В. Н. Пильщиков

Лекции по курсу
«Алгоритмы и алгоритмические языки»

Часть 1: Элементы теории алгоритмов

Москва, 2024

Содержание

Лекция 1. АЛГОРИТМЫ

План лекции:

1. ВВОДНАЯ ЧАСТЬ

Вначале я хочу вкратце рассказать о том, чему посвящён курс «Алгоритмы и алгоритмические языки», какие темы мы будем рассматривать.

Тематика лекций

Факультет ВМК готовит математиков, способных решать задачи с помощью компьютеров. Поэтому обучение наших студентов ведётся по двум основным направлениям: первое направление – математическое, где студенты изучают математические дисциплины (математический анализ, линейную алгебру и т. д.), а второе направление – программистское, где студенты изучают компьютеры и способы работы с ними. Курс «Алгоритмы и алгоритмические языки» начинает это второе направление. Этот курс полугодовой, в конце семестра (в январе) будет экзамен, причём экзамен будет в письменной форме.

Чему посвящён этот курс?

Прежде всего отмечу, что термины ***ЭВМ*** (*электронная вычислительная машина*) и ***компьютер*** обозначают одно и то же, т. е. являются синонимами. «ЭВМ» – это русское название данных устройств, а «компьютер» – это калька с английского слова *computer*, который переводится как «вычислительная машина». В дальнейшем я буду пользоваться обоими этими терминами.

Так вот, в настоящее время компьютеры получили очень широкое распространение и с их помощью решаются самые разнообразные задачи. Компьютеры кажутся всемогущими. Однако при всём этом надо понимать следующую важную вещь. Сам по себе компьютер ничего не делает. Чтобы заставить его решить какую-либо задачу, надо составить для него программу, указав в ней, какие действия, в каком порядке и над какими данными должен выполнить компьютер. Выполняя такую программу, компьютер и решает задачу; если же программы нет, то и компьютер ничего не делает. А кто составляет эти программы? Человек. Таким образом, всё зависит от человека: составит он программу игры в шахматы – компьютер будет играть в шахматы, составит программу вычисления интеграла – компьютер будет вычислять интегралы и т. д.

Следовательно, составление программ для ЭВМ, т. е. программирование – важнейший аспект использования ЭВМ. Этот вид человеческой деятельности нас и будет интересовать – мы будем учиться составлять программы для ЭВМ. Это основная задача курса. С нею тесно связана и другая задача курса – изучение способов записи программ. Недостаточно только придумать программу, надо ещё её правильно записать на языке, понятном ЭВМ. Изучением таких языков (они называются *алгоритмическими языками* или *языками программирования*) мы также будем заниматься.

Конкретно, темы наших лекций можно разделить на три больших раздела.

1. **Элементы теории алгоритмов**. Здесь мы рассмотрим, что такое алгоритмы (это более общее понятие, чем программы для ЭВМ), каковы их свойства и как они записываются в формальном виде.

2. **Язык Паскаль**. Это основной раздел наших лекций, в рамках которого мы будем учиться составлять программы для ЭВМ и будем детально изучать язык программирования Паскаль, на котором эти программы записываются.

3. **Структуры данных**. Здесь мы познакомимся с такими понятиями, как списки, деревья, стеки, очереди, таблицы и т. д., в виде которых в ЭВМ представляются сложные объекты, используемые в программах.

Отмечу, что в последующих программистских курсах Вы будете изучать архитектуру ЭВМ и язык ассемблера, операционные системы и язык C++, методы трансляции и машинную графику, компьютерные сети и искусственный интеллект, и много других вещей, связанных с программированием и компьютерами.

Литература по лекциям

Теперь я перечислю основные учебники, которые помогут Вам при изучении лекционного материала. (Что касается литературы по семинарам, то её Вам сообщат преподаватели семинаров.)

По разделу «Элементы теории алгоритмов»:

1. Любимский Э.З., Мартынюк В.В., Трифонов Н.П. «Программирование» – М.: Наука, 1980.
2. Корухова Л.С., Шура-Бура М.Р. «Введение в алгоритмы» – М.: МГУ, 1997.

По разделу «Язык Паскаль»:

3. Абрамов В.Г., Трифонов Н.П., Трифонова Г.Н. «Введение в язык Паскаль» – М.: КноРус, 2011.
4. Йенсен К., Вирт Н. «Паскаль. Руководство для пользователя». – М.: «Компьютер», 1993.

По разделу «Структуры данных»:

5. Вирт Н. «Алгоритмы и структуры данных». – СПб.: «Невский диалект», 2001.
6. Ахо А., Хопкрофт Д., Ульман Д. «Структуры данных и алгоритмы». – М.: «Вильямс», 2000.

2. ИНТУИТИВНОЕ ОПРЕДЕЛЕНИЕ АЛГОРИТМА

Алгоритм – это одно из самых важных понятий математики, особенно тех её разделов, что связаны с ЭВМ. С изучения этого понятия мы и начнём наш курс.

2.1 Понятие алгоритма

В первом приближении, алгоритмом называется точное описание того, как решать некоторую задачу. Более развернуто: алгоритм – это точно и полно сформулированная инструкция исполнителю, указывающая, какие действия, в каком порядке и над какими объектами надо выполнить, чтобы решить задачу.

Сделаю несколько замечаний и пояснений.

1) Об алгоритме можно говорить, только если есть задача. Если задачи нет, то нечего решать, следовательно, нет и алгоритма.

2) Следует отличать понятие «алгоритм» от таких понятий, как «решение задачи (процесс)» и «результат задачи (ответ)». Алгоритм – это описание способа решения задачи, план решения. Выполняя этот план, мы тем самым решаем задачу и получаем, в конце концов, результат, ответ задачи.

3) Тот, кто выполняет алгоритм (человек это или компьютер), называется *исполнителем*.

4) Из всех возможных описаний того, как решать задачи, алгоритмы выделяются тем, что это наиболее точное и полное описание.

Несколько слов о названии «алгоритм». Это слово происходит от латинской формы написания имени знаменитого узбекского математика X века Аль Хорезми («из Хорезма»), который сформулировал точные правила выполнения арифметических действий над числами, записанными в десятичной системе (в то время это была новинка, т. к. до этого использовалась запись чисел римскими цифрами). Эти правила и стали называть алгоритмами, а позже так стали называть любые точные правила решения задач.

2.2 Примеры алгоритмов

Теперь рассмотрим несколько примеров алгоритмов.

1. Алгоритм Евклида

Если слово «алгоритм» появилось лишь в средние века, то само понятие алгоритма было известно ещё древним грекам (около 2,5 тыс. лет назад). Они придумали много алгоритмов, которые используются и в наши дни. Пожалуй, наиболее известным из них является алгоритм Евклида – того самого, который знаменит как великий геометр. Этот алгоритм позволяет находить **наибольший общий делитель** двух произвольных натуральных чисел x и y, который мы будем обозначать как НОД(x, y).

В школе Вы изучали понятие НОД, но, тем не менее, напомню, что это такое. Пусть имеются два натуральных числа. Общим делителем этих чисел является натуральное число, на которое делится каждое из них. Например, общими делителями чисел 45 и 30 являются 1, 3, 5 и т. д. Как видно, общих делителей может быть несколько, и представляет интерес найти наибольший из них – это и есть НОД. Например, НОД$(45, 30) = 15$. НОД нужен, например, для приведения дроби, скажем 45/30, к несократимому виду – к виду 3/2.

Так вот, алгоритм Евклида и говорит, как найти $z =$ НОД(x, y). Записывается он в виде следующих двух пунктов:

1. Если $x = y$, то положить z равным x и остановиться, иначе перейти к пункту 2.
2. Если $x > y$, то положить x равным $x - y$, иначе положить y равным $y - x$. В любом случае перейти к пункту 1.

Вот такая запись и является алгоритмом. Я не буду приводить доказательство корректности этого алгоритма (т. е. доказывать, что он действительно даёт НОД) – это можете сделать и Вы сами. Покажу лишь, как этим алгоритмом пользоваться.

Но прежде сделаю одно замечание. Иметь алгоритм – это ещё не значит решить задачу. Алгоритм, как не трудно видеть, – это решение задачи в общем виде, для любых x и y. Но выполнять алгоритм в общем виде мы не можем, т. к. не зная конкретных значений x и y, мы не знаем, равны x и y, и не знаем, что делать дальше. Выполнить алгоритм можно только при конкретных исходных данных. Об этом необходимо всегда помнить.

Давайте выполним алгоритм Евклида при следующих исходных данных: $x = 45, y = 30$

1. $x = y$? Нет $\rightarrow$ к п. 2.
2. $x > y$? Да $\rightarrow$ выполняем действие после слова «то»: $x = 45 - 30 = 15$ $(y = 30)$
   Переходим снова к п. 1 алгоритма.
3. $x = y$? Нет $\rightarrow$ к п. 2.
4. $x > y$? Нет $\rightarrow$ выполняем действие после слова «иначе»: $(x = 15)$ $y = 30 - 15 = 15$
   Переходим к п. 1.
5. $x = y$? Да $\rightarrow z = 15$ и останов.

Итак, ответ: НОД$(45, 30) = 15$.

Если мы применим алгоритм Евклида к x=13 и y=13, то он будет выполняться несколько по-другому:

1. $x = y$? Да $\rightarrow z = 13$ и останов.

Замечание. Если разработка, придумывание алгоритма – вещь творческая, требующая знаний и интуиции, то, как видно из нашего примера, выполнение уже имеющегося алгоритма – вещь чисто механическая. При выполнении алгоритма не надо вникать в суть задачи, не нужна никакая интуиция, а надо лишь точно выполнять указанные инструкции. Это очень важная особенность алгоритмов. Именно она объясняет, почему выполнять алгоритмы может не только человек, но автомат, например компьютер. Компьютер не умеет думать, но указанные инструкции он выполняет точно.

2. Вычисление по формулам

Другим примером алгоритма может служить вычисление корней квадратного уравнения $ax^2 + bx + c = 0$ при условии, что $a \neq 0$ и $D = b^2 - 4ac \geq 0$.

Этот алгоритм записывается в две строки:

$$1.\ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$2.\ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Как видно, здесь всё сводится к вычислению по формулам. Этот пример заслуживает особого внимания потому, что вычисление по формуле – это частный случай алгоритма. Действительно, в формуле четко сказано, какие действия производить, над какими величинами и в каком порядке. То есть это алгоритм. Однако обратное утверждение неверно: не всякий алгоритм сводится к формулам. Например, алгоритм Евклида нельзя записать в виде формулы. Как показывает практика, формулы – это лишь очень маленькое подмножество алгоритмов. К сожалению, довольно редко удаётся выразить решение задач в виде формулы.

Приведённые примеры алгоритмов относятся к математике. Это не случайно, т. к. большинство алгоритмов «живет» именно в математике (отмечу ещё алгоритмы сложения двух целых чисел «столбиком», деление целых чисел «уголком» и т. д.). Однако имеется много алгоритмов и вне математики, даже в нашей обычной жизни, правда, мы не всегда об этом догадываемся. Одна из причин этого – то, что они скрываются под другими названиями: «правила», «инструкции», «рецепты» и т. п.

3. Кулинарные рецепты

Это также алгоритмы, т. к. в них сказано, что делать, в какой последовательности и над какими продуктами, чтобы получить результат – некоторое блюдо.

4. Правило перехода через улицу, где нет светофора

«Посмотрите налево и, если нет приближающегося транспорта, то дойдите до середины улицы, где посмотрите направо и, если нет приближающегося транспорта, перейдите улицу до конца».

Других примеров алгоритмов я пока не буду приводить. Думаю, что и приведенных примеров достаточно, чтобы Вы получили общее представление об алгоритмах.

2.3 Назначение алгоритмов

Теперь поговорим о назначении алгоритмов: зачем они нужны, каково их предназначение?

Алгоритмы – это способ передачи знаний о том, как решать задачи. Пусть кто-то придумал, как решать некоторую задачу, или придумал способ приготовления нового блюда, и пусть он захотел поделиться своим открытием с другими людьми. Тогда он должен описать придуманный способ решения задачи, способ приготовления блюда. Причём это описание должно быть полным и точным, иначе может произойти следующее. Если в описании есть пропуски или что-то неточно объяснено, то другому человеку, действующему по этому описанию, придётся восполнять эти пропуски, устранять неточности. Но в общем случае это можно сделать по-разному, в частности, не так, как думал автор описания. И в результате может получиться не то блюдо, которое придумал автор рецепта: например, вместо сладкого блюда получится горькое. Таким образом, если автор хочет, чтобы и другие люди получали тот же результат, что и он, то он должен дать описание без всяких двусмысленностей и пропусков, то есть должен представить алгоритм.

Вот в такой передаче без искажения знаний о способах решения задач, доступных разным людям, и есть назначение алгоритмов.

Кроме того, поскольку в алгоритмах всё, что необходимо, сказано и сказано чётко, точно, то выполнять алгоритмы можно поручить и автомату, в частности ЭВМ. Таким образом, алгоритм – это ещё и способ передачи знаний от человека к автоматам.

Чтобы алгоритмы соответствовали такому назначению, они должны обладать рядом обязательных свойств. Рассмотрим их.

2.4 Свойства алгоритмов

1. Полнота

Это свойство означает, что в алгоритме должны быть учтены все случаи, которые могут возникнуть при решении задачи, и для каждого такого случая должно быть указано, что надо делать.

Другими словами, алгоритм должен быть описан так, чтобы при его выполнении не нужно было ничего додумывать. Эта полнота описания особенно важна, если исполнителем алгоритма является автомат, компьютер, поскольку он не умеет домысливать и делает только то, что ему сказали, и ничего сверх того. Да и для исполнителей-людей это важное требование, т. к. разные люди по-разному устраняют недомолвки.

С точки зрения этого свойства инструкции по использованию бытовых приборов (например, пылесоса), хотя и очень похожи на алгоритмы, все-таки таковыми не являются. Вы все прекрасно знаете, что, пользуясь этими инструкциями, нужно чуть ли ни на каждом шаге о чём-то догадываться, что-то домысливать.

2. Выполнимость

Это свойство означает, что в алгоритме надо указывать только такие действия, которые может выполнить исполнитель.

Свойство выполнимости требует, прежде всего, чтобы действия, указанные в алгоритме, были понятны исполнителю. Если исполнитель не поймет, что надо делать, то он и не сделает этого. Определить, что́ именно понятно, а что нет, достаточно сложно, но обычно требуется, чтобы описание алгоритма было понятно как можно большему числу людей, чтобы в алгоритме указывались как можно более простые действия.

Однако одной понятности мало. Указанные в алгоритме действия должны быть и практически выполнимыми. В самом деле, действие «подними груз весом в 1 миллион тонн» понятно, но попробуй его выполнить. Алгоритм должен указывать только разумные действия, которые исполнитель действительно может реализовать.

Итак, в алгоритмах можно указывать любые действия, лишь бы они были понятны исполнителю и чтобы он мог их практически выполнить.

3. Однозначность (детерминированность)

Это свойство означает, что кто бы ни выполнял алгоритм и сколько бы раз ни выполнял, он должен при одних и тех же исходных данных получать один и тот же результат.

Такому свойству удовлетворяет, например, алгоритм Евклида: сколько бы раз мы ни применяли его к числам 45 и 30, всегда получим ответ 15.

Однако кулинарные рецепты обычно не обладают этим свойством. Действительно, если двум поварам дать приготовить блюдо по одному и тому же кулинарному рецепту, то вряд ли получатся в точности одинаковые блюда. Почему? Да потому, что в кулинарных рецептах встречаются фразы типа «посоли по вкусу», «вари до готовности» и т. п., которые каждый повар понимает по-своему, и в результате получаются разные блюда. Но вот если эти неточные фразы заменить на строгие указания, например, вместо «посоли по вкусу» написать «положи 10 граммов соли», то кулинарные рецепты станут однозначными.

Как добиться детерминированности алгоритмов?

Во-первых, чтобы весь алгоритм был детерминированным, необходимо, чтобы детерминированными были и указанные в нём отдельные действия. В частности, в алгоритмах не допускается бросание монеты или жребия. Возьмём, к примеру, такую инструкцию: «Брось монету. Если выпадет орел, то отдай 100 рублей, а иначе возьми 100 рублей». Ясно, что если выполнять её несколько раз, то мы будем получать разные результаты – то ли мы обогатимся, то ли разоримся.

Во-вторых, в алгоритме должны быть однозначно сформулированы правила перехода от одного действия к другому. Фразы типа «после этого действия выполни то действие либо другое действие» приводят к неоднозначности, поэтому они недопустимы в алгоритмах.

В-третьих, детерминированность требует, чтобы объекты, с которыми работает алгоритм, были точными. Рассмотрим, к примеру, такую инструкцию: «Если размер этого облака равен размеру того облака, то сделай то-то, иначе – сделай то-то». Так как размер облака – понятие неопределённое, то у одного исполнителя размеры облаков могут совпадать, а у другого нет. Из-за этого получатся разные результаты.

Итак, свойство однозначности предъявляет очень жёсткие требования к алгоритмам. Но это необходимо, если мы хотим, чтобы разные исполнители получали одинаковые результаты, чтобы можно было повторить и проверить результаты.

4. Конечность (результативность)

Это свойство означает, что алгоритм должен давать ответ за конечное число шагов, через конечное время.

Практическая важность этого свойства очевидна: мы не можем ждать результат бесконечно долго.

Этому свойству удовлетворяет, например, алгоритм Евклида. Ведь на каждом шаге мы уменьшаем одно из чисел x и y, причём они всегда остаются положительными (мы всегда вычитаем меньшее из большего), поэтому уменьшать x и y мы можем только конечное число раз.

Противоположный пример. Пусть даны две правильные десятичные дроби:

$a = 0{,}\alpha_1\alpha_2\alpha_3\ldots$

$b = 0{,}\beta_1\beta_2\beta_3\ldots$

и пусть мы хотим узнать, равны ли они. В школьном учебнике приводится такой способ решения этой задачи. Сначала надо сравнить первые цифры после запятых; если они не равны, то числа a и b не равны. Если же первые цифры равны, то далее надо сравнить следующие

цифры после запятой; опять же, если эти две цифры не равны, то числа a и b не равны. Иначе сравниваем третьи цифры и так далее.

Даёт ли этот метод ответ через конечное число шагов? Нет, не всегда. Если a не равно b, то мы, в конце концов, дойдем до пары различных цифр. Но если a и b равны, то и все пары цифр равны и сравнение будет продолжаться бесконечно. Это описание не есть алгоритм.

В связи со свойством конечности возникает такой вопрос: а какова величина этого «конечного числа шагов». Ведь одно дело, когда алгоритм даёт ответ через минуту, и совсем другое дело, когда ответ мы получим через тысячу лет. Однако, несмотря на практическую важность этой величины, в свойстве конечности не накладывается никакого ограничения на конкретную величину числа шагов, через которое алгоритм должен остановиться. Дело в том, что с развитием науки и техники становятся все более совершенными наши орудия труда и потому то, на что раньше уходило много времени, теперь требует мало времени. Например, пусть алгоритм требует выполнения 100 млн. шагов и его выполняет человек (приблизительно 1 операция в секунду), тогда ему потребуется более трех лет (в 1 году около 31 млн. секунд). Но если этот же алгоритм будет выполняться на современной ЭВМ с быстродействием 100 млн. операций в секунду, то потребуется всего лишь 1 секунда. Учитывая постоянный прогресс науки и техники, и не накладывают ограничений на величину «конечного числа шагов». Главное, чтобы оно не было бесконечным.

Таковы свойства, которым должны удовлетворять алгоритмы.

Если подвести итог всему сказанному, то можно дать такое определение алгоритма:

Алгоритм *– это точно сформулированная совокупность правил для исполнителя, указывающая как решать задачу, причём эта совокупность должна удовлетворять свойствам полноты, выполнимости, однозначности и конечности.*

Такое определение принято называть *интуитивным определением алгоритма*. Почему «интуитивным»? Ведь слово «интуитивный» часто противопоставляют словам «точный», «строгий», «формальный». Почему это не строгое определение? Да потому, что в нём используются неточные, нестрогие понятия: что считать правилом, а что не считать? какое правило считать выполнимым, а какое нет? И так далее.

На следующих лекциях мы рассмотрим уже строгие определения алгоритмов, а пока нас удовлетворит и это. Хотя оно и не строгое, но даёт нам представление о том, что такое алгоритм, каковы его особенности.

Сейчас же нас будет интересовать несколько иной вопрос – как записывать алгоритмы? Это следующая тема лекции.

3. СПОСОБЫ ЗАПИСИ АЛГОРИТМОВ

Сегодня мы рассмотрим два способа – словесное описание и блок-схемы.

3.1 Словесное описание

Довольно часто алгоритмы записывают обычными словами с использованием, если надо, соответствующей математической символики. Именно так мы описали алгоритм Евклида.

Такое словесное описание алгоритма выгодно тем, что оно понятно многим людям, т. к. не требует никаких специальных знаний (все мы хорошо знаем свой родной язык). Однако у этого способа есть и недостатки – он часто не точен, не строг, а если алгоритм длинный, то ещё и не нагляден.

Почему не строг? Да потому, что тексты на естественном языке часто двусмысленны, не однозначны, а двусмысленность может быть причиной того, что исполнитель алгоритма сделает не то, что следовало бы. Например, кто кого любит во фразе «Мать любит дочь»? В «мужском» варианте этому предложению соответствуют две фразы: «Отец любит сына» и «Отца любит сын», а вот в «женском» варианте обе эти фразы сливаются в одну.

Различие между «=» и «:=»

Более того, казалось бы, язык математики – образец точности и строгости, но и он иногда бывает двусмысленным. Например, что означает запись «$x = 2$»? Чтобы был понятен смысл вопроса, приведу такую фразу:

если $x = 2$, то $y = 2$, иначе $y = 5$

Ясно, что запись «$x = 2$» означает «проверь, равно ли значение переменной x числу 2». Однако аналогичная запись «$y = 2$» имеет уже иной смысл – «сделай новым значением переменной y число 2». Здесь уже не проверка на равенство, а изменение значения переменной. Распознать, что означает запись «$x = 2$», можно только в контексте (в окружении других слов), а вот вне контекста смысл этой записи нельзя определить точно.

Итак, не только естественный язык, но даже и математическая символика страдает неточностями. Поэтому при словесной формулировке алгоритма надо быть предельно внимательным и избегать неточностей.

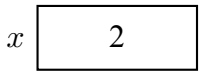
В частности, в языке программирования Паскаль, который мы будет изучать, проблему с двусмысленностью знака «=» принято решать следующим образом: этот знак оставляют за операцией сравнения, а для операции изменения значения переменной вводится новый знак «:=», который читается «присвоить». Следовательно, «$x = 2$» – это проверка, равно ли значение x двум, а «$x := 2$» – это присваивание переменной x нового значения – числа 2 (старое значение при этом теряется). Такими обозначениями мы будем пользоваться в дальнейшем, так что стоит их сразу запомнить.

В данных обозначениях алгоритм Евклида можно переписать так:

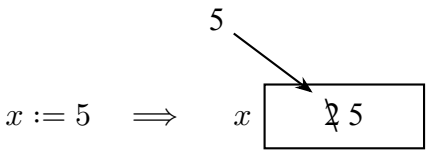
1. Если $x = y$, то $z := x$ и остановиться, иначе перейти к п. 2.
2. Если $x > y$, то $x := x - y$, иначе $y := y - x$. Перейти к п. 1.

