存储终态和对应的动作
  unordered_map<int,NFAstate> statesMap;//存储标号对应状态
}NFA;
```

考虑到我们在 NFA 合并的算法中，状态标号是乱序的，故无法使用 vector 来存储所有的状态，否则将花费很多时间查询相应的状态从而降低效率。为了更快地通过状态编号来查询到对应的状态，我们使用了 unordered_map，其中键存储状态标号，值存储对应的 NFA 状态结构体。此外考虑到每个正规表达式都对应了 NFA 的一个终态，从而对应了一组动作语句，故我们还使用了 map 来同时存储所有的终态和它们对应的动作。

- 存储 DFA 状态的结构体

```
typedef struct  {
  int number = 0;//状态标号
  unordered_set <int> identitySet;//区分不同状态用的集合
  unordered_map <char, int> exEdgesMap;//发出边,键为边上的值，值为下一个状态标号
}DFAstate;
```

DFA 状态的结构体与 NFA 状态的类似，但是不同是，多了一个 unordered_set 来区分不同的 DFA 状态。这是考虑到，在 NFA 确定化的过程中，每一个 DFA 状态都包含了一个 NFA 状态集合，我们通过比较集合是否相同决定是建立一个新的 DFA 状态还是使用之前已存在的相同状态，此处我们直接在 unordered_set 中存储 NFA 状态标号，这样效率较高，但经过性能的测试与分析，我们发现集合的比较操作仍然是较费时的，运行时间占到总 NFA 确定化算法的 15%-30%。

- 存储 DFA 的结构体

