

Lexical Analyzer Programming实验报告

71114223 陈雄辉



2017-1-14

东南大学

计算机科学与技术和软件学院

# Motivation/Aim

基于DFA设计一个具体的词法分析程序，加深对词法分析原理的理解。并掌握在对程序设计语言源程序进行扫描过程中将其分解为各类单词的词法分析方法。

本项目源码已经上传到gihub，地址：

<https://github.com/Bunrning-Bear/compiler_experiment_python>

# Content description

本试验使用基于DFA的方法构造词法分析器，首先自行定义语言，将语言转化为正规表达式，之后转成最优化的DFA，将DFA用数据结构表示出来，并编写对应的驱动程序，进行语法分析

# Ideas/Methods

## 步骤一：使用基于Ubuntu的flex进行词法分析器的构造，对代码构架有直观感受

对应的执行demo和结果存放于\ flex-demo 文件夹中

## 步骤二：设计词法分析器

## 实现方案一（弃用）：

1. **实现思路：**
   1. 人为定义语言和每个单词的正规表达式
   2. 根据正规表达式生成多个NFA，并进行利用去除和子集构造算法构造DFA
   3. 根据弱等价类原则，构造最优化的DFA
   4. 对每个正则表达式生成一个DFA二维字典，行为状态，列为字典元素，dic['word']的值为通过这个边的下一个状态是什么，若达到总结点，则翻译成功
   5. 如果某个字典翻译失败，尝试下一个字典DFA，一直到尝试完成所有的DFA，如果都不行，则这个字符串无法成功翻译，抛出错误；
   6. 过程中需要注意：
      1. 解决同时满足不同优先级的方案，对每个表达式定义优先级，构造DFA的时候按照优先级解决冲突点
      2. 贪婪匹配原则，一直匹配到不能匹配为止
2. **弃用原因：**