Переменная как «ящик»

Раз уж мы заговорили о переменных и присваиваниях, то хочу отметить следующее. В математических задачах переменная, как правило, не меняет своего значения: один раз присвоили ей значение 2 и затем на протяжении всей задачи она и обозначает двойку. В алгоритмах же переменные, напротив, очень часто меняют своё значение в процессе выполнения алгоритма. В связи с этим в программировании переменную лучше всего представлять себе как некий ящик, ячейку (*см. рисунок ниже*). У этого ящика имеется имя – это имя переменной, оно не может меняться (*на рисунке это имя x слева от прямоугольника*); содержимое же ящика – это значение переменной (*число 2 внутри прямоугольника*), оно может меняться.



Тогда запись «$x = 5$» означает проверку, находится ли сейчас в ящике число 5 или нет, а запись «$x := 5$» – уничтожение прежнего значения и запись в ящик нового значения 5.



Мы рассмотрели первый недостаток словесного описания алгоритмов – нестрогость. Теперь о втором недостатке – запутанности, отсутствии наглядности.

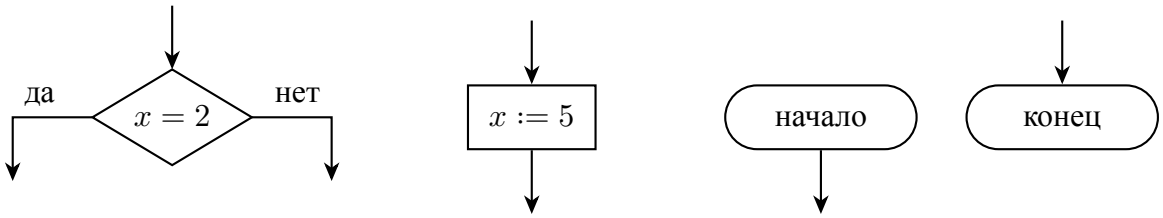
Запутанность проявляется в больших (длинных) алгоритмах. Суть дела в том, что в длинных алгоритмах (в сотни и более пунктов) очень сложно узреть переходы от одного пункта к другому, если они расположены далеко друг от друга; например, трудно сразу найти пункт 759, на который указан переход из пункта 48.

Для устранения этого недостатка была введена специальная форма записи алгоритмов в виде т. н. блок-схем, в которых все переходы для наглядности указываются стрелками. Вот эти блок-схемы мы сейчас и рассмотрим.

3.2 Блок-схемы

Блок-схемой называется графическое изображение алгоритма, в котором каждый этап алгоритма представляется в виде некоторой геометрической фигуры, называемой «блоком», эти блоки соединяются стрелками, показывающими порядок выполнения алгоритма. Внутри каждого блока указываются (в любом виде) действия, которые надо выполнить на соответствующем этапе.

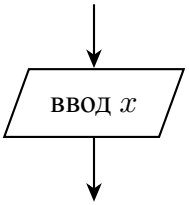
В блок-схемах разные типы действий изображаются разными геометрическими фигурами. В виде ромбиков (*см. а*), которые называются *логическими блоками*, записываются условия (например, сравнение двух чисел), которые необходимо проверить. В виде прямоугольников (*см. б*), которые называются *вычислительными блоками*, записываются действия, связанные с преобразованием информации (например, присваивания переменным).



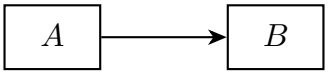
а) *б*) *в*)

Кроме того, поскольку в схеме много блоков, то обязательно надо указать, с какого блока начинается выполнение алгоритма. Для этого пишется слово «начало» в овале (*см. в*), и от него ведётся стрелка к этому первому блоку. Причём «начало» алгоритма может быть только одно, поскольку иначе будет неопределённость (с которого блока начинать?). Аналогично необходимо указывать и то, когда следует прекращать выполнение алгоритма. Для этого от блока, на котором прекращается работа, ведётся стрелка к слову «конец», также заключённому в овал. Блоков «конец» может быть любое число, т. к. закончить выполнение алгоритма можно в любом месте.

Операции ввода-вывода указывают в виде параллелограмма:



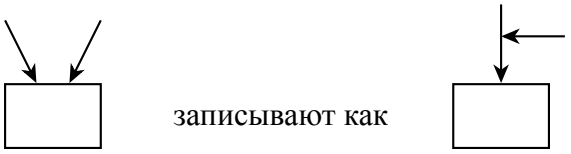
Переходы от блока к блоку указываются стрелками:



Стрелка означает, что, выполнив блок A, следующим необходимо выполнить блок B.

На количество стрелок, выходящих из блоков, накладывается следующее ограничение. Из логического блока всегда должно выходить две стрелки (*см. рис. а*), причём одна помечается словом «да» (или знаком «+»), а другая – словом «нет» (или знаком «−»). Это обусловлено смыслом таких блоков: в них проверяется условие, которое может выполняться (быть истинным) или не выполняться (быть ложным), поэтому мы и переходим либо по стрелке «да», если условие выполняется, либо по стрелке «нет», если условие не выполняется. Из любого вычислительного блока и блока «начало» должна выходить только одна стрелка, иначе будет неоднозначность. Из блока же «конец» стрелок не должно быть.

Теперь о стрелах, входящих в блоки. В блок «начало» не должно входить ни одной стрелки, а в любые остальные блоки может входить любое число стрелок. Причём, если входящих стрелок несколько, то они обычно собираются в единый узел непосредственно перед блоком – так нагляднее:



И ещё один вопрос: в каком порядке размещать блоки на бумаге? В любом, как Вам нравится, но обычно их рисуют сверху вниз в том порядке, как они должны выполняться. При этом надо следить, чтобы стрелки не пересекались, иначе получится не наглядная, запутанная схема, т. е. потеряется основное достоинство блок-схемы.

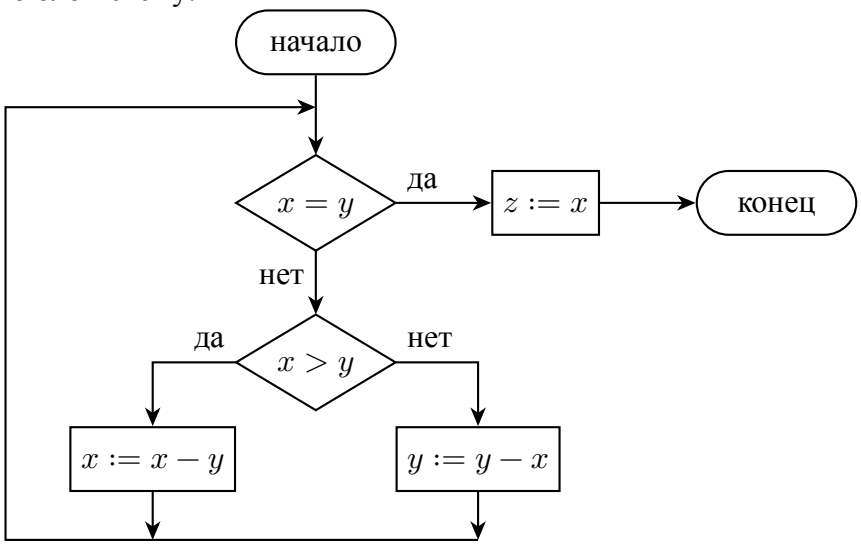
Вот таковы правила записи алгоритмов в виде блок-схем. Теперь рассмотрим примеры такой записи.

1. Алгоритм Евклида

Словесное описание этого алгоритма:

1. Если $x = y$, то $z := x$ и остановиться, иначе перейти к п. 2.
2. Если $x > y$, то $x := x - y$, иначе $y := y - x$. Перейти к п. 1.

Привожу его блок-схему:



2. Алгоритм сложения

Задача: имеется 128 чисел $x_1, x_2, \ldots, x_{128}$; найти S – сумму этих чисел.

Идея решения: положить вначале S равным нулю, а затем по очереди просмотреть все числа x_i и каждое из них добавить к S.

Как записать эту идею в виде точного алгоритма? Новички нередко предлагают следующее словесное описание (для краткости переходы на следующие по порядку пункты явно не указываем):

1. $S := 0$
2. $S := S + x_1$
3. $S := S + x_2$

   …

129. $S := S + x_{128}$

Казалось бы, всё правильно и понятно. Однако это не так. Плохо здесь – наличие многоточия «…». Оно говорит нам приблизительно следующее: «действуй по аналогии» или «заметь

закономерность и действуй согласно ей». Однако заметить, распознать закономерность – это очень сложное интеллектуальное действие и часто неоднозначное.

То, что установить закономерность – дело сложное, показывает такой пример: попробуйте угадать закономерность в следующей последовательности и продолжите её:

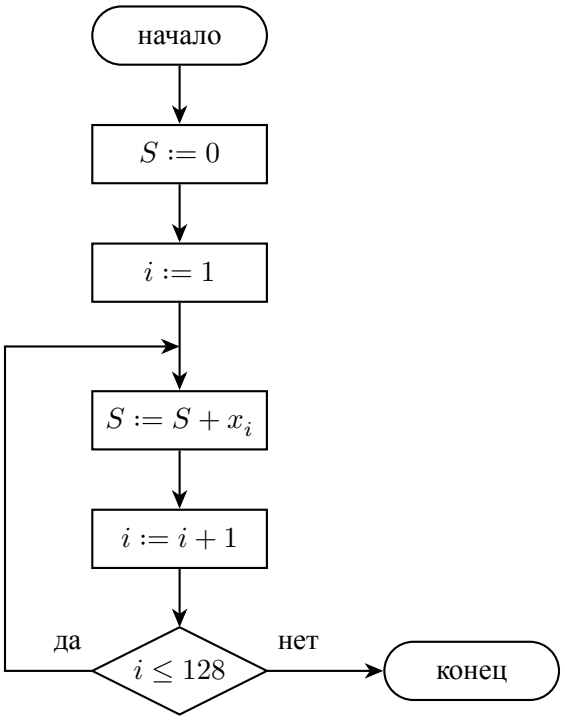
Р Д Т Ч …[1]

А на неоднозначность указывает то, что из выписанных пунктов 2, 3 и 129 можно вывести разные закономерности изменения индекса – не только 1, 2, 3, 4, 5, …, но и 1, 2, 4, 8, 16, … или ещё что-то иное.

А раз действие «угадай закономерность» сложное и неоднозначное, то писать «…» или слова «и т. д.» в алгоритмах категорически запрещено. Запомните это сразу.

Что же делать в нашем примере? Выход напрашивается сам собой: не надо выписывать сто раз однообразное действие $S := S + x_i$, а достаточно выписать его один раз, но сделать так, чтобы оно выполнялось при различных значениях i от 1 до 128.

В виде блок-схемы это записывается так:



На этом я закончу лекцию. В заключение отмечу лишь, что разработка, придумывание алгоритмов – вещь творческая, интеллектуальная и сродни сочинению литературных или музыкальных произведений. Запись же уже придуманных алгоритмов – вещь достаточно техническая, и здесь просто необходимо «набить руку».

[1] *Подсказка*: это первые буквы числительных раз, два, три, четыре, …

Лекция 2. МАШИНА ТЬЮРИНГА

План лекции:

1. Необходимость и способы уточнения понятия алгоритма

2. Машина Тьюринга

3. Возможности машины Тьюринга. Тезис Тьюринга

На предыдущей лекции мы рассмотрели интуитивное определение алгоритма. Как я говорил, это определение не строгое. Сегодня же мы начнем рассматривать уже точные определения алгоритма. Но прежде попытаемся понять, а зачем потребовалось уточнять понятие алгоритма и что, собственно, необходимо уточнять.

1. НЕОБХОДИМОСТЬ И СПОСОБЫ УТОЧНЕНИЯ ПОНЯТИЯ АЛГОРИТМА

1.1 Исторические предпосылки уточнения

Наряду с традиционным стремлением математиков уточнять всё, что попадает в поле их зрения, имелась и другая причина для уточнения понятия алгоритма. Суть её в следующем.

Разработка алгоритмов решения различных задач – одно из важнейших направлений деятельности математиков. За свою многовековую историю математика накопила алгоритмы решения многих задач. Но по мере усложнения рассматриваемых задач стали появляться такие, для которых никак не удавалось найти алгоритмы их решения. Постепенно складывалось мнение, что алгоритмов для решения этих задач не существует. Поэтому вместо поиска алгоритмов для таких задач стали стремиться доказать, что таковых не существует.

Примером такой задачи может служить следующая: можно ли построить алгоритм, на вход которого подаётся описание любой задачи, а он на выходе выдаёт способ (алгоритм) её решения?

описание задачи → | алгоритм | → *способ решения задачи*

Ясно, что существование такого алгоритма маловероятно – уж очень он хорош. Трудно предположить, что для самых разнообразных задач существует единый метод поиска их решения. Если бы существовал такой алгоритм, то нам особенно и делать-то было нечего. Например, захотел доказать теорему – подал на вход такого алгоритма описание теоремы, а он выдал бы её доказательство.

Другой, менее амбициозный пример: можно ли определить для любых двух алгоритмов, эквиваленты ли они или нет? Алгоритмы считаются эквивалентными, если при одинаковых исходных данных они дают одинаковые результаты. И эту задачу никак не удавалось решить, т. е. не удавалось придумать единый способ установления эквивалентности алгоритмов.

Итак, появились задачи, для которых по всей вероятности нельзя построить алгоритм их решения. Поэтому естественным образом встал вопрос о доказательстве несуществования этих алгоритмов. Но для этого, прежде всего, необходимо было уточнить собственно понятие алгоритма. Ведь одно дело – доказать существование алгоритма, для чего достаточно предъявить этот алгоритм, а чтобы убедиться в том, что это действительно алгоритм, достаточно и интуитивного определения алгоритма. И совсем другое дело, когда необходимо доказать, что какой-то алгоритм не существует: имея лишь расплывчатое определение алгоритма, доказать отсутствие алгоритма нельзя, т. к. всегда могут сказать, что в доказательстве рассмотрены не все варианты алгоритмов.

Таким образом, стремление доказать, что для некоторых задач не существует алгоритмов их решения, и было основной причиной уточнения понятия алгоритма.

Потребность в этом стала проявляться особенно остро в начале 20-го века. И вот в 30–50-х годах появилось сразу несколько уточнений понятия алгоритма. Два из них – машины Тьюринга и нормальные алгоритмы Маркова – мы рассмотрим подробно.

1.2 Способы уточнения алгоритмов

Но прежде давайте разберемся, а что собственно необходимо уточнять в понятии алгоритма и как это делается.

Если внимательно присмотреться к интуитивному определению алгоритма, то можно заметить, что уточнять надо главным образом две вещи – понятие правил, т. е. элементарных операций, из которых складывается алгоритм, и понятие объектов, с которыми алгоритм работает.

Уточнение правил

При уточнении понятия правила возникают следующие вопросы. Что является правилом, а что нет? Какие правила выполнимы и детерминированы, а какие нет? Ответить на эти вопросы можно по-разному. Обычно это делается так: ничего не определяют и не уточняют, а просто явно выписывают несколько правил и говорят: «вот эти правила можно использовать в алгоритмах, а другие – нельзя». (Здесь уместно привести следующую аналогию: определить, что такое «цифра» довольно сложно, но можно поступить так: явно выписать десять знаков 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 и сказать: «вот это цифры, а всё иное – не цифры».) При этом в качестве допустимых выбирают настолько простые и однозначно понимаемые правила, что в них никто не усомнится (в этом вы сами скоро убедитесь).

А какие конкретно правила выбирают? Это зависит от вкуса автора точного определения алгоритма. Однако этот выбор всё-таки не произволен: набор таких правил должен быть, как говорят, универсальным, т. е. таким, чтобы с их помощью можно было бы реализовать и все другие правила. Вот это и есть самое трудное в подборе допустимых правил.

Уточнение объектов

Теперь об уточнении понятия объектов, с которыми работают алгоритмы. Сложность здесь в том, что слишком разной природы могут быть те предметы, с которыми работают алгоритмы. Это и числа, это и функции (например, в алгоритмах нахождения производных), это и тексты (в алгоритмах редактирования), это и молекулы (в алгоритмах построения химических веществ), это и сигналы от приборов (в алгоритмах управления технологическими процессами) и т. д. Как же уточнить, стандартизовать всё это многообразие? Выход придумали достаточно простой и вполне естественный. Любой предмет мы считаем известным, если можем описать его словами, а словесное описание – это последовательность символов (букв, цифр, знаков препинания и т. п.). Поэтому вместо того, чтобы рассматривать сами предметы, можно рассматривать их описания, т. е. последовательности символов:

предмет $\rightarrow$ *описание* (*последовательность символов*)

Именно так и поступают при уточнении алгоритмов: объектами их работы считаются последовательности символов и только они. Например, в «химических» алгоритмах используются не сами молекулы, а их химические формулы: H_2O и т. п. В алгоритмах игры в шахматы вместо самих шахматных фигур и доски рассматривается запись расположения фигур на доске в шахматной нотации: Кр а2 (фигура и координаты поля) и т. д.

Отмечу, что в общем случае строго доказать справедливость такой замены предметов на последовательность символов нельзя, поскольку понятие «предмет» расплывчато. Однако практика подтверждает, что, когда надо, такую замену всегда удаётся сделать.

Таким образом, в уточненных определениях алгоритмов объектами их работы являются последовательности символов, а это понятие уже можно определить строго и точно, что мы сейчас и сделаем.

1) ***Символом*** будем называть любой печатный знак.
Это может быть буква, цифра, знак операции, знак препинания, скобки и т. п. Единственное требование к символам: одинаковые символы должны записываться одинаково, а различные – по-разному.

2) ***Алфавитом*** называется любое конечное множество символов.
Алфавит обычно записывается так: $A = \{a, b, +, 9, =\}$, т. е. в фигурных скобках через запятые перечисляются все входящие в него символы. При этом каждый символ указывается только один раз и порядок символов в этом перечне не играет никакой роли, например: $\{a, b\} = \{b, a\}$.

Какие символы включают в алфавит? Те, которые и только которые будут использоваться в рассматриваемой задаче для записи её объектов (исходных данных, промежуточных результатов, окончательного ответа).

3) ***Словом*** назовем конечную последовательность символов из алфавита.
Примеры слов: ab, $a{+}b{=}9$ и т. п. Здесь уже важен порядок следования символов (слово ab не совпадает с ba) и символы могут повторяться ($aabba$).

4) ***Длиной слова*** называется число символов в слове.
Так, длина слова $aabba$ равна 5.

5) На практике удобно ввести понятие ***пустого слова***, т. е. слова, не содержащего ни одного символа и имеющего длину 0. Для этого пустого слова обычно вводят специальное обозначение – Λ (читается «лямбда» – буква греческого алфавита).

Замечание. В нашем определении «слово» – это любая последовательность любых символов. Поэтому у нас *ФШХЦ* или $9a5$ – это слова, тогда как с точки зрения русского языка их не принято считать словами. Более того, если в алфавит включить символ «пробел» (часто обозначается ␣ — «корыто»), то любой текст становится словом в нашем понимании, например: *ЭТО*␣*СЛОВО*. Таким образом, наше определение слова отличается от обычного понимания. Оно более широкое и включает в себя не только обычные слова, но и любые предложения в смысле русского языка, записи химических формул, шахматных партий и т. д.

6) Введём теперь следующее обозначение:
A^* – множество всех слов, составленных из символов алфавита A.
Например, если $A = \{a, b\}$, то $A^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, ...\}$.

Определив понятие «слово», можно теперь сказать, что ***объектами алгоритма*** *являются слова в некотором заданном алфавите и только они*.

Отмечу важную вещь. У многих алгоритмов, как правило, имеется несколько исходных данных. Например, у алгоритма Евклида исходные данные – два натуральных числа. Однако во всех уточнениях алгоритма исходным данным может быть только одно слово. Это не существенное ограничение, поскольку все исходные данные можно выписать друг за другом, разделив их специальными символами (знаками) и всю эту запись рассматривать как единое слово. Например, на вход алгоритму Евклида можно дать такое слово: *45#30*.

Слово, которое подаётся на вход алгоритму, принято называть ***входным словом***. Аналогично, можно считать, что и результат у алгоритма всегда один. Если их несколько, то они объединяются в одно слово. Оно называется ***выходным словом***.

С учётом сказанного, во всех уточнениях алгоритм – это описание способа преобразования одного (входного) слова в другое (выходное) слово:

входное слово $\rightarrow$ [алгоритм] $\rightarrow$ *выходное слово*

Например: алгоритм Евклида преобразует слово *45#30* в слово *15*.

Итак, мы рассмотрели, как может быть уточнено понятие алгоритма. А теперь перейдём к конкретному уточнению, получившему название машины Тьюринга.

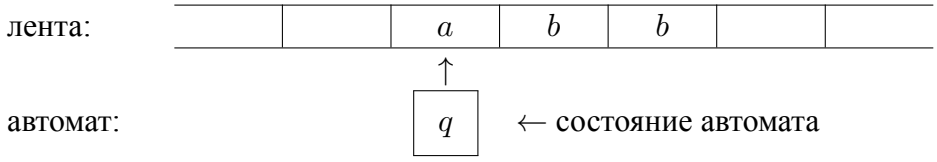
2. МАШИНА ТЬЮРИНГА

В 1936 году английский математик Алан Тьюринг предложил точное определение алгоритма, которое позже получило название *машина Тьюринга* (МТ). Почему именно машина? А потому, что А. Тьюринг придумал некоторую воображаемую машину и определил: то, что умеет делать эта машина, – это и есть алгоритмы, а что не умеет – это не алгоритмы.

Сразу отмечу, что это абстрактная, воображаемая машина, реализовать её на практике нельзя (почему – увидим чуть позднее). Кроме того, это очень примитивная, очень простая машина, применять которую для записи алгоритмов, использующихся на практике, очень неудобно. Всё дело в том, что А. Тьюринг преследовал теоретические цели, а не практические. Он хотел дать такое определение алгоритма, которое было бы удобным в теоретических исследованиях (например, удобным для доказательства существования или не существования алгоритмов, для анализа свойств алгоритмов). А в теоретических исследованиях, чем проще анализируемый объект, тем удобнее с ним работать. Поэтому-то А. Тьюринг и стремился к максимальному упрощению придуманной им машины. Так что не стоит удивляться сверхпростоте машины Тьюринга – это сделано специально и это оправдано. Что же касается уточнений понятия алгоритма, удобных для практических целей, то мы их рассмотрим позднее.

2.1 Структура машины Тьюринга

Машина Тьюринга состоит из двух компонентов – ленты и автомата:



Ленту можно представлять себе как обычную магнитофонную ленту, но бесконечную в обе стороны. (Именно из-за этой бесконечности и нельзя реализовать машину Тьюринга на практике!). Лента разбита на клетки, которые никак не перенумерованы и не именуются. В каждой клетке может быть записан ровно один символ или ничего не записано. Содержимое клетки может меняться – в неё можно записать другой символ или стереть находящийся там символ.

Для удобства пустое содержимое клетки будем называть пустым символом и обозначать специальным символом Λ («лямбда»). В связи с этим указанное содержимое ленты можно представлять себе и так:

	Λ	a	b	b	Λ	

Удобство здесь в том, что вместо длинной фразы «записать символ в клетку или стереть находящийся там символ» можно говорить просто «записать символ в клетку», подразумевая, что запись пустого символа – это стирание прежнего символа.

Автомат – это устройство, похожее на головку магнитофона. В каждый момент он размещается под одной из клеток ленты и видит её содержимое; это т. н. видимая клетка, а находящийся в ней символ – видимый символ. Кроме того, в каждый момент автомат находится в одном из т. н. состояний, которые принято обозначать буквой q с индексами: q_1, q_2 и т. п. Состояние – это нечто вроде канала телевизора; на каждом канале идёт своя передача. Так и у автомата: находясь в каком-то состоянии, он выполняется какую-то определенную операцию.