```
typedef struct {
  unordered_map< int, Rules > endStatesMap;//终态及对应动作
  int startState = 0;//开始状态标号
  vector<DFAstate> statesVec;//存储标号对应状态
}DFA;
```

DFA 结构体的定义也和 NFA 的类似，不同点在于使用了 vector 来存储所有的 DFA 状态。因为 DFA 的状态是按顺序生成的，所以直接使用状态标号当做下标来寻址即可找到对应的状态结构体，这样查询效率相比 unordered_map 肯定是要高的。

算法

- 正规表达式的规范化
    - 非规范正规表达式的规范化

这部分是整个 seuLex 设计中最困难的部分。原因在于我们使用的是原始的 1999 ANSI C 语法对应的 lex 文件，其中包含了非常多复杂的非规范正规表达式形式，下面一一说明每种语法以及对应的规范化方法。

- 分层定义——{x}

Lex 文件中有两部分的正规表达式，其中一部分没有对应动作，有名称：L [a-zA-Z_]。这类正规表达式主要用于分层定义，其会在其余的正规表达式中被引用为：{L}。这类操作符的处理非常简单， 只需要遍历整个正规表达式，提取出花括号中的内容，查找到对应的正规表达式并将其替换花括号及其内容即可。我们在此部采用 map 的数据结构，将对应{ }内容替换成便于之后处理的[ ]内容。

- 多字符或—[x]和[^x]

遇到这类操作符将做如下翻译：

[xyz] -> (x|y|z)

即遍历整个正规表达式，提取中括号中的内容后，首先将所有的斜线转义为正常的转义字符，生成所有需要或起来的字符的集合。随后在每两个字符之间插入或|，最后在两边用小括号( ),替换中括号[ ]及其内容。

此处较难处理的是[^xyz] 这类操作符，其中^表示取补操作，结果为除了xyz 之外其他所有字符或起来。当做补集操作时，我们需要先定义字符全集。在参考了 ANSI 1999 C 标准后，我们定义了 C 语言中所有可能出现的合法字符的集合用于集合的取补操作。在取补获得新的字符集合后，操作同普通的中括号。在此阶段，采用适合做集合差运算的数据类型 set，将全局变量中的

const string ALLSET 解析成全集集合，再与[]内^后元素做差即可

- 引号转义——"xyz"

双引号的作用为转义，将内部的符号都当作字符来处理，因为考虑到其中可能存在例如 (,|等操作符，为了便于后续处理，我们使用的\\"表示来对正规表达式中可能出现的操作符进行转义。

- 斜线转义—— \n

.l 文件中的斜线转义字符是显式的。也就是说.l 文件中的\n，如果我们将其读入到字符串中，它就变成了\\n。这显然不是我们想看到的，我们仅仅想把它当成一个字符\n 处理。故我们需要遍历整个正规表达式，将显式的转义变成我们想要的形式。首先明确.l 文件中所有可能的显式转义情况，然后将其替换为正常的转义形式。

- 任意单个字符—— .

在正规表达式的语法中，用点操作符来表示除了换行符之外的任意单个字符，此时我们可以利用之前实现的中括号处理方法，将.看作[^\n]，这样即可将点操作符规范化。

- 问号和加号—— x? x+

问号操作符表示操作数出现 0 次或一次。此时我们需要涉及到 epsilon 符号，我们使用@表示。则 x? -> (@|x), x+ -> xx*。对于操作数为单个字符的情况，处理起来是非常简单的。较为复杂的是当操作数是被括号括起来的一个整体时，我们需要找到与操作符前的右括号对应的左括号。因此我们需要通左右括号的数目来确认需要处理的内容。此处我们此用一个变量进行计数：当遇到右括号时计数器+1，左括号计数器-1，当计数器为 0 时说明我们已经找到了配对的括号，将括号内的部分当作一个整体进行处理即可。

当我们实现了对以上多种非规范形式正规表达式处理之后，一个新的问题是

如何确定这些操作符（函数）的处理（调用）顺序。我们做出了如下规定:

- 先对带名字的正规表达式进行处理，之后再处理带动作的正规表达式。
- 对带名字的正规表达式仅进行引号转义和花括号替换。
- 操作符按照双引号、花括号、中括号、点、问号和加号、斜线这个顺序进行处理。
    - 加"点"

相比上述正规表达式的规范化处理，加点相对简单了很多，我们仅需要注意以下情况时后面不加点：

- 当前字符为我们定义的转义字符\\
- 当前字符为非转义的（和|
- 当前字符为正规表达式最后一个字符
- 当前字符的后一个字符为|、)和*

- 中缀形式的正规表达式到后缀形式的正规表达式转换

这里我们采用的算法是经典的调度场算法。用于将中缀表达式转换为后缀表达式的经典算法，由艾兹格·迪杰斯特拉引入，因其操作类似于火车编组场而得名。

正规表达式处理好后，转化为后缀主要使用了栈和队列这两个数据结构，大致实现思路如下：

1. 通过引用传递正规表达式数组，逐个取出进行处理。

2. 如果遇到字符就直接 push 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。

3. 如果遇到的是“（”、“）”、“·”、“|”、“*”这些操作符，那么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。

4. 我们定义了转义字符为“·”。所以当读入这个字符的时候会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，这里需要注意的是，如果后面跟着的是“？”、“+”、“{”、“}”、“[”、“]”，为了后续编程方便，则只读入后面的字符而忽略掉前面的转义字符`。

中缀正规表达式处理完成后，把栈中的元素全部放入队列尾部，再把队列

