ثمین مهدی پور ۹۸۳۹۰۳۹

گزارش پروژه اول نظریه زبان ها و ماشین ها

بخش ۱)

فایل ورودی را میخوانیم:

```python
def main():
    dfa_filename = 'DFA_Input_1.txt'
    dfa = parse_dfa_input(dfa_filename)

    input_string = input('Enter a string: ')
    if dfa.accepts(input_string):
        print('Accepted')
    else:
        print('Not accepted')
```

```python
def parse_dfa_input(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    alphabet = lines[0].split()
    states = lines[1].split()
    start_state = lines[2].strip()
    accept_states = lines[3].split()

    transitions = {}
    for line in lines[4:]:
        parts = line.split()
        transitions[(parts[0], parts[1])] = parts[2]

    return DFA(states, alphabet, transitions, start_state, accept_states)
```

با استفاده از این فایل اطلاعات وارد میشود.

از کاربر ورودی میگیریم و با تابع زیر چک میکنیم که قابل قبول است یا خیر:

```
def accepts(self, input_string):
    current_state = self.start_state
    for char in input_string:
        if char not in self.alphabet:
            return False
        current_state = self.transitions.get((current_state, char))
        if current_state is None:
            return False
    return current_state in self.accept_states
```

```
C:\Users\Samin\PycharmProjects\HelloWorld\venv\Script
Enter a string: a
Accepted
```

```
C:\Users\Samin\PycharmProjects
Enter a string: ab
Not accepted
```

بخش۲)

کلاس `NFA` مدلی از یک ماشین پذیرنده متناهی غیرقطعی (NFA) را نشان می‌دهد. این کلاس دارای ویژگی‌های مهمی مانند وضعیت‌ها (`states`)، الفبا (`alphabet`)، گذارها (`transitions`)، وضعیت شروع (`start_state`) و وضعیت‌های قبولی (`accept_states`) است. همچنین این کلاس توابعی مانند `lambda_closure` و `move` و convert_lambda_nfa_to_nfa ، را برای محاسبه نقطه بسته فاصله‌ای (lambda closure) و گذارها دریافتی (move) و تبدیل لامبدا nfa به nfa ارائه می‌دهد.

```python
class NFA:
    def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = accept_states

    def lambda_closure(self, states):
        closure = set(states)
        stack = list(states)

        while stack:
            state = stack.pop()
            lambda_transitions = self.transitions.get((state, 'lambda'))

            if lambda_transitions:
                for epsilon_state in lambda_transitions:
                    if epsilon_state not in closure:
                        closure.add(epsilon_state)
                        stack.append(epsilon_state)
        return closure

    def move(self, states, symbol):
        move_states = set()
        for state in states:
            symbol_transitions = self.transitions.get((state, symbol))
            if symbol_transitions:
                move_states.update(symbol_transitions)
```

کلاس `DFA` مدلی از یک ماشین پذیرنده متناهی قطعی (DFA) را نشان می‌دهد. این کلاس نیز دارای ویژگی‌های مشابه با کلاس `NFA` است. همچنین تابع `to_dfa` در این کلاس، یک NFA را به یک DFA تبدیل می‌کند.

```
class DFA:
    def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = accept_states

    def to_dfa(self, nfa):
        dfa_states = set()
        dfa_start_state = frozenset(nfa.lambda_closure({nfa.start_state}))
        dfa_states.add(dfa_start_state)
        dfa_transitions = {}
        dfa_accept_states = []

        stack = [dfa_start_state]

        while stack:
            current_states = stack.pop()

            for symbol in nfa.alphabet:
                move_states = set()
                for state in current_states:
                    move_states.update(nfa.transitions.get((state, symbol), set()))

                epsilon_closure = set()
                for move_state in move_states:
                    epsilon_closure.update(nfa.lambda_closure({move_state}))
```

