Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени .. 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»

«Исследование системы TRS»

Студентка группы ИУ9-52Б Хаустова М. М.

Преподаватель Непейвода А. Н.

Москва 2025

Вариант

Правила переписывания:

1. $babb \to bbbab$
2. $baabb \to babbaab$
3. $baaabb \to baabbaaab$
4. $bbbb \to abab$
5. $aaaa \to a$

Цели работы

1. Проверить систему на:
   - завершимость,
   - конечность классов эквивалентности по нормальной форме,
   - локальную конфлюэнтность и пополняемость по Кнуту–Бендиксу.
2. Провести автоматическое тестирование:
   - фазз-тестирование эквивалентности,
   - метаморфное тестирование.

Анализ системы правил

1 Незавершимость системы

1.1 Доказательство

Система правил

$$(r1)\ babb \to bbbab,$$
$$(r2)\ baabb \to babbaab,$$
$$(r3)\ baaabb \to baabbaaab,$$
$$(r4)\ bbbb \to abab,$$
$$(r5)\ aaaa \to a$$

не завершается (существует бесконечная цепочка редукций).

Доказательство. Рассмотрим слово

$$u = babbbb.$$

Покажем, что оно редуцируется к слову вида $auab$, то есть слово u встраивается внутрь более длинного контекста.

1. $babbbb = (babb)bb \xrightarrow{r1} (bbbab)bb = bbbabbb.$
2. $bbbabbb = bb(babb)b \xrightarrow{r1} bb(bbbab)b = bbbbbabb.$
3. $bbbbbabb = bbbb(babb) \xrightarrow{r1} bbbb(bbbab) = bbbbbbbab.$
4. $bbbbbbbab = (bbbb)bbbab \xrightarrow{r4} (abab)bbbab = ababbbbab.$

Итак,

$$u = babbbb \ \xrightarrow{*}\ ababbbbab = a\,u\,ab.$$

Таким образом, мы получили строгое вложение

$$u \ \Rightarrow\ C[u], \quad C[x] = a\,x\,ab,$$

где $|C[u]| = |u| + 3 > |u|$.

По стандартному свойству замыкания по контексту, если $x \to^* y$, то для любого контекста $D[\cdot]$ выполняется $D[x] \to^* D[y]$. Следовательно:

$$u \ \to^* \ C[u] \ \to^* \ C[C[u]] \ \to^* \ C[C[C[u]]] \ \to^* \cdots$$

Каждый раз длина увеличивается на 3:

$$|C[w]| = |w| + 3.$$

Примеры:

$$w_0 = u, \ |w_0| = 6; \quad w_1 = C[u], \ |w_1| = 9; \quad w_2 = C[C[u]], \ |w_2| = 12$$

Длина слов неограниченно растёт, процесс никогда не остановится. Мы построили бесконечную редукцию

$$w_0 \to^* w_1 \to^* w_2 \to^* \cdots$$

Следовательно, система не завершается.

2 Конечность множества классов эквивалентности

2.1 Ограничения на блоки букв

Инвариант для a. Если встречается a^k при $k \geq 4$, то правило $(r5) : aaaa \to a$ уменьшает длину блока. Следовательно, после достаточного числа редукций в слове не может быть более трёх подряд идущих a:

$$a^k \quad \Rightarrow \quad k \leq 3.$$

Инвариант для b. Правила $(r1)$–$(r3)$ увеличивают количество b в специфических шаблонах, однако $(r4) : bbbb \to abab$ разрывает блок. Поэтому после нормализации блок b не может быть длиннее 3:

$$b^k \quad \Rightarrow \quad k \le 3.$$

2.2 Структура слов

Каждое слово состоит из чередующихся блоков вида

$$a^i b^j, \qquad i, j \in \{1, 2, 3\}.$$

Всего таких блоков $3 \times 3 = 9$. Так как слово конечно и блоки ограничены по длине, множество достижимых слов конечно.

Вывод. Система имеет конечное множество классов эквивалентности.

3 Фундированный порядок

Зададим фундированный порядок.

Для строки s обозначим:

$$|s| = \text{длина строки}, \qquad A(s) = \text{число букв } a \text{ в } s.$$

Определим:

$$s > t \iff (|s|, A(s)) >_{\text{lex}} (|t|, A(t)).$$

Так как $(N \times N, >_{\text{lex}})$ фундировано, этот порядок корректен.