Пару видимый символ S и текущее состояние автомата q будем называть ***конфигурацией*** и обозначать $\{S, q\}$.

Если лента – это пассивная часть машины Тьюринга и используется для хранения информации, то автомат – это активная часть, именно он всё делает. Делать же он может лишь три простых действия:

1) записать в видимую клетку любой символ (изменить содержимое других клеток автомат не может);

2) сдвинуться на одну клетку влево или вправо, или остаться на месте (перепрыгнуть сразу через несколько клеток он не может);

3) перейти в новое состояние.

Вот только эти элементарные действия и умеет выполнять машина Тьюринга. Все более сложные действия должны быть так или иначе разложены на эти элементарные действия.

Ранее, говоря о способах уточнения понятия алгоритма, я отметил, что, определяя алгоритм, обычно явно перечисляют те правила, которые можно указывать в алгоритмах. Так вот, эти правила для машины Тьюринга я сейчас явно и перечислил. Как видно, что они действительно просты, выполнимы и однозначно понимаемы.

2.2 Такт работы машины Тьюринга

Вся работа машины Тьюринга делится на такты: вначале выполняется первый такт, затем второй такт и т. д. В каждом такте МТ, точнее автомат, всегда выполняет следующие три действия:

1) Записывает новый символ S' в «видимую» клетку.

2) Сдвигается на одну клетку влево (обозначение – L, от *left*), либо на одну клетку вправо (R, от *right*), либо остаётся неподвижным (N).

3) Переходит в новое состояние q'.

Замечание. Эти три действия всегда выполняются в указанном порядке.

Формально, действия одного такта записываются так:

S', [L,N,R], q'

где запись [L,N,R] означает, что в этом месте можно выписать любую из букв L, N или R.

Например, запись «$+$, L, q_8» означает, что в «видимую» клетку записывается символ $+$, после чего автомат сдвигается на одну клетку влево и переходит в состояние q_8.

И вот таким образом такт за тактом происходит работа машины Тьюринга. В целом же автомат, переключаясь с одного состояния на другое, «ползает» по ленте и меняет содержимое её клеток, стирая одни символы и записывая другие.

2.3 Программа для машины Тьюринга

Однако сама по себе машина Тьюринга ничего не делает. Чтобы заставить её работать, надо написать для неё программу, указав в ней, что машина должна делать в каждой возможной конфигурации. Такая программа записывается в виде следующей таблицы:

	S_1	S_2	...	S_i	...	S_n	Λ
Q_1							
...							
q_j				S', [L,N,R], q'			
...							
q_m							

Слева перечислены обозначения состояний, в которых может находиться автомат, сверху указаны символы, которые может видеть автомат, а на пересечениях (в ячейках таблицы) выписаны те такты, которые должен выполнить автомат, находящийся в соответствующем состоянии и обозревающий в «видимой» клетке ленты соответствующий символ.

Замечание. Сколько и какие именно символы и состояния указывать в таблице – определяет автор программы; главное, чтобы были перечислены все символы, которые могут появляться на ленте, и все состояния, в которых может находиться автомат. В частности, обязательно надо указывать колонку с пустым символом.

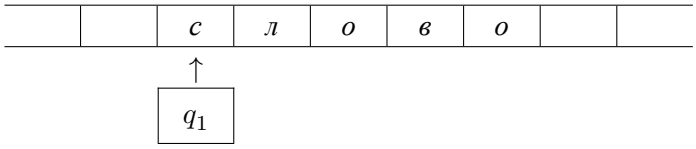
Таким образом, каждая ячейка этой таблицы определяет такт работы машины Тьюринга при одной конфигурации, а в целом таблица определяет поведение МТ при всех возможных конфигурациях. Тем самым, задав такую таблицу, мы полностью определяем повеление машины Тьюринга. *Описать алгоритм в виде МТ – значит предъявить такую таблицу*.[2]

2.4 Правила выполнения программы

Итак, пусть дана программа для машины Тьюринга. Как она выполняется?

1) Прежде всего надо записать на ленте входное слово, т. е. исходное данное для программы. Входное слово – это конечная последовательность символов, записанных в соседних клетках ленты. Внутри слова пустых клеток не должно быть, а слева и справа – только пустые клетки.

2) Надо установить автомат в состояние q_1 и подвести его под первый символ слова:



Если входное слово пустое, то автомат может смотреть в любую клетку, т. к. все они пусты.

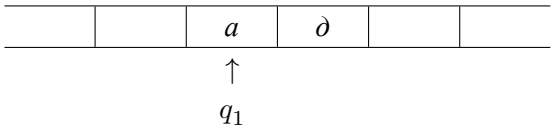
3) Начать выполнение программы: найти в таблице ячейку на пересечении первой строки (т. к. автомат находится в состоянии q_1) и столбца, соответствующего первому символу входного слова, и выполнить указанный такт. После выполнения этого такта автомат окажется в новой конфигурации. Теперь повторяем аналогичные действия для этой конфигурации, и т. д.

Пример.

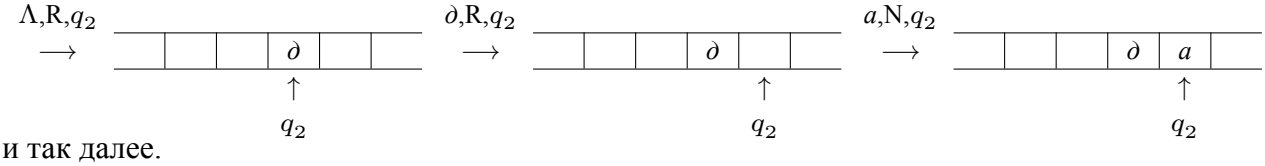
	a	∂	Λ
q_1	Λ, R, q_2	Λ, R, q_1	∂, N, q_1
q_2	a, N, q_2	∂, R, q_2	a, N, q_2

Выполним эту программу для входного слова *ад*.

Исходное состояние ленты и автомата:



Далее имеем:



и так далее.

Введём понятие ***такт останова***. Так называется такт, который ничего не меняет: автомат записывает тот же символ, что и был, остаётся в том же состоянии и не сдвигается, т. е. это такт S, N, q для конфигурации $\{S, q\}$. Например, для нашей конфигурации $\{a, q_2\}$ как раз и будет

[2] Часто машиной Тьюринга называют именно эту таблицу-программу, т. к. остальные элементы (лента и автомат) у всех машин Тьюринга одинаковые.

выполнен такт останова. Попав на такт останова, машина Тьюринга как бы замрёт на месте. Так вот, *по определению считается, что, попав на такт останова, МТ останавливается, завершает свою работу*.

В целом возможны два исхода работы машины Тьюринга по программе:

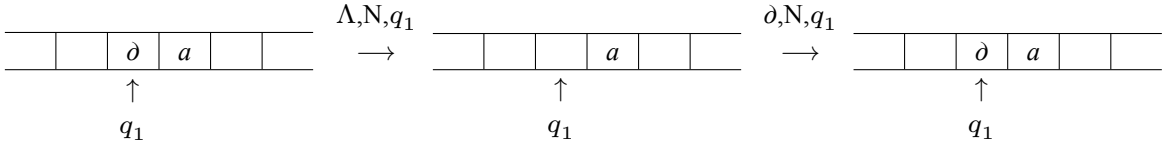
1) Первый исход – «хороший»: это когда в какой-то момент машина Тьюринга попадает на такт останова, т. е. останавливается. В таком случае говорят, что машина Тьюринга ***применима к заданному входному слову***. А то слово, которое в этот момент оказалось на ленте, является выходным словом, т. е. результатом работы машины Тьюринга. В нашем примере МТ применима к входному слову *ад* и результатом будет слово *да*.

Обязательные требования:

- внутри выходного слова не должно быть пустых клеток;
- в момент останова автомат обязан находиться под одним из символов выходного слова (под каким именно – не играет роли), а если оно пустое – под любой клеткой.

Замечание. Эти условия потребуются в дальнейшем для построения композиции машин Тьюринга.

2) Второй исход – «плохой»: это когда машина Тьюринга никак не останавливается, не попадает на такт останова (например, автомат на каждом шаге сдвигается вправо и потому не может остановиться, т. к. лента бесконечная). В этом случае считается, что машина Тьюринга ***неприменима к заданному входному слову***. Ни о каком результате при таком исходе не может идти и речи. Например, возьмём ту же программу, но зададим другое входное слово – *да*:



и т. д. Здесь автомат всё время будет то стирать букву *∂*, то записывать её снова и никогда не остановится.

Как видно, один и тот же алгоритм (программа МТ) может быть применим к одним входным словам (т. е. остановится) и неприменим к другим (т. е. зациклится). Таким образом, применимость/неприменимость алгоритма зависит не только от самого алгоритма, но и от исходных данных.

Важное замечание. Ранее мы говорили о таком свойстве алгоритмов, как конечность (результативность), т. е. о том, что алгоритм должен давать ответ через конечное число шагов. Почему же тогда мы допускаем зацикливание машины Тьюринга? Всё дело в том, что алгоритм обязан давать ответ через конечное число шагов только для «хороших» исходных данных – только для тех, которые относятся к допустимым исходным данным задачи. Если же мы подсунем алгоритму «плохие» исходные данные, то алгоритм уже не обязан останавливаться.

Действительно, почему алгоритм Евклида для нахождения наибольшего общего делителя должен остановиться, если ему дали на входе числа π и e? Ведь для произвольных действительных чисел понятие наибольшего общего делителя не определено.

А поскольку мы никак не ограничиваем входные слова для машины Тьюринга (входным может быть любое слово), то среди них могут быть и «плохие» слова, на которые алгоритм не рассчитан. А раз так, то и надо учитывать, что машина Тьюринга может зациклиться. Для «хороших» входных слов она должна останавливаться и давать ответ, а для «плохих» она не обязана этого делать.

2.5 Соглашения для сокращения записи

Договоримся о некоторых соглашениях, сокращающих запись программы МТ.

1) Если в такте не меняется символ, или не меняется состояние автомата, или автомат не сдвигается, то в соответствующей позиции такта мы не будем ничего не писать.

Например, при конфигурации $\{a, q\}$ следующие пары записей тактов эквивалентны:

a, R, $q_8 \equiv$, R, q_8 (но не Λ, R, q_8)
b, N, $q \equiv b$, ,
a, L, $q \equiv$, L,
a, N, $q \equiv$, , (это такт останова)

2) Если надо указать, что после выполнения некоторого такта машина Тьюринга должна остановиться, то в третьей позиции такта, где указывается новое состояние, будем писать «!». Например, такт «+, L, !» означает следующие действия: записать в «видимую» клетку «+», сдвинуться влево и остановиться. Такт останова « , , » запишется как « , , !».

Формально можно считать, что в программе имеется состояние с названием «!», все клетки которого – это такты останова:

q_m	…	…	…	…
!	, ,	, ,	…	, ,

однако такую строку явно не выписывают, а лишь подразумевают.

Оба этих соглашения необязательны, но они сокращают текст программы и делают её восприятие более лёгким. Например, с использованием сокращений рассмотренная программа будет выглядеть так:

	a	∂	Λ
q_1	Λ, R, q_2	Λ, ,	∂, ,
q_2	, , !	∂, R,	a, ,

3) И ещё одно соглашение, которое полезно для сокращения записи, но уже при описании задач. Мы будем использовать такие обозначения:

P – обозначение входного слова;

A – алфавит входного слова, т. е. набор тех символов, из которых и только которых может состоять P, т. е. $P \in A^*$ (однако в тексте программы могут встречаться и другие символы, т. к. по ходу выполнения программы на ленту можно записывать и иные символы).

2.6 Примеры программ для машины Тьюринга

Приведём два довольно несложных примера МТ, т. к. в мои цели не входит научить Вас писать сложные программы для машины Тьюринга. Этим Вы будете заниматься на семинарах.

Пример 1.

$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Следовательно, P – последовательность из десятичных цифр, т. е. запись неотрицательного целого числа в десятичной системе.

Требуется получить на ленте запись числа, которое на 1 больше чем P, т. е. $P \to P + 1$

Например:

		3	7			$\Longrightarrow$			3	8		
	1	4	9			$\Longrightarrow$		1	5	0		

Идея алгоритма:

1. Подогнать автомат под последнюю цифру числа.
2. Если эта цифра в диапазоне от 0 до 8, то заменить её цифрой, на 1 больше неё, и остановиться.
3. Если же это цифра 9, тогда заменить её на 0 и сдвинуться влево, после чего таким же способом увеличить на 1 предпоследнюю цифру.
4. Особый случай: в P только девятки (например, 999). Тогда автомат будет сдвигаться влево, записывая нули, и, в конце концов, окажется под пустой клеткой. Тогда в эту пустую клетку надо записать 1 и остановиться (результатом будет 1000).

В виде программы МТ эта идея реализуется так:

	0	1	2	3	4	5	6	7	8	9	Λ
q_1	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,L,q_2
q_2	1,,!	2,,!	3,,!	4,,!	5,,!	6,,!	7,,!	8,,!	9,,!	0,L,	1,,!

Пояснения.

1) q_1 – это состояние, в котором автомат «бежит» под последнюю цифру числа. Для этого он всё время движется вправо, какую бы цифру он ни видел. Но здесь есть одна тонкость: когда автомат находится под последней цифрой, то он ещё не знает об этом (ведь он не видит, что записано в соседних клетках) и узнает об этом, только когда проскочит её и попадёт под пустую клетку. Поэтому, дойдя до первой пустой клетки, автомат должен вернуться назад, встав под последнюю цифру, и перейти в состояние q_2 (больше вправо двигаться не надо).

2) q_2 – это состояние, в котором автомат прибавляет 1. Сначала автомат находится под последней цифрой. Если она в диапазоне от 0 до 8, то автомат заменяет её цифрой, на 1 больше её, и останавливается. Но если это 9, то автомат заменяет её на 0 и сдвигается влево, оставаясь в состоянии q_2. Тем самым, он теперь будет прибавлять 1 к предыдущей цифре. Если же автомат сдвинулся влево, а там нет цифры (а есть «пусто»), то он заносит сюда 1 и останавливается.

Пример 2.

Пусть $A = \{a, b, c\}$. Значит, входное слово P – это последовательность из букв a, b и c.

Требуется перенести первую букву P в конец слова.

Например:

	a	b	c	b		$\Longrightarrow$		b	c	b	a	

Идея алгоритма:

1. Запомнить первую букву, а затем стереть её.
2. Перегнать автомат под первую пустую клетку за словом.
3. Записать в неё запомненную букву.

Как бегать вправо, мы уже знаем из предыдущего примера. Здесь возникает вопрос – как запомнить первую букву? Ведь в машине Тьюринга нет никакого запоминающего устройства! Записывать же букву где-то на ленте бессмысленно: как только автомат сдвинется с клетки с этой буквой, он тут же забудет её. Как же запомнить первую букву?

Выход здесь таков – надо использовать разные состояния автомата. Если первая буква – a, то надо перейти в состояние q_2, в котором автомат бежит вправо и записывает в конце a. Если же первой была буква b, то надо перейти в состояние q_3, где всё то же самое, только в конце приписывается буква b. Если же первой была буква c, то надо перейти в состояние q_4, в котором автомат записывает в конце букву c. Следовательно, то, какую первую букву видим, мы фиксируем тем, что переводим автомат в разные состояния. Это типичный приём для МТ.

Программа:

	a	b	c	Λ
q_1	Λ, R, q_2	Λ, R, q_3	Λ, R, q_4	, , !
q_2	, R,	, R,	, R,	a, , !
q_3	, R,	, R,	, R,	b, , !
q_4	, R,	, R,	, R,	c, , !

Других примеров программ для машины Тьюринга я приводить не буду, Вы их будете рассматривать на семинарах. Мы же сейчас поговорим о возможностях машины Тьюринга.

3. ВОЗМОЖНОСТИ МАШИН ТЬЮРИНГА. ТЕЗИС ТЬЮРИНГА

Уже из приведённых примеров видно, что описывать алгоритмы в виде программ для машины Тьюринга – дело очень сложное и требует изощренного ума. Это объясняется тем, что уж слишком примитивна машина Тьюринга, умеет выполнять всего лишь три простейшие операции: записать символ, сдвинуться влево или вправо на одну клетку, перейти в новое состояние.

Естественно возникает вопрос: а многое ли можно сделать с помощью машины Тьюринга, т. е. сколь велики её возможности? Чтобы ответить на этот вопрос рассмотрим сначала ряд свойств машины Тьюринга.

3.1 Композиция алгоритмов

Пусть $\mathcal{A}$ и $\mathcal{B}$ – любые алгоритмы. Тогда их ***композицией***, обозначаемой как $\mathcal{A} \circ \mathcal{B}$, называется последовательное выполнение сначала алгоритма $\mathcal{B}$, а затем алгоритма $\mathcal{A}$. Если обозначить через $\mathcal{C}(P)$ результат применения алгоритма $\mathcal{C}$ к слову P, то композиция алгоритмов $\mathcal{A}$ и $\mathcal{B}$ определяется так ($\forall$ читается как «для всякого»):

$\forall$ слова $P : \mathcal{A} \circ \mathcal{B}(P) = \mathcal{A}(\mathcal{B}(P))$

т. е. к слову P применяется сначала алгоритм $\mathcal{B}$, а затем к полученному результату применяется алгоритм $\mathcal{A}$. (Аналогия из математики: $\sin(\operatorname{tg} x)$.)

Замечание. Если хотя бы один из алгоритмов $\mathcal{A}$ или $\mathcal{B}$ зацикливается, то и алгоритм $\mathcal{C}$ должен зацикливаться.

Вопрос: если $\mathcal{A}$ и $\mathcal{B}$ – программы для машины Тьюринга (т. е. алгоритмы в смысле Тьюринга), то является ли композиция $\mathcal{A} \circ \mathcal{B}$ программой для машины Тьюринга?

Ответ: да, является. Более точно, верна следующая

Теорема: По любым программам $\mathcal{A}$ и $\mathcal{B}$ для машины Тьюринга можно построить такую программу $\mathcal{C}$ для машины Тьюринга, что $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$.

Доказательство: Для простоты считаем, что $\mathcal{A}$ и $\mathcal{B}$ имеют одинаковый алфавит.

Пусть тексты программ алгоритмов $\mathcal{A}$ и $\mathcal{B}$ таковы:

$\mathcal{A}$:

	S_1	…	S_k	Λ
q_1				
…				
q_n				

$\mathcal{B}$:

	S_1	…	S_k	Λ
q_1				
…				
q_m				

Тогда программа $\mathcal{C}$ строится следующим образом:

1. В $\mathcal{A}$ переобозначаем все состояния q_i в p_i (от этого ничего не изменится)
2. В тексте $\mathcal{B}$ все упоминания ! заменим на p_0

3. Составляем следующую таблицу, выписывая $\mathcal{A}$ под $\mathcal{B}$ и добавляя новое состояние p_0:

	S_1	…	S_k	Λ	
q_1					} Текст алгоритма $\mathcal{B}$ с заменой ! на p_0
…					
q_m					
p_0	,L,	…	,L,	,R,p_1	Влево под 1-й символ
p_1					} Текст алгоритма $\mathcal{A}$ с заменой q_i на p_i
…					
p_n					

Так вот, это и есть нужная нам программа $\mathcal{C}$. Действительно, сначала работает верхняя часть таблицы, т. е. программа $\mathcal{B}$, и получается слово $\mathcal{B}(P)$. Но вместо того, чтобы остановиться, мы теперь перейдём в состояние p_0, а в нём автомат движется под первую букву слова $\mathcal{B}(P)$, после чего переходит в состояние p_1.

Замечание. Такой перегон под первую букву необходим, т. к. мы договорились, что все программы начинают выполняться с конфигурации, в которой автомат находится под первой буквой входного слова. Кроме того, мы договорились, что по окончании работы автомат обязательно находится под одной из букв выходного слова, поэтому двигаться к началу этого слова надо справа налево – собственно для этого и нужна эта договорённость, иначе бы мы не знали, куда двигаться.

Итак, автомат видит первую букву слова $\mathcal{B}(P)$ и находится в состоянии p_1. Теперь начинает работать нижняя часть таблицы, т. е. программа $\mathcal{A}$.

Таким образом, действительно сначала проработает $\mathcal{B}$ и получится слово $\mathcal{B}(P)$, а потом к нему применится программа $\mathcal{A}$. Следовательно, $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$. Доказательство окончено.

Простейший пример. Пусть входной алфавит для алгоритмов $\mathcal{A}$ и $\mathcal{B}$ – это $A = \{a, b\}$ и пусть алгоритм $\mathcal{A}$ добавляет в конец входного слова букву a, алгоритм $\mathcal{B}$ заменяет все a на b:

$\mathcal{A}$:

	a	b	Λ
q_1	,R,	,R,	a,,!

$\mathcal{B}$:

	a	b	Λ
q_1	b,R,	,R,	,L,!

Тогда программа для алгоритма $\mathcal{C}$, реализующего композицию $\mathcal{A} \circ \mathcal{B}$, выглядит так:

$\mathcal{C}$:

	a	b	Λ	
q_1	b,R,	,R,	,L,p_0	программа для $\mathcal{B}$
p_0	,L,	,L,	,R,p_1	сдвиг влево под 1-й символ
p_1	,R,	,R,	a,,!	программа для $\mathcal{A}$

Отмечу, что в приведённом доказательстве дана схема построения программы для $\mathcal{C}$, пригодная для любых программ $\mathcal{A}$ и $\mathcal{B}$. Однако это не значит, что данная программа для $\mathcal{C}$ будет наилучшей, оптимальной. Например, в нашем конкретном случае композицию алгоритмов $\mathcal{A}$ и $\mathcal{B}$ можно реализовать и проще:

$\mathcal{C}$:

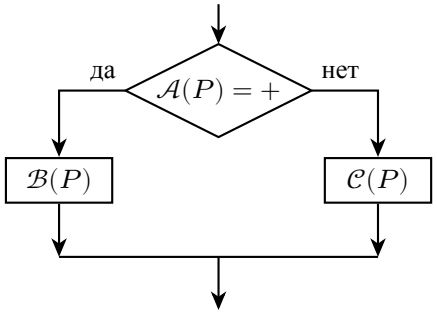
	a	b	Λ
q_1	b,R,	,R,	a,,!

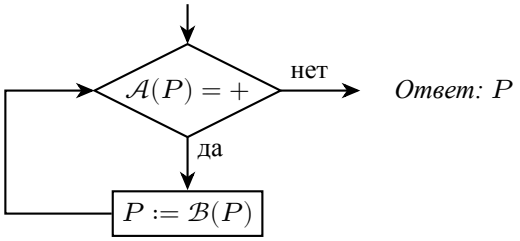
Итак, мы научились строить композицию любых двух машин Тьюринга. Что можно сказать о других способах комбинирования МТ?

3.2 Другие комбинации машин Тьюринга

Аналогично, но более сложно доказывается существование программ и для других комбинаций уже известных программ:

1. *Разветвление*. Если применение алгоритма $\mathcal{A}$ к слову P даёт ответ «+», то применить алгоритм $\mathcal{B}$ к слову P, иначе применить алгоритм $\mathcal{C}$ к слову P (*см. рис. слева*).





2. *Цикл*. Если $\mathcal{A}(P)$ равно «+», то применить алгоритм $\mathcal{B}$ к слову P, полученный результат считать за новое слово P и всё повторить заново. И так до тех пор, пока $\mathcal{A}(P)$ не станет отличным от «+» (*см. рис. справа*).