的值赋给原本传入的正规表达式，也就是修改结束。

- Thompson 算法(将后缀形式的正规表达式转化为 NFA)

后缀形式的正规表达式读入后，要将其转化为 NFA，这里的难点主要在于 NFA 的构造，我们考虑了很多种方式，比如指针方法构建 NFA 连接、基于边来构造 NFA 等方法，但是都遇到了难题。最终决定，让 NFA 只记录初态和终态，而中间状态都通过 `map` 的形式来访问，而每个 NFA 状态仅包含自身标号以及一个 `multimap`，后者主要用于判断通过某个字符或者 epsilon 后能到底哪一个状态标号。通过存储标号的方法使得数据结构轻量化而且简洁化。

读入后缀表达式后，下面来说说 Thompson 算法实现的思路，该算法主要依靠的数据结构是栈，总体思路是：不断取出栈顶的元素进行合并或者其他操作。细化来讲，分为以下几种情况：

1. 如果读到的是字符：则新建两个状态，初态和终态，将初态通过终态的标号连接到终态，连接时使用的 pair 说明了边上的字符，此时的边上的字符就是读入的字符。连接好后构建 NFA，设置好 NFA 的初态和终态，push 进栈中。

2. 读入的是“.”:读入“点”则需要对栈顶的两个元素进行连接操作，找出 downNFA（就是压在栈下面的 NFA）的终态，用 epsilon 与 upNFA（栈顶的 NFA）相连接，并且修改终态，为 upNFA 的终态，同时要注意的一步是要把 upNFA 的状态映射表（通过该表使用状态标号就可获得状态）拷贝到 downNFA 中。最后只把 downNFA 压入栈中，连接操作完成。

3. 读入“*”：读入“*”则需要做循环处理，取出栈顶的 upNFA，创建两个状态，作为初态和终态，用 epsilon 连接初态和终态，再将初态 epsilon 连入 upNFA 初态，将 upNFA 终态 epsilon 连入终态，再将 upNFA 的初态和终态也用 epsilon 连接，最后修改 upNFA 的初态和终态为新建的两个状态，压回栈中。

4. 读入“|”：读入或则需要将栈顶两个元素进行或操作。取出 upNFA 和 downNFA，新建两个状态，为初态和终态，初态通过 epsilon 边连接到 upNFA 和 downNFA 的初态，upNFA 和 downNFA 的终态通过 epsilon 连接到终态，将 upNFA 的 stateMap 拷贝到 down 中，修改 downNFA 的终态和初态，最后将 downNFA 存入栈中。

- 合并 NFA

这里需要注意的是要按照正则表达式的优先级来进行合并，我们采用.l 文件里的正则表达式出现的先后次序来进行合并。所以，我们按照每个小 NFA

最终接收节点的标号，从小到大依次进行合并。具体的算法思路如下：

NFA 的合并类似于构建二叉树，预先取出一个 finalNFA 作为最终的 NFA，循环取出栈顶元素后建立一个初态，通过 epsilon 连接到两个 NFA 的初态上，再修改 finalNFA 的起始状态，添加 finalNFA 的终止状态，最后把 downNFA 的状态 map 拷贝到 finalNFA 中。

- NFA 确定化算法
    - NFAToDFA 确定化过程

      1.DFA 初始状态为 NFA 初始状态的 epsilon 闭包，并将其加入队列

      2.通过维护未处理 NFA 状态集合队列来生成所有的 DFA 状态，每次对队列中第一个状态遍历所有可能出边并求 epsilon 闭包，判断是否为已经存在的 NFA 状态集合(DFA 状态)，并对应进行置入队列/舍弃操作。
    - `epsilon` 闭包求解

      将初始集合中所有状态入队列，并用 hash 数组记录哪些状态已经在集合中

      当队列非空时，执行以下操作
    - 找到队头状态通过 epsilon 边可以到达的状态
    - 队头状态出栈
    - 将这些状态编号中未处理的编号插入到状态集合中，并加入队列，用 hash 数组记录
    - 子集构造 subset_construction

      子集构造的算法非常简单，给定一个字符，从 DFA 状态中的 NFA 状态集合中，找到所有对应该字符的边，并将边另一端的 NFA 状态标号加入到子集中。
    - 确定终态对应的动作 chooseAction

      一个 DFA 状态是否为终态由其中是否包含 NFA 终态决定。如果 DFA 状态中仅包含一个 NFA 终态，则该 DFA 为终态，对应的动作即为该 NFA 终态对应的动作。如果一个 DFA 状态中包含了多个 NFA 终态，按照 lex 语法规定，在.l 文件中靠前的正规表达式拥有更高的优先级，其对应的动作优先执行。故我们的算法如下：

      遍历 DFA 中的所有 NFA 状态，如果该 NFA 状态为终态，如果该终态对应的动作在.l 文件中更靠前则该 DFA 状态对应的动作更新为该动作，若没有找到终态 NFA 则该 DFA 不是终态。

- DFA 的最小化

  DFA 的最小化算法中，我们首先将所有的 DFA 终态都划分到不同的子集中，所有的非终态先放入一个集合中，并将该集合放入处理队列中，在队列中进行新的子集的划分：

  - 取出队头 DFA 子集。
  - 完成同一 DFA 子集的所有状态、所有出边的遍历后判断是否产生了新的子集，若产生了新的子集则直接加入处理队列；若没有产生新的子集，则判断该 DFA 子集是否已经处理过，若是第一次处理则放入队列，来完成向前看的功能。

- Lex.yy.cpp 文件生成

  - 将 lex 程序计算获得的 DFA 结果和 Lex 规则转化为数组输出到语法分析程序 Lex.yy.cpp 中，作为状态转换表和 TOKEN 属性表。
  - 该语法分析程序以 Unit 结构体存储 TOKEN：

