一、 实验目的

掌握从正则表达式构造 NFA 的算法

二、 实验内容与实验要求

一、实验目的

掌握从正则表达式构造 NFA 的算法

二、实验内容

1.输入：一个正则表达式（例如“(a|b)*abb”）；输出：对应的一个 NFA（可以不用图形表示）;

2.编制测试程序;

3.调试程序;

三、 设计方案与算法描述

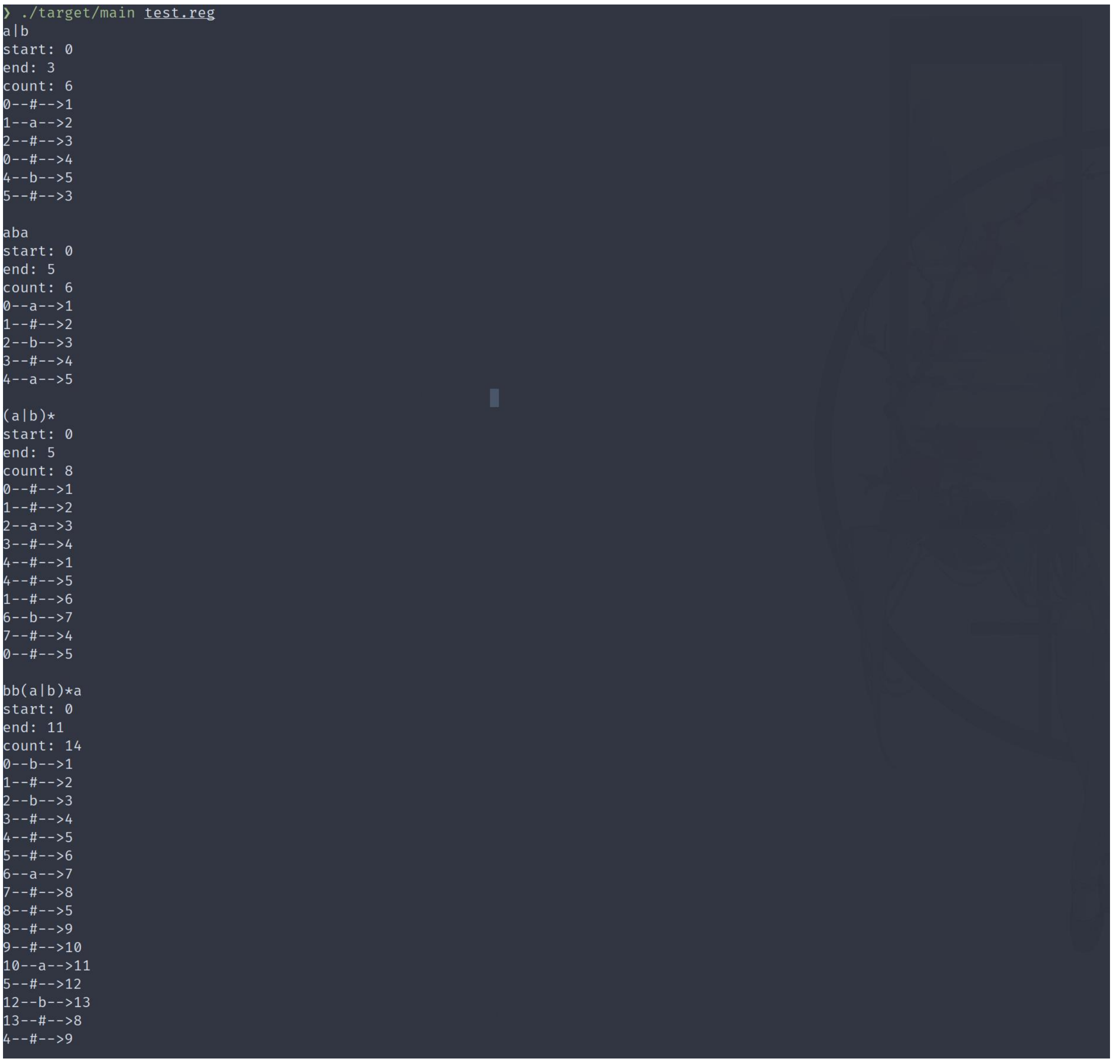
把正则表达式解析成一颗树，再从语法树构建 nfa

四、 测试结果

测试例子

```
#or
a|b
#conn
aba
#closure
(a|b)*
bb(a|b)*a
```

结果



```
❯ ./target/main test.reg
a|b
start: 0
end: 3
count: 6
0--#-->1
1--a-->2
2--#-->3
0--#-->4
4--b-->5
5--#-->3

aba
start: 0
end: 5
count: 6
0--a-->1
1--#-->2
2--b-->3
3--#-->4
4--a-->5

(a|b)*
start: 0
end: 5
count: 8
0--#-->1
1--#-->2
2--a-->3
3--#-->4
4--#-->1
4--#-->5
1--#-->6
6--b-->7
7--#-->4
0--#-->5

bb(a|b)*a
start: 0
end: 11
count: 14
0--b-->1
1--#-->2
2--b-->3
3--#-->4
4--#-->5
5--#-->6
6--a-->7
7--#-->8
8--#-->5
8--#-->9
9--#-->10
10--a-->11
5--#-->12
12--b-->13
13--#-->8
4--#-->9
```

