正则表达式引擎(HASKELL-RE)

上学期学了一些编译器前端的知识，接触到了 `Flex`， `Yacc` 这类的工具，同时也对正则表达式引擎有了很大的兴趣，于是总想着自己实现一个简单的正则表达式引擎，在这里是对这部分知识的一个总结，因为网上有很多资料总感觉缺头少尾，大多数人第一次读龙书[1]也是有些Hold不住，所以，这里拿一个 `入门级` 的正则引擎作为例子，总结一下这部分知识

Introduction

首先介绍一下我的正则引擎的实现情况

```
()       -- 括号的用途我和Russ Cox的功能是一样的，为了优先级考虑
|        -- 可选
*        -- 重复0次以上
+        -- 重复1次以上
?        -- 重复0次或1次
[az]     -- 字符集合
[a-z]    -- 字符范围
[^az]    -- 负值字符集合
[^a-z]   -- 负值字符范围
\ch      -- 功能字符，实现但未添加，可自行扩展
{m, n}   -- 重复次数，未实现，可自行扩展
```

基本功能就是以上所罗列的了，有些未实现的，譬如说 `{m, n}` 可以参考本文自行添加，因为本学期需要考研/实习，重构代码就留到后面再说吧，以上功能的代码大约1000-1500行左右，不算很大的项目，可以作为练手项目进行学习，项目地址Haskell-Re `old-version` 分支

Implementation

首先分析一下工作流程，在大多数课堂上听到老师所讲的方法大约是这样的

```
将正则表达式转化成epsilon-NFA
epsilon-NFA转化为DFA
最小化DFA
DFA模拟(匹配)
```

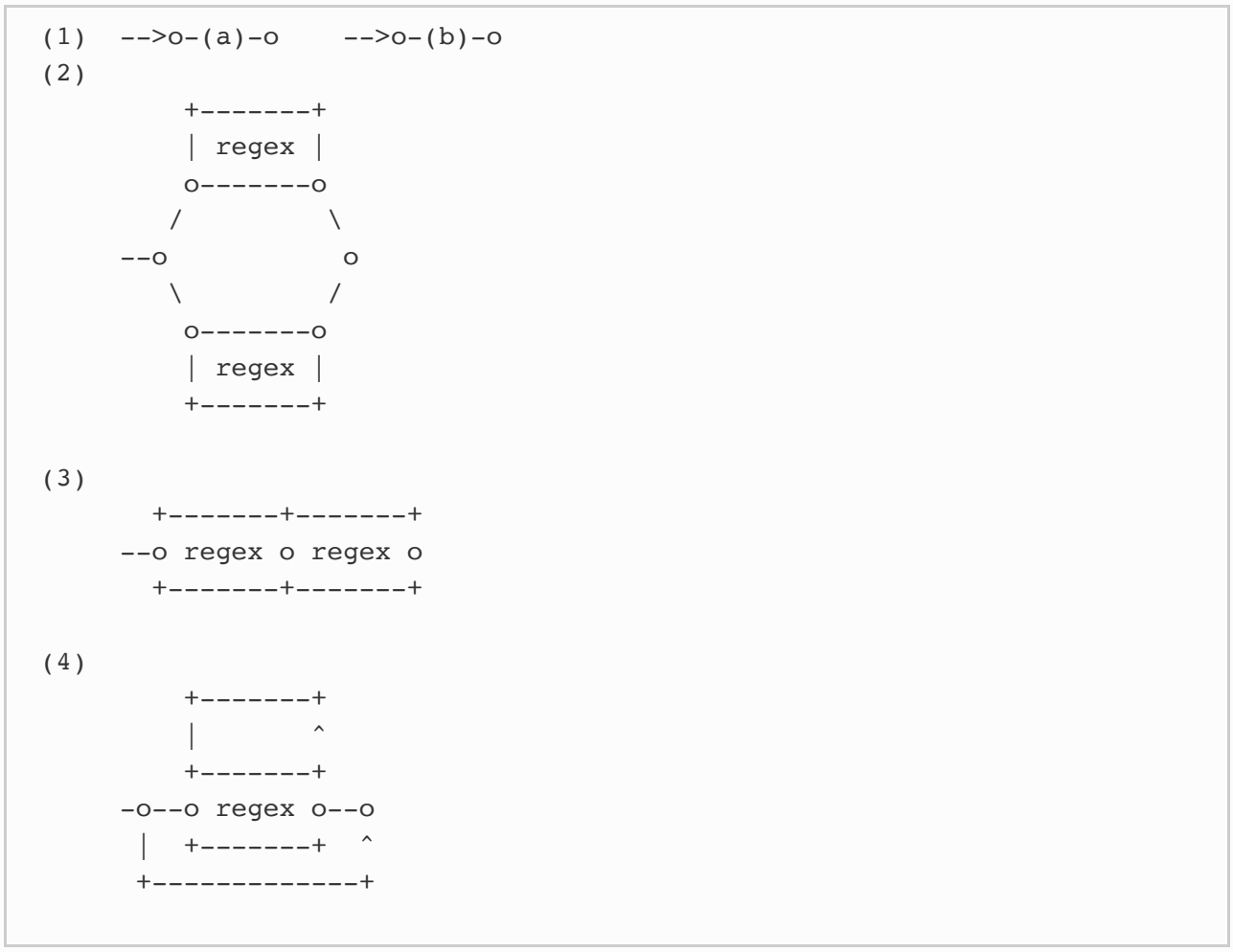
但实际上，我是从底层开始实现的，有点类似从底层一层一层抽象出来的机器，因为如果从解析正则语言直接构造NFA开始，我甚至根本不知道怎么去设计 `NFA` 的数据结构。所以，在我看来正确的做法是从底层的NFA开始，然后构造正则语法，然后将正则表达式解释成我们的正则语法，最后交给 `NFA`， `DFA` 处理就好了