تابع `parse_nfa_input` وظیفه‌ای دارد که ورودی NFA را از یک فایل مشخص خوانده و اطلاعات مربوط به آن را استخراج کند و یک شیء NFA ایجاد کند.

```
def parse_nfa_input(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    alphabet = lines[0].split()
    states = lines[1].split()
    start_state = lines[2].strip()
    accept_states = lines[3].split()

    transitions = {}
    for line in lines[4:]:
        parts = line.split()
        current_state = parts[0]
        s = parts[1]
        if s in alphabet:
            symbol = s
        else:
            symbol = 'lambda'
        next_states = parts[2:]
        transitions.setdefault((current_state, symbol), set()).update(next_states)

    return NFA(states, alphabet, transitions, start_state, accept_states)
```

تابع `write_dfa_output` یک DFA را در یک فایل خروجی ذخیره می‌کند. این تابع مشخصات مربوط به DFA را به همراه گذاره‌ها در فایل خروجی ذخیره می‌کند.

```
def write_dfa_output(dfa, filename):
    with open(filename, 'w') as file:
        file.write(' '.join(dfa.alphabet) + '\n')
        file.write(' '.join(str(sorted(state)) for state in dfa.states) + '\n')
        file.write(str(sorted(dfa.start_state)) + '\n')
        file.write(' '.join(str(sorted(state)) for state in dfa.accept_states) + '\n')
        for (current_states, symbol), next_state in dfa.transitions.items():
            current_states_str = ', '.join(sorted(state for state in current_states))
            next_state_str = ', '.join(sorted(state for state in next_state))
            file.write(f'[ {current_states_str} ] {str(symbol)}  [ {next_state_str} ]\n')
```

تابع `main`، تابع اصلی برنامه است که وظیفه مدیریت فرآیند اجرای برنامه را بر عهده دارد. در این تابع، ابتدا فایل ورودی NFA خوانده می‌شود و سپس با استفاده از توابع `parse_nfa_input` و `to_dfa`، NFA به DFA تبدیل می‌شود. در نهایت، خروجی DFA در یک فایل خروجی ذخیره می‌شود.

```
def main():
    nfa_filename = 'NFA_Input_2.txt'
    dfa_filename = 'DFA_Output_2.txt'

    nfa = parse_nfa_input(nfa_filename)
    nfa = nfa.convert_lambda_nfa_to_nfa()

    dfa = DFA([], nfa.alphabet, {}, '', [])

    dfa = dfa.to_dfa(nfa)

    write_dfa_output(dfa, dfa_filename)


if __name__ == '__main__':
    main()
```

با اجرای این برنامه، فایل NFA_Input۲_ خوانده می‌شود و نتیجه به صورت یک فایل خروجی با نام DFA_Output_۲.txt ذخیره می‌شود.

```
0 1
['q0', 'q2'] ['q0'] ['q1', 'q2'] ['q0', 'q1', 'q2']
['q0']
['q0', 'q2'] ['q0'] ['q1', 'q2'] ['q0', 'q1', 'q2']
[ q0 ] 0  [ q0, q1, q2 ]
[ q0 ] 1  [ q1, q2 ]
[ q1, q2 ] 0  [ q0, q2 ]
[ q1, q2 ] 1  [ q1, q2 ]
[ q0, q2 ] 0  [ q0, q1, q2 ]
[ q0, q2 ] 1  [ q1, q2 ]
[ q0, q1, q2 ] 0  [ q0, q1, q2 ]
[ q0, q1, q2 ] 1  [ q1, q2 ]
```

بخش۳)

- `charType`: یک شاخص برای نوع کاراکترهای عبارت منظم. مقادیر ممکن برای این شاخص شامل: ``SYMBOL`، `CONCAT`، ``UNION`` و `KLEENE` هستند که به ترتیب به ترتیب معادل با کاراکترهای عملگر، عملگر اتصال، عملگر اجتماع و عملگر تکرار صفر یا بیشتر هستند.

- `NFAState`: یک کلاس که وضعیت‌ها (استیت‌ها) را در اتومات نمایش می‌دهد. هر وضعیت دارای یک وضعیت بعدی است که به کلمه ورودی مشخصی وابسته است.

