Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу систем программирования.

«Реализация оператора вывода для СУ – перевода и для переводов, использующих гомоморфизм»

Студент: И. С. Глушатов

Преподаватель: А. С. Семенов

Группа: М8О-207Б-19

Дата:

Оценка:

Подпись:

Москва, 2021

Задание

1. Реализовать оператор вывода для переводов, использующих гомоморфизм
2. Написать структуру СУ – схемы. Реализовать оператор вывода для СУ – перевода.

Гомоморфизмы

Определение: гомоморфизмом называется отображение h: $\sum_1^* \to \sum_2^*$ такое что: $\sum_1$ - входной алфавит, $\sum_2$ - выходной алфавит, и для всех $x \in \sum_1^*$ и $y \in \sum_1^*$ выполняется: $h(x \cdot y) = h(x) \cdot h(y)$

Гомоморфизм можно задать таблицей, где каждому символу из входного алфавита будет соответствовать символ из выходного алфавита.

Алгоритм перевода:
Вход: строка $\omega = \omega_1\omega_2 \dots \omega_n \in \sum_1^*$, где n – длина цепочки; таблица гомоморфизма h: $\sum_1^* \to \sum_2^*$.
Выход: строка $\omega' \in \sum_2^*$ такая что: $h(\omega) = \omega'$

$\omega' := \emptyset$
foreach ($symbol \in \omega$):
    $\omega' := \omega' \cup h(symbol)$
end

Пример

Допустим алфавит $\sum = \{0,1\}$, и мы хотим разобрать цепочку $\omega = 00101$ с помощью гомоморфизма h: $\sum^* \to \sum^*$, заданного таблицей ниже.

a	h(a)
0	1
1	0

Тогда работа алгоритма будет представлять следующее:

Шаг 1. $\omega' := \{\}$
Шаг 2. symbol = 0, следовательно к ω' добавляем h(0) = 1. $\omega' = 1$.
Шаг 3. symbol = 0, следовательно к ω' добавляем h(0) = 1. $\omega' = 11$.
Шаг 4. symbol = 1, следовательно к ω' добавляем h(1) = 0. $\omega' = 110$.
Шаг 5. symbol = 0, следовательно к ω' добавляем h(0) = 1. $\omega' = 1101$.
Шаг 6. symbol = 1, следовательно к ω' добавляем h(1) = 0. $\omega' = 11010$.
Шаг 7. Входная строка считана, следовательно алгоритм закончен. ω' = 11010 – ответ.

Как можно заметить данный перевод осуществляет побитовое отрицание.

Реализация

Класс `myHTable` имеет один атрибут – таблицу, с помощью которой задаётся отображение. При инициализации подается списки входного и выходного алфавита, причем первый элемент первого списка отображается в первый элемент второго и т. д. При несовпадении размеров алфавитов выкидывается ошибка. Метод `debugHTable` выводит таблицу в консоль. Функция `h` осуществляет перевод цепочки, поданную как список символов из входного алфавита, и возвращает

соответствующую цепочку.

```
namespace Translator
{
    class myHTable
    {
        public List<List<Symbol>> table = new List<List<Symbol>>() { new List<Symbol>(),
                                                                     new List<Symbol>()};

        public myHTable(List<Symbol> InputSigma, List<Symbol> OutputSigma)
        {
            if (InputSigma.Count != OutputSigma.Count)
            {
                Console.WriteLine("Неправильно введенная таблица (размеры отличаются)");
                throw new RankException();
            }

            table = new List<List<Symbol>>() { InputSigma, OutputSigma };
        }

        public void debugHTable()
        {
            var a = table[0];
            var h_a = table[1];
            for (int i = 0; i < a.Count; i++)
            {
                Console.Write(a[i]);
                Console.Write(" --- ");
                Console.WriteLine(h_a[i]);
            }
        }

        public string h(List<Symbol> input)
        {
            string output = "";

            if (input.Count == 1 && input[0].symbol == "")
                return "";

            foreach (Symbol symbol in input)
            {
                var s = table[0];
                int ss = s.IndexOf(symbol);

                if (ss == -1)
                {
                    Console.WriteLine("Символ не из алфавита\n
                                                          Входная цепочка не была разобрана");
                    return "";
                }

                output += (table[1])[ss].symbol;
            }

            return output;
        }
    }
```

```
case "10":
    try
    {
        myHTable h_table = new myHTable(new ArrayList() { "0", "1" },
                                        new ArrayList() { "1", "0" });

        Console.WriteLine("\nDebug Homomorphism:");
        h_table.debugHTable();

        Console.WriteLine("\nEnter the line :");
        ArrayList r = new ArrayList(Console.ReadLine().Split(' '));
        Console.WriteLine(h_table.h(r));
    }
    catch (Exception e)
    {
        Console.WriteLine($"\nОшибка: {e.Message}");
    }
    break;
}
```

```
Номер задания: 10

Debug Homomorphism:
0 --- 1
1 --- 0

Enter the line :
1 0 0 1 1 0 0
0 1 1 0 0 1 1
```

