Лекции 3 – 5 Формализация понятия алгоритма

1. ***Неформальное (интуитивное) определение алгоритма***.

1.1. ***Определение***. Под *алгоритмом* в математике понимают точное предписание, задающее вычислительный процесс, ведущий от начальных данных, которые могут варьироваться, к искомому результату. Синоним алгоритма – вычислительная (эффективная) процедура, которая после какого-либо числа шагов (вычислений) приводит к решению поставленной задачи. При этом в интуитивном определении алгоритма слова "вычисления", "вычислительный процесс" понимаются в широком смысле как любой процесс обработки информации: вычисление некоторой величины, поиск решения некоторой (математической) задачи, четкое и ясное предписание по обработке информации.

1.2. ***Замечание***. В рамках данного курса понятие информации считается интуитивно ясным и не дается определения этого понятия. При этом необходимо уточнить, что, имея в виду обработку информации на компьютере, рассматривается только такая информация, которую можно задать в дискретном виде, т.е. конечным числом отдельных знаков (символов). В рамках такого ограничения, предполагается, что в дискретном виде можно задать любую информацию. Более точно: любую информацию в задачах обработки информации можно "подменить" дискретной информацией. Например, цветную картину можно изобразить в виде описания цветов и фактуры пусть большого, но конечного числа отдельных точек (растровое изображение). Следует отметить, что предположение о возможности дискретного представления информации подкреплено тысячелетиями проверенной практикой использования человечеством письменности.

1.3. ***Основные свойства интуитивно определенного алгоритма***.

(1) ***Конечность*** (результативность). *Алгоритм должен заканчиваться за конечное число шагов* и получать результат, обеспечивая решение тех задач, для которых он и создавался. Хотя число шагов не ограничено сверху.

(2) ***Определенность*** (детерминированность). Каждый шаг алгоритма и переход от шага к шагу должны быть точно определены и каждое применение алгоритма к одним и тем же исходным данным должно приводить к одинаковому результату.

(3) ***Простота и понятность***. Каждый шаг алгоритма должен быть четко и ясно определен, чтобы выполнение алгоритма можно было "поручить" любому исполнителю (человеку или механическому устройству).

(4) ***Массовость***. Алгоритм задает процесс вычисления для множества исходных данных (чисел, строк букв и т.п.), он представляет общий метод решения класса задач. Не имеет смысла строить, например, алгоритм нахождения НОД только для чисел 10 и 15.

1.4. Основной недостаток интуитивного определения – не определены базовые понятия: *конечность* (допустимы ли ситуации бесконечного зацикливания), *исходные данные* (какие исходные данные допустимы, как они представляются), *результат*, *исполнитель* (что значит можно поручить любому исполнителю), *класс задач* и др.

2. ***Пример: Алгоритм Евклида*** нахождения наибольшего общего делителя двух целых положительных чисел НОД(*a*, *b*) (в геометрической форме это алгоритм нахождения общей меры двух отрезков).

Даны два целых числа *a* и *b*. Алгоритм Евклида состоит в выполнении следующих шагов:

(1) Разделить нацело *a* на *b*; получить остаток *r*.

(2) Если $r = 0$, то НОД(a, b) = b

(3) Если $r \neq 0$, заменить: a на b, b на r и возвратиться к шагу (1).