3.步骤需要重复重新执行每一个DFA，造成大量的时间浪费，应该想办法更好的归并不同的DFA，以达到更高效的语法翻译

## 实现方案二：

本方案解决了方案一的第三个步骤出现的问题，将所有的正规表达式结合成一个DFA，具体实现方案如下：

* + - 1. 为每个正则表达式构造NFA，并标记每个正则表达式的终结符，需要标记两个信息：

- 正则表达式标号

- 优先级

2. 将多个NFA直接用“或”的方式合并；

3. 通过求闭包和子状态扩展两个方案，进行NFA到DFA的转换，过程需要做一个调整的是：

4. 每一个正则表达式给定一个优先级，状态拓展的时候，如果发现有多个终结符号，取优先级最高的那个

5. 构造化简的DFA：

- 构造弱等价类，但是不能把不同终结点归为同一个弱等价类

6. 根据最后DFA生成查询字典，

- 这个字典是一个二维字典，第一个维度的key表达的是当前的状态位

- 字典的第二个维度的key表达的是某一个状态位下的边，而value表达的是经过这个边[key]，到达的新的状态位

7. 贪婪原则：需要匹配到无法继续匹配[某个状态位没有接下来的key]为止

8. 遍历的过程需要记录成功匹配的正则表达式队列

- 如果匹配多个

- 优先级不同，按照优先级最高的

- 优先级相同，按照最先匹配到的

* 1. **步骤三：设计项目架构（本项目假期想实现一个简单的LEX分析器，所以为此做了一定的初步架构设计）**

- FA-based\main.py

- 调用match函数

- FA-based\global\_value.py

- DFA\_hash\_dictory 字典，定义了表达DFA的字典

- reg\_map：表达式和优先级，附带信息的对应关系

- FA-based\match.py

- 获取下一个读入字符

- 为每次匹配构造成功匹配栈

- 循环调用字典，直到某个字符没有对应的后继状态

- 循环过程中记录匹配成功的字符串和对应的配对表达式编号

- 通确定当前应该选用的正则表达式，

- 输出对应信息

* 1. **步骤四：测试**

# Assumptions

## 实验环境：

Python2.7

Ubuntu14.04 或者windows10均可

要求安装prettytable库，用于格式化输出表格

## 定义合法的语言的正规表达式：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 大类 | 细分类别 | 正规表达式定义 | 对应标识符 | 优先级 |
| 关键字 | If | If | If | 1 |
| then | then | Then | 1 |
| while | do | While | 1 |
| 操作符 | 运算操作符[] | \+|-|\\*|\/|= | operator | 1 |
| 关系操作符[] | >=|<=|==|!= | relation\_op | 2 |
| 间隔符 | 空行（中间定义的表达式）[] | [ \n\t] | delim | 1 |
|  | {delim}+ | whitespace | 1 |
| 分隔符 | 分隔 | [,;] | delimiter | 1 |
| 块分隔符 | [{}] | block | 1 |
| 括号 | [\(\)] | brackets | 1 |
| 数字 | 数位(仅中间定义) | [0-9] | digit | 2 |
| 常量 | ({digit})+ | constant | 2 |
|  |  |  |  | 2 |
| 标识符 | 字母(仅中间定义) | [A-Za-z] | letter | 3 |
| 变量 | {letter}(letter|{digit})\* | identfier | 3 |
| 类型变量 | 系统类型变量 | [int|float] | Type | 1 |

