杭州电子科技大学

《编译原理课程实践》

实验报告

题　　目：词法分析核心算法实现

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：23052322

学　　号：23051214

姓　　名：张逸轩

完成日期：2024-10-28

一、 实验目的

通过实验了解词法分析的过程，正规表达式 ， NFA, DFA 转换。

二、 实验内容与实验要求

词法分析算法实现，含 2 个算法：

（1）实验任务 2.1 正规表达式转 NFA 算法及实现；

（2）实验任务 2.2 NFA 转 DFA 算法及实现；

三、 设计方案与算法描述

（1） 正规表达式转 NFA：

采取 python 实现，将正规表达式从中缀形式变换为后缀形式，然后参考 Thompson 构造法的规则,实现 4 种情况的基本转换（单符号， 连接，并联，闭包），最后合并。以及模拟和验证。

（2）NFA 转 DFA：

采取 python 实现，设计 NFA, DFA 数据结构，计算 epsilon 闭包，使用子集构造法，把 NFA 生成对应的 DFA,并且设计对应的模拟代码，用于测试结果。

四、 测试结果

（1）.正规表达式转 NFA 并且验证

```
hewo@hewo-thinkpad ~/CS/hdu/comp/lab2/lab2-1/src main python3 test.py
testing regex: a(b|c)*
postfix regex: abc|*.
Input: abcbcbc, Accepted: True
Input: accccc, Accepted: True
Input: abbbb, Accepted: True
Input: a, Accepted: True
Input: abc, Accepted: True
Input: bbc, Accepted: False
```

（2）

NFA 转 DFA 并且验证

```
~/CS/hdu/hdu-compile-principles/lab2/lab2-2/src main
> python3 test.py
testing nfa_to_dfa...
string: abc, DFA: True, expect: True, test pass
string: aab, DFA: False, expect: False, test pass
string: ab, DFA: False, expect: False, test pass
string: ac, DFA: False, expect: False, test pass
testing nfa_to_dfa...
string: a, DFA: True, expect: True, test pass
string: ab, DFA: True, expect: True, test pass
string: ac, DFA: True, expect: True, test pass
string: abc, DFA: True, expect: True, test pass
string: acb, DFA: True, expect: True, test pass
string: abcbcbc, DFA: True, expect: True, test pass
string: , DFA: False, expect: False, test pass
string: b, DFA: False, expect: False, test pass
string: ba, DFA: False, expect: False, test pass
testing nfa_to_dfa...
string: ab, DFA: False, expect: False, test pass
string: abb, DFA: True, expect: True, test pass
string: aabb, DFA: True, expect: True, test pass
string: b, DFA: False, expect: False, test pass
string: bb, DFA: False, expect: False, test pass
string: bbb, DFA: False, expect: False, test pass
string: aab, DFA: False, expect: False, test pass
string: ba, DFA: False, expect: False, test pass
```

五、 源代码

（1）.正规表达式转 NFA 并且验证

转换部分：

```python
def add_explicit_concat_operator(expression):
    # 添加显式连接操作符
    output = []
    operators = {'*', '|', '(', ')'}
    for i in range(len(expression) - 1):
        output.append(expression[i])
        # 如果当前字符是操作数或闭包操作后，下一个字符是操作数或'('，则需要加'.'
        if (expression[i].isalnum() or expression[i] == '*') and \
           (expression[i+1].isalnum() or expression[i+1] == '('):
            output.append('.')
    output.append(expression[-1])
    return ''.join(output)

def infix_to_postfix(expression):
    precedence = {'*': 3, '.': 2, '|': 1}  # '.' 表示连接操作
    stack = []
    output = []
    for char in expression:
        if char.isalnum():  # 操作数直接输出
            output.append(char)
        elif char == '(':  # '(' 入栈
            stack.append(char)
        elif char == ')':  # ')' 弹栈直到遇到 '('
            top_token = stack.pop()
            while top_token != '(':
                output.append(top_token)
                top_token = stack.pop()
        else:  # 运算符
            while (stack and stack[-1] != '(' and
                   precedence[stack[-1]] >= precedence[char]):
                output.append(stack.pop())
            stack.append(char)

    while stack:
        output.append(stack.pop())
    return ''.join(output)


