西安财经学院 信息学院

编译原理 实验报告

姓名：陈伯硕
学号：1831050010
班级：计本 1801
指导教师：李薇

成绩：________

实验名称 正规式、自动机的相互转换

实验日期：2020 年 10 月 26 日

一、实验目的

1. 理解正规式、NFA、DFA、DFA 最小化的基本原理；
2. 掌握正规式向 NFA 转换的算法；
3. 掌握 NFA 向 DFA 转换的算法；
4. 掌握 DFA 最小化的算法。

二、实验内容

1. 设计并实现正规式向 NFA 转换的算法；
2. 设计并实现 NFA 向 DFA 转换的算法；
3. 设计并实现 DFA 最小化的算法。

三、实验要求

1. 从下列实验内容中任选一个或几个算法实现；

   （1）设计并实现正规式向 NFA 转换的算法：输入为正规式，输出为一个 NFA。

   （2）设计并实现 NFA 向 DFA 转换的算法：输入为一个 NFA 五元组，输出一个与其等价的 DFA 五元组。

   （3）设计并实现 DFA 最小化的算法：输入为任意一个 DFA，输出此 DFA 等价的状态最少的 DFA。
2. 要求根据算法要求设计合理的数据结构；
3. 编程实现转换过程。

四、 算法设计

4.1 自动机的数据结构与算法

4.1.1 自动机的数据结构

自动机由元组 (Σ, S, S_0, F, f) 组成，其中

1. Σ 为字符的集合, 空串 $\epsilon \notin \Sigma$
2. S 为状态集合
3. $S_0 \in S$ 为初始态
4. $F \subset S$ 是终态的集合
5. $f : S \times (\Sigma \cup \{\epsilon\}) \to S$ 为状态转换函数, 在程序中可以使用哈希表 hash$(S_i, S_{j_k}) = C_{ij_k}$, $\forall c_{ij_k} \in C_{ij_k}$, 使得 $f(S_i, c_{ij_k}) = S_{j_k}$ 在 python 中，hash 函数由字典数 *dict* 据类型直接实现, 对于两个变量的哈希表，可以用字典嵌套字典实现

4.1.2 自动机的算法

设正则表达式 s, t，其对应 NFA 为 $N(s), N(t)$

对于表达式 $s|t$ 对于运算链接 (concatenation)$N(s)N(t)$, 对应的算法如算法 1

算法 1 concatenation 运算

concatenation($N(s)$,$N(t)$)

renameState($N(t)$) $\triangleright$ 修改 $N(t)$ 状态名，防止歧义
merge($N(s), N(t)$) $\triangleright$ 将 $N(t)$ 的状态和转换函数复制给 $N(s)$
$F_s \leftarrow F_t$ $\triangleright$ $N(s)$ 的终态变为 $N(t)$ 的终态 F_t
free($N(t)$) $\triangleright$ 释放 $N(t)$ 的空间
return $N(s)$

并运算 (union)$N(s)|N(t)$

算法 2 union 运算

union($N(s), N(t)$)

```
rename(N(s))
rename(N(t))
copy(N(s), N(t))
new S_0 ▷ 新初态
new S_t ▷ 新终态
addEdge(S_0,S_0^(s),ε) ▷ S_0^(s) 为原 N(s) 的初态
addEdge(S_0,S_0^(t),ε) ▷ S_0^(s) 为原 N(t) 的初态
for S_f ∈ F^(s) ∩ F^(t):
    addEdge(S_f, S_t, ε)
F ← {S_t} ▷ 新的终态集
free(N(t))
return N(s)
```

对于 $s*$ 的形式，表示 s 匹配一次或多次，对自动机来说，无条件地进入终态，无条件返回初态，如算法 3

算法 3 star 运算

star($N(t)$)

```
for S_f ∈ F:
    addEdge(S_f, S_0, ε)
    addEdge(S_0, S_f, ε)
return N(t)
```

4.2 正则表达式解析

正则表达式的结构可以由以下解析式组成

```
<regex>  ::= <term> '|' <regex>
           |  <term>

<term>   ::= { <factor> }

<factor> ::= <base> { '*' }
```

```
<base>   ::= <char>
          |  '\' <char>
          |  '(' <regex> ')'
```