- `ExpressionTree`: یک کلاس که یک درخت بیان عبارت منظم را نمایش می‌دهد. هر گره از درخت دارای نوع کاراکتر و مقدار مربوطه است.

```
class charType:
    SYMBOL = 1
    CONCAT = 2
    UNION = 3
    KLEENE = 4


class NFAState:
    def __init__(self):
        self.next_state = {}


class ExpressionTree:

    def __init__(self, charType, value=None):
        self.charType = charType
        self.value = value
        self.left = None
        self.right = None
```

توابع مهم کد عبارتند از:

- `make_exp_tree`: با گرفتن عبارت منظم ورودی، درخت بیان مربوطه را برمی‌گرداند.

```
def make_exp_tree(regexp):
    stack = []
    for c in regexp:
        if c == "+":
            z = ExpressionTree(charType.UNION)
            z.right = stack.pop()
            z.left = stack.pop()
            stack.append(z)
        elif c == ".":
            z = ExpressionTree(charType.CONCAT)
            z.right = stack.pop()
            z.left = stack.pop()
            stack.append(z)
        elif c == "*":
            z = ExpressionTree(charType.KLEENE)
            z.left = stack.pop()
            stack.append(z)
        elif c == "(" or c == ")":
            continue
        else:
            stack.append(ExpressionTree(charType.SYMBOL, c))
    return stack[0]
```

- `compute_regex`: با گرفتن درخت بیان عبارت منظم، معادل آن NFA را محاسبه می‌کند.

```python
def compute_regex(exp_t):
    # returns E-NFA
    if exp_t.charType == charType.CONCAT:
        return do_concat(exp_t)
    elif exp_t.charType == charType.UNION:
        return do_union(exp_t)
    elif exp_t.charType == charType.KLEENE:
        return do_kleene_star(exp_t)
    else:
        return eval_symbol(exp_t)
```

- `arrange_transitions`: تابع بازگشتی است که ترتیب گذار از وضعیت‌ها و ترانزیشن‌ها را در اتومات تنظیم می‌کند.

```python
def arrange_transitions(state, states_done, symbol_table):
    global nfa

    if state in states_done:
        return

    states_done.append(state)

    for symbol in list(state.next_state):
        if symbol not in nfa['letters'] and symbol != 'λ':
            nfa['letters'].append(symbol)
        for ns in state.next_state[symbol]:
            if ns not in symbol_table:
                symbol_table[ns] = sorted(symbol_table.values())[-1] + 1
                q_state = "Q" + str(symbol_table[ns])
                nfa['states'].append(q_state)
            nfa['transition_function'].append(["Q" + str(symbol_table[state]), symbol, "Q" + str(symbol_table[ns])])

        for ns in state.next_state[symbol]:
            arrange_transitions(ns, states_done, symbol_table)
```

- `write_nfa_output`: این تابع NFA تولید شده را به یک فایل متنی ذخیره می‌کند.

```
def write_nfa_output(nfa, filename):
    with open(filename, 'w') as file:
        file.write(' '.join(nfa['letters']) + '\n')
        file.write(' '.join(str(state) for state in nfa['states']) + '\n')
        file.write(' '.join(nfa['start_states']) + '\n')
        file.write(' '.join(str(state) for state in nfa['final_states']) + '\n')
        for n in nfa['transition_function']:
            qi = n[0]
            s = n[1]
            qo = n[2]
            file.write(f'{qi} {s.replace("λ", "lambda")} {qo} \n')
```

در بخش اصلی برنامه، عبارت منظم ورودی از یک فایل خوانده می‌شود، سپس عبارت منظم ورودی به یک عبارت منظم با فرم پولیش شده تبدیل می‌شود. سپس از روی این عبارت منظم پس از ساختن درخت بیان و محاسبه NFA، اتومات تولید شده به فایل خروجی نوشته می‌شود.

```
if __name__ == "__main__":
    regex_filename = 'RE_Input_3.txt'
    nfa_filename = 'NFA_Output_3.txt'

    with open(regex_filename, 'r') as file:
        lines = file.readlines()

    alphabet = lines[0].split()
    regex = lines[1].strip()
    reg = replace_characters(regex)

    pr = polish_regex(reg)
    et = make_exp_tree(pr)
    fa = compute_regex(et)
    arrange_nfa(fa)
    write_nfa_output(nfa, nfa_filename)
```