3. ***Почему необходимо формальное определение алгоритма***. После того, как в 30-е годы прошлого века была доказана алгоритмическая неразрешимость некоторых задач в разных разделах математики, возникла совершенно новая ситуация. До тех пор, пока ученые верили в возможность построения алгоритмов для всех поставленных задач, не было повода уточнять интуитивное понятие алгоритма. Доказательство существования алгоритмически неразрешимых проблем означало, что существуют классы задач, для решения которых алгоритм не просто не найден, а его не будет найдено никогда. В отличие от конкретных алгоритмов, доказательство алгоритмической неразрешимости, т.е. доказательство невозможности, в котором содержалось бы высказывание *обо всех мыслимых алгоритмах*, потребовало формального уточнения понятия «алгоритм». Не имея формального определения, невозможно говорить обо всех мыслимых алгоритмах.
4. Для формального определения алгоритма требуется предварительно рассмотреть некоторые общие понятия, часто используемые в математике.
   4.1. Множество. Элемент множества. Примеры множеств.
   4.2. Мощность множества. Понятие подмножества. Операции над множествами.
   4.3. Декартово произведение множеств. Отображение множеств.
   4.4. Частично определенная функция.
5. ***Алфавиты и отображения***.
   5.1. *Алфавит* – конечное множество A_p элементов a_i: $A_p = \{a_1, a_2, \ldots, a_p\}$. Элементы алфавита A_p называются *символами*. Последовательность m символов алфавита A_p называется *словом* длины m над алфавитом A_p. Слово длины 0 называется пустым словом и обозначается ε.
   5.2. Слово длины 2 над алфавитом A_p можно рассматривать как символ декартова произведения алфавита A_p на себя: если $a_i \in A_p$ и $a_j \in A_p$, то $a_i a_j \in A_p \times A_p = A_p^2$. Следовательно, если определить $\{\varepsilon\} = A_p^0$, то множество A_p^* всех слов над алфавитом A_n можно представить в виде объединения множеств:

$$A_p^* = \{\varepsilon\} \cup A_p \cup A_p^2 \cup \ldots \cup A_p^m \cup \ldots = \bigcup_{m=0}^{\infty} A_p^m$$

   В объединении $\bigcup_{m=0}^{\infty} A_p^m$ – бесконечно много членов, но бесконечность здесь не *актуальная* (как в математическом анализе), а *потенциальная* в том смысле, что допустимы слова произвольно большой длины. Длину слова $w \in A_p^*$ будем обозначать $|w|$. Итак, A_p^* является счетным множеством (в смысле потенциальной бесконечности).
   5.3. Если в алфавит A включить не только буквы (например, латинские и русские, строчные и прописные), но и цифры, знаки препинания, такие символы как «конец строки», «конец страницы», «знак абзаца», «знак табуляции» и т.п., то любой текст (страницу лю-

бой книги или даже всю книгу)[1] можно представить в виде слова над алфавитом A (или в виде символа алфавита A^*).

5.4. *Кодирование*: представление символов алфавита A словами над алфавитом B.

Утверждение. Для любой пары алфавитов A и B существует простой алгоритм, определяющий взаимно-однозначное отображение $A^* \leftrightarrow B^*$.

В самом деле, пусть для определенности $|A| > |B|$. Будем представлять символы алфавита A словами над алфавитом B (первые $|B|$ символов – однобуквенными словами, следующие $|B|$ символов – двухбуквенными и т.д.). При этом может понадобиться дополнительный символ ι («конец кода символа»). Обратное отображение очевидно.

Из утверждения следует, что любой алфавит, в том числе и двухсимвольный алфавит A_2 (символы 0 и 1) и даже односимвольный алфавит A_1 (обычно в качестве единственного символа алфавита A_1 используется | («палочка»)), пригоден для представления любого текста.

Отметим, что кодирование позволяет ограничиться одним алфавитом. В теории алгоритмов в качестве такого алфавита обычно рассматриваются A_1 или A_2.

6. ***Задача обработки информации***.

6.1. Задача обработки информации – это задача построения частичного отображения (функции) $F : A^* \to A^*$.

Любой алгоритм можно рассматривать как набор правил, позволяющих реализовать требуемое частичное отображение на множестве слов конечного алфавита A.

6.2. *Нумерация*. Каждому элементу (слову) множества A^* можно присвоить его номер – неотрицательное целое число.

Утверждение. Существует взаимно-однозначное отображение (функция) $\#: A^* \to N$, где N – множество целых неотрицательных чисел, которое любому слову $w \in A^*$ ставит в соответствие его номер $n \in N$.

(1) Построение отображения. Пустому слову ε присваивается номер 0 (#(ε) = 0). Каждому односимвольному слову $w = a_i \in A_p^*$ присваивается номер i (i = 1, 2, … p). Произвольному слову $w = a_{i_0} a_{i_1} \ldots a_{i_m}$ длины m присваивается номер

$$\#(w) = i_0 + p \cdot i_1 + p^2 \cdot i_2 + \ldots + p^m \cdot i_m = \sum_{s=0}^{m} p^s \cdot i_s$$ (i_s – номер односимвольного слова $a_{i_s} \in A_p^*$).