```

分类处理：

```python
class State:
    def __init__(self, is_final=False):
        self.is_final = is_final
        self.transitions = {}

    def add_transition(self, symbol, state):
        if symbol in self.transitions:
            self.transitions[symbol].add(state)
        else:
            self.transitions[symbol] = {state}

class NFA:
    def __init__(self, start_state=None):
        self.start_state = start_state
        self.states = set()

    def add_state(self, state):
        self.states.add(state)

def build_nfa_single_symbol(symbol):
    start = State()
    end = State(is_final=True)
    start.add_transition(symbol, end)
    nfa = NFA(start)
    nfa.add_state(start)
    nfa.add_state(end)
    return nfa

def concat_nfa(first_nfa, second_nfa):
    for state in first_nfa.states:
        if state.is_final:
            state.is_final = False  # 第一个NFA的接受状态不再是接受状态
            for symbol, states in second_nfa.start_state.transitions.items():
                for s in states:
                    state.add_transition(symbol, s)

    first_nfa.states.update(second_nfa.states)
    return first_nfa

def parallel_nfa(first_nfa, second_nfa):
    start = State()  # 新的起始状态
    end = State(is_final=True)  # 新的接受状态
    start.add_transition('ε', first_nfa.start_state)
    start.add_transition('ε', second_nfa.start_state)

    # 将所有原始接受状态的转移指向新的接受状态
    for state in first_nfa.states.union(second_nfa.states):
        if state.is_final:
            state.is_final = False
            state.add_transition('ε', end)

    new_nfa = NFA(start)
    new_nfa.states = first_nfa.states.union(second_nfa.states, {start, end})
    return new_nfa

def kleene_star_nfa(nfa):
    start = State()  # 新的起始状态
    end = State(is_final=True)  # 新的接受状态
    start.add_transition('ε', nfa.start_state)
    start.add_transition('ε', end)

    # 所有原始接受状态现在通过ε转移回原始起始状态和新的接受状态
    for state in nfa.states:
        if state.is_final:
            state.is_final = False
```

合并处理

```python
import nfa_part

def build_nfa_from_postfix(postfix_expr):
    stack = []
    for char in postfix_expr:
        if char.isalnum():  # 是操作数，构建单个符号的NFA
            stack.append(nfa_part.build_nfa_single_symbol(char))
        elif char == '.':  # 连接操作
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(nfa_part.concat_nfa(nfa1, nfa2))
        elif char == '|':  # 并联操作
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(nfa_part.parallel_nfa(nfa1, nfa2))
        elif char == '*':  # 闭包操作
            nfa = stack.pop()
            stack.append(nfa_part.kleene_star_nfa(nfa))

    return stack.pop()  # 最后堆栈中剩下的NFA是完整的NFA

```

模拟：

```python
def simulate_nfa(nfa, input_string):
    current_states = set()
    # 初始化当前状态包括起始状态及其通过ε转移可达的所有状态
    add_epsilon_transitions(current_states, nfa.start_state)

    for char in input_string:
        next_states = set()
        for state in current_states:
            if char in state.transitions:
                for next_state in state.transitions[char]:
                    add_epsilon_transitions(next_states, next_state)
        current_states = next_states

    # 检查是否有任何当前状态是接受状态
    return any(state.is_final for state in current_states)

def add_epsilon_transitions(state_set, state):
    if state not in state_set:
        state_set.add(state)
        if 'ε' in state.transitions:
            for next_state in state.transitions['ε']:
                add_epsilon_transitions(state_set, next_state)

```

测试

```python
import nfa_all
import sim
import trans

def test_nfa():
    regex = "a(b|c)*"
    print(f"testing regex: {regex}")
    processed_regex = trans.add_explicit_concat_operator(regex)
    postfix_regex = trans.infix_to_postfix(processed_regex)
    print(f"postfix regex: {postfix_regex}")
    nfa = nfa_all.build_nfa_from_postfix(postfix_regex)

    test_strings = ["abcbcbc", "accccc", "abbbb", "a", "abc", "bbc"]
    for s in test_strings:
        result = sim.simulate_nfa(nfa, s)
        print(f"Input: {s}, Accepted: {result}")

test_nfa()

```

（2）NFA 转 DFA 并且验证

NFA 操作

```python
class NFA:
    def __init__(self, states, transitions, initial_state, accept_states):
        self.states = states  # 所有状态的集合
        self.transitions = transitions  # 转换关系，格式为 {state: {symbol: set(states)}}
        self.initial_state = initial_state  # 初始状态
        self.accept_states = accept_states  # 接受状态集合

    def add_transition(self, from_state, symbol, to_state):
        if from_state not in self.transitions:
            self.transitions[from_state] = {}
        if symbol not in self.transitions[from_state]:
            self.transitions[from_state][symbol] = set()
        self.transitions[from_state][symbol].add(to_state)

def epsilon_closure(nfa, states):
    # epsilon 闭包
    stack = list(states)
    closure = set(states)
    while stack:
        state = stack.pop()

        if 'ϵ' in nfa.transitions[state]:
            for next_state in nfa.transitions[state]['ϵ']:
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
    return closure