这些算法可以用递归实现，我们首先完成以下函数

1. peek() 查看模式串中下一个字符
2. eat(c) 检查下一个字符是否为 c，如果是，将模式串中去掉该元素，即返回第一个字符之后的元素
3. next() 返回下一个元素，并在模式串中去掉该元素

我们从最简单的部分开始，首先通过算法 4 检测 base, 处理转义符，括号和普通单个字母

算法 4 base 部分的解析

base()

if peek() = '(': ▷ 匹配括号
    eat('(') ▷ 处理掉'('
    r ← regex() ▷ 这一部分为另一个正则表达式, 在算法 7实现
    eat(')')
    return r ▷ 此时括号内的表达式的 NFA 是解析结果
else if peek() = '\': ▷ 处理转义符之后的字符
    eat('\')
    esc ← next() ▷ 获得转义符之后的一个字符
    return basicConstruct(esc) ▷ 创建只含有 *esc* 的 NFA
else :
    return basicConstruct(next()) ▷ 只有一种字母的情况

factor 部分由 base 和若干个 * 组成，我们对算法 4 的结果进行 star 运算 (算法 3) 具体过程如算法 5

算法 5 factor 部分的解析

```
factor()
base ← base()    ▷ 从算法 4 中得到 * 之前部分的 NFA
while parttern and peek() = '*':    ▷ 处理所有'*' 字符，并进行相应运算
    eat('*')
    star(base)    ▷ 对 base 进行 star 运算 (算法 3)
return base
```

term 部分由若干个 factor 组成，他们之间用 concatenation 运算链接 (算法 1) 具体实现如算法 6

算法 6 term 部分的解析

```
term()
term ← basicConstruct(ϵ)    ▷ 只有空串转换的 NFA
while pattern and peek() ≠ ')' and peek() ≠ '*': ▷ 下一个字符不能是需要运算符号
    f ← factor() ▷ 读取下一个 term(算法 5)
    term ← concatenation(term, f) ▷ 更新 term
return term
```

最终，我们可以将一个正则表达式分解为 term 或者 term '|' regex 的形式我们对相应 NFA union 运算 (算法 2) 然后得到最终的 NFA 具体细节见算法 7

算法 7 regex 部分的解析

```
regex()
term ← term() ▷ 第一部分为 term 部分
if pattern and peek() = '|':
    regex ← regex() ▷ 下一部分是另一个正则表达式，用自身解析
else :
    return term
```

五、 实验步骤与结果

5.1 代码测试自动机的基本构造

构造自动机类的数据类型和相关方法，并检查状态转换是否正确，运行

```
test = Automata('ab')
test.set_start_state(1)
test.add_final_states(2)
test.add_final_states(2)
test.add_transition(1,2,set(['a','b']))
test.add_transition(1,3,set('b'))
test.draw('../docs/figures/test_automata.pdf')
```

其中打印显示数据类型如下

```
{1: {2: {'a', 'b'}, 3: {'b'}}}
states: {1, 2, 3}
start state:  1
final state:  {2}
transitions:
1->2 on 'a'
1->2 on 'b'
1->3 on 'b'
```