Схемы синтаксически управляемого перевода

Определение: схемой синтаксически управляемого перевода называется пятерка $Tb = (V, \sum, \Delta, P, S)$, где:

V – конечное множество нетерминальных символов.
$\sum$ – конечный входной алфавит.
Δ – конечный выходной алфавит.
P – конечное множество правил вида: $A \rightarrow \alpha, \beta$, где $\alpha \in (V \cup \sum)^*, \beta \in (V \cup \Delta)^*$ и вхождения нетерминалов в цепочку β образуют перестановку вхождений нетерминалов в цепочку α.
S – начальный символ ($S \in V$).

Для исключения неоднозначности при вхождении одинаковых нетерминалов в цепочки α, β будут ставится целочисленные индексы.

Определение: переводом, определяемым схемой Tb, или τ(Tb) называют множество пар цепочек $\{(\omega, \omega') \mid (S, S) \Rightarrow^* (\omega, \omega'), \omega \in \sum^* \text{ и } \omega' \in \Delta^*\}$

Определение: входной грамматикой СУ – схемы Tb называется $G_i = \{V, \sum, P', S\}$, где $P' = \{A \rightarrow \alpha \mid A \rightarrow \alpha, \beta \in P\}$

Определение: выходной грамматикой СУ – схемы Tb называется $G_o = \{V, \Delta, P'', S\}$, где $P'' = \{A \rightarrow \beta \mid A \rightarrow \alpha, \beta \in P\}$

Определение: Выводом СУ – схемы называется построение пары соответствующих цепочек, а полученные пары называются выводимыми парами

Перед описанием алгоритма СУ – перевода нужно обозначить, что функция Derivation возвращает вывод поданной цепочки во входной грамматике G_i данной СУ – схемы. Данная функция может являться, как LL и LR – анализаторами, так и МП – автоматом. В моей реализации использовался метод рекурсивного спуска, как алгоритм нисходящего синтаксического анализа. Он самый простой, но очень неэффективный. Сложность этого алгоритма экспоненциальная и зависит от количества правил для цепочек, выводимых из одинакового нетерминала. Для предотвращения бесконечного цикла при левой рекурсии, используется поиск с итеративным углублением, ограниченный высотой 20 (другими словами, метод не сможет найти разбор цепочки, если для её вывода требуется больше 20 применений правил). Однако от такой условности можно избавиться, если ограничивать глубину динамически и брать за максимальную глубину произведение количества терминалов во входной строке на количество нетерминалов.

Алгоритм Derivation – определения правил для вывода цепочки:
Вход: цепочка $\alpha \in (V \cup \sum)^*$, цепочка $\beta \in \sum^*$, СУ – схема $Tb = (V, \sum, \Delta, P, S)$.
Выход: P' - множество правил из P, по которым из α выводится цепочка β.