(2) Обратное отображение $\#^{-1}$ строится очевидным образом.

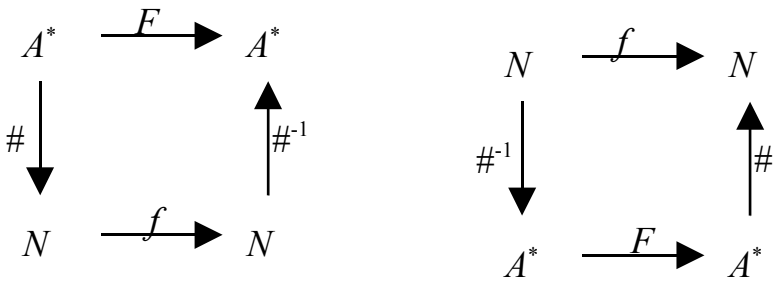
6.3. Нумерация позволяет каждой частичной функции $F : A^* \to A^*$ единственным образом поставить в соответствие частичную числовую функцию $f : N \to N$. В самом деле,

[1] Здесь не указано количество символов алфавита А, это означает, что рассматривается алфавит с произвольным количеством символов, либо алфавит, точное количество символов которого не представляет интереса.

если $v = F(w)$, то $\#(v) = \#(F(w)) = \#(F(\#^{-1}(\#(w)))) = (\#^{-1} \circ F \circ \#)(\#(w)) = f(\#(w))$, следовательно, $\#(v) = f(\#(w))$, где $f = \#^{-1} \circ F \circ \#$[2].

Аналогичным образом можно установить, что $F = \# \circ f \circ \#^{-1}$.

Связь частичных функций F и f можно представить в виде двух коммутативных диаграмм:



Таким образом, показано что:

(1) Каждый алгоритм $F : A^* \to A^*$ определяет частично вычислимую функцию $f : N \to N$

(2) Каждая частично вычислимая функция $f : N \to N$ определяет алгоритм $F : A^* \to A^*$

7. ***Машина Тьюринга***.

7.1. Основная идея предложенного Тьюрингом подхода к уточнению понятия алгоритма, заключается в том, что алгоритмами должны называться те и только те отображения $F : A^* \to A^*$, которые могли бы осуществлять достаточно простые машины-автоматы. Выполнение отображения понимается при этом следующим образом: машине-автомату предъявляется любое исходное слово $w \in A^*$, а она в результате обработки этого слова «выдает» слово $v = F(w)$. Таким образом, каждая машина Тьюринга (МТ) строит отображение $F : A^* \to A^*$ для соответствующей функции F. При этом каждая частичная функция F, для которой можно построить МТ называется *вычислимой по Тьюрингу*. МТ, вычисляющую функцию F, будем обозначать T_F.

7.2. Для задания МТ необходимо задать *алфавит состояний* $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ и *рабочий алфавит* $S = A \cup A'$, где A – алфавит *входных* символов, A' – алфавит *вспомогательных* символов (*маркеров*), который, в частности, может быть пустым.

У МТ есть внешняя память – лента, размеченная на ячейки, каждая из которых может быть «пустой» (пустая ячейка представляется символом Λ (лямбда большое)), либо содержать один символ рабочего алфавита S. МТ связана с лентой посредством управ-

[2] Операция $\circ$ означает композицию (последовательное применение) функций. По определению $(f \circ g)(x) = g(f(x))$.

