Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ОТЧЕТ ПО ПРЕДМЕТУ

Конструирование компиляторов

НА ТЕМУ

Распознавание цепочек регулярного языка - ЛР №1
(вариант № 4)

Преподаватель: Андрей Алексеевич Ступников
Студент: Петрович Милица
Группа: ИУ7-21М
Дата: 12.04.2025.

Постановка задачи

Напишите программу, которая в качестве входа принимает произвольное регулярное выражение, и выполняет следующие преобразования:

1) По регулярному выражению строит НКА.

2) По НКА строит эквивалентный ему ДКА.

3) По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний. Указание. Воспользоваться алгоритмом, приведенным по адресу http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бржозовского

4) Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

Текст программы

```
import re
from collections import defaultdict

# ------------------------------------ Regex to NFA ------------------------------------
-----
class State:
    _id_counter = 0  # статическая переменная

    def __init__(self, name=None):
        if name is None:
            self.name = f"q{State._id_counter}"
            State._id_counter += 1
        else:
            self.name = name

        self.transitions = defaultdict(set)
        self.is_final = False

    def add_transition(self, symbol, state):
        self.transitions[symbol].add(state)

    def __hash__(self):
        return hash(self.name)  # Используем имя для хеширования

    def __eq__(self, other):
        return self.name == other.name  # Сравниваем состояния по имени

    def __repr__(self):
        return f"State({self.name}, final={self.is_final})"


# Класс для НКА
class NFA:
    def __init__(self, start_state, final_state):
        self.start_state = start_state
        self.final_state = final_state
        self.final_state.is_final = True

    def add_transition(self, from_state, symbol, to_state):
        from_state.add_transition(symbol, to_state)
```

```python
# Раскрытие диапазонов в квадратных скобках
def expand_square_brackets(regex):
    def expand_char_class(char_class):
        result = []
        i = 0
        while i < len(char_class):
            if i + 2 < len(char_class) and char_class[i + 1] == '-':
                start = char_class[i]
                end = char_class[i + 2]
                result.extend([chr(c) for c in range(ord(start), ord(end) + 1)])
                i += 3
            else:
                result.append(char_class[i])
                i += 1
        return result

    pattern = re.compile(r'\[([^\]]+)\]')
    while match := pattern.search(regex):
        chars = match.group(1)
        expanded_chars = expand_char_class(chars)
        expanded = '(' + '|'.join(expanded_chars) + ')'
        regex = regex[:match.start()] + expanded + regex[match.end():]
    return regex


# Раскрытие {n}, {n,m}
def expand_braces(regex):
    pattern =
re.compile(r'(\([^)]+\)|\([^()]+\)|\[[^\]]+\]|[^(){}\[\]])\{(\d+)(?:,(\d+))?\}')
    while True:
        match = pattern.search(regex)
        if not match:
            break
        base = match.group(1)
        n = int(match.group(2))
        m = int(match.group(3)) if match.group(3) else None

        if base.startswith('(') and base.endswith(')'):
            base = base
        else:
            base = f'({base})'

        if m is None:
            replacement = base * n
        else:
            options = [''.join([base] * i) for i in range(n, m + 1)]
            replacement = '(' + '|'.join(options) + ')'

        regex = regex[:match.start()] + replacement + regex[match.end():]

    return regex
```

```
# Вставка явной конкатенации (.) между элементами
def insert_concatenation(regex):
    result = ""
    for i in range(len(regex) - 1):
        result += regex[i]
        if (regex[i].isalnum() or regex[i] in ')*+?}') and (
                regex[i + 1].isalnum() or regex[i + 1] == '(' or regex[i + 1] == '['):
            result += '.'
    result += regex[-1]
    return result


# Алгоритм сортировочной станции для преобразования в постфиксную запись
def shunting_yard(regex):
    precedence = {'*': 3, '+': 3, '?': 3, '.': 2, '|': 1}
    output = []
    operators = []
    for char in regex:
        if char.isalnum():
            output.append(char)
        elif char in precedence:
            while operators and operators[-1] != '(' and precedence[operators[-1]] >=
precedence[char]:
                output.append(operators.pop())
            operators.append(char)
        elif char == '(':
            operators.append(char)
        elif char == ')':
            while operators and operators[-1] != '(':
                output.append(operators.pop())
            operators.pop()
    while operators:
        output.append(operators.pop())
    return ''.join(output)
```

```
# Построение НКА из постфиксного выражения
def regex_to_nfa(regex):
    regex = expand_square_brackets(regex)
    regex = expand_braces(regex)
    regex = insert_concatenation(regex)
    regex = shunting_yard(regex)

    stack = []
    for char in regex:
        if char.isalnum():
            start = State()
            end = State()
            start.add_transition(char, end)
            stack.append(NFA(start, end))
        elif char == '*':
            nfa = stack.pop()
            start = State()
            end = State()
            start.add_transition('', nfa.start_state)
            start.add_transition('', end)
            nfa.final_state.add_transition('', nfa.start_state)
            nfa.final_state.add_transition('', end)
            stack.append(NFA(start, end))
        elif char == '+':
            nfa = stack.pop()
            start = State()
            end = State()
            start.add_transition('', nfa.start_state)
            nfa.final_state.add_transition('', nfa.start_state)
            nfa.final_state.add_transition('', end)
            stack.append(NFA(start, end))
        elif char == '?':
            nfa = stack.pop()
            start = State()
            end = State()
            start.add_transition('', nfa.start_state)
            start.add_transition('', end)
            nfa.final_state.add_transition('', end)
            stack.append(NFA(start, end))
        elif char == '|':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            start = State()
            end = State()
            start.add_transition('', nfa1.start_state)