```

DFA 操作

```python
import nfa_pre

class DFA:
    def __init__(self):
        self.states = set()  # DFA状态集
        self.transitions = {}  # DFA状态转换表
        self.initial_state = None  # DFA的初始状态
        self.accept_states = set()  # DFA的接受状态集合

    def add_state(self, state):
        self.states.add(state)

    def add_transition(self, from_state, symbol, to_state):
        from_state = frozenset(from_state)
        if from_state not in self.transitions:
            self.transitions[from_state] = {}
        self.transitions[from_state][symbol] = to_state

    def set_initial_state(self, state):
        self.initial_state = state

    def add_accept_state(self, state):
        self.accept_states.add(state)

def nfa_to_dfa(nfa):
    initial_closure = nfa_pre.epsilon_closure(nfa, {nfa.initial_state})
    dfa = DFA()
    unmarked_states = [initial_closure]  # 未标记的DFA状态集合
    dfa.set_initial_state(frozenset(initial_closure))
    dfa.add_state(frozenset(initial_closure))

    while unmarked_states:
        current_dfa_state = unmarked_states.pop(0)
        # 获取所有符号
        for symbol in set(sym for state in current_dfa_state for sym in nfa.transitions.get(state, {})):
            if symbol == 'ϵ':
                continue

            new_state_set = set()

            for state in current_dfa_state:
                if symbol in nfa.transitions.get(state, {}):
                    new_state_set.update(nfa_pre.epsilon_closure(nfa, nfa.transitions[state][symbol]))

            new_state_frozenset = frozenset(new_state_set)
            if new_state_frozenset not in dfa.states:
                dfa.add_state(new_state_frozenset)
                unmarked_states.append(new_state_frozenset)
                if any(s in nfa.accept_states for s in new_state_frozenset):
                    dfa.add_accept_state(new_state_frozenset)
            dfa.add_transition(current_dfa_state, symbol, new_state_frozenset)

    return dfa

```

模拟与测试

```python
import nfa_pre
import dfa_trans
import sim

def test_nfa_to_dfa(nfa, test_cases):

    print("testing nfa_to_dfa...")

    # 将 NFA 转换为 DFA
    dfa = dfa_trans.nfa_to_dfa(nfa)

    # 执行测试
    results = []
    for input_string, expected in test_cases:
        # 模拟 DFA
        accepts = sim.simulate_dfa(dfa, input_string)
        results.append((input_string, accepts, expected))

    # 输出测试结果
    for result in results:
        print(f"string: {result[0]}, DFA: {result[1]}, expect: {result[2]}, test {'pass' if result[1] == result[2] else 'fail'}")

nfa1 = nfa_pre.NFA(
    states={'q0', 'q1', 'q2', 'q3'},
    transitions={
        'q0': {'a': {'q1'}},
        'q1': {'b': {'q2'}, 'a': {'q3'}},
        'q2': {'c': {'q3'}},
        'q3': {}
    },
    initial_state='q0',
    accept_states={'q3'}
)

test_cases1 = [
    ('abc', True),
    ('aab', False),
    ('ab', False),
    ('ac', False)
]

# a(b|c)*
nfa2 = nfa_pre.NFA(
    states={'q0', 'q1'},
    transitions={
        'q0': {'a': {'q1'}},
        'q1': {'b': {'q1'}, 'c': {'q1'}}
    },
    initial_state='q0',
    accept_states={'q1'}
)

test_cases2 = [
    ('a', True),
    ('ab', True),
    ('ac', True),
    ('abc', True),
    ('acb', True),
    ('abcbcbc', True),
    ('', False),
    ('b', False),
    ('ba', False)
]

nfa3 = nfa_pre.NFA(
    states=set(range(11)),
    transitions={
        'q0': {'ε': {'q1', 'q7'}},
        'q1': {'ε': {'q2', 'q4'}},
        'q2': {'a': {'q3'}},
        'q3': {'ε': {'q6'}},
        'q4': {'b': {'q5'}},
        'q5': {'ε': {'q6'}},
        'q6': {'ε': {'q7'}},
        'q7': {'a': {'q8'}},
        'q8': {'b': {'q9'}},
        'q9': {'b': {'q10'}},
        'q10': {}
    },
    initial_state='q0',
    accept_states={'q10'}
)

test_cases3 = [
    ('ab', False),
    ('abb', True),
    ('aabb', True),
    ('b', False),
    ('bb', False),
    ('bbb', False),
    ('aab', False),
    ('ba', False)
]


# 运行测试函数
test_nfa_to_dfa(nfa1, test_cases1)
test_nfa_to_dfa(nfa2, test_cases2)
test_nfa_to_dfa(nfa3, test_cases3)


```