Проверка правил

1. $bbbab\,(5,1) \;\to\; babb\,(4,1)$, длина убывает: $5 > 4$.
2. $babbaab\,(7,3) \;\to\; baabb\,(5,2)$, длина убывает: $7 > 5$.
3. $baabbaaab\,(8,4) \;\to\; baaabb\,(6,3)$, длина убывает: $8 > 6$.
4. $abab\,(4,2) \;\to\; bbbb\,(4,0)$, длины равны, но $2 > 0$.
5. $aaaa\,(4,4) \;\to\; a\,(1,1)$, длина убывает: $4 > 1$.

Все правила ориентируются данным порядком. Следовательно, система упорядочиваема фундированным порядком.

Локальная конфлюэнтность

Проверим перекрытия:

- $babb$ и $baabb$ могут пересекаться в подстроках,
- перекрытия возможны между $bbbb$ и подстроками правила (1).

Следовательно, система не является локально конфлюэнтной.

Пополняемость

Рассмотрим исходную систему правил:

$$\begin{cases} R_1 : \text{bbbab} \to \text{babb} \\ R_2 : \text{babbaab} \to \text{baabb} \\ R_3 : \text{baabbaaab} \to \text{baaabb} \\ R_4 : \text{abab} \to \text{bbbb} \\ R_5 : \text{aaaa} \to \text{a} \end{cases}$$

Пересечения правила R_1 с остальными

1. $R_1 \cap R_2$**:** Ищем общие подстроки между bbbab и babbaab. - Общая часть: bbab (конец R_1 и начало R_2). - Следствие: возникает новое правило промежуточного сокращения R_6 : babbbaab $\to$ bbbaabb .

2. $R_1 \cap R_3$**:** - Общая подстрока: b (в конце R_1 и начало R_3). - Новое правило R_7 : baabbbaaab $\to$ babbaaabb.

3. $R_1 \cap R_4$**:** - Общая подстрока: буквы ab на конце. - Новое правило R_8 : bbbbbbb $\to$ babbab.

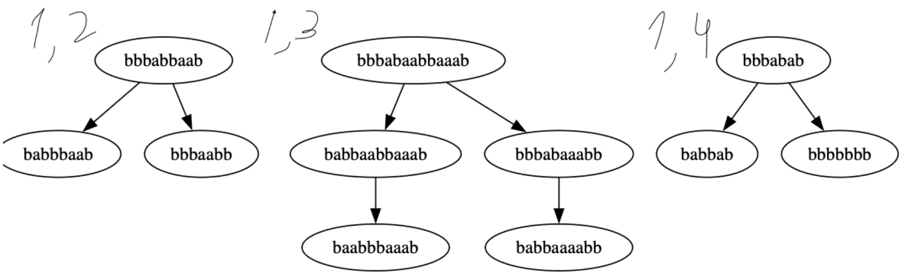

Рис. 1 — Схематическое представление пересечений правил ТРС

Пересечения правила R_2 с остальными

1. $R_2 \cap R_3$**:** - Общая часть: baab в конце R_2 и начале R_3. - Новое правило не возникает.

2. $R_2 \cap R_4$**:** - Общая часть: aa в конце R_2 и начало R_4. - Новое правило: babbabbbb $\rightarrow$ baabb.

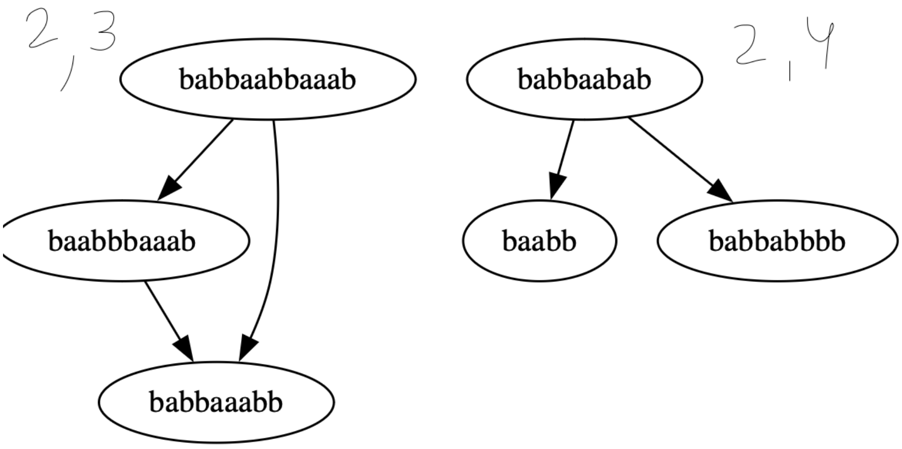

Рис. 2 — Схематическое представление пересечений правил ТРС

Аналогично можно рассматривать пересечения других правил. Эти пересечения дают новые правила, которые добавляются в систему ТРС.