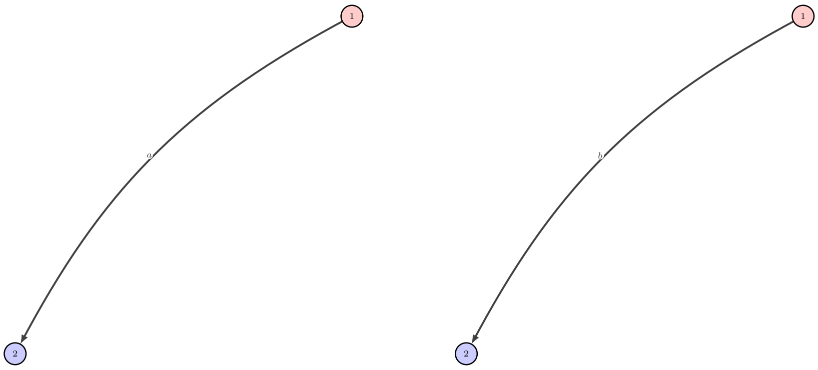
绘图结果如图 1

图 1　测试一个自动机的数据结构

图 1蓝色点代表终态，红色代表初态，其他状态为绿色

5.2 测试自动机的方法和运算

接下来测试自动机的运算，对于图 2a中的自动机 N_1 和图 2b中的自动机 N_2



(a) N_1　　(b) N_2

图 2　简单的自动机

根据算法 3 测试 star($N(s)$) 结果如

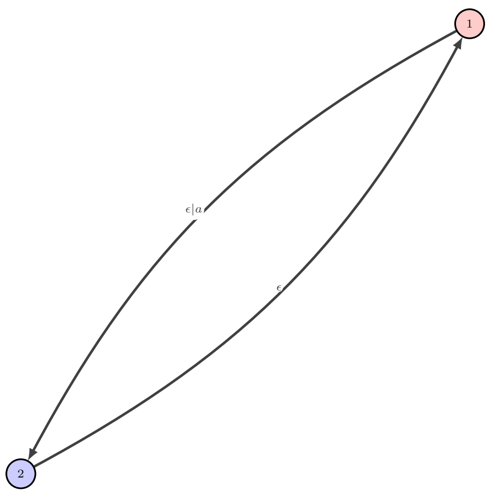


图 3　star 运算测试结果

接下来完成算法 2 并测试 $N(s), N(t)$ 如图 4

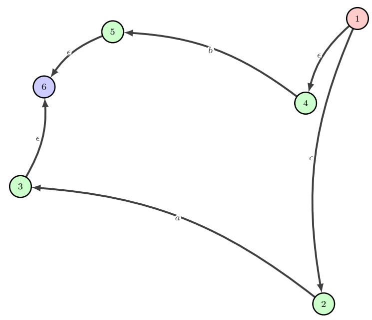


图 4　union 运算测试

最后根据算法 1 完成 concatenation 运算代码并绘图，得

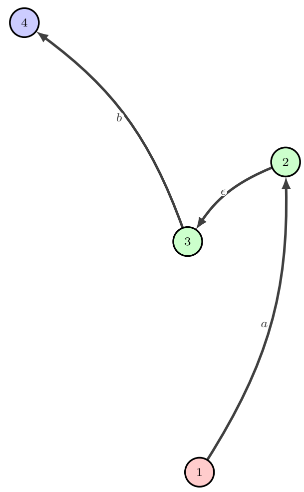


图 5　concatenation$(N(s), N(t))$ **的结果**

5.3 测试正则表达式解析

根据 算法 (4) ~ (7) 完成正则表达式解析算法我们测试正则表达式 $a|b$ 得到图 6

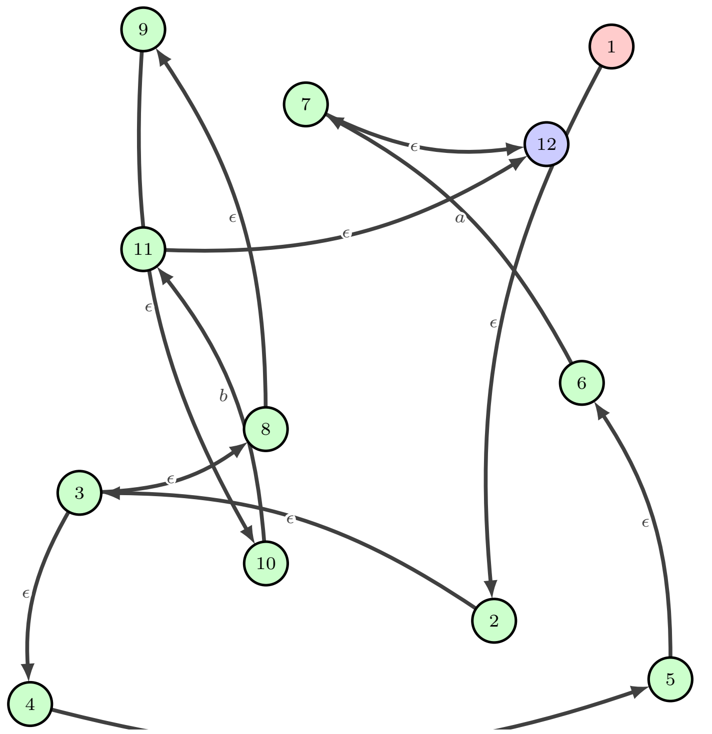


图 6　正则表达式 $a|b$ 生成的 NFA

图 6 中 $6 \rightarrow 7$,$8 \rightarrow 11$ 展示了状态转换的逻辑，但是算法中引入大量空串，使生成 NFA 变得复杂，对于像 $(a|b)*ab$ 这样复杂的表达式 () 视觉上很难识别转换关系需要进一步化简。