3.3 Тезис Тьюринга

О чём говорят эти свойства машин Тьюринга? О том, что если мы имеем программы для выполнения каких-либо действий, то их можно объединять в более сложные программы, которые решают более сложные задачи. Например, имея программу прибавления 1 к целому числу и программу вычитания 1 из целого числа (она строится аналогично), можно написать программу сложения любых целых чисел, для чего надо в цикле вычитать 1 из одного числа и прибавлять 1 к другому числу, а далее уже можно определить программу умножения любых целых чисел, для чего надо зациклить сложение. Аналогично, зациклив вычитание, можно построить программу деления любых целых чисел. Имея же эти программы, можно уже написать программы, которые решают уравнения и многое-многое другое.

Таким образом, несмотря на свою примитивность, машина Тьюринга умеет делать очень многое. Более того, практика показывает, что любой алгоритм можно записать в виде программы для машины Тьюринга. Были попытки придумать алгоритм, который нельзя было бы записать в виде программы МТ, но всякий раз оказывалось, что такой алгоритм сочинить не удаётся.

Всё это позволило А. Тьюрингу выдвинуть следующую гипотезу об универсальности машины Тьюринга:

Тезис Тьюринга: *Любой алгоритм можно представить в виде программы для машины Тьюринга.*

Доказать этот тезис нельзя, т. к. в нём есть неточное понятие «алгоритм» (в интуитивном понимании). Однако имеются важные доводы в его пользу:

1. Как я уже сказал, никому так и не удалось придумать алгоритм в интуитивном понимании, который нельзя было бы представить в виде программы для МТ. То есть никто не построил контрпримера.

2. Я также сказал, что существует несколько уточнений понятия алгоритма, Все они основаны на разных идеях, но оказалось, что все эти определения эквивалентны, т. е. если существует алгоритм согласно одному из этих определений, то существует эквивалентный ему алгоритм и согласно всем другим определениям. Причем этот факт уже строго доказан (т. к. здесь уже используются точные определения). Всё это свидетельствует о том, что во всех этих уточнениях схвачена суть интуитивного понятия алгоритма, что машину Тьюринга действительно можно считать правильным уточнением понятия «алгоритм».

Если согласиться с тезисом Тьюринга, тогда из него вытекает важное следствие: если удалось доказать, что не существует программы МТ для решения какой-либо задачи, то тем самым доказано, что такого алгоритма не существует вообще. А поскольку машина Тьюринга – это точное понятие, то с ним уже можно работать строгими, точными методами.

Лекция 3. НОРМАЛЬНЫЕ АЛГОРИТМЫ МАРКОВА

План лекции:

1. Нормальные алгоритмы Маркова

2. Алгоритмически неразрешимые проблемы

1. НОРМАЛЬНЫЕ АЛГОРИТМЫ МАРКОВА

На прошлой лекции мы рассмотрели одно из уточнений понятия алгоритма – машину Тьюринга. Сегодня мы рассмотрим другое уточнение, которое в начале 50-х годов XX века предложил советский математик Андрей Марков – нормальные алгоритмы. Нормальные алгоритмы Маркова (НАМ) основаны на иных идеях, нежели машина Тьюринга, но уровень их простоты – такой же, поскольку А. Марков также стремился дать такое определение алгоритма, которое было бы удобным в теоретических исследованиях, чем проще – тем лучше. В практических же целях, т. е. для записи решения задач, встречающихся на практике, НАМ не применяются.

Интересной особенностью нормальных алгоритмов Маркова является то, что в них допускается только одно элементарное действие – так называемая подстановкаn(тогда как в машине Тьюринга – три элементарных действия). Однако и этого оказалось достаточным, чтобы получились универсальные алгоритмы. Поэтому в первую очередь рассмотрим эти подстановки.

1.1 Подстановки

Формулой подстановки называется запись вида $\alpha \to \beta$, где α и β – слова из некоторого алфавита (возможно, и пустые). При этом α называется левой частью формулы подстановки, а β – правой частью.

Сама ***подстановка*** (как действие) задается формулой и применяется к некоторому слову P. Суть операции сводится к следующему: в слове P отыскивается часть, совпадающая с левой частью формулы подстановки (т. е. со словом α), и она заменяется на правую часть (т. е. на слово β):

$$P = \boxed{//}\boxed{\alpha}\boxed{\backslash\backslash} \quad \Longrightarrow \quad \boxed{//}\boxed{\beta}\boxed{\backslash\backslash}$$

При этом части слова P слева и справа от α не меняются. Получившееся слово называют результатом подстановки.

Пример: Формула: *ап* $\to$ *олк*

Слово *P*: *папа*

Результат: *полка*

Для наглядности я подчеркнул ту часть слова P, которая совпадает с левой частью формулы подстановки.

Отмечу, что разница между подстановкой и формулой подстановки – такая же, как и между сложением и знаком «+»: сложение – это операция, а «+» – это обозначение данной операции. Аналогично, подстановка – это действие, а формула подстановки – обозначение этого действия.

Теперь кое-что уточним.

1. Если левая часть формулы подстановки входит в слово P, то говорят, что формула ***применима*** к P. Но если α не входит в слово P, то формула считается ***неприменимой*** к P, и подстановка не выполняется.

Например, наша формула подстановки неприменима к слову *карп*. Отмечу при этом, что символы из левой части формулы подстановки должны подряд входить в слово P. Например, буквы *а* и *п* входят в слово *карп*, но не подряд, а в разбивку; это не считается вхождением.

2. Если α входит в слово P несколько раз, то, по определению, заменяется только первое вхождение α в P.

Пример: Формула: *ма* $\to$ *я*
Слово *P*: *мама*
Результат: *яма*

3. Если правая часть формулы подстановки – пустое слово, то подстановка «$\alpha \to$ » сводится к вычёркиванию части α из P. Отмечу, что в формулах подстановки не принято обозначать пустое слово как Λ.

Пример: Формула: *+b* $\to$
Слово *P*: *a+b+c*
Результат: *a+c*

4. Пусть в левой части формулы подстановки находится пустое слово. Встаёт вопрос: как выполняется подстановка « $\to \beta$»? По определению считается, что между двумя любыми символами слова, а также справа и слева от него находятся пустые слова. Таким образом: abb – это то же самое, что $\Lambda a \Lambda b \Lambda b \Lambda$. А раз так, то согласно общему правилу на правую часть должна заменяться самая левая «пустышка», т. е. та, что предшествует первой букве. Следовательно, подстановка по формуле « $\to \beta$» – это приписывание β слева к слову.

Пример: Формула: $\to$ *при*
Слово *P*: *сказка*
Результат: *присказка*

Прошу Вас сразу запомнить следующий важный факт: формула подстановки с пустой левой частью применима к любому слову.

Итак, что такое подстановка, мы узнали. Теперь рассмотрим, что такое НАМ в целом.

1.2 Определение НАМ и правила его выполнения

Нормальным алгоритмом Маркова называется конечный упорядоченный набор формул подстановки:

$$\begin{cases} \alpha_1 \to \beta_1 \\ \alpha_2 \to \beta_2 \\ \cdots\cdots\cdots \\ \alpha_k \to \beta_k \end{cases} \qquad k \geq 1$$

Замечание. Фигурную скобку слева от НАМ можно и не ставить, но с нею нагляднее.

В этих формулах могут использоваться два вида стрелок: обычная стрелка ($\to$) и стрелка «с хвостиком» ($\mapsto$). формула подстановки с обычной стрелкой называется обычной формулой, а со стрелкой «с хвостиком» – ***заключительной*** формулой. В чём разница между этими формулами, мы сейчас увидим.

Пример НАМ:

$$\begin{cases} aa \to a \\ ab \to ba \\ bb \mapsto \\ c \ \ \to cc \end{cases}$$

Записать алгоритм в виде НАМ – значит предъявить такой набор формул.

Теперь рассмотрим правила выполнения НАМ.

1. Прежде всего, задаётся некоторое входное слово P. Где именно оно записано – не важно, в НАМ этот вопрос не оговаривается.

2. Формулы подстановки в НАМ просматриваются сверху вниз и выбирается первая из формул, применимая ко входному слову P, т. е. первая из тех, левая часть которой входит в P. К слову P применяется подстановка, определяемая этой формулой. Получается новое слово P'.

3. Теперь слово P' берётся за исходное и к нему применяется та же самая процедура, т. е. снова сверху вниз просматриваются формулы НАМ, начиная с верхней, и ищется первая формула, применимая к слову P', выполняется соответствующая подстановка и получается новое слово P''. И так далее: $P \Longrightarrow P' \Longrightarrow P'' \Longrightarrow ...$

Обращаю особое внимание: на каждом шаге формулы в НАМ всегда просматриваются, начиная с самой верхней.

Например, выполнение нашего алгоритма для входного слова $abbaa$ будет происходить так (над стрелками указаны номера применённых формул подстановки):

$$abb\underline{aa} \overset{1}{\Longrightarrow} \underline{ab}ba \overset{2}{\Longrightarrow} b\underline{ab}a \overset{2}{\Longrightarrow} bb\underline{aa} \overset{1}{\Longrightarrow} \underline{bb}a \overset{3}{\longmapsto} a$$

Теперь займёмся уточнениями.

1) Если на очередном шаге была применена обычная формула подстановки ($\alpha \to \beta$), то процесс применения НАМ продолжается.

2) Если же на очередном шаге была применена заключительная формула ($\alpha \mapsto \beta$), то после её применения работа алгоритма прекращается. То слово, которое получилось в этот момент, и есть выходное слово, т. е. результат применения НАМ ко входному слову P. Именно этот случай и произошёл у нас в примере: к слову bba мы применили заключительную формулу подстановки «$bb \mapsto$ », поэтому, выполнив подстановку, мы остановились. Выходное слово – a.

Таким образом, разница между обычной и заключительной формулами подстановки в том, что после применения обычной формулы работа НАМ продолжается, а после применения заключительной формулы – прекращается.

А так, и та и другая есть формулы подстановки.

3) Если на очередном шаге к текущему слову неприменима ни одна формула подстановки, то и в этом случае работа НАМ прекращается и ответом считается текущее слово.

Пример. Возьмём алгоритм из предыдущего примера, но в качестве входного возьмём иное слово – $baab$. Тогда имеем:

$$b\underline{aa}a \overset{1}{\Longrightarrow} b\underline{aa} \overset{1}{\Longrightarrow} ba$$

Так как к слову ba не применима ни одна формула подстановки, то НАМ останавливается; выходное слово – ba.

Таким образом, НАМ **останавливается** по двум причинам: ***либо была применена заключительная формула, либо ни одна из формул подстановки не подошла***.

То и другое считается «хорошим» окончанием работы алгоритма. В обоих случаях говорят, что НАМ ***применим*** *ко входному слову*.

4) Однако может случиться и так, что НАМ никогда не остановится. Тогда говорят, что он ***не применим*** *ко входному слову*. В этом случае ни о каком результате нет и речи.

Пример. Пусть алгоритм всё тот же, а входное слово – это bac. Тогда имеем:

$$ba\underline{c} \overset{4}{\Longrightarrow} ba\underline{c}c \overset{4}{\Longrightarrow} ba\underline{c}cc \overset{4}{\Longrightarrow} ba\underline{c}ccc \overset{4}{\Longrightarrow} ... \text{ и т. д.}$$

Значит, наш алгоритм неприменим к слову bac.

1.3 Примеры нормальных алгоритмов Маркова

Теперь рассмотрим несколько примеров записи алгоритмов в виде НАМ. Напомню, что в наших обозначениях A – это алфавит входного слова, т. е. набор символов, которые и только которые могут входить во входное слово, а P – само входное слово.

Пример 1. $A = \{a, b, c\}$. Требуется вместо P получить столько палочек, сколько раз буква c входит в P. Например: $abccac \Longrightarrow |||$.

Идея решения: стереть все a и b, а каждое c заменить на $|$. Это дает такой алгоритм:

$$\begin{cases} a \to \\ b \to \\ c \to | \end{cases}$$

Проверим этот алгоритм на входном слове $abccac$:

$$\underline{a}bccac \overset{1}{\Longrightarrow} bcc\underline{a}c \overset{1}{\Longrightarrow} \underline{b}ccc \overset{2}{\Longrightarrow} \underline{c}cc \overset{3}{\Longrightarrow} |\underline{c}c \overset{3}{\Longrightarrow} ||\underline{c} \overset{3}{\Longrightarrow} |||$$

Останов произошёл из-за того, что неприменима ни одна формула подстановки.

Пример 2. $A = \{a, b, *\}$. Пусть P имеет вид $*Q$, где Q – любое слово из букв a и b. Требуется получить слово $Q*$, т. е. надо перегнать звёздочку в конец входного слова: $*Q \Longrightarrow Q*$. Конкретный пример: $*abb \Longrightarrow abb*$.

Идея решения: будем перетаскивать $*$ постепенно через каждую букву:

$*abb \Longrightarrow a*bb \Longrightarrow ab*b \Longrightarrow abb*$

Как сделать этот перегон с помощью подстановок? Достаточно заметить такой факт: «перепрыгивание» звездочки через какую-то букву ξ – это замена сочетания $*\xi$ на сочетание $\xi*$. Поэтому наш НАМ выглядит так:

$$\begin{cases} *a \to a* \\ *b \to b* \end{cases}$$

Проверим этот НАМ на входном слове abb:

$$\underline{*a}bb \overset{1}{\Longrightarrow} a\underline{*b}b \overset{2}{\Longrightarrow} ab\underline{*b} \overset{2}{\Longrightarrow} abb*$$

Опять же мы здесь остановились, поскольку не подошла ни одна формула подстановки.

Этот пример – вспомогательный, он нужен для решения следующей задачи.

Пример 3. $A = \{a, b\}$. Требуется приписать в конец входного слова P букву a. Например: $abb \Longrightarrow abba$.

Сразу отмечу, что формула вида «$\to a$» не подходит, так как она приписывает букву a в начало P, а не в конец. Чтобы что-то сделать в конце слова, надо этот конец как-то пометить, например, звёздочкой. Для этого введем звёздочку в начало P, затем перегоним её в конец слова P и, наконец, заменим её на a:

$$P \Longrightarrow *P \Longrightarrow P* \Longrightarrow Pa$$

Реализуем эту идею в виде такого НАМ:

$$\begin{cases} \to * \\ *a \to a* \\ *b \to b* \\ * \to a \end{cases}$$

Проверим на входном слове aab:

$$\_abb \overset{1}{\Longrightarrow} \_*abb \overset{1}{\Longrightarrow} \_**abb \overset{1}{\Longrightarrow} \_***abb \overset{1}{\Longrightarrow} \ldots$$

Мы зациклились. Почему? Все дело в первой формуле с пустой левой частью. Напомню, что такая формула применима к любому слову и что формулы в списке всегда просматриваются, начиная с самой верхней, потому наша первая формула всегда срабатывает и не «пускает» нас к другим формулам.

Этот пример показывает, насколько важен в НАМ порядок расположения формул подстановки. В частности, формула с пустой левой частью может располагаться только в самом конце списка. Запомните это сразу!

Учитывая это, переносим первую формулу в конец:

$$\begin{cases} *a \to a* \\ *b \to b* \\ * \to a \\ \to * \end{cases}$$

Снова проверим на слове abb:

$$\_abb \overset{4}{\Longrightarrow} \underline{*a}bb \overset{1}{\Longrightarrow} a\underline{*b}b \overset{2}{\Longrightarrow} ab\underline{*b} \overset{2}{\Longrightarrow} abb\underline{*} \overset{3}{\Longrightarrow} abba$$

Казалось бы, всё сделали и надо останавливаться. Однако у нас в последний раз применилась обычная формула подстановки, поэтому, согласно правилам, работа алгоритма продолжается – снова просматриваются сверху вниз формулы подстановки, снова выбирается первая применимая формула и т. д.:

$$\ldots \overset{3}{\Longrightarrow} \_abba \overset{4}{\Longrightarrow} \underline{*a}bba \overset{1}{\Longrightarrow} a\underline{*b}ba \overset{2}{\Longrightarrow} \ldots$$

В чём дело? А в том, что, заменив $*$ на a, надо было остановить работу алгоритма, а мы этого не сделали. Таким образом, третья формула обязана быть заключительной: $* \mapsto a$. Таким образом, правильный НАМ следующий:

$$\begin{cases} *a \to a* \\ *b \to b* \\ * \mapsto a \\ \to * \end{cases}$$

Этот пример показывает, как важно вовремя остановить алгоритм.

Прежде чем привести следующий пример, я хочу рассказать об одном важном приёме, который используется во многих нормальных алгоритмах. В нашем последнем примере использовался т. н. спецзнак: у нас это была звёздочка, но это может быть и любой другой символ – квадратик, кружочек и т. д. На семинарах Вы увидите, что такие спецзнаки в НАМ встречаются сплошь и рядом, и что без них никак не обойтись. Так вот, я хочу объяснить назначение таких спецзнаков.

Пусть в обрабатываемое нами слово P подслово α входит несколько раз:

P	…	α	…	α	…	α	…

и нам надо заменить одно из этих вхождений α на слово β. Такая замена делается с помощью формулы $\alpha \to \beta$. Однако, если мы применим эту формулу к слову P, то будет заменено первое вхождение α. А что делать, если надо заменить какое-другое вхождение α, скажем второе или последнее? Вот здесь-то на помощь и приходит спецзнак, скажем звёздочка: её надо каким-то образом поместить около нужного нам вхождения α (слева или справа от него – не важно).

Пусть нас интересует последнее вхождение α и пусть звёздочку мы поместили справа от него:

P	…	α	…	α	…	$\alpha*$	…

Зачем это надо? А затем, что звёздочка делает данное вхождение α уникальным: из всех вхождений α только у этого вхождения рядом находится звёздочка. Поэтому, если применим формулу $\alpha* \to \beta$, то на β заменится именно это вхождение α, а не какое-то другое.

Итак, *роль спецзнаков – выделить интересующее нас вхождение некоторой части обрабатываемого слова из числа других таких же частей*.

Воспользуемся этим для решения следующей задачи.

Пример 4. $A = \{a, b, c\}$. Требуется удвоить последнее вхождение буквы a в P. (Если a не входит в P, то ничего не менять.)

Идея решения. Удваивание буквы a реализуется формулой $a \to aa$. Но чтобы она применялась не к первому вхождению буквы a, а к последнему, надо поставить, скажем, справа от последней буквы a спецзнак, например, звёздочку, и применить заключительную формулу $a* \mapsto aa$. Формула должна быть заключительной, поскольку после такой замены надо прекратить работу алгоритма.

Теперь посмотрим, как поместить звёздочку около последнего вхождения буквы a. Поскольку последнее вхождение – это первое вхождение при движении справа налево, то предлагается поставить звёздочку в начало слова, затем перегнать её в конец слова (это мы уже умеем делать), а далее будем перегонять звёздочку влево до ближайшей буквы a.

Кроме того, надо учесть, что в P может и не быть буквы a. Тогда после того, как звёздочка добежит справа налево от конца до начала слова, надо просто уничтожить её и остановиться.

В итоге получается следующий НАМ:

$$\left\{\begin{array}{lll}
a \to a & (1) & \left.\begin{array}{l}\\\\\end{array}\right] \\
b \to b & (2) & \left.\begin{array}{l}\\\\\end{array}\right] \text{ – перегон } * \text{ вправо} \\
c \to c & (3) & \left.\begin{array}{l}\\\\\end{array}\right] \\
b* \to *b & (4) & \left.\begin{array}{l}\\\end{array}\right] \\
c* \to *c & (5) & \left.\begin{array}{l}\\\end{array}\right] \text{ – перегон } * \text{ влево до } a \\
a* \mapsto aa & (6) & \text{ – замена и останов} \\
* \mapsto & (7) & \text{ – останов, если нет } a \\
\to * & (8) &
\end{array}\right.$$

Давайте проверим этот алгоритм на входном слове $abacb$:

$$abacb \overset{8}{\Longrightarrow} *abacb \overset{1,2,3}{\Longrightarrow} \ldots \Longrightarrow abacb* \overset{4}{\Longrightarrow} abac*b \overset{2}{\Longrightarrow} abacb* \overset{4}{\Longrightarrow} abac*b \text{ (!!)}$$

Вместо того чтобы двигаться влево до ближайшей буквы a, звёздочка начала «прыгать» вокруг последней буквы слова. Почему? Дело в том, что формулы $*\xi \to \xi*$[3] перегона звёздочки вправо мешают формулам $\xi* \to *\xi$ перегона звёздочки влево. Отмечу, что перестановка этих групп формул не поможет. Что делать?

Ошибка у нас произошла из-за того, что мы используем символ $*$ в двух разных целях – и чтобы двигаться вправо и чтобы двигаться влево. Так вот, чтобы не было этой ошибки, надо просто ввести ещё один спецзнак, скажем $\#$, распределив между двумя этими спецзнаками обязанности: решётка будет двигаться вправо, а звёздочка– влево. Появиться же звёздочка должна, когда решётка дойдет до конца слова: сначала перед словом поставим решётку; пробежим символом $\#$ до конца слова; преобразуем его в $*$; далее звёздочка выполнит свою работу – пробежит влево до a, удвоит её и завершит работу алгоритма.

Если так и сделать, то получим такой НАМ:

$$\left\{\begin{array}{lll}
\#a \to a\# & (1) & \left.\begin{array}{l}\\\\\end{array}\right] \\
\#b \to b\# & (2) & \left.\begin{array}{l}\\\\\end{array}\right] \text{ – перегон } \# \text{ вправо} \\
\#c \to c\# & (3) & \left.\begin{array}{l}\\\\\end{array}\right] \\
\# \to * & (4) & \text{ – замена } \# \text{ на } * \text{ в конце слова} \\
b* \to *b & (5) & \left.\begin{array}{l}\\\end{array}\right] \\
c* \to *c & (6) & \left.\begin{array}{l}\\\end{array}\right] \text{ – перегон } * \text{ влево до } a \\
a* \mapsto aa & (7) & \text{ – замена и останов} \\
* \mapsto & (8) & \text{ – если нет } a \\
\to \# & (9) &
\end{array}\right.$$

[3] Символ ξ изображает произвольную букву алфавита, так что запись $*\xi \to \xi*$ соответствует трём правилам, по одному правилу для каждой буквы алфавита, без конкретизации порядка правил.

Давайте проверим этот алгоритм на том же входном слове $abacb$:

$$abacb \overset{9}{\Longrightarrow} \#abacb \Longrightarrow \overset{1,2,3}{\ldots} \Longrightarrow abacb\# \overset{4}{\Longrightarrow} abacb* \overset{5}{\Longrightarrow} abac*b \overset{6}{\Longrightarrow} aba*cb \overset{7}{\Longmapsto} abaacb$$

Если же входное слово не содержит букву a, то алгоритм будет выполняться так:

$$cb \overset{9}{\Longrightarrow} \#cb \Longrightarrow \overset{2,3}{\ldots} \Longrightarrow cb\# \overset{4}{\Longrightarrow} cb* \overset{5}{\Longrightarrow} c*b \overset{6}{\Longrightarrow} *cb \overset{8}{\Longmapsto} cb$$

Итак, теперь всё правильно.

На этом я закончу рассматривать примеры нормальных алгоритмов. Другие примеры Вы рассмотрите на семинарах.