# Related FA descriptions

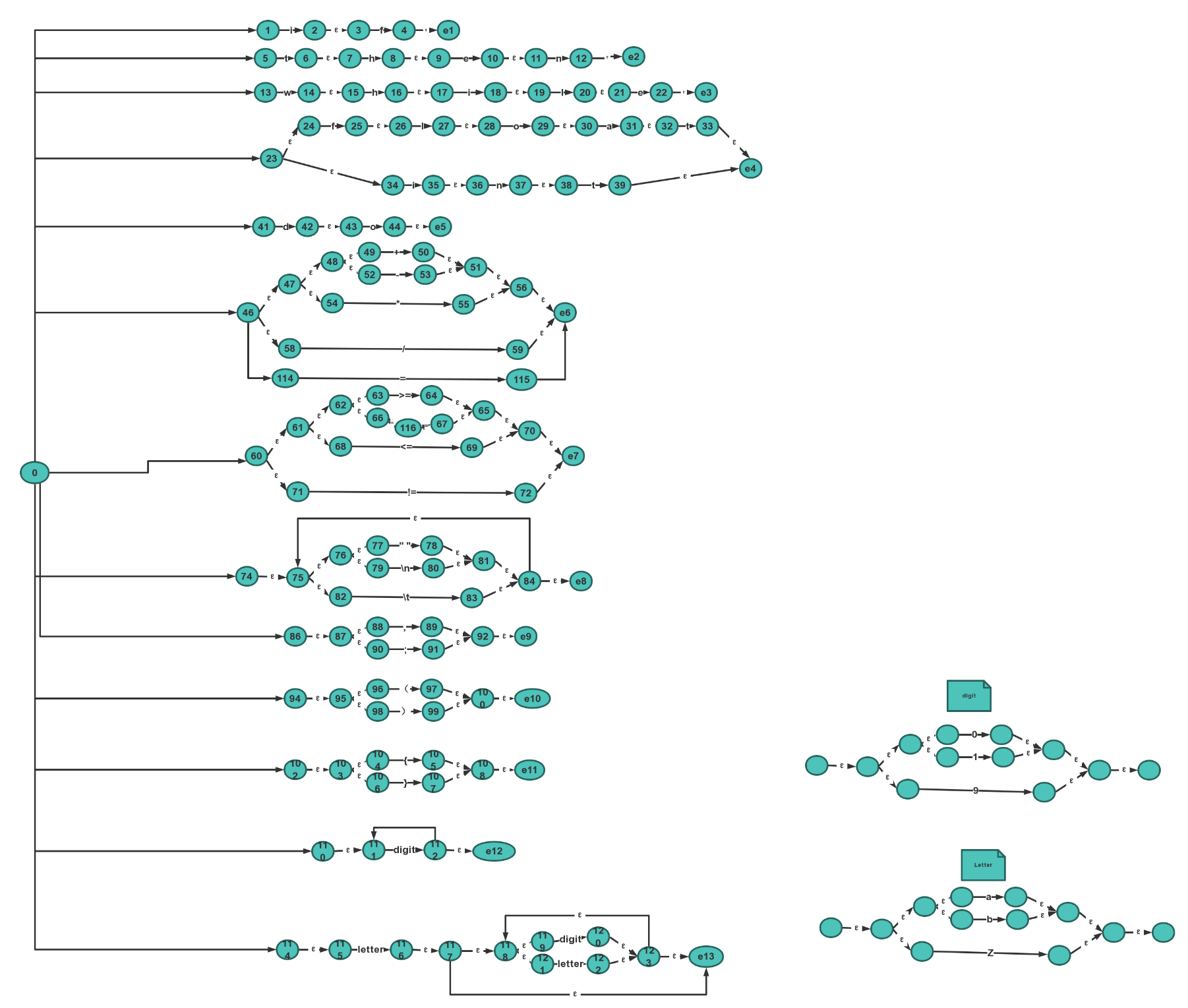
### 将多个NFA通过“或”合成一个NFA

合并原则同单个NFA的构造，唯一的区别是不能合并重点，不同节点的重点应该做不同的标记，图中使用e+数字做标记

由于图比较大，可以使用下列链接查看：

<https://www.processon.com/view/link/586a22cae4b067ce8520c144>

对应高清图片在附件“lex-NFA.png”

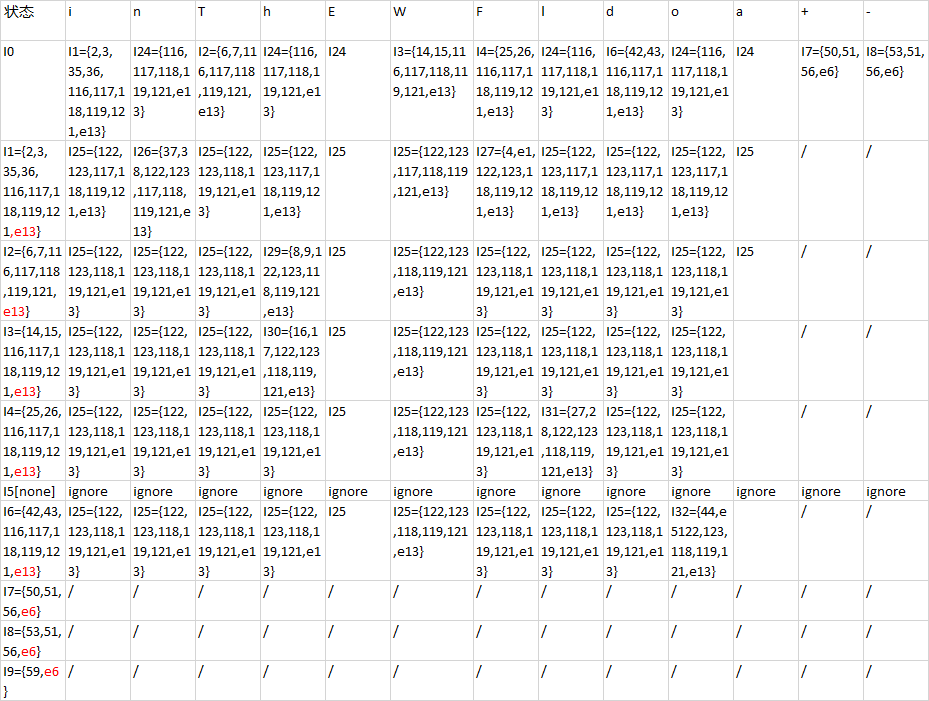


### 表格驱动，利用去除和子集构造算法构造转化表

由于生成的表格比较大，已经导成图片附件“transition table.xlsx”