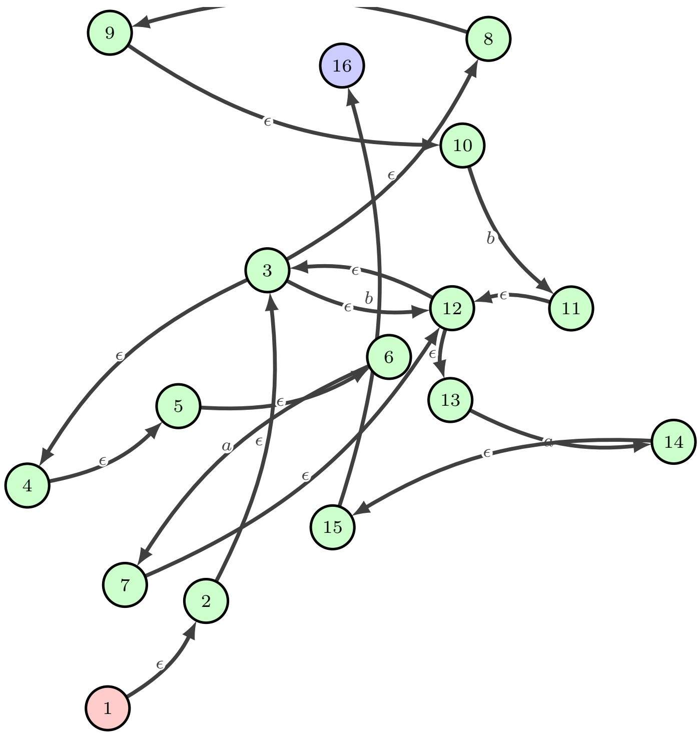


图 7　比较复杂的表达式 $(a|b)*ab$

六、实验总结

这部分实验是一组算法，非常锻炼逻辑能力和设计能力，对数据类型的封装可以应用许多面相对象技术，如实现运算时，可以使用一个对象的方法更新，也可以使用静态方法，基本的自动机类型可以由工厂模式的类方法构造，这一些对以后的程序设计很有帮助。

算法方面，文字的分析很好的展现了递归的简单优雅，利用递归可以快速构造代码，且符合思考的逻辑，但是如果想进一步优化算法，需要利用栈来做进一步分析。

这一类算法的调试，分析都是从小的部分自顶而上构造，自顶而下的思考分析，锻炼了逻辑思维。

对于网上的算法，需要自己的理解分析对于网上的代码需要改造为自己理解的形式，直接不假思索的复制，沉迷于低质量的信息，在我的实验过程中看到了一些经典的讲解 [?] 和一些完整的实现 [?] 这一些都是参考学习的优质资源，比许多明显是别的学生作业的资源优质许多，这些可以对照我们学过的理论 [?] 对编译系统有更深的了解，进而熟悉一套解析的逻辑，增强自己的能力

附录 A 依赖的安装

本文代码的 Atutomata.draw() 方法依赖

```
network2tikz
```

可以通过以下命令在清华镜像网站安装全部

```
pip install -i http://mirrors.aliyun.com/pypi/simple \
--trusted-host mirrors.aliyun.com/pypi/simple/ \
-r requirements.txt
```