ляющей головки (УГ), которая в каждый данный момент располагается под некоторой ячейкой ленты (эта ячейка называется обозреваемой ячейкой (ОЯ). Среди состояний МТ выделяются «начальное состояние» q_0 и «состояние останова» q_s. В начале работы МТ на ленту записываются начальные данные – слова над входным алфавитом A. Лента МТ с записанными на ней данными (например, словом **0010111001011100000**) и текущим положением УГ может быть изображена следующим образом (Λ представляет пустую ячейку ленты, остальные символы представляют самих себя, ОЯ обозначена рамочкой вокруг обозреваемого символа под рамочкой указано текущее состояние УГ):

```
...ΛΛΛΛΛΛΛΛΛΛΛΛΛ00101110010111000000ΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛ...
                   q
```

На каждом такте работы МТ УГ считывает символ из ОЯ, и в зависимости от этого символа и состояния УГ может записать в ОЯ новый символ, перейти в новое состояние, и передвинуться на следующую слева или справа ячейку ленты, либо остаться на месте.

7.3. ***Пример МТ***. Проверка правильности скобочных выражений, т.е. последовательностей открывающих и закрывающих скобок.

Правильным называется скобочное выражение, удовлетворяющее следующим двум условиям: (1) число открывающих скобок равно числу закрывающих, (2) каждая открывающая скобка предшествует парной ей закрывающей скобке. Например, (( ))( ) – правильное скобочное выражение, а)(или (( ) – неправильные скобочные выражения.

Если скобочное выражение правильное, МТ должна записать на ленту результат 1 и остановиться, если нет, – записать 0 и остановиться. Таким образом, после работы МТ на ленте должен быть записан только результат: 0 или 1, и УГ должна быть установлена на ячейке, в которой записан результат.

Рабочий алфавит: $S = \{(,), 0, 1\} \cup \{\Lambda, X\}$ (Λ и X – маркеры).
Алфавит состояний $Q = \{q_0, q_1, q_2, q_3, q_s\}$:

- q_0 – начальное состояние МТ: поиск ближайшей справа закрывающей скобки ")", замена ее на маркер X и переход в состояние q_1. Если, дойдя до конца слова, закрывающей скобки не нашлось – переход в состояние q_2 и движение влево;
- q_s – состояние останова;
- q_1 – поиск парной открывающей скобки при движении влево, замена её на X и переход в q_0;
- q_2 – достигнут конец выражения и при этом не найдено закрывающей скобки: движение влево, стирая все маркеры X; при достижении левого конца выражения (Λ) – запись результата 1; при обнаружении открывающей скобки – переход в состояние q_3, так как выражение неправильное.
- q_3 – выражение неправильное: движение влево, стираем символы (и X, запись результата 0 в первую пустую ячейку и переход в состояние q_s.

Описание данной МТ с помощью «пятерок» вида :

состояние, символ → состояние, символ, направление
направление: L – влево, R – вправо, H – на месте

$\boldsymbol{q_0}$, **(** $\to q_0, (, R$;	$\boldsymbol{q_0}$, **)** $\to q_1, X, L$;	$\boldsymbol{q_0}, \boldsymbol{X} \to q_0, X, R$;	$\boldsymbol{q_0}, \boldsymbol{\Lambda} \to q_2, \Lambda, L$;
$\boldsymbol{q_1}$, **(** $\to q_0, X, R$;	$\boldsymbol{q_1}$, **)** $\to q_1,), L$;	$\boldsymbol{q_1}, \boldsymbol{X} \to q_1, X, L$;	$\boldsymbol{q_1}, \boldsymbol{\Lambda} \to q_3, \Lambda, R$;
$\boldsymbol{q_2}$, **(** $\to q_3, \Lambda, H$;	$\boldsymbol{q_2}$, **)** невозможна;	$\boldsymbol{q_2}, \boldsymbol{X} \to q_1, \Lambda, L$;	$\boldsymbol{q_2}, \boldsymbol{\Lambda} \to q_s, \Lambda, H$;
$\boldsymbol{q_3}$, **(** $\to q_3, \Lambda, L$;	$\boldsymbol{q_3}$, **)** невозможна;	$\boldsymbol{q_3}, \boldsymbol{X} \to q_3, \Lambda, L$;	$\boldsymbol{q_3}, \boldsymbol{\Lambda} \to q_s, 0, H$;

Описание МТ («Пятерки») можно наглядно представить с помощью таблицы:

$\boldsymbol{q_i} \downarrow \setminus \boldsymbol{s_j} \to$	(	)	X	Λ
q_0	$q_0, (, R$	q_1, X, L	q_0, X, R	q_2, Λ, L
q_1	q_0, X, R	$q_1,), L$	q_1, X, L	q_3, Λ, R
q_2	q_3, Λ, H	—	q_2, Λ, L	$q_s, 1, H$
q_3	q_3, Λ, L	—	q_3, Λ, L	$q_s, 0, H$

7.4. Можно показать, что любую МТ можно перестроить таким образом, что она будет, вычислять ту же функцию, что и исходная, но при этом удовлетворять следующим двум условиям (соглашениям или ограничениям):

(1) в начальном состоянии (q_0) УГ установлена на ячейке, в которой записан первый символ исходного слова,

(2) в состоянии останова (q_s) УГ установлена на ячейке, в которой записан первый символ слова-результата функции F, вычисляемой этой МТ.

Такая МТ называется *нормальной* МТ. Переход к нормальным МТ существенно упрощает проблему построения композиции МТ и открывает возможность представления сложных МТ как композиции более простых МТ.

7.5. Диаграммы Тьюринга (ДТ).

7.6. Композиция МТ.

7.7. Понятие Универсальной МТ.

7.8. Проблема останова. Понятие об алгоритмической неразрешимости проблем.

8. ***Нормальные алгоритмы Маркова***

8.1. ***Определение нормального алгоритма Маркова*** (НАМ)

8.1.1. ***Подстановки***. Рассмотрим два непересекающихся алфавита: алфавит основных символов V и алфавит маркеров V'. Мы будем рассматривать множества слов V^* и $(V \cup V')^*$. *Подстановка* $\sigma \to \sigma'$ задает замену в рассматриваемом слове (τ) подслова $\sigma \in (V \cup V')^*$, указанного в левой части подстановки на слово $\sigma' \in (V \cup V')^*$, указанное в правой части подстановки. В результате исходное слово $\tau = \alpha\sigma\beta \in (V \cup V^\wedge)^*$ превращается в слово $\tau' = \alpha\sigma'\beta \in (V \cup V')^*$.

Заметим, что 1) подслова α и β могут быть пустыми (ε); 2) из всех возможных представлений исходного слова τ в виде $\tau = \alpha\sigma\beta$ выбрается такое представление, в котором длина подслова α ($|\alpha|$) минимальна, и для него осуществляется замена. Если обрабатываемое слово (τ) не содержит подслова σ, то подстановка считается *неприменимой*. Символ-стрелка $\to \notin (V \cup V')$ называется *метасимволом*. В некоторых подстановках вместо метасимвола «→» (стрелка) используется метасимвол стрелка с точкой «→.», означающий, что соответствующая подстановка является *терминальной* (завершающей). Иногда для обозначения терминальной подстановки в качестве метасимвола употребляют терминальную стрелку «|→», ее смысл такой же, как и стрелки с точкой «→.» .

8.1.2. ***Нормальный алгоритм Маркова*** задается конечной последовательностью подстановок $\{p_1, p_2, \ldots p_n\}$. При этом:

(1) если применимо несколько подстановок, применяется подстановка, которая встречается в описании алгоритма раньше других;

(2) при подстановке всегда рассматривается самое левое вхождение (длина α минимальна). Например, вхождением пустого слова (σ =) в заданное слово (τ) считается пустой символ перед заданным словом (τ = ασβ, где α = , β = τ).

(3) после применения терминальной подстановки алгоритм завершается;

(4) если ни одна из подстановок неприменима, алгоритм завершается.

(5) если алгоритм завершает работу

8.1.3. ***Пример***. Шифр Юлия Цезаря. V = {a, b, c, …, z}, V' = {*}. j-ая буква латинского алфавита шифруется $j+h$ (mod 26)-ой буквой того же алфавита. Например, для h = 3 подстановки НАМ имеют вид (маркер * помечает текущую шифруемую букву, цифра в скобках – номер правила подстановки):

(1)***A** → **D***, (2)***B** → **E***, (3)***C** → **F***, …, (23)***W** → **Z***, (24)***X** → **A***, (25)***Y** → **B***, (26)***Z** → **C***, (27)***** →**.** , (28) → *