```
Struct Unit{
    string type; //存储 TOKEN 的属性
    string content;//存储 TOKEN 的内容
    int id;  //存储 TOKEN 的 ID
};
```

  - 在 main 中通过 input.get()每次读入一个字符直到文件尾，从 startState 开始每次调用 NextToken 来获取下一个 TOKEN。
  - NextToken 每次也是通过 input.get()读入下一个字符并判断当前状态是否可通过该出边进行转移(check 函数获取字符在 symbolVec 中的下标，便于从 DFA 状态转移表中获取目标转换状态)，若可以则将当前状态压栈，将字符接到 tkstring 尾部并进入下一 DFA 状态；若不可则退出循环。对状态栈一一弹栈直到栈顶状态为终态，同时也通过 input.seekg(-1, ios_base::cur)来回退文件读入指针的一个字符的位置，找到终态便找到了一个 TOKEN；若没有找到终态则会取一个字符标记为 Unknown,退出函数。
  - 找到 TOKEN 后判断是否在 TOKEN 序列中出现过，若没有则调用 add 函数作记录。最后，将 TOKEN 通过 Unit 保存计入词法分析程序的结果 TOKEN 序列中。
  - 完成所有字符的输入后输出 TOKEN 序列的结果。

4 使用说明

4.1 SEU_Lex 使用说明

这里，我们为了缩短编译时间和跨平台下的使用方便，我们为我们的 SEU_Lex 项目编写了 MakeFile，从而利用 CMake 工具方便了编译运行，缩小了 SEU_Lex 工具在占的磁盘空间，方便了用户的调试和使用。

为了进一步说明我们的 SEU_Lex 工具的使用方法，我们首先说明一下使用改工具所需的前期文件。

下载 SEU_Lex.zip 后，解压后，改变当前工作目录到 SEU_Lex，执行 tree 命令可看到，本项目包含的文件如图 4-1-1 所示。