            start.add_transition('', nfa2.start_state)
            nfa1.final_state.add_transition('', end)
            nfa2.final_state.add_transition('', end)
            stack.append(NFA(start, end))
        elif char == '.':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            nfa1.final_state.add_transition('', nfa2.start_state)
            stack.append(NFA(nfa1.start_state, nfa2.final_state))

    return stack.pop()
```

```python
# Удалить все флаги is_final кроме настоящего финального состояния
def clear_is_final(nfa):
    visited = set()
    stack = [nfa.start_state]
    while stack:
        state = stack.pop()
        if state in visited:
            continue
        visited.add(state)
        state.is_final = False
        for transitions in state.transitions.values():
            stack.extend(transitions)
    nfa.final_state.is_final = True


def epsilon_closure(states):
    stack = list(states)
    closure = set(states)
    while stack:
        state = stack.pop()
        epsilon_targets = state.transitions.get('', set())

        # Обработка только если это действительно множество (для NFA)
        if isinstance(epsilon_targets, set):
            for next_state in epsilon_targets:
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
    return closure


# Проверка строки на соответствие НКА
def match_nfa(nfa, string):
    current_states = epsilon_closure({nfa.start_state})
    for char in string:
        next_states = set()
        for state in current_states:
            for next_state in state.transitions.get(char, set()):
                next_states.update(epsilon_closure({next_state}))
        current_states = next_states
    return any(state.is_final for state in current_states)
```

```python
class DFAState:
    def __init__(self, arg, is_final=False):
        if isinstance(arg, (set, frozenset)):
            # Создание из множества состояний НКА
            self.nfa_states = frozenset(arg)
            self.name = "_".join(sorted(str(id(s)) for s in arg))  # уникальное имя по
id состояний
            self.is_final = any(s.is_final for s in arg)
        else:
            # Создание по имени
            self.name = arg
            self.nfa_states = None
            self.is_final = is_final

        self.transitions = {}

    def add_transition(self, symbol, state):
        self.transitions[symbol] = state

    def __repr__(self):
        return f"DFAState({self.name}, final={self.is_final})"


class DFA:
    def __init__(self, start_state):
        self.start_state = start_state
        self.states = []


def get_alphabet(automaton):
    alphabet = set()
    visited = set()
    stack = [automaton.start_state]

    while stack:
        state = stack.pop()
        if state in visited:
            continue
        visited.add(state)

        for symbol, next_states in state.transitions.items():
            if symbol != '':  # исключаем ε-переходы
                alphabet.add(symbol)

            # Поддержка и NFA, и DFA
            if isinstance(next_states, set):  # NFA: множество состояний
                for next_state in next_states:
                    stack.append(next_state)
            else:  # DFA: одно состояние
                stack.append(next_states)

    return alphabet
```

```
def nfa_to_dfa(nfa):
    from collections import deque

    def get_start_closure(start_state):
        # Безопасная проверка на поддержку ε-переходов
        if hasattr(start_state, 'transitions') and
isinstance(start_state.transitions.get('', set()), set):
            return epsilon_closure({start_state})
        else:
            return {start_state}

    start_set = get_start_closure(nfa.start_state)
    start_state = DFAState(start_set)
    dfa_states = {frozenset(start_set): start_state}
    queue = deque([start_set])
    alphabet = get_alphabet(nfa)

    while queue:
        current_set = queue.popleft()
        current_dfa_state = dfa_states[frozenset(current_set)]

        for symbol in alphabet:
            if symbol == '':
                continue  # ε не обрабатываем в ДКА

            next_set = set()
            for state in current_set:
                targets = state.transitions.get(symbol, set())
                next_set.update(targets)

            closure = epsilon_closure(next_set)
            closure_frozen = frozenset(closure)

            if closure_frozen not in dfa_states:
                dfa_states[closure_frozen] = DFAState(closure)
                queue.append(closure)

            current_dfa_state.transitions[symbol] = dfa_states[closure_frozen]

    dfa = type('DFA', (), {})()  # простой объект без класса
    dfa.start_state = start_state
    dfa.states = set(dfa_states.values())
    return dfa


def match_dfa(dfa, string):
    current_state = dfa.start_state
    for char in string:
        if char not in current_state.transitions:
            return False
        current_state = current_state.transitions[char]
    return current_state.is_final
```

```python
def reverse_dfa(dfa):
    all_states = set()
    reversed_transitions = defaultdict(set)
    final_states = []

    # Создаём все состояния и реверсируем переходы
    for state in dfa.states:
        all_states.add(state)
        for symbol, target in state.transitions.items():
            reversed_transitions[target].add((symbol, state))
        if state.is_final:
            final_states.append(state)

    # Создаём новое стартовое состояние, соединённое ε-переходами с финальными
    new_start = State("new_start")
    for fs in final_states:
        new_start.add_transition('', fs)  # ε-переходы


    # Обновляем переходы
    for state in all_states:
        new_trans = defaultdict(set)
        for dest, srcs in reversed_transitions.items():
            if dest == state:
                for symbol, src in srcs:
                    new_trans[symbol].add(src)
        state.transitions = new_trans

    all_states.add(new_start)

    # Сброс финальности у всех
    for state in all_states:
        state.is_final = False
    dfa.start_state.is_final = True  # только он финальный

    return NFA(new_start, dfa.start_state)


def brzozowski_minimization(dfa):
    reversed_nfa = reverse_dfa(dfa)
    reversed_dfa = nfa_to_dfa(reversed_nfa)

    reversed_nfa2 = reverse_dfa(reversed_dfa)
    minimized_dfa = nfa_to_dfa(reversed_nfa2)

    return minimized_dfa
```