Применим построенный НАМ к слову **CAESAR**:

CAESAR (28)→ ***CAESAR** (3)→ **F*AESAR** (1)→ **FD*ESAR** (5)→ **FDH*SAR** (13)→ **FDHV*AR** (1)→ **FDHVD*R** (12)→ **FDHVDU*** (27)→ **FDHVDU**

8.1.4. ***Пример.*** Копирование двоичных чисел: V = {0, 1}, V' = {*, #}

8.2. ***Процедура интерпретации НАМ***

НАМ – это упорядоченная конечная последовательность правил подстановок $\{p_1, p_2, \ldots p_n\}$, которые применяются к строке $\sigma_0 \in V^*$ (*исходная строка*) или к строкам $\sigma_i \in (V \cup V')^*$, $i > 0$ в соответствии со следующей процедурой:

(1) Положить $i = 0$.

(2) Положить $j = 1$.

(3) Если правило p_j применимо к σ_i , то перейти к (5).

(4) Положить $j = j + 1$. Если $j \le n$, то перейти к (3). В противном случае – ***останов.***

(5) Применить p_j к σ_i и найти σ_{i+1}. Положить $i = i + 1$. Если p_j – нетерминальное правило, то перейти к 2. В противном случае – ***останов.***

Говорят, что НАМ ***применим*** к слову σ_0, если в результате применения его к слову σ_0 произойдет ***останов.*** Множество слов в алфавите А, к которым применим заданный НАМ, называется областью применимости этого алгоритма.

Если же, начиная работу над словом σ_0, НАМ зацикливается, то он неприменим к слову σ_0.

8.3. ***Пример***. Сложение чисел в единичной системе счисления. V = {+, 1}, V' = {}. Правила подстановок:

8.3.1. *Первый вариант*: {1+ → +1; +1 → 1; 1→. 1} («выгоняем» плюсы на левый край слова (правило 1) и «стираем» (правило 2), в заключение применяем правило 3 и получаем результат).

Пример применения алгоритма:

σ_0 = «1111+11+111» = «1111+11+111» ⇒ «111+111+111» ⇒ «11+1111+111» ⇒ «1+11111+111» ⇒ «+111111+111» ⇒ «111111+111» ⇒ «111111111» ⇒ «111111111»

8.3.2. *Второй вариант*: {+ → ε; 1→. 1} (стираем все плюсы).

8.4 ***Эквивалентность алгоритмов.***

В приведенном примере для решения одной и той же задачи были составлены два НАМ. Но таких алгоритмов можно было бы привести не два, а три, четыре, … - целый класс алгоритмов. Фактически, решая проблему составления алгоритма для решения конкретной задачи, мы приводим в качестве результата – один из класса эквивалентных алгоритмов, решающих поставленную задачу.

Два алгоритма P и Q в алфавите A ($P : A^* \to A^*, Q : A^* \to A^*$) называются ***эквивалентными***, если области применимости этих алгоритмов совпадают, и для любого слова σ_0 из области применимости эти алгоритмы получают одинаковый результат, т.е. $P(\sigma_0) = Q(\sigma_0)$.

8.5. ***Тезис Маркова***. Любой алгоритм преобразования слов в алфавите *V* может быть представлен нормальным алгоритмом Маркова над алфавитом *V*.

8.6. ***Композиция нормальных алгоритмов Маркова***.

Композиция НАМ. Пусть у нас есть два НАМ *R* и *S*, работающих над исходным словом δ в некотором алфавите *A*. Фактически алгоритм *R* – некоторая функция *f*(δ), примененная к δ, а *S* – соответственно *g*(δ). Тогда их композиция есть суперпозиция функций *g*(*f*(δ)). Как преобразовать правила, чтобы осуществить такую композицию? Отметим, что все это мы делаем для случая, когда *R* и *S* работают в одном алфавите *A*.

В случае, когда оба алгоритма заканчиваются по правилу с **.**, решение имеет следующий вид.

В *R* заменим . на α.