P' := $\emptyset$
if ($\alpha \notin \sum^*$):
    foreach ($X \in \alpha \mid \alpha = [X_1 \dots X_{i-1} X X_{i+1} \dots X_{k_1}] \mid X \in V$):
        foreach ($A \to X_1 \dots X_n, Y_1 \dots Y_m \in P \mid A == X$):
            α' := $[X_1 \dots X_{i-1} X_1 \dots X_n X_{i+1} \dots X_{k_1}]$ // вставка $X_1 \dots X_n$ вместо X

            if ($\alpha' \in \sum^*$ and $\alpha' = \beta$):
                P' := P' $\cup \{A \to X_1 \dots X_n, Y_1 \dots Y_m\}$
                return P'
            end if

            if ($Derivation(\alpha', \beta, Tb) \neq \emptyset$):
                P' := P' $\cup \{A \to X_1 \dots X_n, Y_1 \dots Y_m\} \cup Derivation(\alpha', \beta, Tb)$
                return P'
            end if
        end
        break
    end
    return P'
else:
    return P'
end if

Алгоритм оператора вывода для СУ – схемы:
Вход: цепочка $\omega \in \sum^*$, СУ – схема $Tb = (V, \sum, \Delta, P, S)$.
Выход: цепочка $\omega' \in \Delta^*$ такая что: $\tau(\omega) = \omega'$

P' := Derivation(S, ω, Tb)
ω_{in} := [S]
ω_{out} := [S]
foreach ($A \to X_1 \dots X_n, Y_1 \dots Y_m \in$ P'):
    foreach ($X \in \omega_{in} \mid \omega_{in} = [X_1 \dots X_{i-1} X X_{i+1} \dots X_{k_1}], X \in V$):
        if ($X == A$):
            ω_{in} := $[X_1 \dots X_{i-1} X_1 \dots X_n X_{i+1} \dots X_{k_1}]$ // подставляем $X_1 \dots X_n$ вместо X
            foreach ($Y \in \omega_{out} \mid \omega_{out} = [Y_1 \dots Y_{i-1} Y Y_{i+1} \dots Y_{k_2}], Y \in V$):
                if ($X == Y$):
                    ω_{out} := $[Y_1 \dots Y_{i-1} Y_1 \dots Y_m Y_{i+1} \dots Y_{k_2}]$
                end if
                break
            end
            break
        end if
    end
end

На выходе получаем пару (ω_{in}, ω_{out}), где ω_{in} = ω, а ω_{out} – искомая цепочка ω'.

Пример

Пусть нам дана СУ – схема $Tb = (\{S, A\}, \{0, 1\}, \{a, b\}, P, S)$, где P состоит из правил:

1. $S \rightarrow 0AS, SAa$
2. $A \rightarrow 0SA, ASa$
3. $S \rightarrow 1, b$
4. $A \rightarrow 1, b$

И входная цепочка ω = 0010111, принадлежащая входной грамматике G_i, имеет разбор 1232343. Тогда работа алгоритма будет представлять следующее:

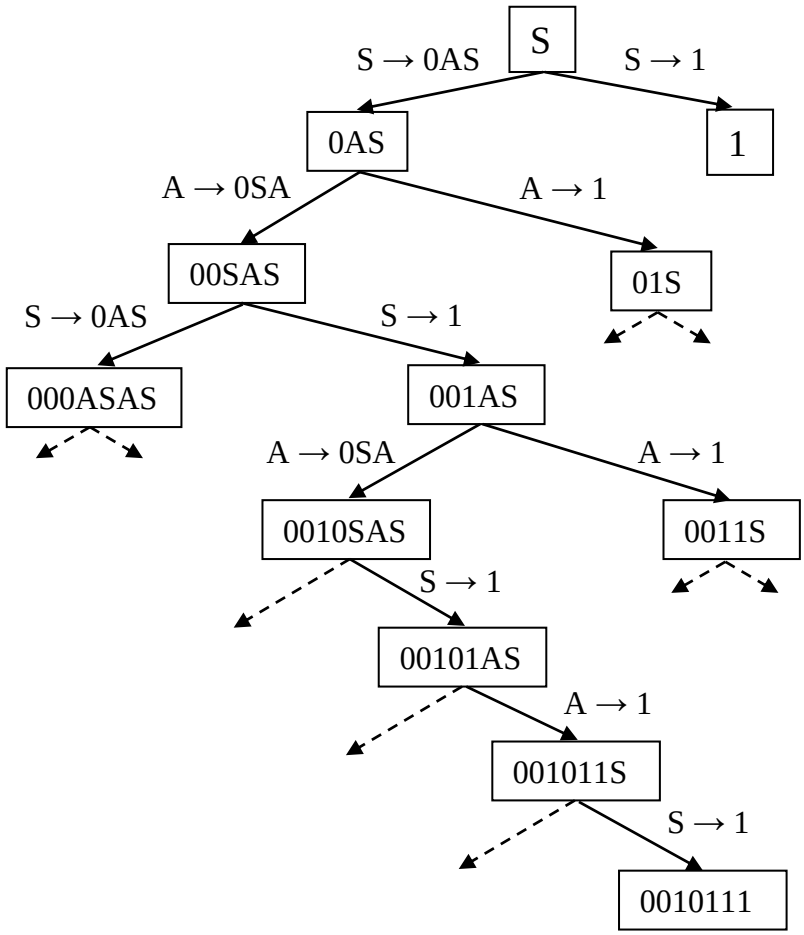

Рис. Поиск в дереве разбора цепочки 0010111 принадлежащей G(L).

Шаг 1. P' = {S → 0AS, SAa; A → 0SA, ASa; S →1, b; A → 0SA, ASa; S →1, b; A →1, b; S →1, b}. ω_{in} = S, ω_{out} = S.

Шаг 2. берем правило S → 0AS, SAa.	ω_{in} = 0AS,	ω_{out} = SAa
Шаг 3. берем правило A → 0SA, ASa.	ω_{in} = 00SAS,	ω_{out} = SASaa
Шаг 4. берем правило S →1, b.	ω_{in} = 001AS,	ω_{out} = SAbaa
Шаг 5. берем правило A → 0SA, ASa.	ω_{in} = 0010SAS,	ω_{out} = SASabaa
Шаг 6. берем правило S →1, b.	ω_{in} = 00101AS,	ω_{out} = SAbabaa
Шаг 7. берем правило A →1, b.	ω_{in} = 001011S,	ω_{out} = Sbbabaa
Шаг 8. берем правило S →1, b.	ω_{in} = 0010111,	ω_{out} = bbbabaa

В итоге получаем вывод (ω_{in}, ω_{out}), где ω_{in} = ω = 0010111, ω_{out} = ω' = bbbabaa – цепочка, соответствующая входной.

Реализация

По определению класс `mySDTSchemata` имеет атрибуты: список нетерминалов, входных и выходных терминалов, правил и начальный символ типа `string`. Правила представляют их себя списки, состоящие из нетерминала типа `string`, который стоит слева в правиле и двух списков – левой и правой цепочек, которые выводятся из данного нетерминала. Правила на вход подаются отдельно с помощью метода `addRule`.

Метод `TranslateByRules` принимает список правил типа `int` и строит вывод левой и правой цепочек по ним. Может упасть при вводе правил, которые не смогут примениться в связи с отсутствием нужного терминала. `Recursive_descent_parser` внутренний метод, осуществляющий разбор цепочки. Этот метод рекурсивный, и в общем случае принимает настоящую цепочку, цепочку, которую нужно разобрать, глубину допустимой рекурсии и массив с правилами разбора, куда будут правила записываться. Метод `Derivation` является оберткой над предыдущим методом и возвращает список правил в том порядке, а котором происходит вывод поданной цепочки по левым правилам СУ – схемы. В итоге `Translate` принимает цепочку, по которой функция строит перевод и делает вывод в консоль.