如果在程序中不使用画图功能，可以忽略安装

附录 B 自动机的相关方法与测试源代码

```python
"""
filename src/Automata.py
reference https://github.com/sdht0/automata-from-regex/blob/master/AutomataTheory.py
"""
from __future__ import annotations # type hint within a class
from typing import *
# see https://stackoverflow.com/questions/41135033/type-hinting-within-a-class
from matplotlib import pyplot as plt
import networkx as nx
```

```python
import matplotlib as mpl
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": ["Helvetica"]})
# for Palatino and other serif fonts use:
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Palatino"],
})


class Automata:
    """
    class to represent a automata

    :param input_alphabet: a set of input symbols
    :type input_alphabet: set,optional

    :ivar empty_string: empty string, denoted by :math:`\epsilon`
    :ivar self.states: a finite states of S
    :ivar self.transitions: 11
    :ivar self.input_alphabet: a set of input symbols
    :ivar self.final_states: the set of final state
    :ivar self.transitions: the transitions functions,
        `translations[f][t] = d` where f is from state,t in to state,
        d is the dict of states where d[state] = set of input symbols
    """
    empty_string = set([r'\epsilon'])

    def __init__(self, input_alphabet: set):
        self.states = set() # a finite states of S
        self.input_alphabet = input_alphabet # a set of input symbols
        self.start_state = None
        self.final_states = set()
        self.transitions = dict()

    # @staticmethod
    # def empty_string() -> str:
    #     r"""get the symbol of empty_string symbol :math:`\epsilon`

    #     :return: r'\epsilon'
    #     :rtype: str
    #     """
    #     return r'\epsilon'
```

```python
def set_start_state(self, state: int):
    """set the start state

    :param state: the label of start state
    :type state: int
    """
    self.start_state = state
    self.states.add(state)

def add_final_states(self, *states):
    """add the final states

    :param states: the list of states
    """
    for state in states:
        self.final_states.add(state)

def add_transition(self, from_state: int, to_state: int, input_symbols: set):
    """add the transition to transfer functions
    (`self.transitions` in the program)

    :param from_state: the begin state
    :type from_state: int
    :param to_state: the next state
    :type to_state: int
    :param input_symbols: the transfer symbols to the next states
    :type input_symbols: set
    """

    self.states.add(from_state)
    self.states.add(to_state)
    if from_state in self.transitions:
        if to_state in self.transitions[from_state]:
            self.transitions[from_state][to_state].update(input_symbols)
        else:
            self.transitions[from_state][to_state] = input_symbols
    else:
        self.transitions[from_state] = {to_state: input_symbols}

def add_transition_from_dict(self, translations: Dict[int, Dict[int, set]]):
    """
    :param translations: translations[f][t] = d where f is from state,t in to state,
                        d is the dict of states where d[state] = set of input symbols
    :type translations: dict
    """
    for from_state, to_states in translations.items():
        for to_state, input_symbols in to_states.items():
```

```python
            self.add_transition(from_state, to_state, input_symbols)

    def __repr__(self):
        """
        display the information of the automata
        """
        trans = ""
        for from_state, to_states in self.transitions.items():
            for to_state, symbols in to_states.items():
                for char in symbols:
                    trans += f"\t{from_state}->{to_state} on '{char}'\n"
            trans += '\n'

        return f"states:\t{self.states}\n" \
            f"start state:\t{self.start_state}\n" \
            f"final state:\t{self.final_states}\n" \
            f"transitions:\n{trans}"

    def rename(self, offset: int) -> None:
        """change the state name to prevent the conflict