В *S* заменим любой символ $\xi \in V \cup V'$ на его «близнеца» $\underline{\xi}$ ($\underline{\xi} \notin V \cup V'$) во всех правилах. Кроме того в *S* заменим . на β. (α и $\beta \notin V \cup V'$). Расширим правила подстановок, добавив правила (для каждой пары символов $\xi \in V$ и $\eta \in V$):

$\xi\alpha \to \alpha\xi$; $\alpha\xi \to \alpha\underline{\xi}$; $\underline{\xi}\eta \to \underline{\xi}\eta$; (группа правил (1)); $\xi\beta \to \beta\underline{\xi}$; $\underline{\beta\xi} \to \beta\xi$; $\underline{\xi\eta} \to \xi\eta$; (группа правил (2)); $\alpha\beta \to .$; далее следуют правила вычисления *S* в «близнецовском» алфавите (система правил (3)), далее следуют правила вычисления *R* в алфавите $V \cup V'$ (система правил (4)).

Тогда последовательность применения правил будет такая. Поскольку исходное слово не содержит ни «близнецов», ни α, ни β, будет работать система правил (4). Как только встретится правило, содержащее α (конечное правило *R*, так как метасимвол . заменен на α) начнет выполняться группа правил (1): вначале произойдет вытеснение α в самую левую позицию слова (после работы (4) где-то в слове находится ровно одна α), затем символы будут преобразовываться в «близнецы», пока исходных символов в слове уже не будет. В близнецовском алфавите может работать только система правил (3). Система правил (3) будет выполняться до остановки *S* (символ β, заменивший **.**). Как только в слове появится β, сработает правило $\alpha\beta \to .$, стоящее раньше (3). Слово после выполнения (2) содержит ровно одну β. Снова β будет вытесняться влево (пока не встанет рядом с α в начале слова), а затем каждый близнец заменится на исходный символ. Наконец, когда не останется ни одного близнеца, сработает правило $\alpha\beta \to$ **.**

(остановка). Результат есть $g(f(\delta))$, где результат снова слово в алфавите V, не содержащее никаких вспомогательных символов («близнецов», α и β).

В случае, когда один из алгоритмов заканчивается по **.**, другой (для определенности – первый) – потому что ни одно из правил неприменимо, нужно ввести в самом конце еще одно правило $\rightarrow \alpha$. В общем случае, между группами правил (3) и (4) нужно добавить группу правил, которая слово $\xi\ldots\eta\alpha$ переводит в $\beta\xi\ldots\eta\alpha$. Такая группа правил аналогична группе (1), при этом используется маркер γ.

8.7. ***Алгоритмическая неразрешимость проверки свойства самоприменимости.***

8.7.1. ***Свойство самоприменимости***. Пусть в некотором алфавите $A = \{a_1, a_2, \ldots a_n\}$ задан НАМ U с помощью системы подстановок. Мы можем рассматривать запись системы подстановок как слово в алфавите $A' = \{a_1, a_2, \ldots a_n, a_{n+1}, a_{n+2}, a_{n+3}\}$, где $a_{n+1} = \rightarrow$, $a_{n+2} =$, $a_{n+3} = .$. Теперь применим систему подстановок U к слову, изображающему этот НАМ в алфавите A'. Если теперь U начнет перерабатывать это слово и остановится, то U будем называть *самоприменимым*, иначе – *не самоприменимым*.

8.7.2. Попытаемся построить такой НАМ W, который для любого слова в алфавите A' вырабатывает слово M, если это слово изображает самоприменимый алгоритм, и слово L, если алгоритм не самоприменим. Т.е. НАМ W распознает свойство самоприменимости для *любого* слова в A'.

Снова «от противного». Предположим, что такой алгоритм W построен. Это значит, что в его системе правил есть правила ... $\rightarrow$. M (1) и ... $\rightarrow$. L (2). Правило (1) срабатывает, когда алгоритм U самоприменим и (2), когда U – не самоприменим.