表格中，合并过程会出现终结边的冲突的情况，此时按照输入时给定的每个标识符的优先级进行剔除，在表格中，使用红色表示选中的终结符，绿色表示显示的非终结符，

下面展示表格的部分截图：



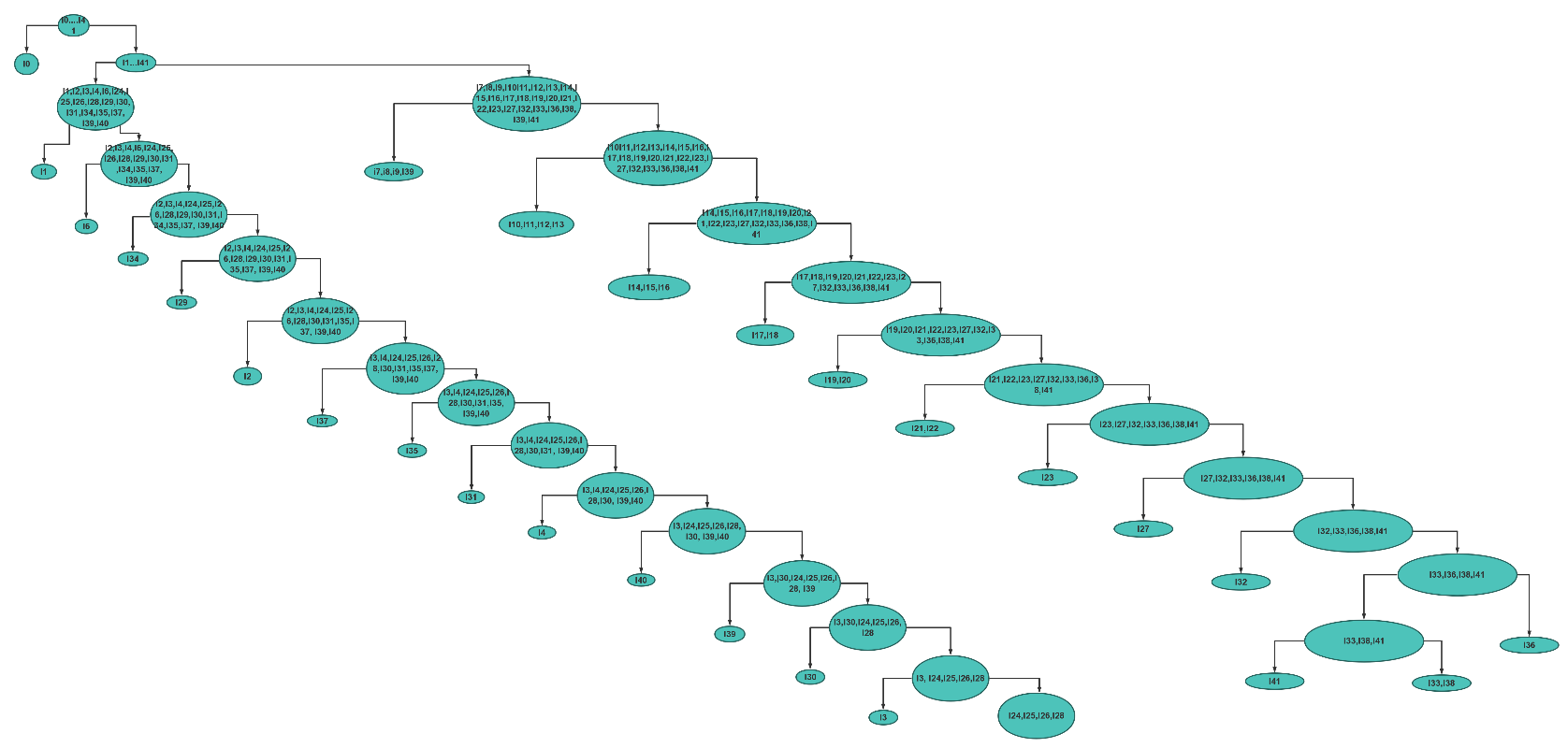
### 根据表格优化DFA

根据transition table，使用弱等价类划分的方法进行状态的合并，与单个DFA的合并方式大体相同，唯一不同的是，要把所有不同的终结点当成不同的弱等价类。

由于图比较大，可以使用下面的链接查看：

<https://www.processon.com/view/link/586a5be9e4b0f7a9c3505044>

高清图片在附件“lex-DFAO.png”