```
AnthonySong-mbp:SEU_Lex_v0 mac$ tree
.
├── CMakeLists.txt
├── DFAMinimization.h
├── GenerateCCode.h
├── Makefile
├── NFAToDFA.h
├── README.md
├── RE_Extraction.h
├── RE_Standardization.h
├── Re2NFA.h
├── SEU_Lex.cpp
├── helper.h
├── lex.l
└── testcase.cpp

0 directories, 13 files
AnthonySong-mbp:SEU_Lex_v0 mac$
```

图 4-1-1：SEU_Lex 的所有相关文件

各个文件的主要功能如表 4-1-1 所示：

表 4-1-1：SEU_Lex 的所有相关文件

文件名	功能
CMakeLists.txt	用于编译
Makefile	用于编译
lex.l	SEU_Lex 的 lex 定义文件，可修改
Re_Extraction.h	lex.l 文件解析

RE_Standarization.h	正则表达式规范化
RE2NFA	构造 NFA
NFAToDFA.h	构造 DFA
DFAMinimization.h	构造最小化后的 DFA
GenerateCode.h	生成词法分析程序代码
SEU_Lex.cpp	总控程序
helper.h	数据结构的定义
testcase.cpp	测试源代码文件

我们的用户需给定以下两个文件。

- 给出符合语法规则的.l 文件

根据.l 的语法规则，定义一个.l 文件作为 Lex 自动生成器的输入。（我们用来模拟测试的.l 文件路径为～\lex.l，和源代码放在同一目录下。）

- 待进行此法分析的源代码文件

这里的源代码文件需符合 ANSI 标准 C 语法规范。

下面详细阐述 SEU_Lex 的使用步骤。

1. 打开终端，切换当前工作目录。如图 4-2-2。

图 4-2-2: SEU_Lex 的使用步骤 1

2. 输入 cmake. 再输入 make all，编译项目。

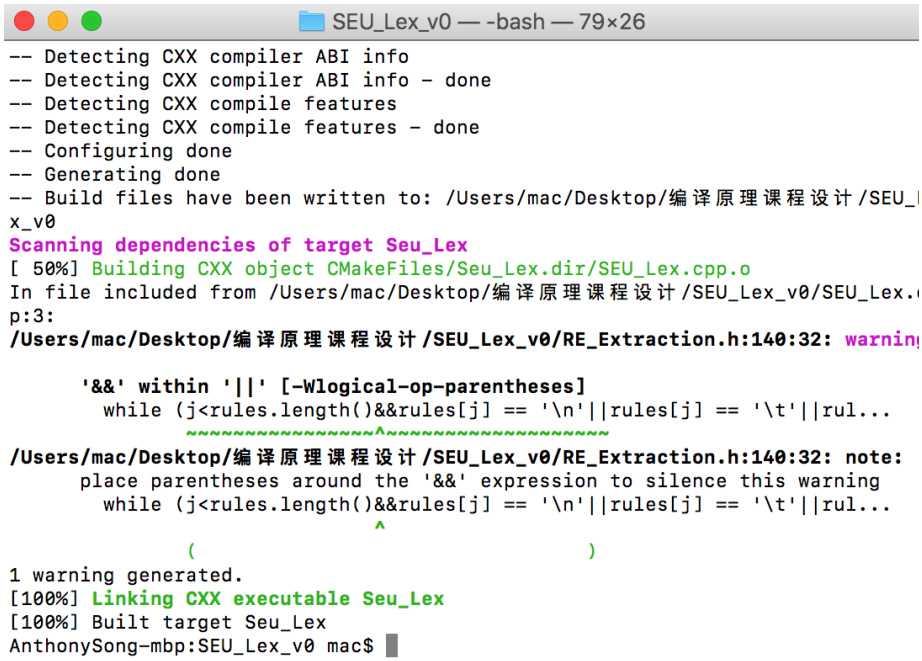


图 4-2-3: SEU_Lex 的使用步骤 2

3. 输入./Seu_Lex lex.l，根据 lex.l 文件，产生词法分析器 SEU_Lex 的 C++代码文件 Lex.yy.cpp。

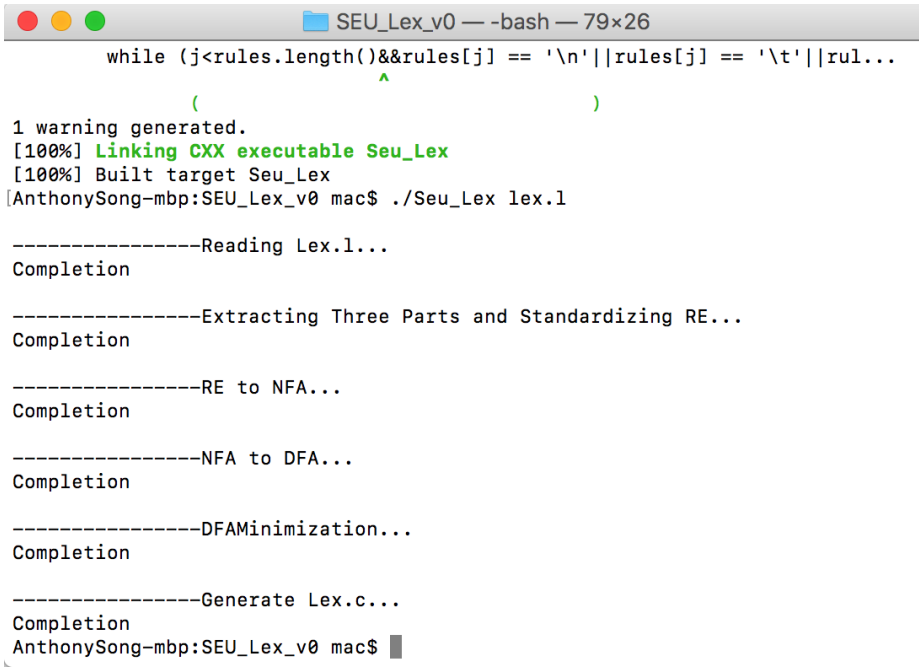


图 4-2-4: SEU_Lex 的使用步骤 3

4. 输入 `g++ -std=c++11 Lex.yy.cpp -o Lexer`，编译生成词法分析器 **SEU_Lex** 的可执行文件。

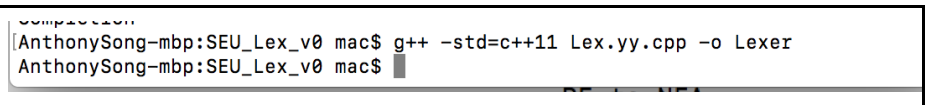


图 4-2-5: SEU_Lex 的使用步骤 4

5. 输入 `./Lexer testcase.cpp`，对测试代码文件进行词法分析，测试 SEU_Lex 的表现。

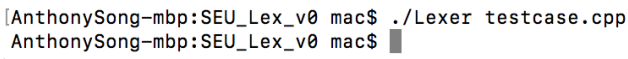


图 4-2-6: SEU_Lex 的使用步骤 5

5 测试用例与结果分析

我们编写了测试文件 testcase.cpp，作为待进行此法分析待程序，使用 lex.l 作为 SEU_Lex 的输入生成 Lexer 可执行文件，并且把 testcase.cpp 作为 Lexer 的输入，观察返回的 Token 序列（Lex_Tokens.txt 文件）是否正确。

- 测试用例（testcase.cpp）

源程序内容如下:

```
int t=0,j=10,tot,ans;
int a[10];

typedef struct node{
    unsigned int q;
    struct node* next;
}Node;

//comment
float add(int x){
    static int p1=1;
    static double p2=2;
    static float p3=3;
    p1+=x;
    p2*=x;
    p3-=x;
    do{
        Node* ptrp1;
        if(ptr->next != 0 && 3>=2.0){
            (*ptr).q %= 2;
        }
        else{
            p1 = p2 < 9 ? p2 : p1 * p2;
            continue;
        }
    }while(i != ++j);
    return 22.2;
}

/*comment2 */
int foo(int* a,const int *b){
```

```
    if(a == b || a!= -0.2 && b << 1){
        return sizeof(int);
    }
    else{
        foo(&a-&b,&b);
        return ;
    }
}

enum week{ Mon = 1, Tues = 2, Wed = 3, Thurs = 4, Fri = 5, Sat = 6, Sun = 7 };

int main(int argc, char const *argv[]){
    tot=10;

    char a[10]={"a","b","c","a","b","c","a","b","c","a","b","c"};

    unsigned a[12];

    int (*(*func[7][8][9])(int*))[5];

    for(int j = 0 ; j< 12; ++j){
        a[j] += j << 1;
    }

    for(int i=1;i<=10;i+=1){
        t=t+i;
        while((j>0&&j<i) || i<8)
            tot=tot*t-j/i;
        add(i);
        tot/=2;
    }
    if(tot==tot&&!(tot!=tot))
        add(tot,para1,& para2);
    switch(j){
        case 3:{
            foo(&a,&b);
            break;
        }
```

```
                case 4:{
                        if(tot==tot&&!(tot!=tot))
                        add(tot,para1,& para2);
                        break;
                }
                case 5:break;
                default:break;
        }

        int a[3][4] = {  {0, 1, 2, 3} , {4, 5, 6, 7} ,  {8, 9, 10, 11}  };

        printf("%d, %d, %d, %d, %d", sizeof(enum week), sizeof(day), sizeof(Mon),
    sizeof(Wed), sizeof(int) );
        for (j=0;j<n2;j++)
          {
                (*p)[j].stu=(char **)malloc (sizeof(char)*3);
                if ((*p)[j].stu == 0)
                        {
                        return false;
                        }
                for (k=0;k<3;k++)
                {
                        (*p)[j].stu[k]=(char *)malloc (sizeof(char)*20);
                        if ((*p)[j].stu[k] == 0)
                {
                return false;
                }
                }
          }
        return 0;
      }
```

返回的 Token 序列为：