1.4 Принцип нормализации

Рассмотрим теперь возможности нормальных алгоритмов Маркова. Как и в случае машины Тьюринга, для нормальных алгоритмов можно доказать, что такие объединения НАМ, как композиция, разветвление и цикл, снова являются нормальными алгоритмами. Следовательно, из простых нормальных алгоритмов Маркова мы можем строить сложные нормальные алгоритмы. Более того, как и в случае с машиной Тьюринга, для любого алгоритма (в интуитивном понимании) удаётся написать эквивалентный ему нормальный алгоритм Маркова, т. е. представить соответствующий список формул подстановок. В этой связи А. Марков выдвинул следующую гипотезу об универсальности НАМ:

Принцип нормализации (тезис Маркова): *Любой алгоритм нормализуем, т. е. для любого алгоритма можно написать эквивалентный ему НАМ.*

Это полный аналог тезису Тьюринга. Он строго не доказывается, но в его пользу есть ряд важных доводов. О них мы уже говорили на предыдущей лекции, рассматривая тезис Тьюринга, поэтому я не буду повторяться.

2. АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ

Теперь воспользуемся нормальными алгоритмами Маркова и рассмотрим задачи, для которых не существует алгоритмов решения. Начнем с определений.

2.1 Определения

Задача (проблема) называется ***алгоритмически разрешимой***, если существует алгоритм её решения. Такой, например, является задача нахождения корней квадратного уравнения: если дискриминант < 0, то корней нет, а иначе – корни вычисляются по известной формуле.

Однако не все задачи имеют алгоритм решения. И дело здесь не в том, что мы пока не знаем алгоритма решения задачи, а в том, что такого алгоритма не существует в принципе. Вот такие задачи и называются алгоритмически неразрешимыми. Итак, задача (проблема) называется ***алгоритмически неразрешимой***, если в принципе не существует алгоритма её решения.

Сейчас мы рассмотрим примеры таких алгоритмически неразрешимых задач, а пока отмечу следующее.

Задачи, которые нельзя решить, известны давно. Например, возьмём известную задачу о *трисекции угла*: дан произвольный угол и надо с помощью циркуля и линейки разделить его на три равных угла. Строго доказано, что сделать этого нельзя. Но здесь мы себя ограничиваем в действиях – разрешено использовать только циркуль и линейку. Если же разрешить пользоваться и другими инструментами, то эта задача решается.

Как видно, многое зависит от допустимых действий, операций. И естественно возникает вопрос: а если разрешить применять любые действия (в пределах разумного, конечно), то можно ли решить любую задачу? Оказывается, нет, не любую. В алгоритмах можно применять любые действия, лишь бы они были практически выполнимыми, но всё равно есть задачи, для которых не существует алгоритмов решения.

В чём же причина этого? В общих словах можно ответить приблизительно так: природа этих задач столь сложна, что единого способа решения их нет. Фактически для каждого конкретного частного случая задачи (для каждых исходных данных) нужен свой способ решения, а таких случаев – бесконечно много. Бесконечное же число алгоритмов нельзя объединить в один алгоритм (отмечу, что конечное число алгоритмов можно объединить в один алгоритм).

В то же время алгоритмическая неразрешимость задачи не означает, что нельзя решить её в частных случаях. Например, задача о трисекции угла легко решается в частном случае – для угла в 90 градусов (угол в 30 градусов строится просто). Но всё дело в том, что для каждого частного случая нужен свой уникальный алгоритм, а вот единого алгоритма решения задачи на все случаи – нет.

Ну а теперь перейдём к конкретным примерам алгоритмически неразрешимых проблем.

2.2 Проблема самоприменимости

Введем понятие ***запись алгоритма***. Пусть имеется нормальный алгоритм Маркова:

$$\begin{cases} \alpha_1 \to \beta_1 \\ \alpha_2 \to \beta_2 \\ \cdots\cdots\cdots \\ \alpha_k \to \beta_k \end{cases}$$

Вытягиваем его в одну линию, разделяя соседние правила подстановки символом «;» (считаем, что этот символ не входит в алфавит алгоритма):

$$\alpha_1 \to \beta_1 \ ; \ \alpha_2 \to \beta_2 \ ; \ ... \ ; \ \alpha_k \to \beta_k$$

Вот такое слово и называется *записью НАМ*.
Например, записью алгоритма

$$\begin{cases} a \to bc \\ b \mapsto c \end{cases}$$

является такой текст:

$$a \to bc \ ; \ b \mapsto c$$

Итак, запись НАМ – это линейная последовательность символов, т. е. слово, а к словам можно применять нормальные алгоритмы. Давайте посмотрим, что будет, если применить наш алгоритм к его записи:

$$\underline{a} \to bc \ ; \ b \mapsto c \overset{1}{\Longrightarrow} \underline{b}c \to bc \ ; \ b \mapsto c \overset{2}{\Longmapsto} cc \to b \ ; \ b \mapsto c$$

Итак, через два шага алгоритм остановился, т. е. он применим к своей записи. Алгоритмы, которые применимы к своей записи, называются ***самоприменимыми***.

Алгоритмы, неприменимые к своей записи (т. е. такие, которые при этом зацикливаются), называются ***несамоприменимыми***.

Пример несамоприменимого алгоритма:

$$\begin{cases} a \to ab \\ b \mapsto c \end{cases}$$

В самом деле, его записью является слово «$a \to ab \ ; \ b \mapsto c$». Применяя алгоритм к этому слову, получаем:

$$\underline{a} \to ab \ ; \ b \mapsto c \overset{1}{\Longrightarrow} \underline{a}b \to ab \ ; \ b \mapsto c \overset{1}{\Longrightarrow} \underline{a}bb \to ab \ ; \ b \mapsto c \overset{1}{\Longrightarrow} ...$$

То есть алгоритм всё время приписывает после a новую букву b и не останавливается.

Замечание. Обратите внимание: применимость алгоритма определяется по отношению к некоторому входному слову; при этом один и тот же алгоритм может быть применим к одним словам и неприменим к другим. Но когда мы говорим о самоприменимости, то здесь разговор идёт

о применимости алгоритма не к любому слову, а к одному вполне конкретному слову – к записи самого алгоритма. Поэтому самоприменимость зависит только от самого алгоритма и ни от чего более.

И ещё одно. Иногда на экзамене студенты дают такое определение: «Алгоритм самоприменим, если он применим к себе». Это неправильно: алгоритм – не слово, а к несловам алгоритмы применять нельзя. Речь идет не о применимости «к себе», а о применимости к записи алгоритма. Запись же – это слово, и потому её можно подать на вход алгоритму.

Итак, алгоритмы делятся на самоприменимые и несамоприменимые. Возникает вопрос: существует ли единый способ распознавания для любого алгоритма, самоприменим он или нет? Это и есть проблема ***самоприменимости***. Более точно она формулируется так:

Существует ли такой алгоритм $\mathcal{A}$, что:

$$\forall \text{ алгоритма } \mathcal{E} : \mathcal{A}(\langle \textit{запись } \mathcal{E}\rangle) = \begin{cases} C, & \text{если } \mathcal{E} \text{ самоприменим,} \\ H, & \text{если } \mathcal{E} \text{ несамоприменим} \end{cases}$$

То, что результатом этого алгоритма являются именно слова C (самоприменим) и H (несамоприменим), не играет ни какой роли; ничего не изменится, если результатом будут слова *ДА* и *НЕТ* или «+» и «−».

Так вот, оказывается, что такого единого способа нет, т. е. проблема самоприменимости алгоритмически неразрешима. Докажем это строго.

Теорема: Не существует указанного алгоритма $\mathcal{A}$.
Доказательство: (от противного):

Предположим, что такой алгоритм существует (обозначим его $\mathcal{A}$). Составим следующий алгоритм $\mathcal{B}$:

$$\begin{cases} C \to C \\ H \mapsto H \end{cases}$$

Этот алгоритм зацикливается, если ему на вход дают слово C, и останавливается, если на входе – слово H.

Рассмотрим теперь композицию алгоритмов $\mathcal{B}$ и $\mathcal{A}$ и обозначим её $\mathcal{K}$: $\mathcal{K} = \mathcal{B} \circ \mathcal{A}$; напомню, что для любого слова P по определению композиции алгоритмов $\mathcal{K}(P) = \mathcal{B}(\mathcal{A}(P))$. Как уже говорилось ранее, композиция двух нормальных алгоритмов есть также нормальный алгоритм. Следовательно, $\mathcal{K}$ – нормальный алгоритм Маркова.

А теперь поставим такой вопрос: алгоритм $\mathcal{K}$ – самоприменим или нет? Попытаемся на него ответить:

а) Предположим, что $\mathcal{K}$ – самоприменим. Тогда имеем:

$$\mathcal{K}(\langle \textit{запись } \mathcal{K}\rangle) = \mathcal{B}(\mathcal{A}(\langle \textit{запись } \mathcal{K}\rangle)) = \mathcal{B}(C) \text{ – зацикливаемся}$$

Следовательно, $\mathcal{K}$ несамоприменим. Итак, предположение, что $\mathcal{K}$ – самоприменим, привело к противоречию, т. е. это предположение ложно.

б) Предположим теперь, что $\mathcal{K}$ – несамоприменим. Тогда имеем:

$$\mathcal{K}(\langle \textit{запись } \mathcal{K}\rangle) = \mathcal{B}(\mathcal{A}(\langle \textit{запись } \mathcal{K}\rangle)) = \mathcal{B}(H) = H$$

Следовательно, $\mathcal{K}$ самоприменим. Таким образом, и в этом случае мы пришли к противоречию.

Что же получилось? Алгоритм $\mathcal{K}$ не является ни самоприменимым, ни несамоприменимым. Но ведь любой алгоритм либо самоприменим, либо несамоприменим. Что это все означает? А то, что такого алгоритма $\mathcal{K}$ просто не существует! А по какой причине он не существует?

Возможны три причины этого:

1) не существует алгоритма $\mathcal{A}$;
2) не существует алгоритма $\mathcal{B}$;
3) не существует композиции $\mathcal{B} \circ \mathcal{A}$.

Но алгоритм $\mathcal{B}$ мы явно предъявили, следовательно, он существует. Кроме того, мы доказали (правда для машины Тьюринга, но это не важно), что если алгоритмы $\mathcal{A}$ и $\mathcal{B}$ существуют, то и их композиция существует и является алгоритмом. Следовательно, и третья причина отпадает.

Остаётся первая причина, а именно, не существует алгоритма $\mathcal{A}$. Но вначале мы предположили, что $\mathcal{A}$ существует, и из этого вывели, что $\mathcal{A}$ не существует, т. е. получили противоречие. Значит, наше первоначальное предположение ложно. Следовательно, $\mathcal{A}$ – не существует.

Итак, мы доказали, что проблема самоприменимости алгоритмически неразрешима, т. е. не существует единого способа, который бы позволял для любого алгоритма определять, самоприменим этот алгоритм или нет.

Но ещё раз напомню, что отсюда не следует, что ни для какого конкретного алгоритма нельзя определить, самоприменим он или нет. Ведь я приводил Вам примеры алгоритмов и показывал, самоприменимы они или нет. Мы лишь доказали, что не существует единого способа решения проблемы самоприменимости для всех алгоритмов сразу.

Отмечу, что сама по себе проблема самоприменимости практически малоценна. Важность доказанной теоремы состоит в том, что опираясь на неё, можно доказать алгоритмическую неразрешимость уже более важных проблем. Одна из них – это проблема останова (или зацикливания) алгоритмов.

2.3 Проблема останова алгоритмов

На практике порой важно знать: зациклится или нет некоторый алгоритм, если его применить к некоторому слову. Естественно, появляется желание придумать единый способ (алгоритм), который бы для любого алгоритма и любого входного слова говорил, остановится или зациклится данный алгоритм, если на вход ему подать данное слово – ***проблема останова***.

Тут, правда, надо уточнить, что будет подаваться на вход этому алгоритму. Обозначим этот интересующий нас алгоритм $\mathcal{A}$. Дело в том, что алгоритму $\mathcal{A}$ нужно передать во-первых, анализируемый алгоритм (назовём его $\mathcal{E}$) и, во-вторых, P – входное слово для алгоритма $\mathcal{E}$. Точнее, на вход $\mathcal{A}$ нужно подать запись алгоритма $\mathcal{E}$ и слово P. Но мы договаривались, что любому алгоритму на вход подаётся только одно слово. Так вот, можно, например, объединить запись $\mathcal{E}$ и слово P в одно слово, разделив их решёткой, т. е. подавать на вход алгоритму $\mathcal{A}$ такое слово: $\langle$*запись* $\mathcal{E}\rangle\#P$. Тогда сама проблема останова формулируется так:

Существует ли такой алгоритм $\mathcal{A}$, что:

$$\forall \text{ алгоритма } \mathcal{E} \text{ и слова } P : \mathcal{A}(\langle\textit{запись } \mathcal{E}\rangle\#P) = \begin{cases} O, & \text{если } \mathcal{E} \text{ применим к } P \\ \mathit{З}, & \text{если } \mathcal{E} \text{ неприменим к } P \end{cases}$$

Докажем, что такой алгоритм не существует.

Слово P любое, возьмём в качестве него запись алгоритма $\mathcal{E}$: $P = \langle$*запись* $\mathcal{E}\rangle$. Тогда алгоритм $\mathcal{A}$ даст ответ O (остановится), если $\mathcal{E}$ применим к слову $\langle$*запись* $\mathcal{E}\rangle$, т. е. если $\mathcal{E}$ самоприменим, и даст ответ *З* (зациклится), если $\mathcal{E}$ неприменим к слову $\langle$*запись* $\mathcal{E}\rangle$, т. е. если $\mathcal{E}$ несамоприменим. Но это значит, что такой алгоритм $\mathcal{A}$ решает проблему самоприменимости, а этого быть не может. Итак, $\mathcal{A}$ не существует.

Следовательно, единого алгоритма, определяющего, зацикливаются или нет алгоритмы, не существует. Это печально, но ничего не поделаешь.

2.4 Проблема эквивалентности алгоритмов

Рассмотрим ещё одну алгоритмически неразрешимую проблему. Это проблема эквивалентности алгоритмов.

Если не вдаваться в тонкости, то два алгоритма $\mathcal{B}$ и $\mathcal{C}$ называются эквивалентными (будем обозначать это как $\mathcal{B} \equiv \mathcal{C}$), если при одинаковых входных словах они выдают одинаковые

выходные слова. Однако, как мы знаем, алгоритмы могут зацикливаться. Кроме того, надо учитывать и то, к каким словам применяются эти алгоритмы. Поэтому нужно более аккуратное определение. Оно такое.

Пусть имеется некоторый алфавит A. Напомню, что через A^* мы обозначаем множество всех слов, составленных из символов алфавита A. Так вот, алгоритмы $\mathcal{B}$ и $\mathcal{C}$ ***эквивалентны*** в алфавите A (на множестве A^*), если к каждому слову из A^* они либо одновременно неприменимы, либо одновременно применимы и выдают одинаковые выходные слова:

$\mathcal{B} \equiv \mathcal{C}$ на A^*, если $\forall P \in A^*$: $\mathcal{B}$ и $\mathcal{C}$ либо оба неприменимы к P,
либо применимы и $\mathcal{B}(P) = \mathcal{C}(P)$

Пример эквивалентных алгоритмов на множестве $A^* = \{a, b\}^*$:

$$\mathcal{B}: \begin{cases} *a \to * \\ *b \to b* \\ * \mapsto \\ \to * \end{cases} \qquad \mathcal{C}: \begin{cases} a \to \end{cases}$$

которые удаляют все буквы a из входного слова.

Пример не эквивалентных алгоритмов на том же множестве:

$$\mathcal{B}: \begin{cases} *a \mapsto \\ *b \to b* \\ \to * \end{cases} \qquad \mathcal{C}: \begin{cases} a \mapsto \end{cases}$$

Эти алгоритмы удаляют из входного слова первое вхождение буквы a, однако эти алгоритмы всё-таки не эквивалентны. Почему они не эквивалентны – постарайтесь определить сами; это Вам в качестве самостоятельной задачи.

Отмечу, что в определении эквивалентности очень важно то, на каком множестве слов A^* определяется эта эквивалентность. Дело в том, что на одном множестве алгоритмы могут быть эквивалентными, а на другом множестве – не эквивалентными. Например, алгоритмы

$$\mathcal{B}: \begin{cases} a \to \\ b \to \\ c \mapsto c \end{cases} \qquad \mathcal{C}: \begin{cases} a \to \\ b \to \\ c \to c \end{cases}$$

эквивалентны на множестве $A^* = \{a, b\}^*$, т. к. они из слов, составленных только из букв a и b, удаляют все буквы, но не эквивалентны на множестве $A^* = \{a, b, c\}^*$, т. к. если во входном слове есть хотя бы одна буква c, то после удаления всех букв a и b алгоритм $\mathcal{B}$ остановится, а алгоритм $\mathcal{C}$ зациклится.

Итак, что такое эквивалентные алгоритмы, мы узнали. Теперь поговорим о проблеме эквивалентности. Не ограничивая общности, будем считать, что алгоритмы $\mathcal{B}$ и $\mathcal{C}$ работают со словами из одного и того же алфавита A. Суть проблемы эквивалентности: хотелось бы построить такой алгоритм $\mathcal{D}$, который по записи любых двух алгоритмов $\mathcal{B}$ и $\mathcal{C}$ говорил, эквивалентны ли они на множестве слов A^* или нет:

$$\mathcal{D}(\langle \textit{запись } \mathcal{B} \rangle \# \langle \textit{запись } \mathcal{C} \rangle) = \begin{cases} Э, & \text{если } \mathcal{B} \equiv \mathcal{C} \text{ на } A^* \\ Н, & \text{иначе} \end{cases}$$

Так вот, строго доказано, что такого алгоритма $\mathcal{D}$ не существует, т. е. нет единого способа распознавания эквивалентности произвольных алгоритмов. Однако я не буду приводить это доказательство, т. к. оно достаточно сложное.

На этом я закончу рассказ про алгоритмически неразрешимые проблемы.

В заключение отмечу следующее. Как я уже говорил, основной причиной для уточнения понятия алгоритма было стремление доказать отсутствие алгоритмов решения тех или иных задач. Так вот, уточнив понятия алгоритма, уже действительно смогли строго доказать, что существуют задачи, у которых нет алгоритмов решения. Некоторые из этих задач мы и рассмотрели.

Лекция 4. МОДЕЛЬНАЯ ЭВМ

План лекции:

1. Структура ЭВМ

2. Оперативная память

3. Машинное представление данных

4. Машинная программа

5. Примеры машинных программ

Рассмотренные нами на предыдущих лекциях машина Тьюринга и нормальные алгоритмы Маркова – это такие способы уточнения понятия алгоритм, которые удобны для теоретических исследований – для доказательства существования и несуществования алгоритмов, для изучения свойств алгоритмов и т. п. Однако применять их для записи решений практических задач невыгодно: алгоритмы в виде машин Тьюринга или НАМ даже для простых задач получаются длинными и запутанными. Поэтому на практике используются другие способы записи алгоритмов, к изучению которых мы и переходим. Сегодня мы рассмотрим запись алгоритмов в виде т. н. машинных программ, т. е. программ для ЭВМ.

Детальным изучением ЭВМ Вы займетесь в следующем семестре, а сейчас я хочу лишь вкратце познакомить Вас с ними. Но прежде чем перейти к рассказу про ЭВМ, сделаю следующее замечание.

Существовало и существует много различных моделей ЭВМ, каждая из которых имеет свои особенности. Изучить все модели ЭВМ – вещь нереальная. Но это и не нужно, т. к. основные принципы работы всех ЭВМ одинаковы, а они отличаются, в общем-то, только частностями. Поэтому достаточно изучить только одну ЭВМ, чтобы понять, как работают и все остальные ЭВМ.

Однако для первого краткого знакомства с ЭВМ изучение даже одной конкретной машины не подходит – уж слишком много технических особенностей в каждой ЭВМ, в которых легко «утонуть». Поэтому мы рассмотрим не какую-то реальную ЭВМ, а упрощённую, так сказать, модельную ЭВМ, в которой оставлены лишь наиболее важные свойства реальных компьютеров, а второстепенные детали опущены. Для нас сейчас главное – понять принципы работы ЭВМ, а не изучать детали.

Итак, переходим к изучению такой модельной ЭВМ.[4]

1. СТРУКТУРА ЭВМ

Под структурой ЭВМ понимается её состав и взаимодействие её частей. Эту структуру можно условно изобразить в виде такой схемы (*см. рис.*).

Рассмотрим вкратце каждую из частей ЭВМ.

Устройства ввода. Через них в машину поступает вся информация из внешнего мира, от человека. У одной ЭВМ может быть несколько устройств ввода, причём разных типов. Основными типами устройства ввода сейчас являются клавиатура и мышь, которые, я думаю, известны большинству из вас.

Устройства вывода. Через них информация передается из ЭВМ во внешний мир, человеку. Этих устройств также может быть несколько и разных типов. Основные типы устройств вывода – это дисплей (экран) и принтер (печатающее устройство).

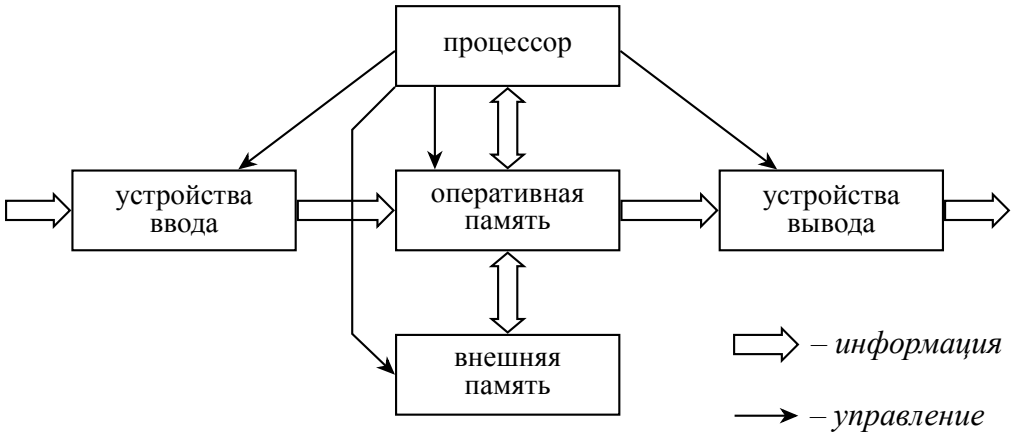
Для хранения информации, с которой работает ЭВМ, в ней имеется запоминающее

[4]При обсуждении понятия «алгоритм» и свойств алгоритмов мы говорили, что важно сформулировать алгоритм так, чтобы исполнитель смог его понять и выполнить. Исполнителем программ, которые мы будем писать, является ЭВМ. Поэтому нам важно знать возможности ЭВМ, исполнителя наших программ.

устройство, которое называется памятью. Практически во всех ЭВМ эта память бывает двух видов – оперативная (или внутренняя) память и внешняя память.

Оперативная память – это быстрое, а потому дорогое и небольших размеров запоминающее устройство. Здесь находится информация, с которой ЭВМ работает в данный момент.

Что именно здесь хранится? Во-первых, те данные, с которыми работает ЭВМ, а во-вторых, алгоритм решения задачи, по которому работает ЭВМ. Как и машина Тьюринга, ЭВМ сама по себе ничего не делает; чтобы заставить её работать, надо написать для неё программу. Когда мы рассматривали машину Тьюринга, то не интересовались тем, где находится программа для неё. Это была воображаемая машина, и где записана программа – было не столь важно. Но когда мы имеем дело с реальной машиной, то сразу возникает вопрос о местонахождении программы для неё. Так вот, программа для ЭВМ хранится в оперативной памяти, как и данные для этой программы.