### 根据等价类划分的结果画出DFA图

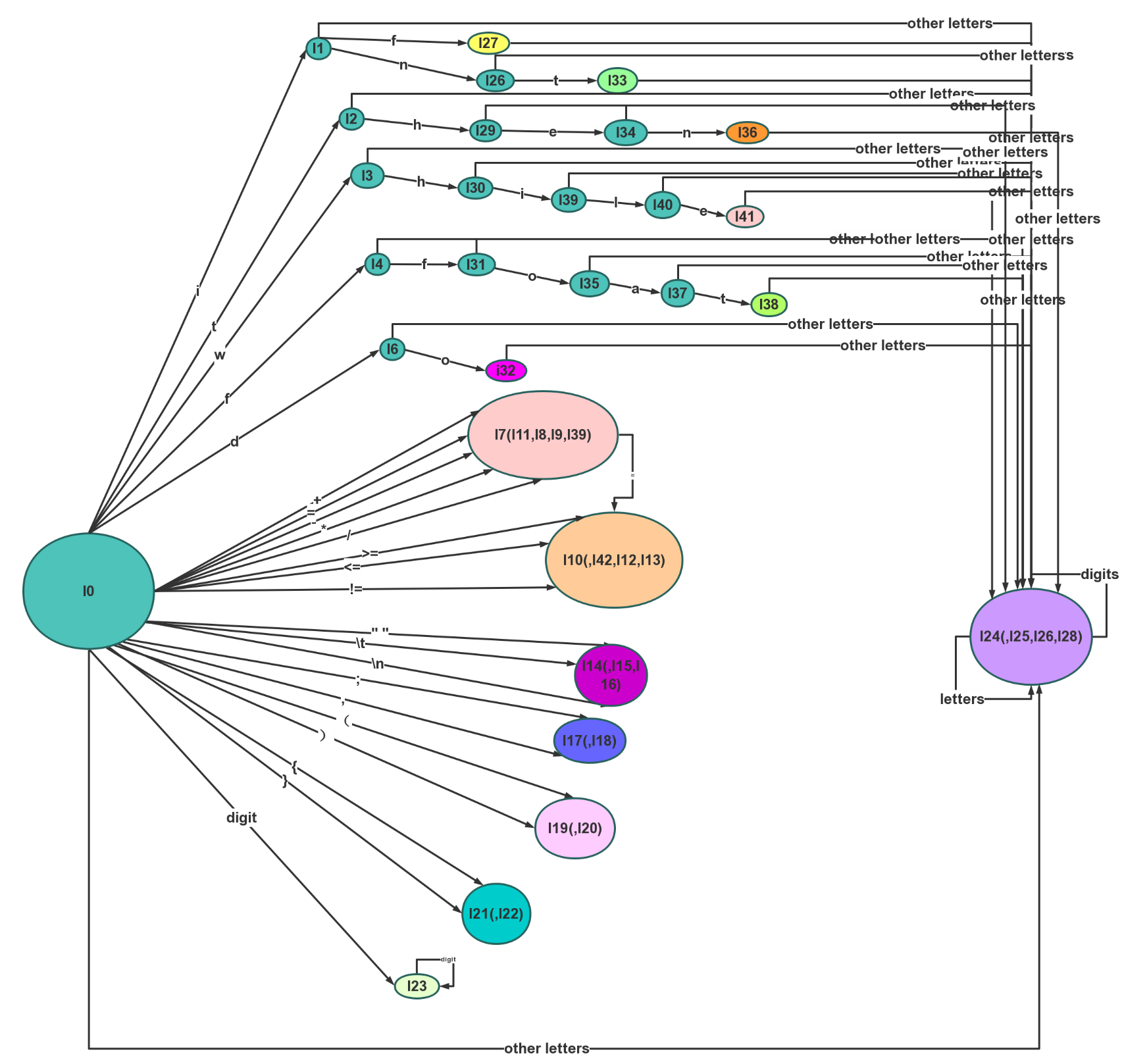
与单个DFA的构造一致，唯一不同的是有多个终结点，要区别标记，

由于图比较大，可以使用下列链接查看：

<https://www.processon.com/view/link/586a7750e4b0dc8df79d2443> ，

对应高清图片在附件“lex-DFA.png”

下面为缩略图：



# Description of important Data Structures

项目的数据结构被存储在\FA-based\Global.py中，

* DFA\_hash\_dic用字典的形式保存DFA
* node\_reg\_mapping 用来完成状态和状态终结点所需要完成的动作（本例皆为输出对应信息）的映射

# Description of core Algorithms

项目的核心代码存储于\FA-based\Match.py中

#!usr/bin/env python

#coding=utf-8

# match.py

from Global import DFA\_hash\_dic, node\_reg\_mapping, key\_mapping\_regex

import logging

import re

class MatchHandler(object):

def \_\_init\_\_(self,fp):

self.\_current\_status='0'# 匹配器当前的状态位

self.\_succ\_list = []# 匹配成功的结果队列

self.\_fp = fp # fp come from open("file path",'r')

self.\_end\_file = False# 文件是否到达结尾的状态字

def \_mapping\_key\_unit(self, single\_char):

"""被Match\_one\_char函数调用的内部函数，用来检查该字符能否在DFA中得到匹配

Args:

single\_char: 输入的字符

Returns:

如果匹配，就输出对应的匹配的key，否则输出None

"""

key = None

if DFA\_hash\_dic[self.\_current\_status].has\_key(single\_char):

key = single\_char

else:

for (temp\_key, value) in key\_mapping\_regex.items():

# find if this single char satisfied with some kind of regex expression.

if re.match(value,single\_char):

key = temp\_key

break

# find a key, then judge if this is a key in dictory

if key and not DFA\_hash\_dic[self.\_current\_status].has\_key(key):

key = None

return key

def \_match\_one\_char(self, single\_char):

"""匹配单个字符，返回下一个状态

Args:

single\_char: 输入的字符

Returns:

如果匹配，就输出下一个状态，否则输出None

"""

key = self.\_mapping\_key\_unit(single\_char)

if not key:

return None

else:

next\_status = DFA\_hash\_dic[self.\_current\_status][key]

return next\_status

# get a end node, record in succ\_list.

def \_append\_succ\_list(self, status, str):

"""将匹配成功的状态和对应的信息存储入succ\_list中，用于匹配结果的输出

Args:

status: 'exx'

str:'float'

Returns:

None

"""

prefix = status[0]

if prefix == 'e':

# next status is a end node, record in succ\_list.

info = {'status':status,'text':str}

self.\_succ\_list.append(info)

def \_update\_status(self, next\_status):

"""匹配结束之后，需要更新状态字到下一个状态

Args:

next\_status:更新的状态值

Returns:

None

"""

self.\_current\_status = next\_status

def \_get\_next\_char(self):

"""Get next char of fp.

when EOF is encountered, return None.

Args:

None

Returns:

next char or None

"""

# fp come from open("file path",'r')

result = self.\_fp.read(1)

if result:

return result

else:

self.\_end\_file = True

return None

def reset\_file\_pointer(self):

"""单个词素匹配结束，在输入流中回退一位重新匹配

"""

self.\_fp.seek(self.\_fp.tell() - 1)

def \_list\_mapping(self):

"""匹配结束之后，从匹配结果中寻找优先级最低的作为匹配结果显示

"""

if self.\_succ\_list == []:

return None

else:

# find the min level regex as the match string.

min\_level = 99

min\_index = -1

for index, item in enumerate(self.\_succ\_list):

status = item['status']

level = node\_reg\_mapping[status]

if level < min\_level:

min\_level = level

min\_index = index

min\_status = self.\_succ\_list[min\_index]['status']

result = {

'status':min\_status,

'text':self.\_succ\_list[min\_index]['text'],

'info':node\_reg\_mapping[min\_status]['info']

}

return result

def \_reset(self):

"""一个正规表达式匹配结束之后,重置初始状态和匹配成功的list

"""

self.\_current\_status='0'

self.\_succ\_list = []

def \_match\_one\_reg(self):

"""

- 获取下一个读入字符

- 为每次匹配构造成功匹配栈

- 循环调用字典，直到某个字符没有对应的后继状态

- 循环过程中记录匹配成功的字符串和对应的配对表达式编号

- 通确定当前应该选用的正则表达式，

- 输出对应信息

"""

next\_char = self.\_get\_next\_char()

current\_text = next\_char

while(1):

if self.\_end\_file:

# get the end of file

break

next\_status = self.\_match\_one\_char(next\_char)

if next\_status == None:

# failed or finfish, depend on succ\_list is empty or not.

# reset read pointer

self.reset\_file\_pointer()

break

else:

# find a match token.

self.\_append\_succ\_list(next\_status, current\_text)

self.\_update\_status(next\_status)

next\_char = self.\_get\_next\_char()

if next\_char == None:

break

current\_text = current\_text+next\_char

# match finish, now mapping.

result = self.\_list\_mapping()

self.\_reset()

if not result:

# return the fail match string

return False,current\_text

else:

return True,result

def match\_driver(self):

"""驱动词法分析器的驱动器，主函数只需要运行这个方法即可

"""

result\_queue = []

while(True):

if self.\_end\_file:

return result\_queue

success, result = self.\_match\_one\_reg()

logging.info("success:%s, result: %s"%(success,result))

if not success:

result\_queue.append("match error with: text\"%s\""%result)

return result\_queue

else:

result\_queue.append(result)

# Use cases on running

# Problems occurred and related solutions