```
int     INT   1
t  IDENTIFIER  3
=  =     4
0  CONSTANT    5
,  ,     6
j  IDENTIFIER  7
=  =     4
```

```
10 CONSTANT    8
, ,     6
tot     IDENTIFIER  9
, ,     6
ans     IDENTIFIER  10
; ;     11
int     INT   1
a  IDENTIFIER  14
[  [     15
10 CONSTANT    8
]  ]     16
;  ;     11
typedef  TYPEDEF     17
struct   STRUCT      18
node     IDENTIFIER  19
{  {     20
unsigned UNSIGNED    22
int      INT   1
q  IDENTIFIER  23
;  ;     11
struct   STRUCT      18
node     IDENTIFIER  19
*  *     24
next     IDENTIFIER  25
;  ;     11
}  }     26
Node     IDENTIFIER  27
;  ;     11
//test   SINGLECOMMENT     28
float    FLOAT 29
add      IDENTIFIER  30
(  (     31
int      INT   1
x  IDENTIFIER  32
)  )     33
{  {     20
static   STATIC      34
int      INT   1
p1 IDENTIFIER  35
=  =     4
1  CONSTANT    36
;  ;     11
static   STATIC      34
double   DOUBLE      37
p2 IDENTIFIER  38
=  =     4
2  CONSTANT    39
;  ;     11
static   STATIC      34
float    FLOAT 29
p3 IDENTIFIER  40
=  =     4
3  CONSTANT    41
;  ;     11
```

```
p1 IDENTIFIER  35
+= ADD_ASSIGN  42
x  IDENTIFIER  32
;  ;     11
p2 IDENTIFIER  38
*= MUL_ASSIGN  43
x  IDENTIFIER  32
;  ;     11
p3 IDENTIFIER  40
-= SUB_ASSIGN  44
x  IDENTIFIER  32
;  ;     11
do DO    45
{  {     20
Node     IDENTIFIER  27
*  *     24
ptrp1    IDENTIFIER  46
;  ;     11
if IF    47
(  (     31
ptr      IDENTIFIER  48
-> PTR_OP      49
next     IDENTIFIER  25
!= NE_OP 50
0  CONSTANT    5
&& AND_OP      51
3  CONSTANT    41
>= GE_OP 52
2.0      CONSTANT    53
)  )     33
{  {     20
(  (     31
*  *     24
ptr      IDENTIFIER  48
)  )     33
.  .     54
q  IDENTIFIER  23
%= MOD_ASSIGN  55
2  CONSTANT    39
;  ;     11
}  }     26
else     ELSE  56
{  {     20
p1 IDENTIFIER  35
=  =     4
p2 IDENTIFIER  38
<  <     57
9  CONSTANT    58
?  ?     59
p2 IDENTIFIER  38
:  :     60
p1 IDENTIFIER  35
*  *     24
p2 IDENTIFIER  38
```

```
;  ;     11
continue CONTINUE    61
;  ;     11
}  }     26
}  }     26
return   RETURN      62
22.2     CONSTANT    63
;  ;     11
}  }     26
struct   STRUCT      18
miemie   IDENTIFIER  64
{  {     20
char     CHAR  65
m  IDENTIFIER  66
=  =     4
"b"      STRING_LITERAL    67
;  ;     11
extern   EXTERN      68
double   DOUBLE      37
n  IDENTIFIER  69
=  =     4
-2019.33 CONSTANT    70
;  ;     11
}  }     26
;  ;     11
/*test2*/     BLOCKECOMMENT     71
int      INT   1
main     IDENTIFIER  72
(  (     31
)  )     33
{  {     20
char     CHAR  65
=  =     4
"YangMieMieZuiQiang!"     STRING_LITERAL    73
;  ;     11
tot      IDENTIFIER  9
=  =     4
10 CONSTANT    8
;  ;     11
for      FOR   74
(  (     31
int      INT   1
i  IDENTIFIER  75
=  =     4
1  CONSTANT    36
;  ;     11
i  IDENTIFIER  75
<= LE_OP 76
10 CONSTANT    8
;  ;     11
i  IDENTIFIER  75
+= ADD_ASSIGN  42
1  CONSTANT    36
)  )     33
```