Замечание. Несколько слов о соотношении терминов «алгоритм» и «программа». Это, вообще говоря, синонимы, т. е. обозначают одно и то же. Однако часто между ними все же проводят различие: алгоритм используют как общий термин для обозначения любых точных правил решения задачи, а программой принято называть алгоритм, предназначенный для выполнения на ЭВМ. Если мы не собираемся исполнять алгоритм на ЭВМ, то его не принято называть программой.

Внешняя память по своим характеристикам противоположна оперативной памяти: она более медленная, менее дорогая и больше по размерам. Здесь хранится информация, которая вообще-то нужна ЭВМ, но не в данный момент. В настоящее время основным типом внешней памяти являются всякого рода магнитные диски – дискеты, «жесткие» диски (т. н. винчестеры), компакт-диски и др.

В принципе ЭВМ может работать и без внешней памяти, но на практике её роль очень велика. Дело в том, что из-за высокой цены размер оперативной памяти сравнительно невелик, и хранить здесь долго и всю необходимую информацию не удаётся. Поэтому, если в оперативной памяти не хватает места, то часть информации из оперативной переносится во внешнюю память, где она хранится, пока снова не потребуется, а оперативная память в это время используется для хранения более нужной информации. Так что внешняя память используется как архив – для хранения той информации, которая в данный момент времени пока не нужна ЭВМ.

Кроме различия в скорости, цене и размерах, между оперативной и внешней памятью есть ещё два очень существенных различия. Во-первых, ЭВМ может работать только с той информацией, что находится в оперативной памяти, а с информацией из внешней памяти непосредственно работать не может. Если всё же необходимо работать с информацией, расположенной во внешней памяти, прежде всего надо переписать эту информацию в оперативную память, только после этого ЭВМ может с ней работать. Во-вторых, оперативна память – это кратковременная память, информация в ней сохраняется только на время выполнения программы.

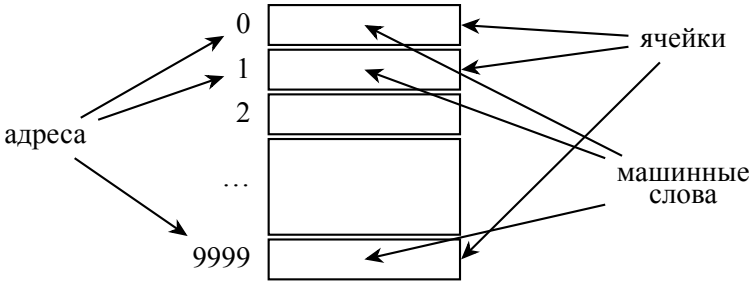
Как только программа кончает работать, так оперативная память сразу очищается и отдаётся в распоряжение следующей программе. Внешняя память – это долговременная память, информация в ней сохраняется и тогда, когда программа уже кончила работать. Размеры внешней памяти позволяют выделить в ней место для каждого программиста. И если он запишет сюда чего-нибудь сегодня, то и через неделю эта информация будет находиться там же. Поэтому, если надо что-то сохранить до следующего раза, то надо это «что-то» записать во внешнюю память.

Процессор («тот, кто осуществляет процесс») – это «мозг» ЭВМ, он всем «заправляет». Во-первых, он управляет работой всех других устройств ЭВМ, координирует их работу. В частности, именно процессор посылает сигналы устройствам ввода-вывода, чтобы те начали или окончили ввод-вывод, именно он управляет обменом информацией между оперативной и внешней памятью. Во-вторых, именно процессор выполняет те программы, которые находятся в оперативной памяти.

Таковы основные устройства, из которых состоит ЭВМ. Нас будут интересовать, главным образом, оперативная память и представление данных в ней, а также то, как выглядят и выполняются машинные программы.

2. ОПЕРАТИВНАЯ ПАМЯТЬ

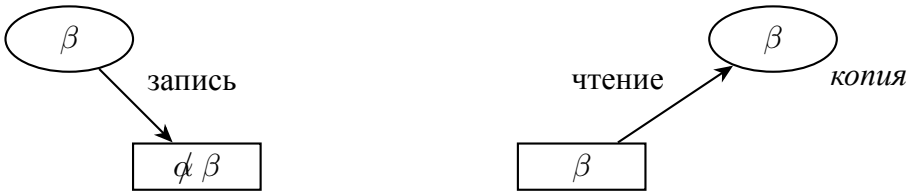
Оперативная память состоит из отдельных элементов, называемых ***ячейками***. Её можно представить себе как последовательность таких ячеек:



Все ячейки нумеруются начиная с нуля. Будем считать, что в нашей модельной ЭВМ имеется 10 тысяч ячеек, поэтому номер последней ячейки – 9999. (В реальных современных ЭВМ – миллионы и миллиарды ячеек.) Эти номера принято называть ***адресами ячеек***. Адрес ячейки – это нечто вроде её имени; все ссылки на ячейки – только по адресам: если надо, чтобы ЭВМ что-то записала в какую-либо ячейку, то даётся команда «запиши в ячейку с таким-то адресом» и т. п.

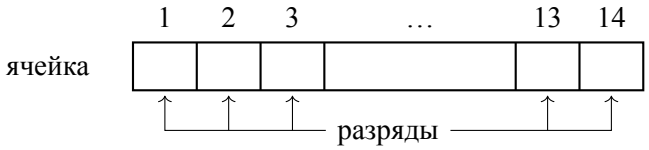
Содержимое ячейки, т. е. записанную в ней информацию, называют ***машинным словом***. Это может быть число, один или несколько символов, инструкция машинной программы и т. д.

По отношению к ячейкам в ЭВМ применяется только две операции: запись в ячейку и чтение из ячейки. Записать новое машинное слово в ячейку – значит уничтожить прежнее слово и поместить новое слово (*см. рис., слева*). Считывание же из ячейки означает копирование находящегося там сейчас машинного слова, само слово остается в ячейке неизменным (*см. рис., справа*):



Отмечу попутно, что считать машинное слово из ячейки или записать его в ячейку можно только целиком. Считать или записать только часть машинного слова нельзя.

Теперь поговорим о структуре ячеек. Каждая ячейка делится на более мелкие части, которые называются ***разрядами***. Будем считать, что в нашей модельной ЭВМ ячейка состоит из 14 разрядов (почему именно 14 – поймёте чуть позже), которые будем нумеровать слева направо начиная с 1:



Далее. Что может быть записано в разряде? На семинарах Вы уже познакомились с двоичной системой счисления, в которой для записи чисел используется всего две цифры – 0 и 1. По ряду причин этот двоичный способ представления информации технически наиболее выгоден, именно он и используется в реальных ЭВМ. Таким образом, в реальных ЭВМ в каждом разряде может храниться либо 0, либо 1, поэтому реальные машинные слова – это последовательности нулей и единиц. Однако для людей такая форма записи информации непривычна. Поэтому, чтобы при первом знакомстве с ЭВМ облегчить Вам жизнь, я буду придерживаться обычной десятичной системы счисления. Следовательно, в нашей модельной ЭВМ в каждом разряде может храниться одна из десятичных цифр, поэтому машинное слово у нас – это последовательность из 14 десятичных цифр, например: 12345678901234. В этом основное упрощение нашей модельной ЭВМ, главное отличие её от реальных машин.

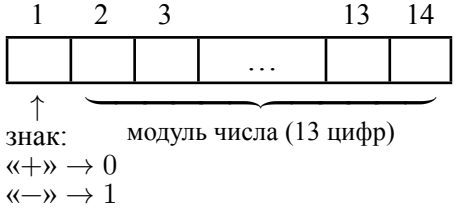
Итак, любая информация, которую мы хотим сохранить в памяти модельной ЭВМ, должна быть так или иначе представлена в цифровом виде, записана десятичными цифрами и только ими. Вот и давайте рассмотрим, как это делается.

3. МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

Начнём с представления чисел. Сразу отмечу, что в реальных ЭВМ различаются представления целых чисел и действительных чисел вообще (в чём причина этого, я скажу позже). Чтобы не отличаться от реальных ЭВМ, сохраним это различие и мы.

3.1 Представление целых чисел

При выборе представления целых чисел возникает только одна проблема – как избавиться от знака числа, от знаков $+$ и $-$? Ведь это не цифры, а мы всё должны записывать в цифровом виде. Решается эта проблема так. Договоримся отводить под знак числа самый левый (1-й) разряд ячейки и будем сюда вместо плюса записывать 0, а вместо минуса – 1.[5] Все остальные разряды отводим для записи модуля числа, причём, если в числе меньше 13 цифр, то слева добавляем нули (другими словами, число прижимается к правому краю ячейки):



Примеры:

$+12345$	0	0 0000 0001 2345
-12345	1	0 0000 0001 2345

[5] В современных компьютерах используется другой способ представления отрицательных чисел.

Как видно, в ячейку можно записать только 13-значные числа. Поэтому в нашей ЭВМ могут быть представлены только целых чисел от $-(10^{13}-1)$ до $+(10^{13}-1)$, числа же вне этого диапазона записать в ячейку нельзя.

Отмечу, что это – особенность не только нашей ЭВМ, но и всех реальных компьютеров. Память (а тем более ячейка) имеет конечный размер, поэтому, даже если на одно число отводить всю оперативную память, всё равно найдутся такие большие числа, что их не удастся записать в ЭВМ.

3.2 Представление вещественных чисел

Сразу отмечу, что в математике принят термин «действительные числа», а вот в программировании их почему-то называют «вещественными числами». Чтобы не нарушать эту традицию, я буду придерживаться термина «вещественные числа».

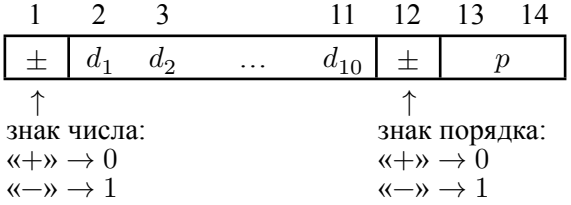
С представлением вещественных чисел связано две проблемы: проблема знака и проблема запятой, отделяющей целую часть от дробной. Как решить первую задачу, мы уже знаем, а вторая проблема решается следующим образом. Прежде всего замечу, что заменять запятую на какую-то цифру нельзя, т. к. тогда будет непонятно, как отличить настоящую цифру от обозначения запятой. Выход из положения – вообще не писать запятую. Как это удаётся сделать, мы сейчас и увидим.

Пусть дано число вещественное число $x \neq 0$ (о нуле разговор особый). Прежде всего представим его в виде с нулевой целой частью и с ненулевой первой цифрой в дробной части:

$$x = \pm 0{,}d_1 d_2 d_3 \ldots \cdot 10^{\pm p}, \quad \text{где } d_1 \neq 0$$

Например: $-105{,}82 = -0{,}10582 \cdot 10^{+3}$. Такое представление числа называется ***нормализованным***. Отмечу также, что цифровую часть числа, т. е. $d_1 d_2 d_3 \ldots$, принято называть ***мантиссой***, а показатель степени $\pm p$ – ***порядком***.

Поскольку в таком представлении у всех чисел части «0,» и « $\cdot$ 10» одни и те же, то их можно явно не указывать, а лишь подразумевать. Явно надо указывать только те части, что меняются от числа к числу, а это – знак числа, мантисса и порядок. Вот именно эти три части и записываются в ячейку:



Примеры:

$-2002{,}503 = -0{,}2002503 \cdot 10^{+4} \rightarrow$

1	20025 03000	0	04

$+0{,}0034567 = +0{,}34567 \cdot 10^{-2} \rightarrow$

0	34567 00000	1	02

Что касается нуля, то он, по определению, представляется нулевым машинным словом, т. е. состоящим из 14 нулей:

$0{,}0 \rightarrow$

0	00000 00000	0	00

что соответствует представлению $+0{,}0 \cdot 10^{+0}$.

Сделаю несколько замечаний.

1) На порядок вещественного числа мы отвели два разряда, поэтому максимально допустимый порядок – 99. Таким образом, диапазон представимых чисел: от -10^{99} до $+10^{99}$. Это очень большой диапазон, но всё равно есть числа (с порядком более 99), которые не представимы в памяти ЭВМ.

2) В общем случае в записи вещественных чисел присутствует бесконечное количество цифр. Но размер ячейки ограничен, и мы не можем записать в неё все цифры мантиссы (у нас, например, сохраняется только 10 цифр мантиссы). Поэтому, хотим мы того или нет, в памяти ЭВМ вещественные числа в общем случае представляются с погрешностью, приближённо.

3) Из того, что вещественные числа представляются приближённо, следует, что и операции над вещественными числами в общем случае выполняются приближённо, хотя бы из-за потери точности в промежуточных результатах:

$$(1/3) \cdot 3 \Rightarrow 0{,}3333333333 \cdot 3 = 0{,}9999999999 \neq 1$$

Итак, в общем случае вещественные числа представляются приближённо и операции над ними выполняются приближённо. От этого никуда не деться, это следствие конечного размера памяти ЭВМ.

4) Теперь вспомним, что целые числа являются частным случаем вещественных чисел, поэтому целые числа могут быть представлены так же, как и вещественные числа. Например:

$$12345 = 12345{,}0 = +0{,}12345 \cdot 10^{+5} \rightarrow$$

0	12345 00000	0	05

Но мы помним, что целые числа могут быть представлены и по-другому. Естественно возникают вопросы: Зачем нужно два представления для целых чисел? Зачем особо выделяют целые числа? Причина в следующем.

Как я только что сказал, вещественные числа представляются приближённо и операции над ними выполняются приближённо. Это же касается и целых чисел, если их рассматривать как частный случай вещественных чисел. Поэтому, например, число 12345 в памяти ЭВМ может быть представлено не только точно, но и как 12345,0001 или как 12344,9999, а результатом деления 4 на 2 может оказаться не 2, а 1,9999. Так вот, если мы согласны на приближенное представление целых чисел и приближенное выполнение операций над ними, тогда можно представлять целые числа как вещественные. Но если мы хотим точного представления и точного выполнения операций над целыми числами, то целые необходимо представлять «по-своему». Для этого во всех ЭВМ и введено особое представление для целых чисел, при котором они записываются точно и все операции над ними выполняют точно.

3.3 Представление символьной информации

Кроме чисел, ЭВМ работает и с символьной информацией – с отдельными символами и с текстами типа «*a+b-c=10a*». Как такая информация записывается в ЭВМ в виде цифр?

Решение очень простое: каждому символу ставят в соответствие некоторое целое число (оно называется кодом символа), которое и записывается в ячейку. Возможные примеры кодов:

$a \rightarrow 01, b \rightarrow 02, c \rightarrow 03, \ldots, z \rightarrow 26$, + $\rightarrow 27$, - $\rightarrow 28$, = $\rightarrow 29, 0 \rightarrow 30, 1 \rightarrow 31$ и т. д.

Текст же, т. е. последовательность символов, кодируется выписыванием подряд кодов его символов:

a + b - c = 1 0 a $\rightarrow$ 01 27 02 28 03 29 31 30 01

Но здесь возникает одна проблема: у нас в ячейку помещается 14 цифр, а код символа содержит две цифры. Можно, конечно, записывать в каждую ячейку по одному коду:

```
00 0000 0000 0001  a
00 0000 0000 0027  +
00 0000 0000 0002  b
.........................
```

Это т. н. ***неупакованное*** представление текста. Однако оно невыгодно – слишком много места пропадает даром. Выгоднее «запихать» в одну ячейку столько кодов, сколько «влезет». В нашей ЭВМ в одной ячейке можно разместить 7 символов (в виде кодов). Если одной ячейки не хватит для представления текста, то его продолжают в следующих по порядку ячейках:

```
01 27 02 28 03 29 31  –  a+b-c=1
30 01 00 00 00 00 00  –  0a
```

(00 – код, не соответствующий никакому символу; его ставят, чтобы дополнить ячейку до конца.) Это т. н. ***упакованное представление***.

Какому из этих двух представлений текстов отдать предпочтение? Это зависит от того, что для нас важнее – экономия памяти или экономия времени. Дело в том, что упакованное представление даёт выигрыш по памяти, но проигрывает по времени. Как я уже говорил, считывать из ячейки и записывать в неё можно только всё машинное слово целиком, поэтому при использовании упакованного представления приходится «исхитряться», чтобы считывать по отдельности коды символов из ячейки и записывать их по отдельности обратно, на что тратится время. При неупакованном же представлении этого делать не надо, поэтому операции при неупакованном представлении выполняются быстрее.

Итак, упакованное представление даёт выигрыш по памяти, но проигрывает по времени, а неупакованное представление, наоборот, выигрывает по времени, но проигрывает по памяти. Поэтому выбор представления зависит от того, что для нас важнее – экономия памяти или экономия времени.

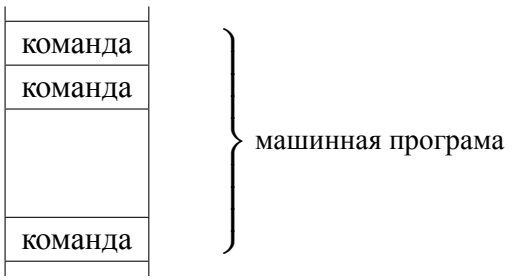
Мы рассмотрели машинное представление чисел и символьной информации. Аналогично можно придумать способы записи в цифровом виде и для любой другой информации. Но мы этим заниматься не будем.

4. МАШИННАЯ ПРОГРАММА

Мы рассмотрели, как представляются данные в ЭВМ. Теперь перейдем к вопросу о машинном представлении алгоритмов, которые также должны записываться в цифровом виде.

Любая ЭВМ, а точнее – её процессор, умеет выполнять около сотни простых операций таких, как сложить два числа, разделить одно число на другое, сравнить два числа и т. п. Но сама по себе ЭВМ не будет выполнять эти операции. Чтобы заставить ЭВМ что-либо делать, надо написать для неё программу (алгоритм), указав, какие операции надо выполнить, в каком порядке и над какими данными. Такая программа называется ***машинной программой***. Она составляется человеком, записывается на специальном языке, называемом *машинным языком*, и вводится в оперативную память. Когда программа уже записана в оперативной памяти, то процессор начнет её выполнять, ЭВМ начнет что-то делать. (Будем считать, что в ЭВМ есть специальная кнопка «начать выполнять программу».)

Что же представляет собой машинная программа? Это последовательность из так называемых *команд*, каждая из которых записывается в одну ячейку, причём соседние команды записываются в соседние ячейки памяти:



Команды выполняются по очереди: сначала первая, затем вторая и т. д. Это так называемый *естественный порядок* выполнения команд.

Машинные команды

Содержательно, каждая команда – это приказ автора программы машине выполнить одну из тех операций, которые она умеет выполнять. Например, первая команда может быть приказом «сложить такие-то два числа», вторая – «разделить одно число на другое» и т. п. Выполняя эти приказы, ЭВМ и решает задачу.

Что надо указывать в команде?

Смысл большинства операций, которые умеет выполнять ЭВМ, можно выразить так:

$$A \otimes B \Rightarrow C$$

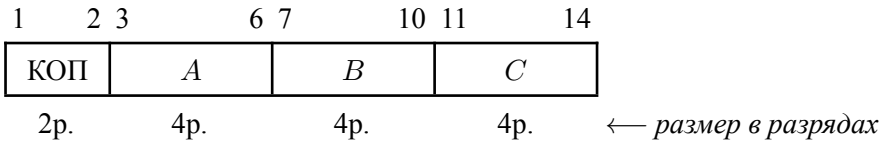
т. е. берутся две величины из ячеек с адресами A и B, над ними выполняется некоторая операция $\otimes$ (например, вычитание) и полученный результат записывается в ячейку с адресом C. Поэтому в команде, т. е. приказе на выполнение операции, надо указывать 4 вещи: какую именно операцию надо выполнить, над какими двумя величинами и куда записать результат. Отмечу, что исходные данные операции и её результат обычно называют ***операндами***, т. е. то, над чем выполняется операция.

Как всё это записывается в цифровом виде?

Начнём с операции. Конечно, нельзя писать слово «сложить» или знак $+$, т. к. это не цифры. Поэтому делают так: все операции, которые умеет выполнять ЭВМ, нумеруют двухзначными числами, например: сложение $\sim$ 01, вычитание $\sim$ 02, умножение $\sim$ 03 и т. д., и в команде указывают номер нужной операции. Такой номер называется ***кодом операции*** (КОП).

Теперь об операндах. Так как все данные находятся в каких-то ячейках памяти, то в команде указываются адреса тех ячеек, в которых хранятся эти исходные данные для операции и указывается адрес ячейки, в которую надо поместить результат. Обратите внимание: указываются не сами операнды, а адреса ячеек, где они находятся.

Итак, в команде указывается КОП и три адреса. При этом на КОП отводится 2 разряда, а на каждый адрес – по 4 разряда. В ячейке всё это записывается так:



Как видно, команда занимает 14 разрядов; именно под команды и подогнан размер ячеек в нашей модельной ЭВМ.

Рассмотрим примеры команд. Если 01 – КОП сложения, тогда команда

01	0205	4036	5980

означает следующий приказ компьютеру: сложить числа из ячеек с адресами 0205 и 4036 и записать полученную сумму в ячейку с адресом 5980.

Другой пример. Как я уже говорил, команды программы выполняются в том порядке, как они записаны в ячейках памяти. Но иногда этот естественный порядок выполнения команд надо нарушить и после какой-то команды нужно выполнить не следующую по порядку команду, а какую-то другую. Учитывая это, вводятся т. н. *команды перехода*. Примером может служить команда, называемая «переходом по равенству»:

10	2008	2009	0050

Смысл такой: надо сравнить два числа из ячеек с адресами 2008 и 2009 и, если они равны, следующей выполнить команду, расположенную в ячейке 0050. Если же числа не равны, то ничего не делать, а просто перейти к следующей по порядку команде.

Третий пример. Выполняя машинную программу, ЭВМ сама по себе не остановится. Чтобы прекратить работу ЭВМ, надо заставить её выполнить специальную команду останова:

99	0000	0000	0000

Выполнение этой команды и приводит к завершению работы ЭВМ.

Вот так записываются и выполняются команды. Способ записи программ в указанном цифровом виде и называется ***машинным языком.***

Система команд модельной ЭВМ

Совокупность всех команд, которые умеет выполнять ЭВМ, принято называть ***системой команд*** ЭВМ. Мы рассмотрим не все команды, а только те из них, которые потребуются нам в примерах.

Для сокращения записи введём следующие обозначения:

A, B, C – адреса ячеек (1-го, 2-го и 3-го операндов), указываемые в командах

a, b, c – содержимое этих ячеек

$x \Rightarrow C$ – записать машинное слово x в ячейку с адресом C

перейти к C – следующей выполнить команду, записанную в ячейке с адресом C

Замечание. Так как у нас числа делятся на целые и вещественные, а они представляются по-разному, то в ЭВМ каждая арифметическая операция представлена двумя командами – одна для работы с целыми числами, а другая – для работы с вещественными числами. Смешивать типы операндов в этих командах нельзя. Учитывая это, я буду указывать КОПы парами – для целых операндов и для вещественных.