`CreateParseTree` возвращает дерево вывода поданной цепочки по левым правилам. Задание для следующего пункта.

```
namespace Translator
{

    class SDTProduction: Production {

        public List<Symbol> TranslateProd;
        public SDTProduction(Symbol S0, List<Symbol> RHSin, List<Symbol> RHSout): base(S0, RHSin)
        {
            TranslateProd = RHSout;
        }
    }

    class mySDTSchemata
    {
        public List<Symbol> V;
        public List<Symbol> Sigma;
        public List<Symbol> Delta;
        public List<SDTProduction> Productions;
        public Symbol S0;

        public mySDTSchemata(List<Symbol> NTS, List<Symbol> IPTS, List<Symbol> OPTS, Symbol FS)
        {
            V = NTS;
            Sigma = IPTS;
            Delta = OPTS;
            if (!V.Contains(FS))
                throw new Exception($"Начальный символ {FS} не принадлежит множеству нетермина-
лов");
            this.S0 = FS;
            Productions = new List<SDTProduction>();
        }

        public void addRule(Symbol NotTerminal, List<Symbol> Chain1, List<Symbol> Chain2)
        {
            if (!V.Contains(NotTerminal))
                throw new Exception($"В правиле {NotTerminal} --- ({Utility.convert(Chain1)},
{Utility.convert(Chain2)}) нетерминал {NotTerminal} не принадлежит множеству нетерминалов");

            foreach (Symbol symbol in Chain1)
            {
                if (!V.Contains(new Symbol(symbol.symbol.Split('_')[0])) && !Sig-
ma.Contains(symbol))
                    throw new Exception($"В правиле {NotTerminal} --- ({Utility.convert(Chain1)},
{Utility.convert(Chain2)}) символ {symbol} не принадлежит множеству нетерминалов или входному ал-
фавиту");
            }
```

```
            foreach (Symbol symbol in Chain2)
            {
                if (!V.Contains(new Symbol(symbol.symbol.Split('_')[0])) && !Del-
ta.Contains(symbol))
                    throw new Exception($"В правиле {NotTerminal} --- ({Utility.convert(Chain1)},
{Utility.convert(Chain2)}) символ {symbol} не принадлежит множеству нетерминалов или выходному
алфавиту");
            }

            Productions.Add(new SDTProduction(NotTerminal, Chain1, Chain2));
        }

        public void debugSDTS()
        {
            Console.WriteLine("\n");
            Console.WriteLine($"Нетерминальные символы: {Utility.convert(V)}");
            Console.WriteLine($"Входные символы: {Utility.convert(Sigma)}");
            Console.WriteLine($"Выходные символы: {Utility.convert(Delta)}");
            Console.WriteLine($"Начальный символ: {S0}");
            Console.WriteLine();

            for (int i = 0; i < Productions.Count; i++)
            {
                Console.WriteLine($"{i+1}. {Productions[i].LHS} --- ({Utili-
ty.convert(Productions[i].RHS)}, {Utility.convert(Productions[i].TranslateProd)})");
            }
        }

        public void TranslateByRules(List<int> RulesNumbers)
        {

            var Chain1 = new List<Symbol>() { S0 };
            var Chain2 = new List<Symbol>() { S0 };
            Console.Write("\n");
            Console.WriteLine($"Вывод в данной СУ-схеме по правилам: {Utili-
ty.convertInt(RulesNumbers)}");
            Console.Write($"({Utility.convert(Chain1)}, {Utility.convert(Chain2)})");

            foreach (var rule in RulesNumbers)
            {
                int real_rule = rule - 1;

                if (real_rule >= Productions.Count || real_rule < 0)
                    throw new Exception($"Несуществует правила с номером {real_rule}.");

                var NT = Productions[real_rule].LHS;
                var l = new List<Symbol>(Productions[real_rule].RHS);
                var r = new List<Symbol>(Productions[real_rule].TranslateProd);

                // поиск нужного нетерминала
                int index1 = -1;
                int index2 = -1;
                {
                    for (int i = 0; i < Chain1.Count; i++)
                    {
                        Symbol chain = Chain1[i];
                        if (chain.symbol == NT.symbol || chain.symbol.Split('_')[0] == NT.symbol)
                        {
                            index1 = i;
                            NT = chain;
                            break;
                        }
                    }

                    for (int i = 0; i < Chain2.Count; i++)
                    {
                        Symbol chain = Chain2[i];
                        if (chain.symbol == NT.symbol || chain.symbol.Split('_')[0] == NT.symbol)
                        {
                            index2 = i;
```

```
                            break;
                        }
                    }
                }

                Chain1.RemoveAt(index1);

                var ll = l;
                var rr = r;
                for (int i = 0; i < ll.Count; i++)
                {
                    var symbol = ll[i];
                    if (!Sigma.Contains(symbol)) { // если терминал
                        int max = 0;
                        foreach (var ss in Chain1)
                        {
                            if (ss.symbol.Split('_')[0] == symbol.symbol.Split('_')[0])
                                if (!V.Contains(ss))
                                    if (max < Convert.ToInt32(ss.symbol.Split('_')[1]))
                                        max = Convert.ToInt32(ss.symbol.Split('_')[1]);
                        }
                        max++;
                        Chain1.Insert(index1, new Symbol(symbol.symbol.Split('_')[0] + "_" +
max.ToString()));
                        rr[rr.IndexOf(symbol)] = new Symbol(symbol.symbol.Split('_')[0] + "_" +