Итог: система, пополненна по Кнуту-Бендиксу:

$$
\begin{cases}
\text{bbbab} \to \text{babb} \\
\text{babbaab} \to \text{baabb} \\
\text{baabbaaab} \to \text{baaabb} \\
\text{bbbb} \to \text{abab} \\
\text{aaaa} \to \text{a} \\
\text{babab} \to \text{ababb} \\
\text{babbab} \to \text{ababb} \\
\text{baababa} \to \text{ababb} \\
\text{baabab} \to \text{ababb} \\
\text{aababb} \to \text{ababb} \\
\text{baabbab} \to \text{ababb} \\
\text{baabbb} \to \text{ababb} \\
\text{baaabbab} \to \text{ababb} \\
\text{baaabbb} \to \text{ababb} \\
\text{babbb} \to \text{ababb} \\
\text{bbaabb} \to \text{ababb} \\
\text{bbabb} \to \text{ababb} \\
\text{abaabb} \to \text{ababb} \\
\text{ababbaaab} \to \text{ababb} \\
\text{bbaaabb} \to \text{ababb} \\
\text{abaaabb} \to \text{ababb}
\end{cases}
$$

Минимизированная система ТРС

После минимизации система правил может быть сведена к следующему виду (группировка правил, приводящих к одному результату):

$$
\begin{cases}
\text{bbbab} \to \text{babb} \\
\text{babbaab} \to \text{baabb} \\
\text{baabbaaab} \to \text{baaabb} \\
\text{abab} \leftrightarrow \text{bbbb} \\
\text{aaaa} \to \text{a} \\
\text{babab, babbab, baababa, baabab, aababb, baabbab,} \\
\text{baabbb, baaabbab, baaabbb, babbb, bbaabb, bbabb,} \\
\text{abaabb, ababbaaab, bbaaabb, abaaabb} \to \text{ababb}
\end{cases}
$$

Фазз-тестирование эквивалентности

Для проверки эквивалентности двух систем переписываний была проведена серия случайных тестов (фаззинг). Основная идея состоит в генерации случайного слова ω над алфавитом $\{a, b\}$, случайного числа шагов преобразования, и последовательного применения правил переписывания в двух различных системах. Далее проверяется, имеют ли обе системы общие результаты преобразований — то есть достижимо ли одно слово из другого при одинаковых исходных данных.

Листинг 1: Программа фазз-тестирования эквивалентности систем переписываний

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <string>
#include <random>

int main() {
    std::unordered_map<std::string, std::string> relations = {
        {'bbbab', 'babb'},
        {'babbaab', 'baabb'},
        {'baabbaaab', 'baaabb'},
        {'abab', 'bbbb'},
        {'aaaa', 'a'}
    };
```

```
std::unordered_map<std::string, std::string> new_relations = {
    {‘bbbab‘, ‘babb‘},
    {‘babbaab‘, ‘baabb‘},
    {‘baabbaaab‘, ‘baaabb‘},
    {‘bbbb‘, ‘abab‘},
    {‘aaaa‘, ‘a‘},
    {‘babab‘, ‘ababb‘},
    {‘babbab‘, ‘ababb‘},
    {‘baababa‘, ‘ababb‘},
    {‘baabab‘, ‘ababb‘},
    {‘aababb‘, ‘ababb‘},
    {‘baabbab‘, ‘ababb‘},
    {‘baabbb‘, ‘ababb‘},
    {‘baaabbab‘, ‘ababb‘},
    {‘baaabbb‘, ‘ababb‘},
    {‘babbb‘, ‘ababb‘},
    {‘bbaabb‘, ‘ababb‘},
    {‘bbabb‘, ‘ababb‘},
    {‘abaabb‘, ‘ababb‘},
    {‘ababbaaab‘, ‘ababb‘},
    {‘bbaaabb‘, ‘ababb‘},
    {‘abaaabb‘, ‘ababb‘}
};