        :param offset: offset the number
        :type offset: int
        """
        self.states = set(i+offset for i in self.states)
        self.start_state += offset
        self.final_states = set(i+offset for i in self.final_states)

        # change the transition
        new_transitions = dict()
        for from_state, to_states in self.transitions.items():
            new_transitions[from_state+offset] = dict()
            for to_state in to_states.keys():
                new_transitions[from_state+offset][to_state+offset] = \
                    self.transitions[from_state][to_state]

        self.transitions = new_transitions

    def draw(self, save='temp.pdf',seed:int=None) -> None:
        """
        draw the graph

        :param save: save the save path (`reference <https://stackoverflow.com/a/20382152>`_)
        :type save: str
        :param seed: the node location random seed
        :type seed: int
```

```
    if you haven't installed network2tikz,
    you need install it by

    .. code-block:: bash

        pip install -U network2tikz

    """
    from network2tikz import plot
    nodes = list(self.states)
    node_colors = [
        'green!20' if node not in self.final_states else 'blue!20' for node in self.states]
    node_colors[nodes.index(self.start_state)] = "red!20"
    edges = []
    edge_labels = []
    for from_state, to_states in self.transitions.items():
        for to_state, symbols in to_states.items():
            edges.append((from_state, to_state))
            labels = []
            for symbol in symbols:
                labels.append(symbol)
            edge_labels.append("| ".join(labels))

    plot((nodes, edges), save,
        #  layout="spring_layout",
         seed=seed,
         canvas=(10,10),
         node_label_as_id=True,
         node_color=node_colors,
         edge_label=edge_labels,
         edge_math_mode=True, edge_directed=True, edge_curved=0.2,
         edge_label_position='left')

@classmethod
def empty_construct(cls):
    """construct a empty construct of a automata

    :return: the empty automata
    :rtype: Automata
    """
    return cls.basic_construct(set([r'\epsilon']))

@classmethod
def basic_construct(cls, symbol: set):
    """construct NFA with a single symbol

    :param symbol: the symbol
```

```python
        :type symbol: str
        :return: a NFA
        :rtype: Automata
        """
        basic = Automata(symbol)
        basic.set_start_state(1)
        basic.add_final_states(2)
        basic.add_transition(1, 2, set(symbol))
        return basic

    @staticmethod
    def star_operation(nfa):
        """process the star operation

        .. note::

            the nfa is changed after call the method

        :param nfa: the previous NFA
        :type nfa: Automata
        :return: the new NFA after processing star operation
            that means add two string in the begin state and end state
        :rtype: Automata
        """
        for final_state in nfa.final_states:
            nfa.add_transition(nfa.start_state, final_state,
                               set([r"\epsilon"]))
            nfa.add_transition(final_state, nfa.start_state,
                               set([r"\epsilon"]))

        return nfa

    @staticmethod
    def concatenation(basic: Automata, addition: Automata) -> Automata:
        """union two Automata

        :param basic: this Automata will be changed after union
        :type basic: Automata
        :param addition: This Automata will be deleted after union
        :type addition: Automata
        :return: [description]
        :rtype: Automata
        """
        # to manage the state name conflict
        offset = max(basic.states)
        addition.rename(offset)
```

```python
        basic.add_transition_from_dict(addition.transitions)
        for pre_final in basic.final_states:
            basic.add_transition(pre_final, addition.start_state,
                                 Automata.empty_string)

        basic.final_states = addition.final_states