max.ToString());

                    } else {
                        Chain1.Insert(index1, symbol);
                    }
                    index1++;
                }

                Chain2.RemoveAt(index2);
                Chain2.InsertRange(index2, rr);

                Console.Write($" => {rule}\n({Utility.convert(Chain1)},
                                                                {Utility.convert(Chain2)})");
            }

            Console.WriteLine();
        }

        private bool Recursive_descent_parser(List<Symbol> current_chain,
                                  List<Symbol> real_chain, int depth, ref List<int> answer)
        {

            if (depth == 0)
            {
                if (Utility.IsSameArrayList(current_chain, real_chain))
                    return true;

                return false;
            }


            if (Utility.IsSameArrayList(current_chain, real_chain))
            {
                return true;
            }
            else // иначе ищем нетерминал, к которому можем применить правило
            {
                var NT = new Symbol("");
                foreach (var symbol in current_chain)
                {
                    if (V.Contains(new Symbol(symbol.symbol.Split('_')[0])))
                    {
                        NT = new Symbol(symbol.symbol.Split('_')[0]);
                        break;
                    }
```

```
            }

            for (int i = 0; i < Productions.Count; i++)
            {
                var rule = Productions[i];
                if (rule.LHS.symbol == NT.symbol)
                {
                    var next_chain = new List<Symbol>(current_chain);
                    int index = next_chain.IndexOf(NT);
                    next_chain.RemoveAt(index);

                    var r = new List<Symbol>(rule.RHS);
                    for (int j = 0; j < r.Count; j++)
                    {
                        r[j] = new Symbol(r[j].symbol.Split('_')[0]);
                    }
                    next_chain.InsertRange(index, r);

                    if (Recursive_descent_parser(next_chain, real_chain, depth-1,
                                                                        ref answer))
                    {
                        answer.Add(i+1);
                        return true;
                    }
                }
            }

            return false;
        }

    }

    public List<int> Derivation(List<Symbol> left_chain)
    {
        var s = new List<int>();

        for (int i = 2; i < 8; i++)
        {
            bool flag = Recursive_descent_parser(new List<Symbol>() { S0 }, left_chain, i,
ref s);
            if (flag)
            {
                break;
            }
            else
            {
                s.Clear();
            }
        }

        s.Reverse();
        return s;
    }

    public void Translate(List<Symbol> left_chain)
    {
        Console.WriteLine();
        Console.Write("Входная строка: ");
        Console.WriteLine(Utility.convert(left_chain));

        var s = Derivation(left_chain);

        Console.Write("Её вывод во входной грамматике: ");
        foreach (int rule in s)
        {
            Console.Write(rule);
            Console.Write(" ");
        }
        Console.WriteLine();
```

```
            TranslateByRules(s);
        }
    }
}
```

```
case "11":
    try
    {
        mySDTSchemata sdt = new mySDTSchemata(new ArrayList() { "S", "A" },
                                              new ArrayList() { "0", "1" },
                                              new ArrayList() { "a", "b" },
                                              "S");

        sdt.addRule("S", new ArrayList() { "0", "A", "S" }, new ArrayList() { "S", "A", "a" });
        sdt.addRule("A", new ArrayList() { "0", "S", "A" }, new ArrayList() { "A", "S", "a" });
        sdt.addRule("S", new ArrayList() { "1" }, new ArrayList() { "b" });
        sdt.addRule("A", new ArrayList() { "1" }, new ArrayList() { "b" });

        Console.Write("\nDebug SDTranslator:");
        sdt.debugSDTS();

        sdt.TranslateByRules(new ArrayList() { 1, 2, 3, 2, 3, 4, 1, 4, 3 });
        sdt.Translate(new ArrayList() { "0", "0", "1", "0", "1", "1", "1"  });

        /*Tree tr = sdt.CreateParseTree(new ArrayList() { "0", "0", "1", "0", "1", "1", "1" });
        tr.Print();
        tr.PrintCrown();
        tr.Transform();
        tr.Print();
        tr.PrintCrown();*/

    }
    catch (Exception e)
    {
        Console.WriteLine($"\nОшибка: {e.Message}");
    }
    break;
```

```
Номер задания: 11

Debug SDTranslator:

Нетерминальные символы: S A
Входные символы: 0 1
Выходные символы: a b
Начальный символ: S

1. S --- (0 A S, S A a)
2. A --- (0 S A, A S a)
3. S --- (1, b)
4. A --- (1, b)

Вывод в данной СУ-схеме по правилам: 1 2 3 2 3 4 1 4 3
(S, S) => 1
(0 A_1 S_1, S_1 A_1 a) => 2
(0 0 S_2 A_1 S_1, S_1 A_1 S_2 a a) => 3
(0 0 1 A_1 S_1, S_1 A_1 b a a) => 2
(0 0 1 0 S_2 A_1 S_1, S_1 A_1 S_2 a b a a) => 3
(0 0 1 0 1 A_1 S_1, S_1 A_1 b a b a a) => 4
(0 0 1 0 1 1 S_1, S_1 b b a b a a) => 1
(0 0 1 0 1 1 0 A_1 S_1, S_1 A_1 a b b a b a a) => 4
(0 0 1 0 1 1 0 1 S_1, S_1 b a b b a b a a) => 3
(0 0 1 0 1 1 0 1 1, b b a b b a b a a)

Входная строка: 0 0 1 0 1 1 1
Её вывод во входной грамматике: 1 2 3 2 3 4 3

Вывод в данной СУ-схеме по правилам: 1 2 3 2 3 4 3
(S, S) => 1
(0 A_1 S_1, S_1 A_1 a) => 2
(0 0 S_2 A_1 S_1, S_1 A_1 S_2 a a) => 3
(0 0 1 A_1 S_1, S_1 A_1 b a a) => 2
(0 0 1 0 S_2 A_1 S_1, S_1 A_1 S_2 a b a a) => 3
(0 0 1 0 1 A_1 S_1, S_1 A_1 b a b a a) => 4
(0 0 1 0 1 1 S_1, S_1 b b a b a a) => 3
(0 0 1 0 1 1 1, b b b a b a a)
```

