Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 1

по курсу «Теория формальных языков»

Студент группы ИУ9-51Б Григорян Д. А.

Преподаватель Непейвода А. Н.

Москва 2025

1 Задание

По имеющейся SRS определить:

- завершимость

- конечность классов эквивалентности по НФ (для построения эквивалентностей считаем, что правила могут применяться в обе стороны). Если их конечное число, то построить минимальную систему переписывания, им соответствующую.

- локальную конфлюэнтность и пополняемость по Кнуту-Бендиксу

По SRS $\mathcal{T}$ строится другая SRS $\mathcal{T}'$, которая должна сохранять те же классы эквивалентности. Если исходная SRS завершима, то правила в $\mathcal{T}'$ должны удовлетворять условию убывания левой части относительно правой по выбранному вами фундированному порядку $>$.

Провести автоматическое тестирование предполагаемой эквивалентности двух указанных SRS.

Фазз-тестирование эквивалентности: строится случайное слово ω и случайная цепочка переписываний его в ω' по $\mathcal{T}'$. Проверить, можно ли получить ω' из ω (или наоборот) в рамках правил $\mathcal{T}'$.

Метаморфное тестирование: выбрать инварианты, которые должны сохраняться (либо монотонно изменяться) при переписывании в рамках $\mathcal{T}'$. Порождать случайную цепочку переписываний над случайным словом в $\mathcal{T}'$ и проверить выполнимость инвариантов. Как минимум два разных инварианта.

Вариант 9

1) aaaa $\rightarrow$ ab

2) abbb $\rightarrow$ bba

3) babb $\rightarrow$ bb

4) aabb $\rightarrow$ aaba

5) bbbaa $\rightarrow$ bb

6) aaabab $\rightarrow$ baabb

7) baabb $\rightarrow$ aabab

8) baabab $\rightarrow$ bab

9) bbabab $\rightarrow$ bb

10) baabaab $\rightarrow$ a

11) bab → baaaab

2 Проверка завершимости

Рассмотрим слово

$$w_1 = aaabab$$

$$aaa\underline{bab} \xrightarrow{11} aaabaaaab = w_2$$

$$aaab\underline{aaaa}b \xrightarrow{1} aaababb = w_3$$

$$aaa\underline{bab}b \xrightarrow{11} aaabaaaabb = w_4$$

$$aaab\underline{aaaa}bb \xrightarrow{1} aaababbb = w_5$$

$$aaa\underline{babb}b \xrightarrow{3} aaabbb = w_6$$

$$a\underline{aabb}b \xrightarrow{4} aaabab = w_1$$

Таким образом, получаем **циклическую последовательность** переписываний:

$$w_1 \to w_2 \to w_3 \to w_4 \to w_5 \to w_6 \to w_1 \to \dots$$

3 Локальная конфлюэнтность и пополняемость по Кнуту-Бендиксу

SRS $\mathcal{T}$ локально не конфлюэнтна, так как из слова ‘aaaa‘ можно получить 2 разных нормальных формы: ‘aba‘ и ‘aab‘.

Исследуем систему $\mathcal{T}$ на число классов эквивалентности.
Было сделано предположение, что в исходной системе всего 2 нормальные формы ‘a‘ и ‘bb‘. Программно, используя roads.py удалось проверить могут ли lhs перейти в одну из нормальных форм.

Предварительно получили, что:

1. aaaa → (aaaa → ab на позиции 0)
   ab → (a → baabaab на позиции 0)
   baabaabb → (bb → babb на позиции 6)
   baabaababb → (baabab → bab на позиции 3)
   baababb → (baabab → bab на позиции 0)
   babb → (babb → bb на позиции 0)
   Нормальная форма достигнута: bb

2. abbb → (abbb → bba на позиции 0)
   bba → (bb → bbbaa на позиции 0)
   bbbaaa → (bba → abbb на позиции 1)
   babbbaa → (babb → bb на позиции 0)
   bbbaa → (bbbaa → bb на позиции 0)
   Нормальная форма достигнута: bb

3. babb → (babb → bb на позиции 0)
   Нормальная форма достигнута: bb

4. aabb → (a → baabaab на позиции 0)
   baabaababb → (baabab → bab на позиции 3)
   baababb → (baabab → bab на позиции 0)
   babb → (babb → bb на позиции 0)
   Нормальная форма достигнута: bb