Набор тестов

```
regexes = {
    '[a-z]{3}': ["abc", "xyz", "ab", "abcd"],
    '(a|b)+c': ["aac", "abc", "bbbc", "c", "abb"],
    '[0-9]{3,4}': ["12", "123", "1234", "1", "12345"],
    '[A-Ca-c]*': ["", "AaBbCc", "abcAC", "D", "abcx"],
    '[13579]?0': ["0", "10", "30", "11", "00", "21"],
    'a*': ['', 'a', 'aa', 'aaa', 'b'],
    'b+': ['', 'b', 'bb', 'bbb', 'bbj'],
    'c?': ['', 'c', 'cc', 'ccc', 'c1']
}
```

1) Для регулярного выражения **[a-z]{3}**
   - abc – ожидается True
   - xyz – ожидается True
   - ab – ожидается Flase
   - abcd – ожидается False

2) Для регулярного выражения **(a|b)+c**
   - aac – ожидается True
   - abc – ожидается True
   - bbbc – ожидается True
   - c – ожидается Flase
   - abb – ожидается False

3) Для регулярного выражения **[0-9]{3,4}**
   - 12 – ожидается Flase
   - 123 – ожидается True
   - 1234 – ожидается True
   - 1 – ожидается Flase
   - 12345 – ожидается False

4) Для регулярного выражения **[A-Ca-c]***
   - ‘’ – ожидается True
   - AaBbCc – ожидается True
   - abcAC – ожидается True
   - D – ожидается Flase
   - abcx – ожидается False

5) Для регулярного выражения **[A-Ca-c]***
   - ‘’ – ожидается True
   - AaBbCc – ожидается True
   - abcAC – ожидается True
   - D – ожидается Flase
   - abcx – ожидается False

6) Для регулярного выражения **[13579]?0**
   - 0 – ожидается True
   - 10 – ожидается True
   - 30 – ожидается True
   - 11 – ожидается Flase
   - 00 – ожидается False
   - 21 – ожидается False