        del addition
        return basic

    @staticmethod
    def union(basic: Automata, parallel: Automata) -> Automata:
        """handle the regex s|t by union these NFA

        :param basic: the NFA will change after union
        :type basic: Automata
        :param parallel: the NFA will be deleted after union
        :type parallel: Automata
        :return: The new NFA based on `basic`
        :rtype: Automata
        """
        # rename the two graph
        basic.rename(offset=1)
        offset = max(basic.states)
        parallel.rename(offset)

        # update edges
        basic.add_transition_from_dict(parallel.transitions)

        # update the start
        new_start_state = min(basic.states) - 1
        basic.add_transition(new_start_state,
                             basic.start_state, Automata.empty_string)
        basic.add_transition(new_start_state, parallel.start_state,
                             Automata.empty_string)
        basic.set_start_state(new_start_state)

        # handle the final states
        new_final_state = max(parallel.states)+1
        pre_finals = basic.final_states.union(parallel.final_states)
        for pre_final in pre_finals:
            basic.add_transition(
                pre_final, new_final_state, Automata.empty_string)
        basic.final_states = set([new_final_state])

        del parallel
        return basic
```

```python
if __name__ == "__main__":
    def figure_path(s):
        return f"../reports/regex_parser/figures/{s}.pdf"

    # basic test
    test = Automata(set('ab'))
    test.set_start_state(1)
    test.add_final_states(2)
    test.add_final_states(2)
    test.add_transition(1, 2, set(['a', 'b']))
    test.add_transition(1, 1, set('b'))
    print(test.transitions)
    print(test)
    test.draw('../reports/regex_parser/figures/test_automata.pdf',seed=2) # 2
    """ output
    {1: {2: {'a', 'b'}, 1: {'b'}}}
    states: {1, 2}
    start state:  1
    final state:  {2}
    transitions:
            1->2 on 'a'
            1->2 on 'b'
            1->1 on 'b'
    """
    print(test.transitions)
    test.rename(3)
    print(test)
    """output
        {1: {2: {'a', 'b'}, 1: {'b'}}}
        states: {4, 5}
        start state:  4
        final state:  {5}
        transitions:
                4->5 on 'a'
                4->5 on 'b'
                4->4 on 'b'
        """

    # test basic construct
    test1 = Automata.basic_construct(set(['a']))
    test1.draw(save="../reports/regex_parser/figures/basic_a.pdf",seed=1)
    print(test1)
    """
    states: {1, 2}
    start state:  1
    final state:  {2}
```

```
transitions:
       1->2 on 'a'
"""

# test star operation
test1 = Automata.star_operation(test1)
print(test1)
test1.draw('../reports/regex_parser/figures/test_star.pdf',seed=1)
r"""output
   states: {1, 2}
   start state:  1
   final state:  {2}
   transitions:
          1->2 on '\epsilon'
          1->2 on 'a'

          2->1 on '\epsilon'
"""

# test link operation
test1 = Automata.basic_construct(set(['a']))
test2 = Automata.basic_construct(set(['b']))
test2.draw(figure_path('basic_b'),seed=1)
print(Automata.concatenation(test1, test2))
test1.draw(save=figure_path('test_concatenation'),seed=2) # 2
r"""output
   states: {1, 2}
   start state:  1
   final state:  {2}
   transitions:
          1->2 on 'a'
          1->2 on '\epsilon'

          2->1 on '\epsilon'
"""

# test parallel union
test1 = Automata.basic_construct(set(['a']))
test2 = Automata.basic_construct(set(['b']))
test3 = Automata.union(test1, test2)
test3.draw(save=figure_path('test_union'),seed=79744993) # 1111
# import random
# for i in range(50):
#    s = int(random.random() * 100000000)
#    test3.draw(f'/tmp/{s}.pdf',seed=s)
print(test3)
r"""output
```

```
    states: {1, 2, 3, 4, 5, 6}
    start state:  1
    final state:  {6}
    transitions:
           2->3 on 'a'

           4->5 on 'b'

           1->2 on '\epsilon'
           1->4 on '\epsilon'

           3->6 on '\epsilon'

           5->6 on '\epsilon'
"""
```