五、 源代码

```
#include <cassert>
#include <cstdint>
#include <iostream>
#include <iterator>
#include <memory>
#include <optional>
#include <set>
#include <string>
#include <string_view>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

using std::array;
using std::move;
using std::optional;
using std::shared_ptr;
using std::string;
using std::string_view;
using std::unordered_map;
using std::unordered_set;
using std::vector;

constexpr char EPSILON = '#';

struct Parser;

struct NFA {
public:
  struct Node {
  public:
    struct AdjEnrty {
      bool valid;
      char via;
      Node *to;
      AdjEnrty(char via, Node *to) : via(via), to(to), valid(true) {}
      AdjEnrty() : via('\0'), to(nullptr), valid(false) {}
    };

    // 对于正则表达式生成的 nfa 来说, 每一个节点至多有两个出边
    struct Adj : array<AdjEnrty, 2> {
      void insert(char via, Node *to) {
```

```
    if (auto &entry0 = this->at(0); !entry0.valid) {
      entry0 = AdjEnrty(via, to);
    } else if (auto &entry1 = this->at(1); !entry1.valid) {
      entry1 = AdjEnrty(via, to);
    } else {
      assert(false);
    }
  }
};

static constexpr int UNALLOC_ID = -1;
Node() : id(UNALLOC_ID), adj() {}
void set_to(char via, Node *to) { this->adj.insert(via, to); }
bool is_terminal() {
  return !this->adj.at(0).valid && !this->adj.at(1).valid;
}

void alloc_state(int &allocator) {
  if (this->id != UNALLOC_ID) {
    return;
  } else {
    this->id = allocator;
    allocator++;
    if (auto &entry0 = this->adj.at(0); entry0.valid) {
      entry0.to->alloc_state(allocator);
    }
    if (auto &entry1 = this->adj.at(1); entry1.valid) {
      entry1.to->alloc_state(allocator);
    }
  }
}

void print(unordered_set<Node *> &visited) {
  if (visited.find(this) != visited.end()) {
    return;
  }
  visited.insert(this);
  if (auto &entry0 = this->adj.at(0); entry0.valid) {
    std::cout << this->id << "--" << entry0.via << "-->" << entry0.to->id
              << std::endl;
    entry0.to->print(visited);
  }
  if (auto &entry1 = this->adj.at(1); entry1.valid) {
    std::cout << this->id << "--" << entry1.via << "-->" << entry1.to->id
```

```
                  << std::endl;
        entry1.to->print(visited);
      }
    }

    int id;
    Adj adj;
  };
  Node *start;
  Node *end;
  int cnt;
  NFA(Node *start, Node *end) : start(start), end(end), cnt(0) {}
  NFA *alloc_state() {
    auto allocator = 0;
    this->start->alloc_state(allocator);
    cnt = allocator;
    return this;
  }
  void print() {
    auto visited = unordered_set<Node *>();
    std::cout << "start: " << this->start->id << std::endl;
    std::cout << "end: " << this->end->id << std::endl;
    std::cout << "count: " << this->cnt << std::endl;
    this->start->print(visited);
  }
};

struct RegExp {
  virtual NFA to_nfa() { assert(false); }
  virtual string to_string() { assert(false); }
};

struct CharExp : RegExp {
  char ch;
  CharExp(char ch) : ch(ch) {}
  NFA to_nfa() override {
    auto start = new NFA::Node();
    auto end = new NFA::Node();
    start->set_to(ch, end);
    return NFA(start, end);
  }
  string to_string() override { return string(1, ch); }
};
```

```
struct ClosureExp : RegExp {
  RegExp *inner;
  ClosureExp(RegExp *inner) : inner(inner) {}
  NFA to_nfa() override {
    auto nfa = inner->to_nfa();
    auto inner_start = nfa.start;
    auto inner_end = nfa.end;
    auto start = new NFA::Node();
    auto end = new NFA::Node();
    start->set_to(EPSILON, inner_start);
    start->set_to(EPSILON, end);
    inner_end->set_to(EPSILON, inner_start);
    inner_end->set_to(EPSILON, end);
    return NFA(start, end);
  }
  string to_string() override { return "(" + inner->to_string() + ")*"; }
};

struct OrExp : RegExp {
  RegExp *case_a;
  RegExp *case_b;