7) Для регулярного выражения **a***
   - ‘’ – ожидается True
   - a – ожидается True
   - aa – ожидается True
   - aaa – ожидается True
   - b – ожидается False

8) Для регулярного выражения **b+**
   - ‘’ – ожидается False
   - b – ожидается True
   - bb – ожидается True
   - bbb – ожидается True
   - bbj – ожидается False

9) Для регулярного выражения **c?**
   - ‘’ – ожидается True
   - c – ожидается True
   - cc – ожидается False
   - ccc – ожидается False
   - c1 – ожидается False

Результаты выполнения программы

```
Регулярное выражение: '[a-z]{3}'
Строка 'abc' соответствует MIN автомату DFA: True
Строка 'xyz' соответствует MIN автомату DFA: True
Строка 'ab' соответствует MIN автомату DFA: False
Строка 'abcd' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: '(a|b)+c'
Строка 'aac' соответствует MIN автомату DFA: True
Строка 'abc' соответствует MIN автомату DFA: True
Строка 'bbbc' соответствует MIN автомату DFA: True
Строка 'c' соответствует MIN автомату DFA: False
Строка 'abb' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: '[0-9]{3,4}'
Строка '12' соответствует MIN автомату DFA: False
Строка '123' соответствует MIN автомату DFA: True
Строка '1234' соответствует MIN автомату DFA: True
Строка '1' соответствует MIN автомату DFA: False
Строка '12345' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: '[A-Ca-c]*'
Строка '' соответствует MIN автомату DFA: True
Строка 'AaBbCc' соответствует MIN автомату DFA: True
Строка 'abcAC' соответствует MIN автомату DFA: True
Строка 'D' соответствует MIN автомату DFA: False
Строка 'abcx' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------
```

```
Регулярное выражение: '[13579]?0'
Строка '0' соответствует MIN автомату DFA: True
Строка '10' соответствует MIN автомату DFA: True
Строка '30' соответствует MIN автомату DFA: True
Строка '11' соответствует MIN автомату DFA: False
Строка '00' соответствует MIN автомату DFA: False
Строка '21' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: 'a*'
Строка '' соответствует MIN автомату DFA: True
Строка 'a' соответствует MIN автомату DFA: True
Строка 'aa' соответствует MIN автомату DFA: True
Строка 'aaa' соответствует MIN автомату DFA: True
Строка 'b' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: 'b+'
Строка '' соответствует MIN автомату DFA: False
Строка 'b' соответствует MIN автомату DFA: True
Строка 'bb' соответствует MIN автомату DFA: True
Строка 'bbb' соответствует MIN автомату DFA: True
Строка 'bbj' соответствует MIN автомату DFA: False
--------------------------------------------------------------------------------

Регулярное выражение: 'c?'
Строка '' соответствует MIN автомату DFA: True
Строка 'c' соответствует MIN автомату DFA: True
Строка 'cc' соответствует MIN автомату DFA: False
Строка 'ccc' соответствует MIN автомату DFA: False
Строка 'c1' соответствует MIN автомату DFA: False
```

Анализ результатов и краткие выводи по работе

В рамках данной работы был разработан и реализован парсер регулярных выражений, способный преобразовывать регулярные выражения в недетерминированные конечные автоматы (НКА) и детерминированные конечные автоматы (ДКА), а также минимизировать полученные автоматы с помощью алгоритма Бржозовского.

Одним из основных этапов работы стало создание алгоритма, который преобразует регулярные выражения в НКА. В процессе были учтены все основные конструкции, такие как:

- Диапазоны символов (например, [a-z]),
- Повторения (например, {n,m}),
- Операции объединения (|), конкатенации и замыкания.

Реализованный алгоритм корректно обрабатывает широкий спектр регулярных выражений, включая более сложные конструкции с ограничениями на количество повторений символов.

На следующем этапе была осуществлена детерминизация НКА, что позволило улучшить производительность автоматов при проверке строк на соответствие. Этот процесс был реализован с использованием метода ε-замыкания, позволяющего эффективно переходить между состояниями при

наличии ε-переходов, и обеспечил корректную работу детерминированных автоматов для любых регулярных выражений.

Для минимизации ДКА был использован алгоритм Бржозовского, который включает два этапа:

1. Разворот автомата,
2. Детерминизация и повторный разворот.

Алгоритм позволил значительно сократить количество состояний в ДКА, что повлияло на эффективность проверки строк на соответствие регулярным выражениям. Минимизация уменьшила объем памяти, необходимый для хранения автоматов, и ускорила их работу.