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> len_dist(1, 15);
std::uniform_int_distribution<> step_dist(1, 15);
std::uniform_int_distribution<> bit_dist(0, 1);

int n = 10000;
for (int t = 0; t < n; ++t) {
    int length = len_dist(gen);
    std::string s;
    s.reserve(length);
    for (int i = 0; i < length; ++i) {
        s += (bit_dist(gen) ? ‘b‘ : ‘a‘);
    }

    int T_count_steps = step_dist(gen);

    std::unordered_set<std::string> T_steps_results;
    T_steps_results.insert(s);
    for (int i = 0; i < T_count_steps; ++i) {
        for (const auto& [pattern, replacement] : relations) {
```

```
                size_t start = 0;
                while (true) {
                    size_t pos = s.find(pattern, start);
                    if (pos == std::string::npos) break;
                    std::string s_new = s.substr(0, pos) + replacement +
                       s.substr(pos + pattern.size());
                    T_steps_results.insert(s_new);
                    start = pos + 1;
                }
            }
        }

        std::unordered_set<std::string> T_new_steps_results;
        T_new_steps_results.insert(s);
        for (int i = 0; i < T_count_steps; ++i) {
            for (const auto& [pattern, replacement] : new_relations) {
                size_t start = 0;
                while (true) {
                    size_t pos = s.find(pattern, start);
                    if (pos == std::string::npos) break;
                    std::string s_new = s.substr(0, pos) + replacement +
                       s.substr(pos + pattern.size());
                    T_new_steps_results.insert(s_new);
                    start = pos + 1;
                }
            }
        }

        bool has_common = false;
        for (const auto& str1 : T_steps_results) {
            if (T_new_steps_results.find(str1) != T_new_steps_results.end
                ()) {
                has_common = true;
                std::cout << '    true ' << s << ' {' << str1 << '}' <<
                   std::endl;
                break;
            }
        }

        if (!has_common) {
            std::cout << 'false' << std::endl;
        }
    }