5. bbbaa → (bbbaa → bb на позиции 0)
   Нормальная форма достигнута: bb

6. aaabab → (aaabab → baabb на позиции 0)
   baabb → (bb → babb на позиции 3)

baababb → (baabab → bab на позиции 0)
babb → (babb → bb на позиции 0)
Нормальная форма достигнута: bb

7. baabb → (bb → babb на позиции 3)
   baababb → (baabab → bab на позиции 0)
   babb → (babb → bb на позиции 0)
   Нормальная форма достигнута: bb

8. baabab → (aabab → baabb на позиции 1)
   bbaabb → (ab → aaaa на позиции 3)
   bbaaaaab → (aaaa → ab на позиции 2)
   bbabab → (bbabab → bb на позиции 0)
   Нормальная форма достигнута: bb

9. bbabab → (bbabab → bb на позиции 0)
   Нормальная форма достигнута: bb

10. baabaab → (baabaab → a на позиции 0)
    Нормальная форма достигнута: a

11. bab → (bab → baaaab на позиции 0)
    baaaab → (aaaa → ab на позиции 1)
    babb → (babb → bb на позиции 0)

После было проверено, может ли ‘bb‘ перейти в ‘a‘, и оказалось, что может:
bb → (bb → babb на позиции 0)
babb → (bab → baaaab на позиции 0)
baaaabb → (ab → aaaa на позиции 4)
baaaaaaab → (aaaa → ab на позиции 2)
baabaab → (baabaab → a на позиции 0)
Нормальная форма достигнута: a

Таким образом, в исходной системе всего 1 класс эквивалентности - ‘a‘.

Код программы roads.py представлен в Листинге 1.

```
from collections import deque

rules = [
    ("aaaa", "ab"),
    ("abbb", "bba"),
    ("babb", "bb"),
    ("aabb", "aaba"),
    ("bbbaa", "bb"),
    ("aaabab", "baabb"),
    ("baabb", "aabab"),
    ("baabab", "bab"),
    ("bbabab", "bb"),
    ("baabaab", "a"),
    ("bab", "baaaab"),
]

def find_all_substring_indices(word, sub):
    indices = []
    i = 0
    while i <= len(word) - len(sub):
        if word[i:i+len(sub)] == sub:
            indices.append(i)
        i += 1
    return indices

def reduce_with_path(start_word):
    queue = deque()
    queue.append((start_word, []))
    seen = set()

    while queue:
        word, path = queue.popleft()
        if word in seen:
            continue
        seen.add(word)

        if word == "a": #or word =="bb":
            path.append((word, "norm", None))
            return path

        for a, b in rules:
            for idx in find_all_substring_indices(word, a):
                new_word = word[:idx] + b + word[idx+len(a):]
                queue.append((new_word, path + [(word, f"{a} → {b}", idx)]))
            for idx in find_all_substring_indices(word, b):
```

```
                    new_word = word[:idx] + a + word[idx+len(b):]
                    queue.append((new_word, path + [(word, f"{b} → {a}", idx)]))
    return None

word_to_test = "bb"
path = reduce_with_path(word_to_test)

if path:
    for step in path:
        word, action, idx = step
        if action == "norm":
            print(f"Нормальная формадостигнута : {word}")
        else:
            print(f"{word} → ({action} напозиции {idx})")
else:
    print(f"Слово '{word_to_test}' несводитсякнормальнойформе ")
```

Построим по алгоритму Кнута-Бендикса систему $\mathcal{T}'$, которая сохраняет те же классы эквивалентности. Рассмотрим систему

1) aaaa $\to$ ab

2) abbb $\to$ bba

3) babb $\to$ bb

4) aabb $\to$ aaba

5) bbbaa $\to$ bb

6) aaabab $\to$ baabb

7) baabb $\to$ aabab

8) baabab $\to$ bab

9) bbabab $\to$ bb

10) baabaab $\to$ a

11) baaaab $\to$ bab (было bab $\to$ baaaab)

Здесь слова сначала сравниваются по длине, если длина одинаковая, то они упорядочиваются лексикографически, где ‘a‘ < ‘b‘.