Вывод

Перевод с помощью гомоморфизма является самым простым методом перевода, который может использоваться, к примеру, при кодировании информации или при расчетах простых хэш-функций и т. п. Однако примитивность метода, не позволяет работать с более сложными переводами, к примеру, из инфиксной записи выражения в постфиксную и т. д.

СУ – схемы позволяют задавать сложные переводы и являются базовыми для теории переводов, на которой базируются интегрированные схемы компиляции. Представляя выражение в виде дерева вывода цепочки, принадлежащей входной грамматике, можно достаточно простым алгоритмом преобразовывать в дерево вывода соответствующей цепочки, принадлежащей выходной грамматике. Проблема перевода же возникает в процессе разбора входных цепочек, так как не всегда цепочки могут разбираться детерминированными автоматами или принадлежать определенной грамматике, к примеру для LL или LR анализаторов. Поэтому я использовал неэффективный метод рекурсивного спуска, который позволяет разбирать большое количество видов КС – грамматик.

Ссылка на GitHub с библиотечными файлами с полной реализацией разобранных структур и алгоритмов: https://github.com/Igor743646/c_sharp

Литература

1. А. С. Семенов. Порождающие и распознающие системы формальных языков с примерами на языке программирования C#. Москва 2021.
2. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. Издательство “МИР”. Москва 1978.
3. А. Ахо, М. Лам, Р. Сети, Дж. Ульман. Компиляторы. Принципы, технологии и инструментарий. Второе издание. Москва · Санкт-Петербург · Киев 2008.
4. Э. А. Опалева, В. П. Самойленко. Языки программирования и методы трансляции. Издательство “БХВ - Петербург” 2005.
5. Определения дерева разбора и кроны дерева разбора. [Электронный ресурс], URL: https://neerc.ifmo.ru/wiki/index.php?title=Контекстно-свободные_грамматики,_вывод,_лево-_и правосторонний_вывод,_дерево_разбора
6. Понятие гомоморфизма в теории формальных языков. [Электронный ресурс], URL: https://www.mccme.ru/free-books/pentus/pentus.pdf