  OrExp(RegExp *case_a, RegExp *case_b) : case_a(case_a), case_b(case_b) {}
  NFA to_nfa() override {
    auto nfa_a = case_a->to_nfa();
    auto nfa_b = case_b->to_nfa();
    auto start = new NFA::Node();
    auto end = new NFA::Node();
    start->set_to(EPSILON, nfa_a.start);
    start->set_to(EPSILON, nfa_b.start);
    nfa_a.end->set_to(EPSILON, end);
    nfa_b.end->set_to(EPSILON, end);
    return NFA(start, end);
  }
  string to_string() override {
    return "(" + case_a->to_string() + "|" + case_b->to_string() + ")";
  }
};

struct ConnExp : RegExp {
  RegExp *head;
  RegExp *tail;
  ConnExp(RegExp *head, RegExp *tail) : head(head), tail(tail) {}
  NFA to_nfa() override {
    auto head_nfa = head->to_nfa();
    auto tail_nfa = tail->to_nfa();
    head_nfa.end->set_to(EPSILON, tail_nfa.start);
    return NFA(head_nfa.start, tail_nfa.end);
  }
  string to_string() override { return head->to_string() + tail->to_string(); }
};

struct Parser {
  Parser(string_view input) : stream(input) {
    auto ch = this->stream.next();
    assert(ch != EOL);
    this->cur = ch;
  }

  RegExp *parse() {
    auto ret = this->parse_exp();
    assert(!this->next());
    return ret;
  }

private:
  struct Stream {
  public:
    string_view input;
    Stream(string_view input) : input(input) {}
    char next() {
      if (this->input.empty()) {
        return EOL;
      }
      return this->pop();
    }

  private:
    char pop() {
      auto ret = this->input[0];
      this->input = this->input.substr(1);
      return ret;
    }
  };

  static constexpr char STAR = '*';
  static constexpr char OR = '|';
  static constexpr char LEFT_PAREN = '(';
  static constexpr char RIGHT_PAREN = ')';
  static constexpr char EOL = '\0';

  Stream stream;
  char cur;
  bool next() {
    this->cur = this->stream.next();
    return this->cur != EOL;
  }
  bool has_next() { return this->cur != EOL; }

  RegExp *parse_exp() {
    auto left = this->parse_term();
    while (this->has_next() && this->cur == OR) {
      assert(this->next());
      auto right = this->parse_term();
      left = new OrExp(left, right);
    }
    return left;
  }

  RegExp *parse_term() {
    auto head = this->parse_atomic();
    while (this->has_next() && this->cur != OR && this->cur != RIGHT_PAREN) {
      auto tail = this->parse_atomic();
      head = new ConnExp(head, tail);
    }
    return head;
  }


  // 单个字符构成的字符表达式或者是被括号包裹的表达式

  RegExp *parse_atomic() {
    if (this->cur == LEFT_PAREN) {
      assert(this->next());
      auto inner = this->parse_exp();
      assert(this->cur == RIGHT_PAREN);
      if (this->next() && this->consume_star()) {
        return new ClosureExp(inner);
      } else {
        return inner;
      }
    } else {
      assert(this->cur != RIGHT_PAREN && this->cur != OR && this->cur != STAR &&
             this->cur != EOL);
      auto regexp = new CharExp(this->cur);
      if (this->next() && this->consume_star()) {
        return new ClosureExp(regexp);
      } else {
        return regexp;
      }
    }
  }


  // 检测否有*, 如果有则消耗掉直到 this.cur != *返回 true,

  // 否则返回 false
  bool consume_star() {
    if (this->cur == STAR) {
      while (this->next() && this->cur == STAR) {
      }
      return true;
    } else {
      return false;
    }
  }
};
#include "../../common/comm.hpp"
#include "./nfa_from_regexp.hpp"
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <iostream>

int main(int argc, char *argv[]) {
  assert(argc == 2);
  auto file = argv[1];
  auto content = Util::read_file_to_string(file);
  auto regexs = Util::lines(content);
  for (auto regex : regexs) {
    std::cout << regex << std::endl;
    Parser(regex).parse()->to_nfa().alloc_state()->print();
    getchar();
  }
  return 0;
}
```