```
{ {     20
t  IDENTIFIER  3
=  =     4
t  IDENTIFIER  3
+  +     77
i  IDENTIFIER  75
;  ;     11
while    WHILE 78
(  (     31
(  (     31
j  IDENTIFIER  7
>  >     79
0  CONSTANT    5
&& AND_OP      51
j  IDENTIFIER  7
<  <     57
i  IDENTIFIER  75
)  )     33
|| OR_OP 80
i  IDENTIFIER  75
<  <     57
8  CONSTANT    81
)  )     33
tot      IDENTIFIER  9
=  =     4
tot      IDENTIFIER  9
*  *     24
t  IDENTIFIER  3
-  -     82
j  IDENTIFIER  7
/  /     83
i  IDENTIFIER  75
;  ;     11
add      IDENTIFIER  30
(  (     31
i  IDENTIFIER  75
)  )     33
;  ;     11
tot      IDENTIFIER  9
/= DIV_ASSIGN  84
2  CONSTANT    39
;  ;     11
}  }     26
if IF    47
(  (     31
tot      IDENTIFIER  9
== EQ_OP 85
tot      IDENTIFIER  9
&& AND_OP      51
!  !     86
(  (     31
tot      IDENTIFIER  9
!= NE_OP 50
tot      IDENTIFIER  9
```

```
)  )      33
)  )      33
add       IDENTIFIER  30
(  (      31
tot       IDENTIFIER  9
)  )      33
;  ;      11
switch    SWITCH      87
(  (      31
j  IDENTIFIER  7
)  )      33
{  {      20
case      CASE  88
3  CONSTANT    41
:  :      60
break     BREAK 89
;  ;      11
case      CASE  88
4  CONSTANT    90
:  :      60
break     BREAK 89
;  ;      11
case      CASE  88
5  CONSTANT    91
:  :      60
break     BREAK 89
;  ;      11
default   DEFAULT     92
:  :      60
break     BREAK 89
;  ;      11
}  }      26
return    RETURN      62
0  CONSTANT    5
;  ;      11
}  }      26
```

可以看出 Token 序列满足要求，且成功地处理了多行注释的情况。

6 课程设计总结（包括设计的总结和需要改进的内容）

在整个 SEU_Lex 的开发过程中，我们三人分工合作。按照 SEU_Lex 的功能模块的执行顺序，自然地将流程分为三个部分：lex.l 到正则表达式、正则表达式到 NFA 和 NFA 到最小化到 DFA。整个流程思路清晰明了，各个部分到算法简洁自然，这样我们后续到整体整合调试提供了基础。

在各个模块到开发伊始，我们商讨了各个部分所需要到数据结构，在参考了学长所定义的数据结构之后，我们充分考虑到程序运行的效率和所消耗的内存空间大小，在做到一定的权衡利弊之后，我们最终确定了我们所认为的最优的数据结构定义，这不但为我们程序的在可接受时间内执行完成提供了保障，也为我们后期的整体调试整合提供了方便。最终，我们将整个 SEU_Lex 生成 Lexer 的运行时间控制在几秒钟之内。

在各个模块的分工开发过程中，我们三人努力确保各自模块之间的低耦合性，重点放在各自模块的接口输入输出的确立，从而为我们最终的整合调试提供了便捷。我们三人虽然各模块独自开发，但是仍会保持充分的交流和协作，从而推动了项目的有条不紊的进展，从中我们不但巩固了词法分析的流程和各步骤使用的常见算法，进一步加深了对 C++的 STL 中定义的常见数据结构使用方法，而且，更重要的是，我们深刻意识到团队合作中交流的重要性，这为我们今后在项目开发团队中开展工作打下了基础。

最后，对于未来的改进方向，我们将着重于实现 lex 的高阶功能，如 yymore()，yywap()等函数功能的实现，以及对更多非规范化正则表达式的解析，如正则表达式的分组机制等。

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。