Операция	КОП цел	КОП вещ	Действие
Арифметические			
Сложение	01	21	$a + b \Rightarrow C$
Вычитание	02	22	$a - b \Rightarrow C$
Умножение	03	23	$a * b \Rightarrow C$
Деление	–	24	$a \,/\, b \Rightarrow C$ (деления целых нет)
Переходы			
Переход по $=$	10	30	если $a = b$, то перейти к C
Переход по $\neq$	11	31	если $a \neq b$, то перейти к C
Переход по $<$	12	32	если $a < b$, то перейти к C
Переход по $\leq$	13	33	если $a \leq b$, то перейти к C
Ввод-вывод и преобразование			
Ввод	07	27	ввод числа и запись в яч. C
Вывод	08	28	вывод числа из яч. A
Перевод Ц $\rightarrow$ В	09	–	преобразование a в вещ. и запись в C
Перевод В $\rightarrow$ Ц	–	29	округление a до целого и запись в C
Другие операции			
Пересылка	00		$a \Rightarrow C$ (a – любого типа)
Останов	99		стоп

Замечания:

1) Если в команде какой-то адрес не задействован (например, адрес второго операнда в пересылке), то можно указывать любой адрес (обычно указывают нулевой – 0000). Но все 14 разрядов нужно указать обязательно.

2) Переходы по $>$ и $\geq$ не нужны, т. к. переход по $a > b$ – это переход по $b < a$, а переход по $a \geq b$ – это переход по $b \leq a$.

3) Как именно осуществляется ввод и вывод чисел, мы не будем уточнять.

5. ПРИМЕРЫ МАШИННЫХ ПРОГРАММ

Теперь мы готовы написать полные машинные программы.

Пример 1.

Ввести целые числа a и b и вещественное число c, вычислить величину d:

$$d = \frac{a^2 - b}{c + 3{,}14159}$$

и вывести d.

Отмечу, что взять и сразу написать машинную программу для решения даже простой задачи достаточно сложно. Поэтому обычно машинную программу составляют в три этапа. Я их сначала выпишу, а поясню потом на нашем примере:

1. планирование операций;
2. распределение памяти;
3. кодирование команд.

Кроме того, при составлении машинной программы обычно выписывают табличку из трёх колонок [*см. ниже*]. В левой колонке будем указывать адреса ячеек, в которых размещаются команды нашей программы, в средней колонке будем записывать сами команды в цифровом виде (содержимое ячеек), а в правой колонке – комментарии, т. е. действия, которые должна выполнять программа.

Итак, составляем программу для нашей задачи.

1) ***Планирование операций***. Как видно из приведенного списка команд, ЭВМ умеет складывать два числа, умножать два числа и т. д., но не умеет выполнять более сложные действия, например, не может сама вычислить нашу формулу. Поэтому наша первая задача – разложить вычисление по формуле на последовательность более простых действий – тех, которые понимает ЭВМ. Выписывать эти действия сразу в виде цифровых команд очень сложно, проще сначала записать эти операции в привычной символьной форме, а уж потом перевести их на машинный язык. Эти операции в символьном виде будем записывать в правой колонке. Это и есть этап планирования операций.

Наша программа должна последовательно ввести a, b и c. Затем вычислить числитель: возвести a в квадрат, записать в d; и т. д.

Обращу Ваше внимание на два момента.

1. Результат вычисления знаменателя запишем во вспомогательную ячейку p. Изменять входные данные (то есть значения a, b и c) не рекомендуется.
2. В нашей ЭВМ нет операции деления целых чисел. Поэтому перед делением числитель надо из целого числа преобразовать в вещественное число.

2) ***Распределение памяти***. Итак, какие операции должна выполнить ЭВМ, мы узнали, тем самым мы узнали, какие КОПы надо указывать в командах. Но этого недостаточно, чтобы записать команды в цифровом виде. Дело в том, что в командах надо указывать адреса тех ячеек, где находятся величины a, b и т. д. Определение этих адресов и называется распределением памяти.

Кто определяет эти адреса? Этим занимается автор программы. Какие это адреса? На время выполнения программы вся память находится в распоряжении программы, и автор программы может занимать какие угодно ячейки. Обычно переменные программы располагают компактно, в подряд идущих ячейках.

Кроме того, надо определить, в каких ячейках будет размещаться сама программа.

Давайте выделим под нашу программу самые первые ячейки памяти – начиная с адреса 0. Это позволяет нам полностью заполнить левую колонку нашей таблицы.

Под числа же давайте выделим такие ячейки:

$$a \sim 0100 \quad b \sim 0101 \quad c \sim 0102 \quad d \sim 0103 \quad p \sim 0104$$

Но этого мало. В формуле есть число 3,14159; такие явно заданные числа называют константами программы. Оно не является исходным данным для программы, поэтому не должно вводиться. Что с ним делать? Надо задать его нам самим – выделить под него ячейку и записать его туда в машинном виде. Какая это ячейка? Любая, но обычно константы размещаются сразу за последней командой программы.

3) ***Кодирование команд***. Вот теперь уже всё готово, чтобы написать программу на машинном языке. Запись команд в цифровом виде принято называть кодированием команд, кодированием программы.

адрес	команда / число	операции
0000	07 0000 0000 0100	ввод a
0001	07 0000 0000 0101	ввод b
0002	27 0000 0001 0102	ввод c
0003	03 0100 0100 0103	$a * a \Rightarrow d$
0004	02 0103 0101 0103	$d - b \Rightarrow d$
0005	21 0102 0010 0104	$c + 3{,}14159 \Rightarrow p$ (вспомог. ячейка)
0006	09 0103 0000 0103	$d_{\text{цел}} \Rightarrow d_{\text{вещ}}$
0007	24 0103 0104 0103	$d \,/\, p \Rightarrow d$
0008	28 0103 0000 0000	вывод d
0009	99 0000 0000 0000	останов
0010	03 1415 9000 0001	$3{,}14159 = +0{,}314159 \cdot 10^{+1}$

Отмечу, что в память ЭВМ записывается только средняя колонка, т. к. именно в средней колонке мы написали содержимое ячеек памяти. Две остальные мы выписали для себя, для наглядности, в ЭВМ они не попадают. Как именно программа попадает в память ЭВМ, мы не будет уточнять, это не столь принципиально для первого знакомства с ЭВМ.

Пример 2.

Ввести целое число n (≥ 2) и вычислить факториал $f = n!$, после чего напечатать это значение.

Опять действуем в том же порядке: 1) планируем операции, т. е. определяем, какие операции должна выполнить программа, и заполняем правую колонку; 2) распределяем память под программу, переменные и константы; 3) кодируем – записываем команды в цифровом виде.

Распределения памяти: $n \sim 2000$, $f \sim 2001$, $i \sim 2002$.

Программа:

адрес	команда / число	пояснения
0000	07 0000 0000 2000	ввод n
0001	00 0008 0000 2001	$1 \Rightarrow f$
0002	00 0008 0000 2002	$1 \Rightarrow i$
0003	03 2002 2001 2001	L: $i * f \Rightarrow f$
0004	01 2002 0008 2002	$i + 1 \to i$
0005	13 2002 2000 0003	если $i \leq n$, то перейти к L
0006	08 2001 0000 0000	вывод f
0007	99 0000 0000 0000	останов
0008	00 0000 0000 0001	1 как целое

Другие примеры машинных программ я приводить не буду. Уже из двух приведенных программ видны следующие особенности ЭВМ.

В ЭВМ допускается использование более крупных операций, чем в машинах Тьюринга и нормальных алгоритмах Маркова, а, следовательно, составлять программы для ЭВМ проще, и они короче. Кроме того, ЭВМ – это реально работающие автоматы, тогда как машина Тьюринга и нормальные алгоритмы Маркова – это воображаемые конструкции. Таким образом, ЭВМ, конечно, лучше в смысле практического использования.

Однако у ЭВМ и много недостатков. Основные из них:

а) используется непривычная для людей цифровая форма записи операций. Для людей более понятна символьная запись операций (как в правой колонке), чем запись цифрами;

б) операции в ЭВМ, конечно, более мощные, чем в машинах Тьюринга и нормальных алгоритмах Маркова, но всё-таки не столь мощные как хотелось бы. Например, мы привыкли сразу выписывать формулы, а в ЭВМ их приходится разбивать на более мелкие действия.

Так вот, желая избавиться от этих недостатков, люди придумали более удобные средства записи алгоритмов – т. н. *алгоритмические языки*. Но об этом мы поговорим на следующей лекции.

В заключение отмечу радостную для Вас вещь: тема «Модельная ЭВМ» не войдет в число экзаменационных вопросов (экзамен по ЭВМ у Вас будет в следующем семестре). Рассказал же я про ЭВМ для того, чтобы те из Вас, кто не знал этого, получили общее представление о том, как устроена и как работает ЭВМ, и чтобы в дальнейшем Вы понимали, почему различаются целые и вещественные числа, что такое упакованное и неупакованное представление и т. д.

Лекция 5. АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ. МЕТАЯЗЫКИ

План лекции:

1. Алгоритмические языки (краткий обзор)

2. Металингвистические формулы

3. Синтаксические диаграммы

4. Запись вещественных чисел в Паскале

1. АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ (краткий обзор)

Достоинства и недостатки машинного языка

На прошлой лекции мы рассмотрели машинные программы, т. е. запись алгоритмов на машинном языке – языке, понятном ЭВМ. Там я рассказал о достоинствах и недостатках машинных языков. Напомню их. Если сравнивать машинный язык с машиной Тьюринга и нормальными алгоритмами Маркова, то, конечно, машинный язык более удобен, в нём допускается использование более крупных элементарных операций (сложение, умножение и т. д.), чем в машине Тьюринга и нормальных алгоритмах Маркова, и этих операций не одна, как в нормальных алгоритмах Маркова, и не три, как в машине Тьюринга, а десятки. Поэтому программы на машинном языке получаются короче, писать их легче. Важно и то, что ЭВМ – это реально работающее устройство, тогда как машина Тьюринга и нормальные алгоритмы Маркова реально не существуют, это воображаемые конструкции.

Однако у машинного языка и много недостатков. Главных – два. Во-первых, в машинном языке используется непривычная для людей цифровая форма записи операций. Мы привыкли к символьной записи операций, а не к цифровой. Во-вторых, хотя в ЭВМ и используются более крупные операции, чем в машине Тьюринга или нормальных алгоритмах Маркова, для практических целей это всё же слишком мелкие операции. Например, мы привыкли выписывать формулу сразу целиком, а на машинном языке вычисление по формуле приходится разбивать на последовательность более мелких операций.

Итак, машинный язык, конечно же, более удобный способ записи алгоритмов, чем машина Тьюринга и нормальные алгоритмы Маркова, но всё же он недостаточно удобен.

Достоинства алгоритмических языков

Когда появились первые ЭВМ, а они появились в середине 1940-х годов, машинный язык был единственным языком для записи программ, люди были рады и ему. Но постепенно они осознали его недостатки и придумали, как избавиться от них, придумали более удобные способы записи программ – придумали ***алгоритмические языки***, или ***языки программирования***.

Вообще говоря, алгоритмический язык – это любой язык для записи алгоритмов, поэтому машина Тьюринга, нормальные алгоритмы Маркова и машинный язык – это алгоритмические языки. Но обычно под алгоритмическим языком понимают язык описания алгоритмов, удобный для применения на практике. Именно в этом смысле я и буду использовать этот термин.

Удобство алгоритмических языков достигается за счёт использования более привычной для людей символьной формы записи операций, за счёт того, что сами эти операции более крупные, чем в машинном языке.

Для примера приведу небольшую программу на языке Паскаль, который начнём изучать со следующей лекции. Она решает ту же задачу, что и первая машинная программа предыдущей лекции: ввести целые числа a и b и вещественное число c, вывести величину d, вычисленную по формуле

$$d = \frac{a^2 - b}{c + 3{,}14159}.$$

Программа на Паскале:

```
program EXAMPLE (input,output);
  var a,b: integer; c,d: real;
begin
  read(a,b,c);
  d:=(a*a-b)/(c+3.14159);
  write(d)
end.
```

Сделаю несколько замечаний по этой программе.

1) Как видно, в Паскале используются слова английского языка. Конечно, для тех, кто не знает его или знает плохо, понимать такой текст трудно. Но в настоящее время английский язык является мировым языком в программировании, и во всех алгоритмических языках используются слова именно из английского языка. К этому необходимо привыкнуть. Тем более в Паскале используется всего 3–4 десятка английских слов, и выучить их не так уж трудно. Для начала же приведу перевод этой программы на русский язык:

```
программа ПРИМЕР (ввод,вывод);
  перем a,b: целое; c,d: веществ;
начало
  ввести(a,b,c);
  d:=(a*a-b)/(c+3.14159);
  вывести(d)
конец.
```

Здесь уже легко понять смысл программы и без дополнительных пояснений.

2) Видно также, что при записи программ на Паскале используется обычная математическая символика, правда, с некоторыми отличиями. Например, формулы должны быть вытянуты «в линию». Это обусловлено тем, что вводить информацию в ЭВМ с клавиатуры можно лишь символ за символом, поэтому ввести многоэтажную формулу не удастся. Есть и другие отличия (знак умножения обозначается не крестиком, а звёздочкой и т. д.). Во всём остальном формулы и операции записываются как обычно.

Разнообразие алгоритмических языков

Далее. Первый алгоритмический язык появился в 1956 г., он назывался Фортран. Потом придумали и много других алгоритмических языков, сейчас их насчитывается несколько тысяч. Правда, подавляющее большинство из них мало распространено.

Естественно возникает вопрос: а зачем столько алгоритмических языков придумали? Почему нельзя обойтись одним алгоритмическим языком? Ответ такой. Людям приходится решать на ЭВМ самые разнообразные задачи – математические, инженерные, экономические, биологические, лингвистические и т. д. И то, что удобно для одних задач, далеко не удобно для других. Придумать же один алгоритмический язык, который был бы удобен для всех задач, не удаётся. Были попытки создать такой язык, но ничего хорошего из этого не получилось. Поэтому пошли по другому пути: для каждого класса задач стали придумывать свои языки.

Однако это не означает, что алгоритмический язык, ориентированный на один класс задач, нельзя использовать и для других задач, что на нём нельзя описать какие-то алгоритмы. Уж если язык машины Тьюринга универсален (т. е. позволяет описывать любой алгоритм), то тем более универсальны и более мощные алгоритмические языки. Дело здесь в другом: просто каждый алгоритмический язык разрабатывается для своего круга задач и для этих задач он очень удобен, а для других – не очень.

Трансляторы (переводчики)

Теперь я расскажу о т. н. трансляторах.

Итак, люди придумали алгоритмические языки, которые более удобны для записи программ, чем машинные языки. Но возникает вопрос: кто же будет выполнять программы, написанные на алгоритмическом языке? Ведь ЭВМ понимает только свой, машинный язык, а всё остальное она не понимает. А раз так, то и выполнять программы, записанные на алгоритмическом языке, она не может. Зачем же тогда писать программы на алгоритмическом языке, если их нельзя выполнить на ЭВМ?

Эта проблема решается так же, как и у человека, которому надо понять речь на неизвестном ему иностранном языке. В такой ситуации он использует переводчика с иностранного языка на свой родной язык. Аналогично и с алгоритмическими языками. Сначала пишут программу на алгоритмическом языке, а затем переводят её в эквивалентную программу на машинном языке, после чего эту машинную программу и выполняет ЭВМ.

А кто занимается таким переводом? Очевидно, этим может заняться специальный человек, но это невыгодно – лишняя работа, уж лучше сразу писать на машинном языке. Поэтому перевод возложили на саму ЭВМ. А сможет ли она это сделать? Да. Дело в том, что алгоритмические и машинные языки – это формальные, точные языки и потому существуют точные правила перевода с алгоритмического языка на машинный. А раз так, то можно написать алгоритм такого перевода и поручить его выполнение ЭВМ.

Именно так и делают. Кто-то один раз пишет машинную программу такого перевода, и эту программу записывают во внешнюю память ЭВМ. Эта программа называется ***транслятором*** (переводчиком). А затем поступают так. Человек пишет программу на алгоритмическом языке, вводит её текст в оперативную память ЭВМ и одновременно считывает из внешней памяти в оперативную транслятор с этого алгоритмического языка и передаёт ему управление. Так как транслятор – это программа на машинном языке, то ЭВМ может её выполнить. Транслятор просматривает текст программы на алгоритмическом языке строчку за строчкой, и строит эквивалентные машинные команды, записывая их в другое место оперативной памяти. Например, вместо *read*(a, b, c) транслятор построит такие команды (при следующем распределении памяти: $a \sim 0100, b \sim 0101, c \sim 0102$):

`07 0000 0000 0100` – ввод a как целого
`07 0000 0000 0101` – ввод b как целого
`27 0000 0000 0102` – ввод c как веществ.

В конце концов, в оперативной памяти окажется ещё одна программа, которая эквивалентна исходной программе на алгоритмическом языке, но записана на машинном языке. А такую программу уже может выполнять ЭВМ. Именно она и выполняется (построив её, транслятор передаёт управление на её первую команду), именно она вводит исходные данные, именно она вычисляет выводит результаты.

Условно всё это можно изобразить так:

Таким образом, мы вводим в ЭВМ программу на алгоритмическом языке, но на самом деле выполняется не она, а другая программа – эквивалентная ей машинная программа. Так решается проблема с исполнением программ на алгоритмических языках, этих «иностранных» для машины языках.

Сделаю несколько замечаний о трансляторах.

1) На перевод (трансляцию) программы с алгоритмического языка на машинный язык уходит довольно много времени. На этом мы теряем время. Но это с лихвой компенсируется выигрышем на этапе составления программы: написать программу на алгоритмическом языке в несколько раз легче и быстрее, чем на машинном языке. Поэтому при использовании алгоритмических языков времени от начала составления программы до её выполнения и получения результатов уходит значительно меньше, чем при использовании машинного языка.

2) В будущем Вы познакомитесь с тем, как работают трансляторы. Пока лишь отмечу, что транслятор – это очень сложная программа, содержащая десятки тысяч команд. Над созданием трансляторов трудятся группы высококвалифицированных программистов в течение нескольких месяцев и даже лет. Но на такие затраты необходимо идти, т. к. без транслятора любой алгоритмический язык – мертвый язык: писать программы на нём можно, а выполнять их на ЭВМ – нет. Кроме того, транслятор разрабатывается только один раз, а затем уж вовсю используется.

3) На разных ЭВМ разные машинные языки. В этом ещё один недостаток машинных языков: программа, написанная на языке одной ЭВМ, не может быть выполнена на другой ЭВМ. Её приходится переписывать заново. Алгоритмические языки же освобождает от этого недостатка. Они являются, как говорят, *машинно-независимыми*: программа на алгоритмическом языке может быть выполнена на разных ЭВМ. Достигается это за счёт того, что для каждой ЭВМ пишут (один раз!) свой транслятор с этого алгоритмического языка, переводящий программы на машинный язык данной, конкретной ЭВМ.

Об описании алгоритмических языков

Теперь поговорим об описании алгоритмических языков.

Как я сказал, удобство алгоритмических языков достигается как за счёт привычной символьной формы записи, так и за счёт использования большого и разнообразного набора операций. Но всё это оборачивается тем, что алгоритмический язык становится довольно-таки сложным хозяйством, и возникает проблема точного и строгого описания уже его самого, проблема создания специальных средств описания этого хозяйства.

Некоторые из этих способов мы сегодня рассмотрим. Но прежде я хочу объяснить, что именно надо описывать в алгоритмическом языке и как это делается.

Необходимо же описывать четыре аспекта алгоритмического языка: алфавит, синтаксис, семантику и прагматику.

1) **Алфавит**.

Прежде всего, надо описать алфавит алгоритмического языка, т. е. указать, какие символы можно использовать при записи программ на алгоритмическом языке, а какие – нельзя. Как описывается алфавит, мы уже знаем: явно перечисляются все символы, входящие в алфавит.

Обычно в алфавит алгоритмического языка входят латинские буквы (у нас в стране – и русские буквы), цифры, знаки операций ($+$, $-$, $<$, $>$, $=$ и т. д.), знаки препинания (. , : ; и т. д.) и другие символы.

2) **Синтаксис**.

Из символов алфавита можно составить самые разные тексты, в том числе и бессмысленные с точки зрения алгоритмического языка. Также как из русских букв можно составить тексты *чхфи* или *моя стул*, которые с точки зрения русского языка незаконны. Так вот, ***синтаксис*** – это правила, указывающие, какие тексты допустимы в языке, а какие – нет. Коротко говоря, синтаксис отвечает на вопрос: «Как правильно записывать тексты?»

Каким способом описывается синтаксис алгоритмического языка? Можно просто сформулировать правила словами на естественном языке (на русском, английском и т. п.), как это происходит в школе при изучении русского или иностранного языка. Но при этом есть опасность, что описание будет неточное, двусмысленное. Приведу такой пример.

В алгоритмическом языке в качестве имён переменных можно использовать слова, составленные из букв и цифр и начинающиеся с буквы, например: *x*, *abc*, *дом3кв8* и т. п. Такие конструкции принято называть ***идентификаторами***. Словесно идентификатор обычно определяется так:

Идентификатор – это последовательность букв и цифр, начинающаяся с буквы.

Казалось бы хорошее и точное определение. Но давайте вдумаемся в него. Текст *abc* – это идентификатор или нет? Вообще-то это идентификатор, но вот согласно данному определению – нет. Дело в том, что в этом определении сказано «… из букв и цифр …», а союз «и» обычно понимается как «и то, и другое». Следует ли отсюда, что в идентификаторе должны быть одновременно и буквы, и цифры? Если да, то *abc* – это не идентификатор, т. к. в нём нет цифр. Другой пример: *x* – это идентификатор? Вообще-то да, но согласно данному определению – нет. Дело в том, что это определение требует, чтобы была последовательность символов, а в обычной речи мы не применяем термин «последовательность» к одному элементу.

Как видно, не столь уж и точное это определение. А это одно из простейших определений, нужных для описания алгоритмического языка, с более сложными конструкциями ситуация усугубляется. Поэтому и стремятся избежать словесных описаний, для чего придумали формальные способы описания синтаксиса. Два из них мы сегодня рассмотрим.

3) **Семантика**.

Указать только синтаксис при описании алгоритмического языка – недостаточно. Мало знать, что такой-то текст правильно записан, надо ещё знать, а что этот текст означает, каков его смысл. Так вот, ***семантика*** – это правила, определяющие смысл каждого синтаксически правильного текста. Коротко говоря, семантика отвечает на вопрос: «Что это означает?»

Для семантики алгоритмического языка также придумали формальные методы описания, но они очень громоздки и неудобны.[6] Поэтому на практике они используются редко, и обычно семантику описывают словами естественного языка, стремясь, конечно, избегать неточностей и двусмысленностей. И мы будем так делать. Формальные же способы описания семантики Вы будете изучать на старших курсах.

4) **Прагматика**.

Вообще говоря, для описания алгоритмического языка достаточно описать алфавит, синтаксис и семантику. Но при практическом использовании алгоритмического языка, а особенно при его изучении, очень важно знать прагматику языка. ***Прагматика*** отвечает на вопрос «Зачем это нужно?», говорит о назначении тех или иных конструкций языка, о том, когда и как они используются. Без знания ответов на эти вопросы, обычно бывает трудно понять, зачем та или иная конструкция включена в язык, в каких ситуациях следует её использовать.

Со строгими методами описания прагматики дело обстоит совсем плохо. Пока никто не придумал никакие формальные методы, даже плохие. Поэтому прагматика описывается пока только словесно, на естественном языке.

Вот те четыре аспекта алгоритмического языка, которые необходимо описывать. Далее мы рассмотрим формальные методы только для описания синтаксиса алгоритмического языка.