Упростим сначала правила 6 и 7 по принципу a $\to$ b, b $\to$ c, тогда a $\to$ c. Получим:

1) aaaa $\to$ ab

2) abbb $\to$ bba

3) babb $\to$ bb

4) aabb $\to$ aaba

5) bbbaa $\to$ bb

6) aaabab $\to$ aabab

7) baabab $\to$ bab

8) bbabab $\to$ bb

9) baabaab $\to$ a

10) baaaab $\to$ bab

Рассмотрим критические пары

1. Из ‘baaab‘ можно получить ‘bab‘ и ‘bb’, поэтому добавим правило ‘bab‘ $\to$ ‘bb‘. Можно убрать правило ‘baaaab‘ $\to$ ‘bab‘.

2. Из ‘babb‘ можно получить ‘bb‘ и ‘bbb‘, добавим правило ‘bbb‘ $\to$ ‘bb‘.

3. Из ‘aaabab‘ можно получить ‘aaaba‘ и ‘aaba‘, добавим правило ‘aaaba‘ $\to$ ‘aaba‘. Правило ‘aaabab‘ $\to$ ‘aabab‘ убираем.

4. Из ‘baabab‘ можно получить ‘bb‘ и ‘baaba‘, добавим правило ‘baaba‘ $\to$ ‘bb‘. Правило ‘baabab‘ $\to$ ‘bb‘ убираем.

5. Из ‘bbbaa‘ можно получить ‘aaba‘ и ‘bb‘, добавим правило ‘aaba‘ $\to$ ‘bb‘.

6. Из ‘abbb‘ можно получить ‘bba‘ и ‘abb‘, добавим правило ‘bba‘ $\to$ ‘abb‘. Правило ‘abbb‘ $\to$ ‘bba‘ убираем.

7. Из ‘aaaba‘ можно получить ‘bba‘ и ‘abb‘, добавим правило ‘bba‘ $\to$ ‘abb‘. Правило ‘abbb‘ $\to$ ‘bba‘ убираем.

8. Из ‘aaaaa‘ можно получить ‘aba‘ и ‘aab‘, добавим правило ‘aba‘ $\to$ ‘aab‘.

9. Из ‘baabaab‘ можно получить ‘a‘ и ‘bb‘, добавим правило ‘bb‘ $\to$ ‘a‘. Правило ‘baabaab‘ $\to$ ‘a‘ убираем.

10. Из ‘abb‘ можно получить ‘aa‘ и ‘a‘, добавим правило ‘a‘ $\to$ ‘a‘. Правило ‘abb‘ $\to$ ‘a‘ убираем.

11. Из ‘aaaa‘ можно получить ‘ab‘ и ‘a‘, добавим правило ‘ab‘ $\to$ ‘a‘. Правило ‘aaaa‘ $\to$ ‘ab‘ убираем.

12. Из ‘bab‘ можно получить ‘ba‘ и ‘a‘, добавим правило ‘ba‘ $\to$ ‘a‘. Правило ‘bab‘ $\to$ ‘a‘ убираем.

Критических пар больше нет, то есть система конфлюэнтна, также все действия сохраняли классы эквивалентности, то есть полученная система $\mathcal{T}'$ эквивалентна исходной и имеет вид после привидения к минимамльной:

aa $\to$ a

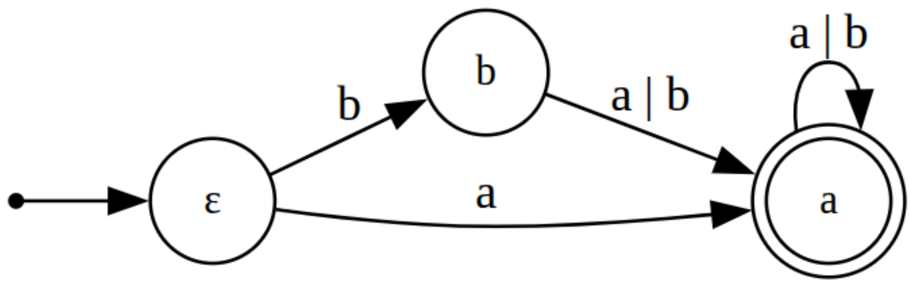
ab $\to$ a

ba $\to$ a

bb $\to$ a

4 Классы эквивалентности

Рассмотрим классы эквивалентности полученной системы $\mathcal{T}'$, так как они совпадают с классами эквивалентности исходной SRS, то $\mathcal{T}$ имеет такие же классы эквивалентности.