附录 C　正则表达式的解析

```python
from Automata import Automata


class RegexParser:
    """
    store and parse a apttern

    .. code-block:: text

       <regex> ::= <term> '|' <regex>
                |  <term>

       <term> ::= { <factor> }

       <factor> ::= <base> { '*' }

       <base> ::= <char>
              |  '\\' <char>
              |  '(' <regex> ')'


    :param pattern: the pattern to match the string
    :type pattern: str

    :ivar self.pattern: the pattern
    :ivar self.NFA: the NFA machine
    """
```

```python
    # alphabet = set([chr(i) for i in range(65, 91)])\
    #     .union([chr(i) for i in range(97, 123)])\
    #     .union([chr(i) for i in range(48, 58)])


    def __init__(self, pattern: str):
        """store and parse a apttern

        :param pattern: the pattern to match the string
        :type pattern: str
        """
        self.pattern = pattern
        # self.NFA = self.build_NFA()

    # def build_NFA(self):
    #     """build a NFA from pattern create :class:`Automata.Automata`

    #     :return: the NFA of the current pattern

    #     :rtype: Automata.Automata
    #     """
    #     language = set()
    #     self.buffer = []
    #     self.automata = []
    #     previous = r'\epsilon'
    #     for char in self.pattern:
    #         if char in self.alphabet:
    #             pass
    #             # TODO
    #     return None

    def peek(self) -> str:
        """returns the next item of input without consuming it;

        :return: the next character
        :rtype: str
        """
        return self.pattern[0]

    def eat(self, item:str) -> None:
        """eat(item) consumes the next item of input, failing if not equal to item.



        :param item: the next item
        :type item: str
        :raises RuntimeError: get the wrong letter.
        """
```

```python
        if(self.peek() == item):
            self.pattern = self.pattern[1:]
        else:
            raise RuntimeError(f"expect: {item}; got {self.peek()}")

    def next(self) -> str:
        """returns the next item of input and consumes it;

        :return: the next character
        :rtype: str
        """
        c = self.peek()
        self.eat(c)
        return c

    def parse_base_part(self) -> Automata:
        """check the cases encountered

        .. code-block:: text

            <base> ::= <char>
                    |  '\\' <char>
                    |  '(' <regex> ')'

        :return: Automata of this part
        :rtype: Automata
        """
        if self.peek() == '(':
            self.eat('(')
            r = self.parse_regex()
            self.eat(')')
            return r

        elif self.peek() == '\\':
            self.eat('\\')
            esc = self.next()
            return Automata.basic_construct(esc)
        else:
            return Automata.basic_construct(self.next())

    def parse_factor_part(self) -> Automata:
        base = self.parse_base_part()

        while(self.pattern and self.peek() == '*'):
            self.eat('*')
            base = Automata.star_operation(base)
```

```python
        return base

    def parse_term_part(self) -> Automata:
        """check that it has not reached the boundary of a term or the end of the input:

        .. code-block:: text

            <term> ::= { <factor> }

        :return: the NFA of this part
        :rtype: Automata
        """
        factor = Automata.empty_construct()
        while(self.pattern and self.peek() != ')' and self.peek() != '|'):
            next_factor = self.parse_factor_part()
            factor = Automata.concatenation(factor, next_factor)

        return factor

    def parse_regex(self) -> Automata:
        """For regex() method, we know that we must parse at least one term,
        and whether we parse another

        .. code-block::text

            <regex> ::= <term> '|' <regex>
                      |  <term>

        :return: the NFA
        :rtype: Automata
        """
        term = self.parse_term_part()
        if(self.pattern and self.peek() == '|'):
            self.eat('|')
            regex = self.parse_regex()
            return Automata.union(term, regex)
        else:
            return term



if __name__ == "__main__":
    def figure_path(s):
        return f"../reports/regex_parser/figures/{s}.pdf"

    # test the cases of the only letter
    test1 = RegexParser("a")
```

```
print(test1.parse_base_part())
"""output
    states: {1, 2}
    start state:  1
    final state:  {2}
    transitions:
            1->2 on 'a'
"""

# test the escape symbol
test2 = RegexParser("\*")
print(test2.parse_base_part())
"""output
    states: {1, 2}
    start state:  1
    final state:  {2}
    transitions:
            1->2 on '*'
"""

# test parse factor part
test3 = RegexParser('a*')
print(test3.parse_factor_part())
r"""output
    states: {1, 2}
    start state:  1
    final state:  {2}
    transitions:
            1->2 on '\epsilon'
            1->2 on 'a'

            2->1 on '\epsilon'
"""

test4 = RegexParser('ab')
print(test4.parse_term_part())
r"""output
    states: {1, 2, 3, 4, 5, 6}
    start state:  1
    final state:  {6}
    transitions:
            1->2 on '\epsilon'

            3->4 on 'a'

            2->3 on '\epsilon'
```

```
          5->6 on 'b'

          4->5 on '\epsilon'
"""
nfa1 = RegexParser('(a|b)').parse_regex()
print(nfa1)
# # which is best
# import random
# import os
# import time
# l = []
# for i in range(100):
#     s = int(random.random() * 100000000)
#     nfa1.draw(f'/tmp/ab/{s}.pdf',seed=s)
#     l.append(s)
# time.sleep(10)
# for s in l:
#     os.system(f"pdftoppm /tmp/ab/{s}.pdf /tmp/ab/{s} -png")
nfa1.draw(save=figure_path("a|b"),seed=79870681)

# test a complex
nfa2 = RegexParser('(a|b)*ab').parse_regex()
nfa2.draw(save=figure_path('complex'),seed=53138909)
```