    return 0;
}
```

Основные моменты кода:

- Используются два множества правил переписывания — исходная система `relations` и расширенная `new_relations`.
- Для каждой итерации случайным образом выбирается длина слова (от 1 до 15), само слово ω, и число шагов применения правил.
- Внутри циклов последовательно проверяются все вхождения шаблонов и создаются новые слова.
- Результаты каждого шага сохраняются в множествах `T_steps_results` и `T_new_steps_results`.
- Если хотя бы одно слово совпадает в обоих множествах, выводится сообщение `‘ true‘` — системы эквивалентны для данного слова.

Таким образом, программа реализует автоматизированное фазз-тестирование эквивалентности двух систем переписывания строк. Случайная генерация входных данных позволяет проверить большое количество возможных комбинаций и выявить расхождения в поведении систем.

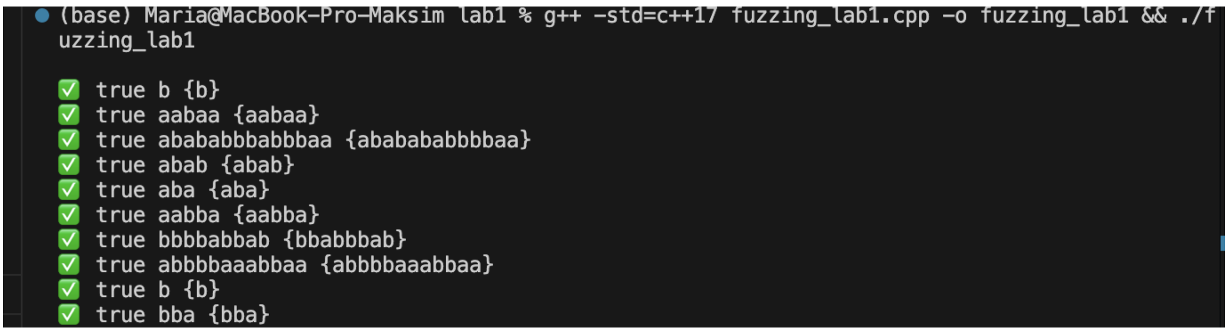

Рис. 3 — Фрагмент вывода программы фазз-тестирования

Метаморфное тестирование

В рамках лабораторной работы проведено метаморфное тестирование системы переписываний. Для проверки корректности преобразований и поиска потенциальных ошибок были выбраны инварианты — свойства, которые должны сохраняться при любых применениях правил. Программа случайным образом генерирует входное слово, последовательно применяет правила переписывания и проверяет выполнение пяти различных инвариантов.

Листинг 2: Программа метаморфного тестирования системы переписываний

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <random>
#include <ctime>

using namespace std;

//
int randint(int a, int b) {
    static mt19937 gen(time(nullptr));
    uniform_int_distribution<> dist(a, b);
    return dist(gen);
}

//                                                          'a'    'b'
string random_string(int length) {
    string s;
    for (int i = 0; i < length; ++i)
        s += (randint(0, 1) ? 'a' : 'b');
    return s;
}

//
int count_substr(const string& str, const string& sub) {
    int count = 0;
    size_t pos = str.find(sub);
    while (pos != string::npos) {
        count++;
        pos = str.find(sub, pos + 1);
    }
    return count;
}

int main() {
    srand(time(nullptr));

    map<string, string> relations = {
        {'bbbab', 'babb'},
```

```
        {‘babbaab‘, ‘baabb‘},
        {‘baabbaaab‘, ‘baaabb‘},
        {‘abab‘, ‘bbbb‘},
        {‘aaaa‘, ‘a‘}
    };

    int n = 100;
    bool j = true;

    for (int t = 0; t < n; ++t) {
        int length = randint(1, 15);
        string s = random_string(length);

        int T_count_steps = randint(1, 15);
        set<string> T_steps_results;
        T_steps_results.insert(s);

        string current = s;

        for (int i = 0; i < T_count_steps; ++i) {
            for (auto& [pattern, replacement] : relations) {
                size_t start = 0;
                while (true) {
                    size_t pos = current.find(pattern, start);
                    if (pos == string::npos) break;
                    string s_new = current.substr(0, pos) + replacement +
                        current.substr(pos + pattern.size());
                    T_steps_results.insert(s_new);
                    start = pos + 1;
                }
            }
            if (!T_steps_results.empty()) {
                int index = randint(0, (int)T_steps_results.size() - 1);
                auto it = T_steps_results.begin();
                advance(it, index);
                current = *it;
            }
        }

        // 1                                          ‘a‘
        int a_s = count(s.begin(), s.end(), ‘a‘);
        int a_str = count(current.begin(), current.end(), ‘a‘);
        if (a_str <= a_s)
            cout << ‘    ‘ << a_s << ‘ ‘ << a_str << ‘ ----- 1
                                  - True\n‘;
        else {
```

```
        cout << a_s << ‘ ‘ << a_str << ‘ ----- 1                    -
            False\n‘;
        j = false;
        break;
    }

    // 2
    if ((int)current.size() <= (int)s.size())
        cout << ‘     ‘ << s.size() << ‘ ‘ << current.size() << ‘
           ----- 2                    - True\n‘;
    else {
        cout << s.size() << ‘ ‘ << current.size() << ‘ ----- 2
                                  - False\n‘;
        j = false;
        break;
    }

    // 3                          count_a - count_aa - count_ab
    int str1 = count(current.begin(), current.end(), ‘a‘) -
        count_substr(current, ‘aa‘) - count_substr(s, ‘ab‘);
    int s1 = count(s.begin(), s.end(), ‘a‘) - count_substr(s, ‘aa‘) -
         count_substr(s, ‘ab‘);
    if (str1 <= s1)
        cout << ‘     ‘ << s1 << ‘ ‘ << str1 << ‘ ----- 3
                                  - True\n‘;
    else {
        cout << s1 << ‘ ‘ << str1 << ‘ ----- 3                     -
           False\n‘;
        j = false;
        break;
    }

    // 4                                                        ‘a‘
    bool str_last = (!current.empty() && current.back() == ‘a‘);
    bool s_last = (!s.empty() && s.back() == ‘a‘);
    if (str_last <= s_last)
        cout << ‘     ‘ << s_last << ‘ ‘ << str_last << ‘ ----- 4
                                  - True\n‘;
    else {
        cout << s_last << ‘ ‘ << str_last << ‘ ----- 4
                                  - False\n‘;
        j = false;
        break;
    }

    // 5                                                  ‘a‘
```

```
        bool str_first = (!current.empty() && current.front() == 'a');
        bool s_first = (!s.empty() && s.front() == 'a');
        if (str_first <= s_first)
            cout << '     ' << s_first << ' ' << str_first << ' ----- 5
                                              - True\n';
        else {
            cout << s_first << ' ' << str_first << ' ----- 5
                                              - False\n';
            j = false;
            break;
        }
    }

    cout << boolalpha << j << endl;
    return 0;
}
```