Таким образом в SRS $\mathcal{T}$ ровно 1 класс эквивалентности ‘a‘.

5 Фазз-тестирование эквивалентности

Код программы представлен в Листинге 2.

```
import random
import re
from collections import Counter

class RewriteSystem:
    def __init__(self, rules, alphabet=None, name="SRS"):
        self.rules = rules
        self.alphabet = alphabet or []
        self.name = name
        self._closure_cache = {}

    def apply_rules_once(self, word):
        results = set()
        for lhs, rhs in self.rules:
            for match in re.finditer(f"(?={re.escape(lhs)})", word):
                start = match.start()
                new_word = word[:start] + rhs + word[start+len(lhs):]
```

```
                results.add(new_word)
        return results

    def _bfs(self, start, target_set=None):
        if target_set is None and start in self._closure_cache:
            return self._closure_cache[start]

        if target_set is not None and start in target_set:
            return True
        seen = {start}
        frontier = [start]
        while frontier:
            new_frontier = []
            for w in frontier:
                for nw in self.apply_rules_once(w):
                    if nw not in seen:
                        if target_set is not None and nw in target_set:
                            return True
                        seen.add(nw)
                        new_frontier.append(nw)
            frontier = new_frontier

        if target_set is None:
            self._closure_cache[start] = seen
            return seen

        return False

    def closure(self, start):
        return self._bfs(start)

    def is_reach(self, start, target_set):
        return self._bfs(start, target_set)


class Experiment:
    def __init__(self, system_main, system_check, alphabet, word_length=17,
        steps=8):
        self.system_main = system_main
        self.system_check = system_check
        self.alphabet = alphabet
        self.word_length = word_length
        self.steps = steps
        self.rng = random.SystemRandom()

    def rand_word(self):
```

```
        return "".join(self.rng.choices(self.alphabet, k=self.word_length))

    def apply_rand_rules(self, word):
        for _ in range(self.steps):
            matches = [(i, lhs, rhs)
                       for lhs, rhs in self.system_main.rules
                       for i in range(len(word) - len(lhs) + 1)
                       if word[i:i+len(lhs)] == lhs]
            if not matches:
                break
            i, lhs, rhs = self.rng.choice(matches)
            word = word[:i] + rhs + word[i+len(lhs):]
        return word

    def equivalent_via_T1(self, w1, w2):
        cl1 = self.system_check.closure(w1)
        cl2 = self.system_check.closure(w2)
        return bool(cl1 & cl2)

    def generate_tests(self, n_tests=5):
        for _ in range(n_tests):
            word = self.rand_word()
            rewritten_word = self.apply_rand_rules(word)

            equivalent = self.equivalent_via_T1(word, rewritten_word)

            yield (word, rewritten_word, equivalent)

    def run_tests_summary(self, n_tests=5):
        counter = Counter()
        for word, rewritten_word, equivalent in self.generate_tests(n_tests):
            print("Исходное слово:", word)
            print("Переписанное слово:", rewritten_word)
            print(f"{self.system_check.name}: {equivalent}\n")
            counter["total"] += 1
            counter["success" if equivalent else "failure"] += 1
        return counter

T = [
    ("aaaa", "ab"),
    ("abbb", "bba"),
    ("babb", "bb"),
    ("aabb", "aaba"),
    ("bbbaa", "bb"),
    ("aaabab", "baabb"),
    ("baabb", "aabab"),
```

```
        ("baabab", "bab"),
        ("bbabab", "bb"),
        ("bab", "baaaab"),
        ("baabaab", "a"),
]

T1 = [
        ("aa", "a"),
        ("bb", "a"),
        ("ab", "a"),
        ("ba", "a"),
]

alphabet = ["a", "b"]

system_main = RewriteSystem(T, alphabet, name="T")
system_check = RewriteSystem(T1, alphabet, name="T1")

experiment = Experiment(system_main, system_check, alphabet, word_length=20,
    steps=8)

summary = experiment.run_tests_summary(n_tests=5)
print(summary)
```

6 Метаморфоное тестирование

Пусть w — слово, а $N(s, w)$ — число вхождений подстроки s в w

1. Определим величину

$$J(w) = \Big(N(b, w) + N(ab, w) + N(ba, w) + N(aba, w) + N(abb, w) + \\ + N(bba, w) + N(bbb, w)\Big) \bmod 2$$

Тогда:

$$J(w_{\text{start}}) = J(w_{\text{end}}).$$

Каждое правило из T меняет сумму вхождений подстрок $b, ab, ba, aba, abb, bba, bbb$ на чётное число.