Regular Syntex Data Structure

我的正则语法参考了Regular Expression and Automata using Haskell(3)，但为了扩展其他功能，这里我加了一个 `group`，其他功能也可以依此添加数据结构

```haskell
data Regex = Epsilon
           | Literal Char    -- 这里得益于Haskell的设计是基于UTF-8的
           | Alt Regex Regex -- alternative
           | Con Regex Regex -- concatenation
           | Star Regex      -- (re)*
           | Plus Regex      -- (re)+
           | ZOO Regex       -- (re)? zero or one
```

至于如何生存NFA就十分简单了，就是那十分经典的做法，字符、表达式连接、表达式分支、表达式重复，按照集合的规则添加状态和边就可以了



Top-Down Parsing

我在初学的时候，总是想知道正则表达式是怎么解析成 `epsilon-NFA` 的，参考了Russ Cox的 `C语言描述` 又感到有些不太适合我，因为他将正则表达式直接根据优先级转成了后缀表达式，之后遍历一次表达式就拿两个栈完成了 `epsilon-NFA` 的构建。有些一头雾水。不过Russ Cox的文章确实很有参考意义。首先，构造 `epsilon-NFA` 在我看来，有两种都算简单操作，但其实殊途同归

- Russ Cox的构造后缀表达式
- 构建AST

我的实现方法选择了后者，主要是我个人对后缀表达式没什么好感，虽然两种方法思想都是一致的，在构建AST上，我参考了Stack Overflow的回答，一个PCRE的简单的BNF语法，利用语法树，我们之后还可以根据需求添加更多的功能，这一点，我认为仅仅用后缀表达式是没办法比的

```
<RE>             ::= <union> | <simple-RE>
<union>          ::= <RE> "|" <simple-RE>
<simple-RE>      ::= <concatenation> | <basic-RE>
<concatenation>  ::= <simple-RE> <basic-RE>
<basic-RE>       ::= <star> | <plus> | <elementary-RE>
<star>           ::= <elementary-RE> "*"
<plus>           ::= <elementary-RE> "+"
<elementary-RE>  ::= <group> | <any> | <eos> | <char> | <set>
<group>          ::= "(" <RE> ")"
<any>            ::= "."
<eos>            ::= "$"
<char>           ::= any non metacharacter | "\" metacharacter
<set>            ::= <positive-set> | <negative-set>
<positive-set>   ::= "[" <set-items> "]"
<negative-set>   ::= "[^" <set-items> "]"
<set-items>      ::= <set-item> | <set-item> <set-items>
<set-items>      ::= <range> | <char>
<range>          ::= <char> "-" <char>
```

这下，我们的工作简单多了,我们只要按照我们之前定义的NFA数据结构把这颗树中的结点解析到 `NFA` 中就行了，这里需要注意一个问题，就是老生常谈的左递归(left-recursion)问题，大约在 `RE` 和 `union` 这里，按照常规操作消除左递归就没什么问题了,至于生成了NFA后，参考我上一篇文章就可以了，顺提一嘴， `Monadic Parser` 确实很有用，写几个Parser后，根据这个思想也可以应用到很多地方

Conclusion

这个项目本身只有1000多行代码，但是确实包含了很多算法问题，我实现的自动机也是十分低效的，因为我的状态机的边没有表示成范围，所以在构造类似 `[a-z]` 或者 `[^a-z]` 这类的东西时，生成的NFA是根据 `(a|b|c|d|...|z)` 来生成的，在我上一篇文章中已经指出了问题所在，这个学期可能没有时间重构代码了，不过，基于这个思想，很多超出 `Type-3` 语法的扩展正则表达式也可以实现了

Reference

(1)(Compilers: Principles, Techniques, and Tools (2nd Edition))(https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811)

(2)(Regular Expression Matching Can Be Simple And Fast)(https://swtch.com/~rsc/regexp/regexp1.html)

(3)(Regular Expressions and Automata using Haskell)(https://www.cs.kent.ac.uk/people/staff/sjt/craft2e/regExp.pdf)