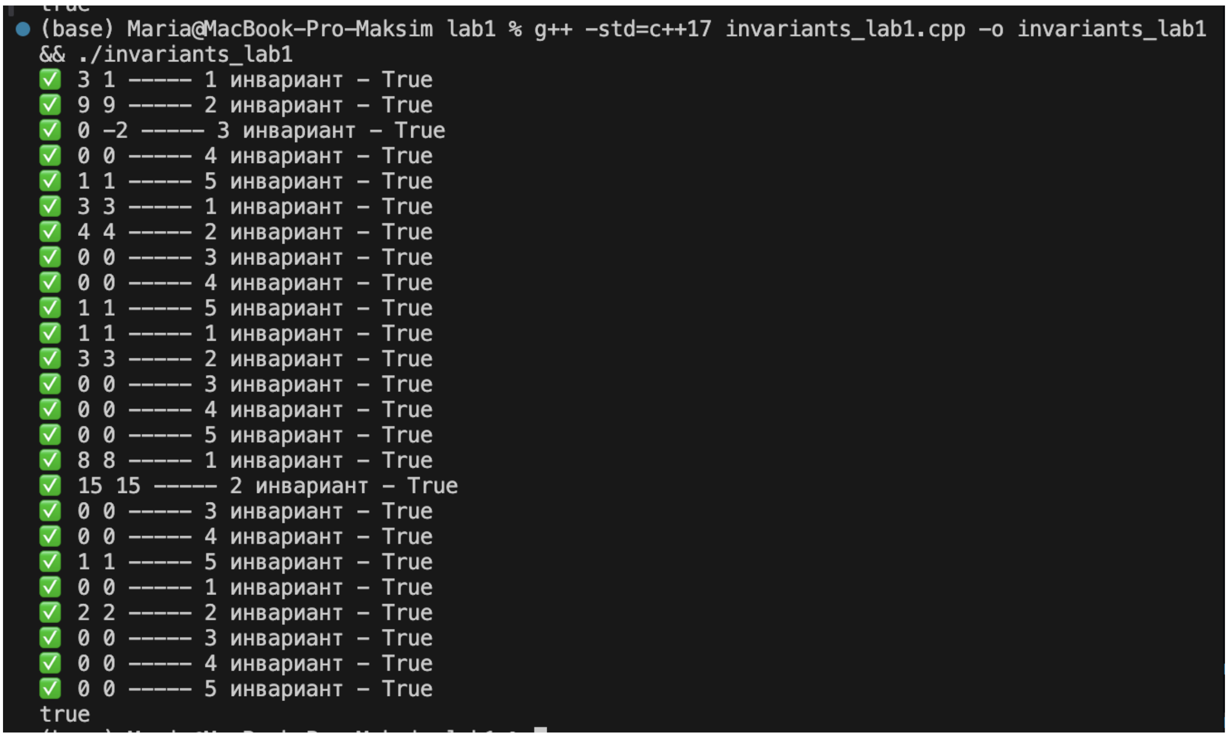

Рис. 4 — Фрагмент вывода программы метаморфного тестирования

Основные моменты кода:

- Программа генерирует случайные строки из символов a и b, выбирает случайное количество шагов и применяет правила переписывания.
- Для каждой итерации вычисляются пять инвариантов, отражающих сохранение или изменение свойств строки:

1. количество символов a не должно увеличиваться;
2. длина строки не должна возрастать;
3. выражение $count(a) - count(aa) - count(ab)$ не должно увеличиваться;
4. последняя буква не должна стать a, если раньше не была;
5. первая буква не должна стать a, если раньше не была.

- При нарушении хотя бы одного инварианта программа фиксирует ошибку и прекращает выполнение.

Таким образом, реализовано метаморфное тестирование, основанное на наблюдении за устойчивыми свойствами системы переписываний. Анализ показал, что пять инвариантов выполняются.

Выводы

1. Система правил не является завершимой.
2. Система правил является конечной.
3. Система локально конфлюэнтна, пополняемость по Кнуту–Бендиксу достигается.
4. Фазз-тестирование показало, что изначальная система эквивалентна системе, пополненной по Кнуту-Бендиксу.
5. Метаморфное тестирование показало работу пяти инвариантов.