2. Определим величину:

$$\begin{aligned}K(w) &= -2N(a,w) - N(b,w) + 2N(aa,w) + N(ab,w) + N(ba,w)\\ &\quad + N(aba,w) + N(abb,w) + N(bba,w) + N(bbb,w).\end{aligned}$$

Тогда:

$$K(w_{\text{start}}) = K(w_{\text{end}}).$$

$$\Big(K(b) = -1, \quad K(w) = -2 \ \text{для любого } w \neq b\Big)$$

Код программы представлен в Листинге 3.

```
import random
import re
from collections import Counter

class RewriteSystem:
    def __init__(self, rules, alphabet=None, name="SRS"):
        self.rules = rules
        self.alphabet = alphabet or []
        self.name = name

    def apply_rules_once(self, word):
        results = set()
        for lhs, rhs in self.rules:
            for match in re.finditer(f"(?={re.escape(lhs)})", word):
                start = match.start()
                new_word = word[:start] + rhs + word[start+len(lhs):]
                results.add(new_word)
        return results

    def apply_random_steps(self, word, steps=8):
        rng = random.SystemRandom()
        for _ in range(steps):
            matches = [(i, lhs, rhs)
                       for lhs, rhs in self.rules
                       for i in range(len(word)-len(lhs)+1)
                       if word[i:i+len(lhs)] == lhs]
            if not matches:
                break
            i, lhs, rhs = rng.choice(matches)
            word = word[:i] + rhs + word[i+len(lhs):]
        return word

class Experiment:
```

```python
    def __init__(self, system, alphabet, word_length=17, steps=8):
        self.system = system
        self.alphabet = alphabet
        self.word_length = word_length
        self.steps = steps
        self.rng = random.SystemRandom()

    def rand_word(self):
        return "".join(self.rng.choices(self.alphabet, k=self.word_length))

    def run_metamorphic_tests(self, n_tests=10, invariants=None):
        invariants = invariants or []
        counter = Counter()
        for _ in range(n_tests):
            word = self.rand_word()
            rewritten = self.system.apply_random_steps(word, self.steps)
            success = True
            for inv in invariants:
                if not inv(word, rewritten):
                    success = False
                    print("Не прошелинвариант :", inv.__name__)
            counter['total'] += 1
            counter['success' if success else 'failure'] += 1
        return counter


T = [
    ("aaaa", "ab"),
    ("abbb", "bba"),
    ("babb", "bb"),
    ("aabb", "aaba"),
    ("bbbaa", "bb"),
    ("aaabab", "baabb"),
    ("baabb", "aabab"),
    ("baabab", "bab"),
    ("bbabab", "bb"),
    ("bab", "baaaab"),
    ("baabaab", "a"),
]

alphabet = ["a", "b"]
system_main = RewriteSystem(T, alphabet, name="T")

def N(s, w):
    L = len(s)
    return sum(1 for i in range(len(w)-L+1) if w[i:i+L] == s)
```

```python
def J(w):
    return (N("b", w) + N("ab", w) + N("ba", w) + N("aba", w) + N("abb", w) +
        N("bba", w) + N("bbb", w)) % 2

def inv_J(start, end):
    return J(start) == J(end)

def K(w):
    return (-2*N("a", w) - N("b", w) + 2*N("aa", w) + N("ab", w) + N("ba", w)
            + N("aba", w) + N("abb", w) + N("bba", w) + N("bbb", w))

def inv_K(start, end):
    return K(start) == K(end)

experiment = Experiment(system_main, alphabet, word_length=12, steps=8)
summary = experiment.run_metamorphic_tests(
    n_tests=100000,
    invariants=[inv_J, inv_K],
    )
print(summary)
```