Заменим теперь W на W', заменив правило (1) на два правила $\{. M \rightarrow \alpha M; \alpha \rightarrow \alpha\}$ (1′) : мы заменили . на α и добавили новое правило, которое зацикливает W'. Таким образом, алгоритм W' зацикливается, если U - самоприменим, и вырабатывает L, если U – не самоприменим. Применяем теперь W' к самому себе. Снова возможны два варианта:

1. W' - самоприменим: произойдет результативная остановка. Но тогда W' - не самоприменим, поскольку вырабатывает слово L, которое и обозначает факт не самоприменимости.

2. W' - не самоприменим: остановки не произойдет. Но ведь это как раз обозначает факт самоприменимости. Противоречие.

8.8. ***Заключительные замечания***

8.8.1. Конечно, можно построить и универсальный НАМ U, который мог бы *интерпретировать* любой нормальный алгоритм, включая самого себя. Он строится приблизительно в той же манере, что и универсальная машина Тьюринга.

8.8.2. Можно доказать эквивалентность двух формальных систем Тьюринга и Маркова конструктивным путем: построить универсальную МТ, которая могла бы интерпретировать любой НАМ и, наоборот, построить универсальный НАМ, который интерпретирует любую МТ.

8.8.3. Уже отмечалось, что решить задачу можно различными путями: существуют различные НАМ решения одной и той же задачи. Конечно, важно уметь *распознавать эквивалентность* двух НАМ. Но проблема построения алгоритма, который

может определить эквивалентность любых двух НАМ, алгоритмически неразрешима.

9. ***Об универсальных алгоритмах***

9.1. Идея универсальных машин и алгоритмов широко используется в компьютерах. По существу программы (алгоритмы) почти никогда не пишутся в системе команд компьютера. Программы пишут программы для некоторой абстрактной машины (например, Си-машины, которая изучается в данном курсе). Программа переносится на реальный компьютер некоторым алгоритмом (компилятором), причем такой перенос, как правило, является многоступенчатым (т.е. с использованием некоторых промежуточных интерпретаторов). В процессе компиляции многократно производится преобразование информации из одного представления в другое. Причем все эти представления ориентированы на бесконечные машины.

9.2. Но реальный компьютер и реальный интерпретатор (Си) – конечные машины: все виды их памяти конечны (более того, известны их верхние границы). На конечных машинах возникает новый тип останова, когда для выполнения программы не хватает памяти. На самом деле так было бы и для МТ, если бы лента МТ, была ограничена.

9.3. В этой связи алгоритмическая неразрешимость утрачивает смысл. Действительно, эти проблемы связаны с бесконечностью ленты Тьюринга (хотя любая машина использует конечное число клеток ленты, но всегда найдется машина, которая использует на клетку больше – нет верхней границы), аналогично, со словом в алгоритмах Маркова. Вспомним также суммирование неизвестного количества чисел, находящихся на неизвестном (но конечном) расстоянии друг от друга. На конечной ленте нет такой проблемы. Действительно, поскольку в компьютере конечное число клеток (битов) памяти, то, следовательно, на нем в принципе может быть только конечное число программ (алгоритмов), хотя фактически большое $\approx n!$, где n - число битов. И казалось бы нам удается избежать такой опасности как неразрешимость. Но там нас подстерегает другая беда – сложность. Ведь опасность может быть в объеме памяти, когда нельзя выполнить программу на данном компьютере из-за нехватки памяти. Другая опасность – время. Не имеет смысла иметь программу расчета траектории ракеты, если за время вычисления ракета уже попадет в цель и, следовательно, ее нельзя будет поразить противоракетой. Факт временной сложности используется широко, например, в системе защиты информации. Конечно, в криптографии любой шифр может быть потенциально раскрыт, но если дешифровка требует полмиллиона лет суммарной работы всех компьютеров в мире, то шифр можно считать абсолютно надежным. Через полмиллиона лет зашифрованное письмо просто утратит всякий смысл.

9.4. Поэтому при составлении программы для компьютера, необходимо оценивать ее временную и пространственную сложность (сложность исследуется и в формальных системах, но для реальных программ – это жизненно необходимо). Необходимо иметь хотя бы грубые оценки сложности составляемых и применяемых программ. Более тонкие оценки дают возможность сравнивать между собой эквивалентные программы и находить среди них подходящие (а иногда и наиболее подходящую) для данных условий.