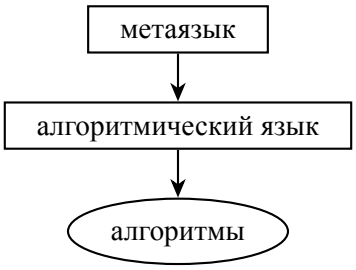
2. МЕТАЛИНГВИСТИЧЕСКИЕ ФОРМУЛЫ

Начну с названия. Что такое «металингвистический»? *Мета* переводится как *над*, а *лингва* – как *язык*, получается *надъязык*, но обычно его называют *метаязык*. Так что металингвистические формулы – это формулы (фразы, предложения) метаязыка.

А что такое метаязык? В общем случае языком называют любой способ описания чего-либо. Например, алгоритмический язык – это язык для описания алгоритмов. Но алгоритмический язык в свою очередь тоже надо описывать. Для этого нужен язык, на котором мы и будем

[6] Сейчас общепризнанным методом описания семантики алгоритмических языков является аксиоматика Ч. А. Хоара, использующая формулы алгебры логики.

описывать алгоритмический язык. Вот такой язык, предназначенный для описания другого языка, и называется ***метаязыком***:



Отмечу, что, вообще говоря, для описания самого метаязыка нужен свой метаязык – так сказать, метаметаязык, для описания которого нужен метаметаметаязык и так до бесконечности. Казалось бы неразрешимая проблема. Но на практике дело обстоит не столь безнадежно, т. к. метаязык для описания синтаксиса алгоритмического языка столь прост, что для него не надо придумывать специальный метаметаязык, а достаточно в этом качестве использовать обычный естественный язык, который мы хорошо знаем.

Существуют разные метаязыки. Мы будем рассматривать вариант, который получил название **БНФ** – ***формулы Бэкуса–Наура***, в честь двух ученых – авторов этого языка. Я сначала опишу этот метаязык, а затем уж расскажу, как с его помощью описывается синтаксис алгоритмических языков.

Формулы Бэкуса-Наура

В БНФ используются следующие конструкции.

1) ***Терминальные символы*** – это символы алфавита алгоритмического языка, т. е. символы, которые и только которые могут встречать в текстах на алгоритмическом языке.

Слово «терминальный» означает «окончательный», терминальные символы – те символы, из которых, в конце концов, и будут записываться программы.

2) ***Метапеременная*** – любой текст в уголках. Пример: ⟨*число*⟩. Метапеременная – это название некоторого множества текстов; указать метапеременную в формуле БНФ – значит сказать, что в этом месте может быть выписан любой текст из этого множества. Обычно метапеременная соответствует понятию описываемого алгоритмического языка; в естественном языке (например, русском) также есть такие понятия, например ⟨*имя существительное*⟩ или ⟨*словосочетание*⟩.

Чтобы лучше понять смысл метапеременной, приведу следующий пример. Все мы знаем, как трудно запомнить правила оформления официальных бумаг, например, заявлений. В связи с этим часто составляют образцы, шаблоны, показывающие, как правильно писать эти бумаги. Например, образец заявления может быть таким:

Декану ⟨*название факультета*⟩
⟨*звание*⟩⟨*ФИО*⟩
от студента группы ⟨*номер*⟩
⟨*ФИО*⟩

ЗАЯВЛЕНИЕ.

. . . . . .

Те слова, которые обязательно надо писать в заявлении, выписаны здесь явно (таково, например, слово *Декану*); это терминальные символы. Но есть такие места, где могут быть записаны

разные слова, в зависимости от конкретных условий (например, номер группы). В такое место образца вставляются не сами слова, а название того класса слов, которые можно употреблять в данном месте. Это и есть метапеременная. Вместо неё при составлении заявления нужно подставлять любой текст из соответствующего множества вариантов.

Аналогичное происходит и при описании синтаксиса алгоритмического языка. Например, если надо указать, что в каком-то месте необходимо выписать число, но какое – заранее неизвестно, то мы пишем в уголках ⟨*число*⟩.

Важно, чтобы название метапеременной соответствовало сути понятия описываемого языка. Например, в образце заявления использована метапеременная ⟨*ФИО*⟩ (фамилия, имя, отчество), а не, например, метапеременная ⟨*мп3*⟩. Поскольку синтаксис алгоритмического языка мы описываем для людей, нужно сделать это описание максимально понятным.

Отмечу особо, что конструкции *число* и ⟨*число*⟩ – это не одно и то же. Если слово *число* выписано без уголков, то это означает, что в данном месте должна стоять последовательность из пяти символов *ч*, *и*, *с*, *л*, *о*. Причём эта последовательность не имеет никакого отношения к числам. Если же мы имеем в виду именно числа, то должны писать в уголках ⟨*число*⟩.

3) ***Знак*** $::=$ – эта конструкция из трёх символов читается как «есть по определению» и используется именно в этом смысле.

4) ***Знак*** $|$ – вертикальная черта читается «или» и используется для разделения вариантов: «или то, или другое».

Вот из этих четырёх конструкций и строятся формулы БНФ. Каждая формула имеет вид:

$$\alpha ::= \beta_1 \mid \beta_2 \mid ... \mid \beta_n \quad (n \geq 1)$$

Здесь α, указанная в левой части формулы – это определяемое понятие; левая часть обязательно должна быть метапеременной. В правой части формулы указывается последовательность альтернатив, где β_i – любые последовательности (возможно, пустые) из терминальных символов и метапеременных.

Смысл формулы таков – она говорит, что правильная запись того, что мы назвали α, есть, по определению, либо текст β_1, либо текст β_2, …, либо текст β_n, и что других вариантов нет.

Пример 1.

⟨*цифра*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Смысл: цифра – это либо символ 0, либо символ 1, …, либо символ 9. Все иное цифрой не является.

Пример 2.

⟨*кортеж*⟩ ::= ( ) | (⟨*элемент*⟩ , ⟨*элемент*⟩)

⟨*элемент*⟩ ::= *а* | *а* ⟨*цифра*⟩ *а*

Смысл: кортеж – это либо текст ( ), либо текст, состоящий из скобки «(», за которой следует нечто, названное элементом, за ним следует запятая, снова элемент и скобка «)». Элементом же является либо буква *а*, либо три символа, первый и последний из которых – символы *а*, между ними – любая цифра.

Пример кортежей: ( ) $(a, a8a)$ (a, a)

Не кортежи: (a) $(c, a3a)$ $[a, a]$

Замечания:

1) Если в правой части формулы используется метапеременная, то эту метапеременную надо обязательно описать, т. е. выписать формулу, в левой части которой стоит эта метапеременная. Оставлять какую-либо метапеременную неописанной нельзя.

2) В последовательности (⟨*элемент*⟩, ⟨*элемент*⟩) дважды использована одна и та же метапеременная ⟨*элемент*⟩. Это не значит, что оба её вхождения надо заменять на один и тот же

текст. Вместо каждого вхождения можно подставлять «свой элемент». Запомните это сразу: метапеременная может быть заменена на любой текст, который ей соответствует.

3) О различии терминов «понятие» и «метопеременная». В примере 2 описываются понятия кортеж и элемент. Записи $\langle$*кортеж*$\rangle$ и $\langle$*элемент*$\rangle$ в левых частях формул – это метапеременные, использованные для определения соответствующих им понятий. Чтобы не путаться, нужно просто запомнить, что название понятий в БНФ записываются в угловых скобках (таким образом они становятся метапеременными).

4) Ещё раз отмечу, что БНФ описывает только синтаксис, т. е. то, как правильно записывать то или иное понятие, и ничего не говорят про смысл, семантику этого понятия. Поэтому из указанных формул непонятно, что означает запись «(a, a)», а лишь понятно, что это правильная запись кортежа.

Рекурсивные БНФ

В рассмотренных выше примерах, одни понятия определялись через другие. Но допускаются и формулы, где понятие определяется «через себя», т. е. где одна и та же метапеременная входит как в левую, так и в правую часть формулы. Например:

$$\alpha ::= \beta \mid \gamma\alpha$$

Такие формулы называются рекурсивными (рекурсия – это определение через себя). Рассмотрим примеры рекурсивных описаний.

Пример 3.

Определим понятие «целое без знака», понимая под этим любую непустую последовательность цифр, например: 0, 73, 1985 и т. п.

Можно попытаться дать такое определение:

$$\langle \textit{целое без знака} \rangle ::= \langle \textit{цифра} \rangle \mid \langle \textit{цифра} \rangle\langle \textit{цифра} \rangle \mid \langle \textit{цифра} \rangle\langle \textit{цифра} \rangle\langle \textit{цифра} \rangle \mid \ldots$$

То есть, целое без знака – это либо одна цифра, либо две, либо три и т. д. Но такое определение не пройдет, т. к. в формуле разрешено выписывать лишь конечное число правых частей, а в данном случае их придется выписывать бесконечное число раз.

Поэтому нужно придумать иное определение. А для этого достаточно заметить следующий факт: если мы имеем целое, скажем 12, то, приписав к нему справа (или слева) цифру, скажем, 5, снова получим целое – 125. Это и подсказывает вид нужной формулы:

$$\langle \textit{целое без знака} \rangle ::= \langle \textit{цифра} \rangle \mid \langle \textit{целое без знака} \rangle\langle \textit{цифра} \rangle$$

Смысл: целое без знака – это либо цифра, либо целое, к которому справа приписана цифра.

Проверим, является ли текст 125 правильной записью целого без знака. Согласно данной формуле, есть два варианта правильной записи этого понятия. Первый вариант – первая альтернатива – говорит, что целым является одна цифра. Нам это не подходит, т. к. у нас три цифры. Значит, остаётся только второй вариант, который требует, чтобы 12 было целым без знака, а 5 – цифрой. Последнее верно, поэтому надо доказать, что 12 – целое без знака. Опять возможны два варианта, первый из которых не подходит, поэтому рассматриваем второй вариант, который требует, чтобы 1 было целым без знака, а 3 – цифрой. Последнее верно, поэтому доказываем, что 1 – целое без знака. Теперь уже проходит первая альтернатива, согласно которой одна цифра – это целое. Итак, мы показали, что текст 125 удовлетворяет нашему определению.

Это же можно доказать и в «обратную сторону»: 1 – цифра, а значит, это целое без знака; приписывая к этому целому справа цифру 2, мы снова получаем целое; приписывая же к целому 12 цифру 5, мы опять получаем целое без знака.

Далее. Обращаю Ваше внимание на то, что в правильной рекурсивной формуле обязательно должна быть хотя бы одна нерекурсивная ветвь (т. е. не содержащая определяемое понятие), иначе мы попросту зациклимся («зарекурсивимся»). В нерекурсивных ветвях описываются простейшие варианты понятия, а вот в рекурсивных ветвях показывается, как из простых вариантов строятся более сложные варианты.

Пример 4.

Определим строго понятие «идентификатор», о котором я уже упоминал при рассказе о неоднозначности понимания словесного описания:

⟨*идентификатор*⟩ ::= ⟨*буква*⟩ | ⟨*идентификатор*⟩⟨*буква*⟩ |
⟨*идентификатор*⟩⟨*цифра*⟩

⟨*буква*⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

Смысл: идентификатор – это либо одна буква, либо уже имеющийся идентификатор, к которому справа приписана буква или цифра. Отсюда следует, что *x*, *abc*, *ac8w3* – это идентификаторы, а *8c* – не идентификатор.

Замечание. Если все альтернативы формулы не уместились в одной строке, часть формулы переносят на следующую строку (как формулы в математике). Важное отличие: знак альтернативы | повторять не нужно! Если написать в конце одной строки | и повторить его в начале следующей строки, формально получим два знака | | подряд, между ними пустая цепочка символов, пустая альтернатива.

Расширение БНФ

Ради сокращения записи в БНФ введём ещё одно обозначение – фигурные скобки { }. Очень часто в БНФ приходится определять последовательность из одних и те же элементов, скажем из α, причём в этой последовательности может быть 0, 1, 2 и более элементов. Такая последовательность (назовем её ⟨*посл_из_*α⟩) определяется следующим образом:

⟨*посл_из_*α⟩ ::= ⟨*пусто*⟩ | ⟨*посл_из_*α⟩ α

⟨*пусто*⟩ ::=

(*Замечание*: во второй формуле всего одна правая часть, причём она пустая.)

Так вот, договоримся обозначать эту последовательность как $\{\alpha\}$. Следовательно, запись $\{\alpha\}$ означает любой из следующих текстов:

⟨*пусто*⟩, α, $\alpha\alpha$, $\alpha\alpha\alpha$ и т. д.

Отмечу попутно, что запись $\{\alpha\beta\}$ означает любой из следующих текстов:

⟨*пусто*⟩, $\alpha\beta$, $\alpha\beta\alpha\beta$, $\alpha\beta\alpha\beta\alpha\beta$ и т. д.,

но не $\alpha\alpha\beta\beta$, $\alpha\alpha\alpha\beta\beta\beta$ и т. п.!

Например, с использование фигурных скобок формулу для понятия «целое без знака» можно переписать так:

⟨*целое без знака*⟩ ::= ⟨*цифра*⟩{⟨*цифра*⟩}

Смысл: целое без знака – это цифра, за которой следует любая последовательность из цифр, возможно и пустая.

В то же время формула

⟨*целое без знака*⟩ ::= {⟨*цифра*⟩}

ошибочна, т. к. допускает пустой текст, который, конечно, не является записью целого числа без знака.

Пример 5.

⟨*сумма*⟩ ::= ⟨*буква*⟩{+ ⟨*буква*⟩}

Примеры правильных записей понятия «сумма»: a $b+c$ $c+a+b$

БНФ и описание синтаксиса алгоритмического языка

Познакомившись с тем, что такое БНФ, теперь посмотрим теперь, как они используются для описания синтаксиса алгоритмического языка.

Любой правильный текст на алгоритмическом языке принято называть программой. Поэтому выписывается формула, определяющая понятие программа: в левой части стоит метапеременная ⟨*программа*⟩, а в правой части указывается её структура. Например, для языка Паскаль эта формула выглядит так:

⟨*программа*⟩ ::= ⟨*заголовок программы*⟩ ⟨*блок*⟩ .

Появилось две новых метапеременных, их тоже надо описать:

⟨*заголовок программы*⟩ ::= `program` ⟨*имя*⟩ `(`⟨*список файлов*⟩`);`

⟨*блок*⟩ ::= ⟨*раздел описаний*⟩ ⟨*раздел операторов*⟩

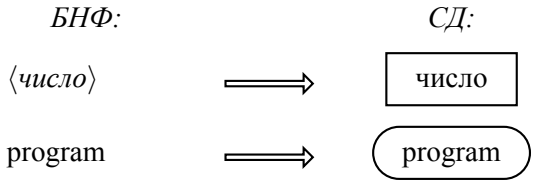
Опять появились новые метапеременные. Теперь уже для них нужно выписывать определения и т. д. Синтаксис языка считается полностью описанным, если приведены формулы для всех метапеременных.

3. СИНТАКСИЧЕСКИЕ ДИАГРАММЫ

Синтаксические диаграммы (СД) – эта другой способ описания синтаксиса, придуманный Никлаусом Виртом, автором языка Паскаль. По сути дела, это те же формулы БНФ, но записанные в наглядной графической форме – в виде диаграмм с овалами, прямоугольниками и стрелками. Поэтому я буду фактически приводить правила перевода тех или иных конструкций из БНФ в СД.

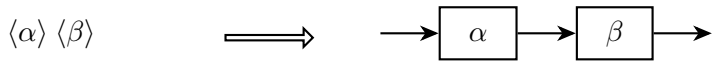
Правила записи синтаксических диаграмм

1. В диаграмме понятие заключается в прямоугольник, а явно заданный текст (т. е. текст из терминальных символов) – в овал:

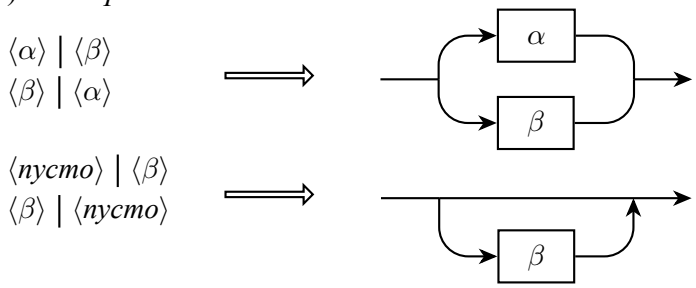


2. Эти прямоугольники и овалы соединяются стрелками в нужной последовательности. Возможны следующие способы соединения:

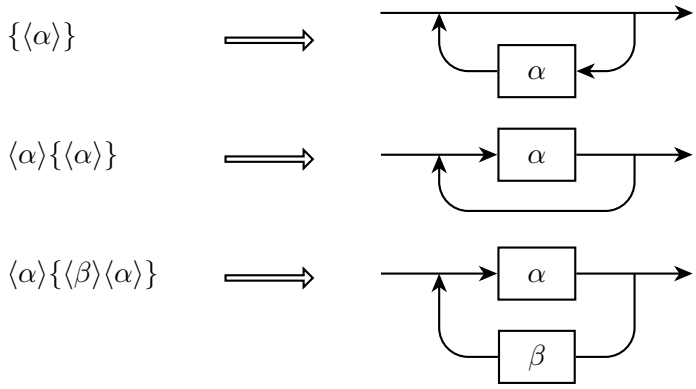
а) *последовательное соединение*:



б) *альтернативы*:



в) *повторения*:



3. Чтобы описать понятие в виде синтаксической диаграммы, надо указать название этого понятия (без уголков) и провести от него стрелку к диаграмме, изображающей по указанным правилам правую часть соответствующей формулы БНФ:

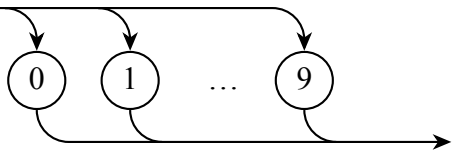
понятие ⟶ *диаграмма*

Считается, что некоторый текст удовлетворяет такой диаграмме, если, начав с левого конца диаграммы и двигаясь по стрелкам, мы найдем путь к правому концу диаграммы, которому соответствует наш текст.

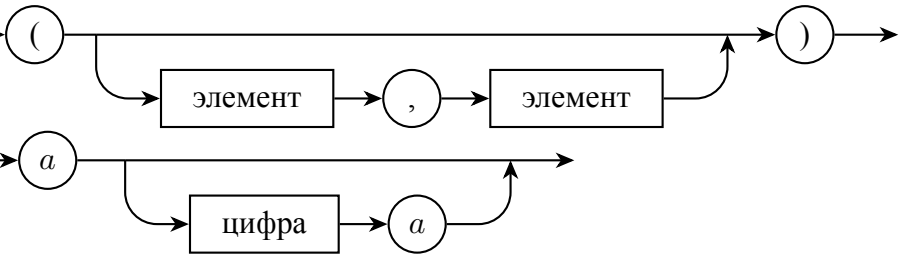
Примеры синтаксических диаграмм

Определим средствами синтаксических диаграмм понятия, которые определили ранее с помощью БНФ.

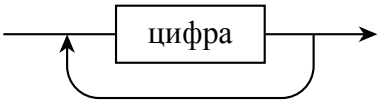
Пример 1. цифра



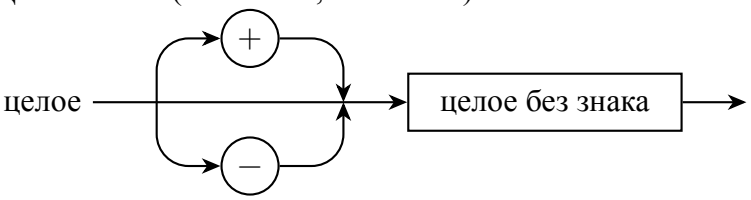
Пример 2. кортеж



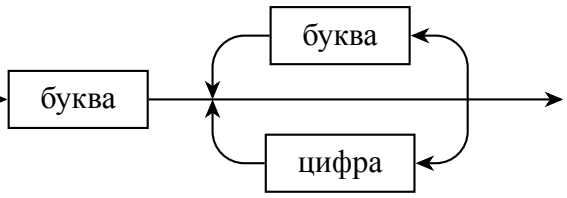
Пример 3. целое без знака



Пример 3′. Целое число (возможно, со знаком)

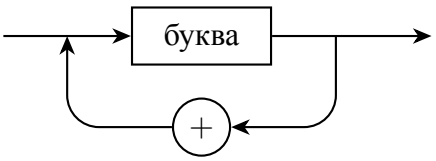


Пример 4. идентификатор



Диаграмму для понятия *буква* я не буду приводить, т. к. она аналогична СД для *цифры*.

Пример 5. сумма



Что лучше – БНФ или СД? Трудно сказать. Синтаксические диаграммы, конечно, нагляднее, но зато БНФ компактнее, занимают меньше места, поэтому в книгах чаще используют БНФ, а не СД. Можно сказать, что синтаксические диаграммы ориентированы на людей, а БНФ – на ЭВМ; они более пригодны для компьютерной обработки. В дальнейшем мы будем пользоваться как БФН, так и СД – в зависимости от того, что в конкретной ситуации удобнее.

Попутно отмечу, что в книге В. Г. Абрамова и Н. П. и Г. Н. Трифоновых «Введение в язык Паскаль» из-за типографских соображений СД даются несколько в ином виде, чем приводил я и чем они обычно приводятся.

4. ЗАПИСЬ ВЕЩЕСТВЕННЫХ ЧИСЕЛ В ПАСКАЛЕ

В оставшееся время я хочу немного забежать вперед и, воспользовавшись синтаксическими диаграммами, показать, как в языке Паскаль описываются вещественные числа (это тема из следующей лекции).

Мы уже знаем, что в компьютерах различаются целые и вещественные числа. Это различие переносится практически во все алгоритмические языки, в частности и в язык Паскаль. Как записываются целые числа, я уже показал в примере $3'$. Теперь же посмотрим, как записываются вещественные числа.

В Паскале вещественные числа записываются как десятичные дроби, но при этом есть два отличия.

1) Вместо запятой, отделяющей целую часть от дробной, пишут точку (так принято в США и Англии, эта традиция перешла во все алгоритмические языки):

$3{,}14 \to 3.14$

2) Для записи очень больших и очень маленьких чисел (например, в физике и химии) мы часто используем 10-й порядок: вместо 2000000 пишем $2 \cdot 10^6$, а вместо 0,000002 пишем $2 \cdot 10^{-6}$. В Паскале также можно использовать этот способ записи, но делается это несколько иначе: вместо «$\cdot 10$» пишут `E` (от *exponent* – показатель степени), а сам показатель опускают вниз, в строку:

$2 \cdot 10^6 \to$ `2E6` или `2E+6`

$2 \cdot 10^{-6} \to$ `2E-6`

В общем случае в записи вещественного числа указываются три вещи – целая часть (вместе со знаком числа), дробные часть (вместе с точкой) и порядок. Например, в числе

```
-314.159E+2
```

«`-314`» – целая часть, «`.159`» – дробная часть, «`E+2`» – порядок. При этом целая часть указывается обязательно, а вот либо дробную часть, либо порядок можно опускать (но кто-то из них должен присутствовать, чтобы число было вещественным, а не целым):

`+3.14159` – нет порядка

`2E-8` – нет дробной части

Приведу также примеры неправильных записей вещественных чисел и то, как их преобразовать, чтобы получить правильную запись:

`.14` – нет целой части (надо: `0.14`)

`2.` – после точки должна быть цифра (надо: `2.0`)

`-314` – это целое число, а не вещественное (надо `-314.0` или `-314E0`)

`E2` – это идентификатор, а не число (надо: `1E2`)

Теперь я уже приведу точный синтаксис записи вещественных чисел в Паскале: