Истинность и доказуемость

Сергей С. Степанов

Ниже приведены достаточно общие рассуждения, имеющим скорее философский характер. При определённом размышлении, некоторые вещи и идеи, в силу привычности кажущиеся очевидными, таковыми на самом деле не являются. Осознание этого не только исключительно интересно, но и даёт импульс для дальнейших размышлений.

Мы сконцентрируемся на таких фундаментальных понятиях математики, как доказательство, алгоритм и множество. Будет проанализирован мощный, но требующий осторожности метод рассуждения от противного. При помощи простого высокоуровневого варианта машины Тьюринга мы обсудим некоторые понятия теории алгоритмов. После этого переберёмся в канторовский рай и совершим небольшую, но достаточно критическую экскурсию по теории множеств. В заключение мы обсудим теорему Гёделя о неполноте математики, понятие истины, и связанные с ними проблемы построения искусственного интеллекта.

Необходимо предупредить, что ряд утверждений, приведенных ниже, не разделяется многими математиками, поэтому к ним необходимо относится предельно критично. Однако, именно в этом и состоит цель – пробудить, иногда, возможно в провокационной форме, два самых важных свойства – умение удивляться и сомневаться.

Последняя версия документа может быть найдена по адресу http://synset.com. Все замечания и предложения просьба присылать по почте math@synset.com или оставлять на сайте, на страницах обсуждения.

---

v. 0.1, 2008-2009 г., (printed: 21 марта 2011 г.)

I Математика, от мамонтов до наших дней

• В жизни нам редко удаётся убедить кого-либо в чём-либо. Сколько существует людей – столько будет и мнений. Поэтому часть людей, неудовлетворённых такой ситуацией, решила существенно сузить тему обсуждений, но при этом научиться убеждать друг друга, приходя к единому мнению, которое торжественно назвали "Истина". Так появилась математика.

То, что объединение кучек из "двух" и "трёх" камней всегда будет приводить к такому же результату, как и добавление камня к куче из "четырех" камней, было абсолютно очевидным. Только полностью лишённые права считаться математиками, могли оспаривать истинность этого утверждения. В такой однозначности и неизбежности ощущалась тень улыбки Всевышнего, и, конечно, математические занятия также начали носить признаки священнодействия.

К тому же, затем выяснилось, что некоторые математические истины очень полезны для суетных дел человеческих. Можно пересчитать овец в стаде, честно разделить мамонта на части, или предсказать затмение Солнца, подняв свой авторитет перед соплеменниками. Однако, не смотря на очевидную полезность, математика многих привлекает не по этому. Им нравится убеждать и быть убеждаемыми, т.е. заниматься *доказательствами.*

От первого проблеска математической мысли до её современных вершин, доказательства носят психологический характер. По образному выражению Владимира Андреевича Успенского "*психологическое доказательство* – это рассуждение, убедительное на столько, что вы готовы других людей убеждать при помощи *этого же* рассуждения."

Древние индийские математики, тот факт, что площадь круга равна радиусу умноженному на половину длины окружности, доказывали так:

Естественно, подобная лаконичность носит налёт снобизма и клановости. Предполагается, что детали, любой посвящённый додумает сам. В частности разработает понятие бесконечно малых, предельного перехода, а возможно, и придёт к определению числа π. Похожая ситуация, к сожалению, присутствует и в некоторых современных трактатах.

• Математика началась с понятий множества, числа и равенства. Вряд ли кто-то высекал на стенах пещеры каракули "$1 = 2 \to 2 = 1$", но осознание того, что различные, очень не похожие друг на друга объекты окружающего мира, собранные вместе, обладают чем то общим, создало первого математика. Для всех не математиков два мамонта, безусловно, не тоже, что два саблезубых тигра, или две симпатичные самки. Однако, если отбросить все различия, то между тем что осталось можно поставить знак равенства и назвать числом.

"Бог создал натуральные числа, а всё остальное придумал человек". С этим утверждением Леопольда Кронекера трудно согласиться. На самом деле Бог создал Всё. А всё остальное, в том числе и натуральные числа, придумал человек. Понятие *числа* возникло в результате некоторого возбуждения нейронов в мозгу математика, которое, по удивительному замыслу Божьему, имеет примерно такую же конфигурацию и в головах других математиков. Практически одновременно появились понятия "сложить", "меньше", "больше" и "не равно". Для этого достаточно было насыпать несколько кучек камней и сказать – СМОТРИ.

Математики быстро научились вычитать, отбирая из кучи камней часть, и поняли, что эта *операция* обратна сложению. В результате решения задачи честного раздела кучи камней на несколько равных частей, возникла операция деления, а некоторые странные, ни как поровну не делящиеся совокупности, были названы простыми. Умножение обратное делению, имело также и самостоятельное существование, воплощённое в виде прямоугольника, выложенного одинаковыми камнями. Начали появляться первые знаковые системы, формализующее рассуждение.

Следующий прорыв возник из простого вопроса: "$2+3=5,\ \ 5-3=2$, а чему равно $3-5$?". И просто посмотреть на кучку камней, чтобы на него ответить, недостаточно! Необходимо было невероятное напряжение мысли для *придумывания* таких понятий как "ноль" и "отрицательное число", а затем – фантастическая харизма, для убеждения других, что рассуждать о них, вообще говоря, тоже можно. К счастью, сейчас отрицательным числам учат задолго до того, как ребёнок поймёт, что он хочет быть математиком. В этом возрасте у человеческого существа очень развито умение удивляться, однако сомневаться оно почти не умеет. Поэтому в отрицательные числа ребёнок сначала верит, а затем привыкает к ним. Став же взрослым, он теряет способность удивляться. А ведь если у натуральных чисел есть хоть какие-то прообразы в окружающем мире, то отрицательных чисел там просто нет! Даже "привычный" ноль, на самом деле, также загадочен, как и хлопок Будды одной ладонью.

Просто *изобретать* новые математические понятия (объекты) конечно неинтересно. Интересно *открывать* те свойства и следствия, которые следуют из их существования. При этом необходимо уметь убежать других в их истинности, т.е. доказывать. Из ничего вывести всё можно. Но тяжело. Поэтому, кроме объектов потребовались исходные "очевидные" истины, при помощи которых можно доказывать другие. Благодаря Евклиду возник *аксиоматический метод*. Он тоже основывался на *психологическом доказательстве*, однако, при этом в основу рассуждения клалось небольшое число истин, из которых выводилось бесконечное множество других истин. Эта очень красивая идея стала очередным прорывом в математической мысли, определив её развитие на последующие тысячелетия.

Очень часто новые сущности возникали при попытках расширить действие операций в которых участвуют уже существующие объекты. Так появились отрицательные и рациональные числа. При помощи последних стало возможным описывать музыкальные гармоники, сколь угодно малые величины и геометрические построения. В мире везде усматривался их след. И при этом они были лишь парой целых чисел m и n! Вся гармония рухнула в один миг, когда кто то из учеников Пифагора доказал (убедив даже Учителя), что $\sqrt{2} \neq m/n$. Т.е. существует объект, определяемый формулой $\sqrt{2} \cdot \sqrt{2} = 2$, который не является рациональным числом. Потрясение было на столько велико, что пифагорейцы засекретили это *открытие*. Однако, такими волюнтаристическими методами остановить математическую мысль нельзя. На свободу вырвались сначала алгебраические числа подобные $\sqrt{2}$, а затем и такие монстры, как π и e, оказавшиеся не только не рациональными, но даже и не алгебраическими. Весь этот быстро разрастающийся зоопарк объединили в понятие действительного числа.

Одновременно с ним в математику пришла *бесконечность*. Она, конечно, была там и раньше. Неисчислимы звёзды на небе и капли в океане. Евклид знал, что простых чисел бесконечно много. Но все эти бесконечности были очень безобидными. Бесконечность "непрерывного" таила множество опасностей для главного орудия – убедительного математического рассуждения. Для их преодоления потребовалось изобретение исключительно полезного для практики исчисления бесконечно малых. Математики и физики научили друг друга дифференцировать, интегрировать, и это было хорошо.

Однако, настоящего математика нужды практики интересуют меньше всего. Он хочет убеждать и быть убеждённым. Поэтому пришёл Кантор.

Георг Кантор формализовал исходное понятие математической мысли – теорию множеств. Многие из его доказательств очень психологичны:

$$\text{СМОТРИ}: \qquad \begin{array}{ccccccc} 1 & 2 & 3 & 4 & \ldots & n & \ldots \\ \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \\ 2 & 4 & 6 & 8 & \ldots & 2n & \ldots \end{array}$$

Идея соответствия между двумя множествами и возможность "пересчитать" элементы множества при помощи натуральных чисел была простой и очень красивой, особенно когда оказалось, что пересчитать можно даже всюду плотные рациональные числа. После того, как всё было пересчитано, неожиданно выяснилось, что действительные числа несчётны. Вслед за этим посыпались логические парадоксы, часть которых была известна ещё древним, но рассматривалась как некий курьёз. В парадоксах проводятся некоторые убедительные рассуждения, приводящие к противоположным результатам. Под угрозой оказалась самая привлекательная черта математики – убедительность её доказательств!

Пришлось разработать методы *формальных теорий*, при помощи которых *часть* психологических доказательств можно записать в виде последовательности символов, которую, используя некоторый *алгоритм*, может проверить кто угодно не понимая *смысла* этих символов. Это была вершина вершин. Возникла возможность проводить *формальные доказательства*. Конечно совсем избавиться от психологии не получилось, но многие доказательства стали ещё убедительней, а математика приобрела статус исключительно точной науки.

После совсем небольшого периода торжества, Курт Гёдель ошарашил всех теоремой о том, что в математике "*существуют истинные формулы, которые невозможно доказать*." Это было на столько неожиданно и скандально, что появилось множество философских спекуляций, от утверждений об ограниченной познаваемости мира, до выводов о невозможности построения искусственного интеллекта. В последнем случае объявляется, что *человеческое мышление способно распознать истинность некоторого утверждения, даже если не может его доказать*. В тоже время компьютер, действуя формально (алгоритмически), не способен на такие рефлексивные откровения. Мы закончим эту главу анализом подобных заблуждений, но сначала подробно проанализируем понятия алгоритма, множества и формальной теории.

Теоремы "о неполноте" и "о несчётности" являются яркими примерами психологических доказательств, полученных при помощи метода рассуждений от противного. Поэтому, с его анализа мы и начнем.

II Доказательство от противного

Доказательство от противного – мощный и часто используемый в математике метод. Предположив, что некоторый факт (объект) является истинным (существует), и придя к противоречию, мы заключаем, что факт ложен (объект не существует). Рассмотрим несколько примеров.

• **Теорема Евклида** о бесконечности простых чисел $p_1, p_2, ...$ является классическим и самым простым рассуждением от противного:

> *Не существует самого большого простого числа* p_{max}.
>
> $\Diamond$: Пусть это не так, и самое большое простое число p_{max} существует. Построим число $p = p_1 \cdot p_2 \cdot ... \cdot p_{max} + 1$. Оно не делится ни на одно p_i, и больше чем p_{max}. Мы пришли к противоречию, следовательно, самого большого простого числа (как объекта!) не существует и простых чисел бесконечно много. $\square$

Заметим, что p не обязательно простое, так как его простой множитель может находится между p_{max} и p, но всё равно будет большим p_{max}.

• **Теорема об иррациональности** $\sqrt{2}$

> *Не существует натуральных* m *и* n*, таких, что* $\sqrt{2} = m/n$.
>
> $\Diamond$: Пусть это не так. Сократим общие множители у m, n, и возведём всё в квадрат: $2 = m^2/n^2$. Отсюда следует, что $m^2 = 2 \cdot n^2$ является чётным числом, поэтому m тоже чётно и представимо при помощи некоторого натурального k, как $m = 2 \cdot k$. Подставляя m в исходное соотношение, получаем $n^2 = 2 \cdot k^2$, а, следовательно, и n чётно. Но это противоречит тому, что мы сократили все общие множители, а значит таких m и n не существует. $\square$

Психологическая убедительность обоих доказательств не вызывает сомнений. Тем не менее, необходимо помнить, что получив противоречие, мы не всегда доказываем то, что *хотим* доказать. Противоречие не обязательно свидетельствует об ошибочности исходной посылки. Его может дать любое из утверждений использовавшихся при доказательстве. Особенно их много в теореме об иррациональности $\sqrt{2}$. Однако, они на столько "очевидны", что мы считаем ошибочной именно исходную посылку.

Видно, что схема доказательства приведенных теорем одинаковая. Мы показываем, что некоторый объект не существует, если предположение о его существовании приводит к противоречию.

• **Проблема Брадобрея**. В некоторой деревне все мужчины бреются либо сами, либо у брадобрея. Брадобрей (мужчина) бреет только тех, кто сам не бреется. Сформулируем теорему:

Брадобрей бреет себя сам.

◊ Пусть это не так, и брадобрей себя не бреет. Тогда он должен бриться у брадобрея. Значит брадобрей бреет себя. □

Сделав отрицание теоремы, и получив противоречие, мы должны прийти к выводу, что теорема верна. Но совершенно ясно, что это не так, и мы можем построить не только обратное доказательство, но и прямое: "если брадобрей бреется сам, то он не может бриться у брадобрея ...". В этом случае вновь получается противоречие.

Приведенное описание деревни со строгими правилами принадлежит Бертрану Расселу, как популярная формулировка проблем, возникающих в попытке *определить* "множество всех тех множеств, которые не содержат себя в качестве своего элемента". Мы умышленно явный парадокс представили в виде теоремы, чтобы продемонстрировать простой факт:

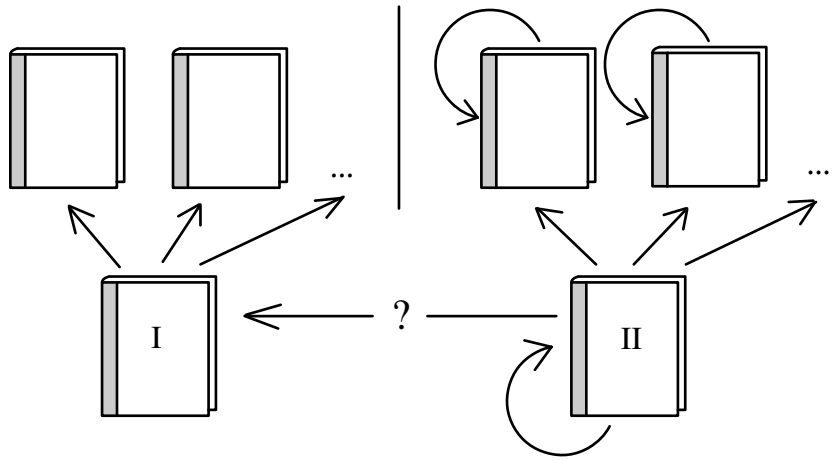
Получение противоречия в доказательстве от противного может свидетельствовать не об истинности теоремы, а о противоречивости объектов которые участвуют в её формулировке.

Другими словами, нельзя сказать: "возьмём множество всех множеств ..." и докажем "теорему о том, что ..." Сначала необходимо убедиться, что объект, о котором будет идти речь в теореме, существует. В частности, деревня, описанная Расселом, существовать не может. Конечно, возникает вопрос – "а что значит существовать или не существовать, и где не существовать?" Есть объект, определённый выше, и мы можем использовать его при построении новых объектов и теорем о них...

Дело в том, что математическое рассуждение явно или не явно исходит из некоторых аксиом. Именно аксиомы задают свойства объекта. Если в фиксированной системе аксиом поменять хотя бы одну аксиому, может получиться объект с совершенно другими свойствами. Понятно, что произвольно задавать аксиомы нельзя. Они не должны быть *противоречивыми*, иначе никакого объекта определять не будут. Или, другими словами, – объект определяемый при помощи противоречивых аксиом не существует.

Подробнее мы обсудим элементы формальных аксиоматических систем в следующем разделе, где снова проанализируем проблему брадобрея. Сейчас же рассмотрим ещё одну версию того же парадокса.

• **Проблема Библиотекаря**. Существует Библиотека с книгами. Любая книга внутри своего текста может упомянуть сама себя (например, в списке литературы привести свое название). Соответственно все книги можно разделить на две группы. В первую попадают книги, которые на себя не ссылаются, а во вторую – ссылающиеся на себя книги. Кроме этого, существуют две книги, являющиеся каталогами всех книг Библиотеки. Первый каталог перечисляет все те книги, которые на себя не ссылаются, а второй, наоборот – все ссылающиеся на себя книги:



Сформулируем теперь теорему:

Первый каталог содержит в списке книг себя.

◊ Пусть это не так. Тогда первый каталог содержится во втором (все книги перечислены в обоих каталогах и каталог есть книга). Но во втором каталоге перечисляются только самоссылающиеся книги, и первого каталога там быть не может. Мы пришли к противоречию, следовательно теорема верна. □

Если мы остановимся на этом этапе, то получим заведомо неверный вывод. Понятно, что первый каталог на себя ссылаться не может (он является каталогом не самоссылающихся книг). Как и в случае с брадобреем, мы можем провести как обратное доказательство (от противного), так и прямое. И оба раза получить противоречие.

О чём оно говорит? Понятно, что не об истинности или ложности теоремы. Веря в то, что два различных доказательства должны всегда приводить к одному и тому же, мы вынуждены сделать вывод: *объект Библиотека*, с заданными свойствами, *существовать не может*.

Любая ссылка на "естественность" или "видимую не противоречивость" исходных определений не достойна математика, так как это уже эмоции. Единственный путь – попытаться уйти от психологических формулировок и доказательств к формальным.

• **Парадокс лжеца**. Вся математика состоит из логических утверждений. При этом логика математики бинарна. Утверждение "$2 < 3$" или истинно или ложно. Третьего не дано. Именно эта бинарность придаёт математическому доказательству ту чудесную убедительность, ради которой всё и затевалось. Введем обозначение того, что некое логическое утверждение A является истинным: $\mathbf{True}(A)$.

На самом деле обозначение **True** излишне, так как записывая в качестве аксиомы или посылки некоторое утверждение A, мы предполагаем его истинность. Однако, такое обозначение будет удобно для дальнейшего. *Определим* высказывание:

$$L \ : \ \neg\mathbf{True}(L),$$

где "$\neg$" – знак логического отрицания, а после двоеточия идёт *определение* утверждения L. Оно является вариантом парадокса лжеца: "L – истинно, если не истинно L". Сформулируем следующую теорему:

Утверждение L является истинным: L=И.

$$\Diamond_1: \text{ пусть } L = \text{Л} \quad => \quad \mathbf{True}(L) = \text{Л} \quad => \quad L = \neg\mathbf{True}(L) = \text{И}. \ \square$$

(Далее "=>" означает логический вывод; "И" – истина, "Л" – ложь). В доказательстве от противного, мы пришли к противоречию. Поэтому исходная посылка $L = \text{Л}$ не верна и, следовательно, теорема верна. Однако понятно, что это не так. Мы можем провести доказательство и в прямом направлении:

$$\Diamond_2: \text{ пусть } L = \text{И} \quad => \quad \mathbf{True}(L) = \text{И} \quad => \quad L = \neg\mathbf{True}(L) = \text{Л} \ \square,$$

и снова прийти к противоречию. Таким образом, мы не способны ни доказать ни опровергнуть теорему и ходим по замкнутому кругу.

Причина этого, как и раньше, состоит в том, что объект L *используемый при формулировке теоремы* противоречив и, следовательно, не может существовать. Второй вопрос, почему объект L, построенный столь "конструктивным" образом, не существует? Вокруг парадокса лжеца ломают копья со времен древних греков. Самое простое объяснение состоит в том, что при определении L используется *бесконечная рекурсия*. Объект определяется сам через себя и при этом, говоря языком программирования, нет точки остановки этой рекурсии. Например, компьютер не смог бы оперировать с таким определением и просто "завис" бы. Это же рискует сделать и человек пытающийся глубоко вдуматься в L. Поэтому, такое построение (определение) объекта L просто некорректно.

III Формальные доказательства

• Прийдя к двойным противоречиям в задачах о брадобрее, библиотеке и лжеце, мы сделали вывод, что объекты использующиеся при формулировки теоремы не существуют. Чтобы глубже понять утверждение "не существуют", необходимо определить понятие "формальное доказательство". Более последовательно эта тема будет обсуждена во второй главе, а сейчас мы сделаем небольшой и стремительный её обзор.

Почему психологическое доказательство убеждает большинство математиков? Потому, что в своей основе оно содержит небольшое число типовых схем рассуждения. К этим схемам привыкли и *договорились* считать их убедительными, т.е. логичными. Таким образом математика использует *логику*, т.е. допустимые методы убеждения $\ddot\smile$.

Кроме логики, психологические доказательства используют естественный язык. А это очень не однозначный способ выражения мысли. Он хорош для поэзии, но не для математики. Одно и тоже слово или фраза могут выражать для разных людей, в разном контексте совершенно различный смысл. Например дискуссии на тему парадокса лжеца в форме – "некто утверждает, что он лжец" часто порождают множество заявлений лежащих вне математики: "лжец врёт регулярно, но не всегда", или "в утверждении смешивается два разделённых во времени действия – произнесение фразы и её оценка говорящим", и т.д. Всё это может быть верным для некоторого подмножества людей с точки зрения обыденных суждений, но не имеет никакого отношения к математике.

Так как математика должна однозначно высказываться об однозначно определённых сущностях, ей необходимо, *по возможности*, отказаться от естественного языка. Точнее, она должна говорить при помощи символов, не имеющих для человека никаких чувственных и многозначных аналогов. В более экстремистской форме это звучит так:

> Предметом математики являются только объекты и суждения, однозначным образом выражающиеся на формальном языке символов, которые сами по себе не имеют никакого смысла. То, что так записаться не может – математикой не является.

Произвольный математический объект из некоторого класса (с определёнными свойствами) мы обозначим маленькими латинскими буквами из конца алфавита x, y, z. Конкретный объект обозначается буквами из начала алфавита a, b, c или специальными символами $0, 1, \varnothing$, и т.п. Всегда можно договориться о синтаксисе и однозначно отличать константные сущности от переменных (произвольных).

Сущности могут вступать между собой в некоторые отношения, в результате чего возникают *высказывания* о них. Высказывания могут быть константными: "$1 < 2$", брадобрей – мужчина: "$M(a)$", или переменными: $x < 0$, некоторый житель деревни x мужчина "$M(x)$". В последнем случае они называются *предикатами*. Любые высказывания считаются либо истинными либо ложными. В случае предикатов, при одних x, y, z они истинны, а при других ложны. Для натуральных чисел 0,1,2,... предикат $x < 1$ истинен при $x = 0$, и ложен при всех остальных. Равенство $x = y$ это тоже предикат, истинный когда сущности x и y совпадают.

Из простых высказываний (константных или переменных) можно создавать сложные высказывания при помощи логических связок. Так фразу "если A то B", мы заменяем на формальную формулу: $A \to B$. Так, утверждение "если x больше 4-х, то оно больше 2-х" имеет следующий вид: $(x > 4) \to (x > 2)$. Естественно, x может быть и не больше 4-х, но в целом эта формула для натуральных чисел всегда истинна, так как *если* выполняется *посылка* $(x > 4)$, то будет истинно и *следствие* $(x > 2)$. Символ "$\to$" называется *импликацией*. Имея две формулы вида $A \to B$ и $B \to C$, мы можем записать новую формулу $A \to C$:

$$A \to B,\ B \to C \qquad => \qquad A \to C.$$

Если из A следует B, а из B следует C, *то* из A следует C. Это пример *вывода* или *формального доказательства*. Т.е. если нам необходимо доказать формулу $A \to C$, то сначала потребуется доказать формулы $A \to B$ и $B \to C$. Эта схема доказательства считается универсальной и применимой к любым высказываниям, будь то арифметика, геометрия или мир безбородых мужчин. Проведение такого формального доказательства не требует понимания *смысла* высказываний $A,\ B$ и C.

Кроме импликации вводится ещё ряд логических связок для получения составных высказываний "и" $(\&)$, "или" $(\vee)$, "не" $(\neg)$. Если два высказывания одновременно истинны или ложны, мы говорим, что они эквиваленты $A \leftrightarrow B$. Так, справедлив ещё один вид формального доказательства (вывода нового утверждения):

$$A \to B,\ \ B \to A \qquad => \qquad A \leftrightarrow B.$$

Аксиомами является набор формул, которые в данной теории предполагаются истинными. Естественно, не любые формулы могут лечь в основание теории, а только те, которые *непротиворечивы*. Это означает, что из них нельзя вывести заведомо ложные формулы. К таким относят, например, $A \leftrightarrow \neg A$ ("A эквивалентно не A") или $A \,\&\, \neg A$.

• Рассмотрим теперь парадокс брадобрея. Утверждение о том, что "некий x бреет y-ка" обозначим предикатом $S(x,y)$. В этом случае $S(x,x)$ означает, что "x бреет сам себя". Введём константу "a" – имя брадобрея (фиксированный объект) и запишем аксиомы теории:

$$\neg S(x,x) \to S(a,x), \qquad\qquad S(a,x) \to \neg S(x,x).$$

Первая аксиома гласит: "если *любой* x сам себя не бреет, то его бреет a". В силу принципиальности брадобрея, считается верной и вторая аксиома: "любой кого бреет брадобрей не бреет себя".

Записав аксиомы, мы можем отдать их анализировать человеку, который о *содержательном смысле* предиката $S(x,y)$ и константы a ничего не знает. Для него это абстрактные символы, составляющие формулы объявленные истинными (аксиомы). Однако, по совместной договорённости он знает, что если есть две формулы вида $A \to B$, $B \to A$, то можно записать формулу $A \leftrightarrow B$. Что он и делает:

$$S(x,x) \leftrightarrow \neg S(a,x).$$

Затем этот математик использует правило конкретизации. Если в формуле есть переменная (произвольный объект), то на её место можно поставить любую константу, например "a". Проделав это, он получит всегда ложную формулу вида $A \leftrightarrow \neg A$. Ему ничего не остаётся, как вынести обвинительный приговор – представленная система аксиом противоречива и не определяет никакого предиката $S(x,y)$.

• С парадоксом лжеца ситуация чуть более сложная. Первоначально он имел форму высказывания "любой критянин лжец", что записывается так: $C(x) \to L(x)$, где $C(x)$: "x – это критянин ", и $L(x)$: "x – лжец". Проблема появляется, если произносящий эту фразу сам является критянином. Точнее, если он единственный в мире критянин $\ddot\smile$, так как утверждение $\neg(C(x) \to L(x))$ означает "не любой критянин лжец" (т.е. существуют не лжецы).

Однако, даже если критянин один (т.е. вместо предикатов стоит просто высказывания C, L), определённые проблемы вызывает запись заявления о том, что C "произносит эту фразу". Можно, например, *определить* сущность L (чем бы она не являлась) при помощи 3-х аксиом:

$$\begin{array}{ll} L \to \neg(C \to L) & : \text{если } L \text{ истинно (лжец), то } C \to L \text{ не истинна,} \\ \neg L \to (C \to L) & : \text{если } \neg L \text{ (не лжец), то фраза } C \to L \text{ истинна,} \\ C & : \text{констатация факта, что это критянин.} \end{array}$$

Как мы увидим во второй главе, эти три аксиомы противоречивы и из них одновременно следует L и $\neg L$. Хотя попарно, они являются вполне допустимыми системами.

• Теперь утверждение "объект не существует" приобретает понятный смысл. По *договорённости*, в обсуждениях могут присутствовать (существовать) только объекты заданные при помощи непротиворечивой системы аксиом. "Убедительность доводов" типа "ну как же, есть деревня, есть брадобрей, ..." теперь ни на кого не произведёт впечатления.

Надо понимать, что это Игра, которая имеет свои Правила. Играющие должны их придерживаться. Вот и всё. Второй вопрос – "почему эта Игра позволяет предсказывать затмения Солнца или поведение пыльцы под микроскопом?" Но это уже вопрос из совершенно другой Игры.

Задание объектов при помощи аксиом не единственный метод. Второй подход – это рекурсивное определение объекта самого через себя. Так, если при помощи аксиом в арифметике мы *задали* операцию получения следующего числа: $x+1$ (добавления камня), то можно *определить* функцию сложения чисел x и y при помощи следующих двух формул:

$$\begin{aligned} &a(x,0) = x \\ &a\big(x,y+1\big) = a(x,y)+1. \end{aligned}$$

В предыдущем разделе, аналогично, при помощи рекурсии был определён объект $L = \neg\mathbf{True}(L)$. Но, в отличие от функции сложения это определение некорректно, так как не содержит выхода из рекурсии типа $a(x,0) = x$. Это несколько иная *алгоритмическая некорректность*.

Хотя разнообразными алгоритмами математики пользовались с незапамятных времён, его понятие, как математического объекта, возникло недавно. Рядом с ним стоит ещё одно понятие – конструктивный объект, т.е. объект который можно *явно* задать (определить) в виде *конечной* последовательности символов. При этом не важно: пользуемся ли мы буквами или цифрами. Переобозначением их можно заменить друг другом. Важна только конечность предъявления этого объекта.

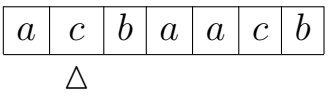
На самом деле конструктивными является большинство математических объектов, первым из которых была кучка камней на земле в пещере. Натренированная на них математическая интуиция осмеливается размышлять также об объектах неконструктивных. Правда, часто только для того, чтобы тут же доказать их бессмысленность (пусть существует, не знаем какое, самое большое простое число...). Однако, иногда противоречия сразу не возникают, и неконструктивные объекты получают от сообщества математиков право на существование.

Алгоритмы являются тем миром, где разрешены только конструктивные объекты. Компьютер может работать с символами алгебры, геометрическими построениями, однако в основе всего лежат натуральные числа и вычислимые функции, которые заданы на множестве этих чисел.

IV Вычислимые функции и их алгоритмы

Вычислимой называется функция $f(x_1, ..., x_n)$ от одного или нескольких натуральных аргументов, натуральные значения которой получаются в результате выполнения некоторого *алгоритма*. Определить термин "алгоритм" без синонимичных ссылок на себя же самого сложно. Фразы типа "дискретные, однозначные последовательные действия..." не являются полностью удовлетворительными. Поэтому считается, что "алгоритм", как и "множество" – это понятие первичное. Для определённости будем считать, что алгоритм это математическая модель уже *существующего* физического устройства – цифрового *компьютера*. У него идеализируется память, которая может быть сколь угодно большой. Естественно, исключаются и форс-мажорные ситуации - перегрев, блондинки, чашки с кофе и т.п. Поэтому вычислимой будет функция, для которой можно составить *программу*, т.е. дать *конечное* определение, позволяющее вычислить функцию при помощи этого идеального устройства.

Программа – это конечная строка символов из некоторого, опять же конечного алфавита. Компьютер имеет вычислитель (маркер), который *однозначным* образом перемещается по строке (тексту программы). Это перемещение не обязательно линейно, и при достижении некоторого символа возможен переход на несколько символов вперёд или назад.



Таким образом есть программа (строка "*acbaac*") и вычислитель (маркер), который на данном шаге выполнения алгоритма указывает на один символ – инструкцию для компьютера (на рисунке выше это треугольник, стоящий на инструкции "*c*").

Программы записываются на некотором формальном *языке* программирования. Это может быть язык машины Тьюринга, ассемблер для процессоров с бесконечной разрядностью, или C++, в котором нет ограничения на размер целочисленных переменных. Формальный язык определяется при помощи *синтаксических правил*. Это означает, что всегда можно отличить "бессмысленную" последовательности символов от тех, по которым может перемещаться вычислитель. Ситуация полностью аналогична, например, арифметике, где правильно построенная формула "$(2 < 1)$" отличима от неправильной: "$2)(1 <$".

Одна и та же функция может быть описана при помощи *различных* алгоритмов, каждый из которых может быть представлен *различными* программами (использующими различный алфавит и синтаксис).

Кроме программы и вычислителя существует *память*, в которую могут помещаться натуральные числа, вычисляющиеся по мере выполнения алгоритма. Память можно считать некоторым бесконечным упорядоченным множеством $\mathcal{M} : \{m_0, m_1, m_2, ...\}$. Величины m_i называются "*ячейками памяти*". Перед началом работы алгоритма, все m_i, для определённости, равны нулю. В момент "запуска" алгоритма на выполнение, в первые n элементы памяти $\mathcal{M}$ помещаются значения n входных аргументов функции $f(x_1, ..., x_n)$. Порядок перемещения вычислителя по тексту программы зависит как от символов программы, так и от значений чисел в памяти (начальных или "промежуточных").

Таким образом, компьютер состоит из *конечной* программы $\mathcal{P}$, вычислителя $\mathcal{C}$, указывающего на выполняемую инструкцию, и *неограниченной* памяти $\mathcal{M}$, в которую могут помещаться и считываться натуральные числа. Ниже приведен пример трёх последовательных шагов некоторого алгоритма, вычисляющего функцию одной переменной $f(x)$ при $x = 5$:

$\mathcal{P}:$	$\boxed{a}\boxed{c}\boxed{b}\boxed{a}\boxed{a}$	$\mathcal{P}:$	$\boxed{a}\boxed{c}\boxed{b}\boxed{a}\boxed{a}$
$\mathcal{C}:$	$\triangle$	$\mathcal{C}:$	$\triangle$
$\mathcal{M}:$	$\{5, 0, 0, ...\}$	$\mathcal{M}:$	$\{5, 2, 0, ...\}$

$\mathcal{P}:$	$\boxed{a}\boxed{c}\boxed{b}\boxed{a}\boxed{a}$
$\mathcal{C}:$	$\triangle$
$\mathcal{M}:$	$\{0, 2, 0, ...\}$

При вычислении значения любой функции, компьютер получает на вход значение аргументов этой функции и помещает их в память $\mathcal{M}$. Затем он начинает перемещать вычислитель по тексту программы, изменяя значения чисел в памяти. При достижении некоторого символа происходит остановка вычислителя и выдача значения функции. Вообще говоря, возможна ситуация, когда перемещение по строке программы никогда не заканчивается, или вычислитель дошёл до конца строки не встретив символа остановки. В этих случаях считается, что значение функции не *определено*. Другими словами, вычислимая функция не является обязательно всюду определённой функцией.

Функция – это "ящик" с несколькими входами x, и одним выходом y:

$$y = f(x_1, x_2, ..., x_n): \qquad x_1, x_2, ..., x_n \to \boxed{f} \to y.$$

Фразы "один вход" или "один выход" не являются принципиальными. Понятно, что n упорядоченных целых чисел $(x_1, x_2, ..., x_n)$ всегда можно представить как одно, и наоборот. Для этого достаточно взять n первых *простых чисел* p_i и вычислить $x = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdot 7^{x_4} \cdot 11^{x_5} \cdot ... \cdot p_n^{x_n}$. Обратная операция разложения на простые множители одного числа x позволяет восстановить значения всех x_i. Тем не менее, будем считать, что может быть несколько входов, но выход всегда один.

• Компьютер может вычислять некоторые "очевидно вычислимые" функции, например, складывать два числа: $m_1 + m_2$. Они берутся из памяти, а индекс обозначает номер числа в упорядоченной последовательности $\mathcal{M}$. Результат тоже помещается в память, с указанием номера ячейки. Для этого в языке есть операция присвоения: $m_3 \leftarrow m_1 + m_2$; (1-я и 2-я ячейки складываются и отправляются в 3-ю):

$$m_1 \leftarrow 5; \quad m_2 \leftarrow m_1; \quad m_3 \leftarrow m_1 + m_2;$$

(в ячейке m_3 будет 10). Точка с запятой – это разделительный символ (отделяющий инструкции). Пробелы компьютером пропускаются. В качестве номера i ячейки m_i может выступать явно заданное натуральное число: m_2 или число из другой ячейки: m_{m_1}.

Кроме этого, можно выяснить равны ли между собой два числа:

$$\textbf{if}(m_i = m_j)\ \{\mathcal{P}_1\} \quad \mathcal{P}_2$$

Символы $\mathcal{P}_i$ обозначают некоторые куски программы (строки). Если число m_i равно m_j, то вычислитель переходит к выполнению строки $\mathcal{P}_1$, после чего выполняет строку $\mathcal{P}_2$. Если же условие в круглых скобках не выполняется, то $\mathcal{P}_1$ *пропускается* и сразу начинает выполняться строка $\mathcal{P}_2$. Выполнить кусок программы $\mathcal{P}_1$ означает, что вычислитель должен дойти до конца строки, которая её описывает. Фигурные скобки ограничивают блок $\mathcal{P}_1$. Для упрощения разрешим также символ меньше: $\textbf{if}(m_i < m_j)\ \{\mathcal{P}_1\}\ \mathcal{P}_2$, т.е. если $m_i < m_j$ то $\mathcal{P}_1\mathcal{P}_2$, иначе $\mathcal{P}_2$.

Компьютер может перескочить в любую точку программы. Для этого вводится понятие *метка* к которой нужно перепрыгнуть. Так, следующий алгоритм:

$$\mathcal{P}_1 \quad \#1; \quad \mathcal{P}_2 \quad \textbf{goto}(1); \quad \mathcal{P}_3$$

требует от компьютера выполнить $\mathcal{P}_1$, затем *проигнорировать* "#1;", выполнить $\mathcal{P}_2$, а встретив "**goto**(1);", найти в тексте программы "#1;" и начать работать сразу после неё (выполнять $\mathcal{P}_2$). Предполагается, что метка "#1;" с данным номером в программе *есть* и только *одна*. Различных меток "#1;" "#2;",... может быть сколь угодно много.

Последняя, необходимая для описания алгоритмов команда

$$\textbf{return}\ m_i;$$

приводит к остановке работы компьютера с сообщением, что в ячейке m_i находится результат вычисления (т.е. возвращается значение функции). Даже если вычислитель $\mathcal{C}$ дошёл до конца программы $\mathcal{P}$, и остановился, но там нет инструкции **return**, – то алгоритм не закончен!

• Приведенный выше язык составления программ алгоритмов будем называть BASE. Он и объекты $\langle \mathcal{P}, \mathcal{C}, \mathcal{M} \rangle$ *достаточны* для описания *любого* алгоритма. Точнее, наоборот:

> *всё, что выражается в терминах приведенных выше символов, по определению является алгоритмом.*

Для дальнейшего анализа удобно ввести понятие совокупности функций, хотя их всегда можно имитировать в рамках только BASE.

Пусть программа может состоять из упорядоченного множества функций, каждая из которых имеет свой уникальный номер (имя). Программа каждой функции заключается в фигурные скобки, перед которыми идёт её имя с указанием числа аргументов, которых может и не быть:

$$\begin{array}{lllll} f_1() & \{ & \textbf{return } 1; & & \} \\ f_2(m_0) & \{ & m_1 \leftarrow m_0 + 25; & \textbf{return } m_1; & \} \\ f_3(m_0, m_1) & \{ & m_2 \leftarrow m_0 + f_2(m_1); & \textbf{return } m_2; & \} \end{array}$$

Указание аргументов функции в скобках необходимо только для того, чтобы единообразным образом можно было использовать их и в текстах описания алгоритма функции. Другими словами, мы предполагаем, что значение вычислений *некоторой* функции (число) может быть использовано в алгоритме *любой* функции, также как и ячейка памяти (см. выше описание f_3). Всё это уже расширение языка (т.н. BASE с функциями).

Теперь необходимо сделать очень важное дополнение. Когда в тексте встречается вызов функции, вычислитель останавливается и *ждёт* пока она не вернёт значение. Во время ожидания память вычислителя *не изменяется*. Компьютер создаёт *ещё один* вычислитель, который начинает перемещаться по тексту вызываемой функции. Новый вычислитель имеет *собственную память*, которую использует при работе алгоритма. Поэтому вычисление некоторой функции может провести к *последовательному* созданию вычислителей с собственными массивами памяти. Пока последний созданный вычислитель не доберётся до команды **return**, все созданные перед ним находятся в состоянии ожидания.

Приведенное описание синтаксиса программы и её интерпретации компьютером можно перевести на формальный язык. Начало будет выглядеть примерно так. "Пустая строка символов является программой. Если $\mathcal{P}_1$ и $\mathcal{P}_2$ программы, то их последовательное соединение $\mathcal{P}_1\mathcal{P}_2$ это тоже программа. Если $\mathcal{P}$ программа, а i, k целые числа, то "$\mathcal{P}$ $m_i \leftarrow k$;" – программа", и т.д., и т.п. Естественно, подобные описания также являются алгоритмами. Однако мы считаем их на столько простыми и естественными, что будем рады *свести к ним* любые другие, сколь угодно сложные алгоритмы.

V Немного программирования

Рассмотренный язык BASE является достаточно "высокоуровневым" и может быть сведен к ещё более простым элементам. Однако, если бы программисты были ограничены даже перечисленным синтаксисом, их можно было бы только пожалеть. Для повышения выразительности языка вводятся разнообразные расширения обозначений, которые, однако, однозначным образом расшифровываются в терминах базового синтаксиса. Поэтому, перед выполнением программы, её необходимо *компилировать*, т.е. преобразовать входную строку программы с расширенным алфавитом в другую, с более лаконичным. Например, если в тексте встречается символ "$\neq$" (не равно), то необходима такая "компилляция":

$$\textbf{if}(x \neq y)\{\ \mathcal{P}_1\ \}\ \ \mathcal{P}_2 \qquad => \qquad \textbf{if}(x = y)\{\ \textbf{goto}(1);\ \}\ \ \mathcal{P}_1\ \ \#1;\ \mathcal{P}_2;$$

На месте букв x и y могут стоять m_i и m_j с *любыми* индексами i и j. Стрелка "=>" обозначает компиляцию исходной строки в новую, "понимаемую" компьютером. Исходный текст подразумевает, что если $x \neq y$, то выполняется $\mathcal{P}_1\mathcal{P}_2$, если же нет (т.е. $x = y$), то выполняется только $\mathcal{P}_2$. Именно это и происходит в преобразованной программе.

Аналогично можно *определить* инструкции для проведения циклических вычислений. Так, ниже слева приведен исходный программы, которая вычисляет сумму натуральных чисел до числа хранящегося в ячейке m_0 включительно. Справа от черты, та же функция после компилляции (т.е. приведения в исходный синтаксис):

$m_1 \leftarrow 0; \quad m_2 \leftarrow 0;$
while$(m_1 < m_0)\{$
    $m_1 \leftarrow m_1 + 1;$
    $m_2 \leftarrow m_2 + m_1;$
$\}$
return $m_2;$

$m_1 \leftarrow 0; \quad m_2 \leftarrow 0;$
$\#1;$
if$(m_1 < m_0)\{$
    $m_1 \leftarrow m_1 + 1;$
    $m_2 \leftarrow m_2 + m_1;$
    goto$(1);$
$\}$
return $m_2;$

В расширенном синтаксисе считается, что часть программы, находящаяся внутри фигурных скобок, после команды цикла **while**(условие) повторяется до тех пор, пока условие срабатывает (в нашем случае, пока счётчик чисел в ячейке m_1 меньше, чем число в m_0). На самом деле первые строки не нужны, так как по договорённости, перед запуском программы все ячейки её памяти обнуляются.

• Расширение синтаксиса, требующее переделки компиллятора (а это тоже алгоритм!), – не единственный способ усиливать выразительность языка. Например, в базовом синтаксисе нет функции получения предыдущего целого числа (т.е. вычитания единицы). Поэтому можно добавить функцию проводящую такие вычисления, и в других функциях её вызывать. Приведём, например, такой вариант:

$$
\begin{array}{l}
\mathbf{p}(m_0)\{ \\
\quad m_1 \leftarrow 0; \\
\quad \textbf{while}(m_1 < m_0)\{ \\
\quad\quad m_2 \leftarrow m_1 + 1; \\
\quad\quad \textbf{if}(m_2 = m_0)\ \textbf{return}\ m_1; \\
\quad\quad m_1 \leftarrow m_2; \\
\quad \} \\
\}
\end{array}
$$

Для читаемости функция названа "$\mathbf{p}$", однако необходимо помнить, что это на самом деле что-то типа: "f_{137}".

В $\mathbf{p}(m_0)$ происходит перебор всех чисел начиная с нуля (переменная m_1) до тех пор, пока следующее после m_1 число (m_2) не окажется равным входному аргументу m_0. Этот алгоритм назвать быстрым сложно, но нас сейчас это не должно заботить.

Напомню, что мы работаем только с целыми неотрицательными числами (0,1,2,....), поэтому приходим к первой (в длинной цепочке $\ddot\smile$) алгоритмической неприятности. Функция $\mathbf{p}(x)$, хотя описывается алгоритмом (она вычислима), определена не для всех значений x. В частности, для нуля нет предыдущего значения, и $\mathbf{p}(0)$ не доберётся до команды **return**. Если в ячейке m_0 находится 0, то условие в круглых скобках цикла **while** не сработает, и тело цикла (в фигурных скобках) будет пропущено. Вычислитель дойдёт до конца программы функции и остановится, так как команды **return** там нет, но и текст алгоритма закончился.

Физический компьютер может в этот момент замигать лампочками, перегрузить операционную систему или покрыть небо звёздами. Наш идеальный компьютер просто "зависнет". Мы не можем написать в конце программы, например **return** 0;, так как это уже будет другая функция (не предыдущего числа). Нельзя и вернуть не число, а например строку "NAN", так как в других функциях $\mathbf{p}(m_0)$ используется в арифметических вычислениях, да и нет "NAN"-а в нашем синтаксисе. Таким образом:

вычислять, не значит вычислить.

Например, вычислимой будет и такая функция: $\mathbf{f}(m_0)\{$ **while**$(1 = 1)\{\}$ $\}$.

• Определим теперь ещё одну функцию $\mathbf{mult}(x, y)$ умножения двух целых чисел. Для целей дальнейшего анализа мы снова выберем не самый эффективный способ. Однако, он проиллюстрирует важную концепцию программирования – *рекурсию*:

$$
\begin{array}{l}
\mathbf{mult}(x, y)\{ \\
\quad \mathbf{if}(y = 0)\{\ \mathbf{return}\ 0;\ \} \\
\quad \mathbf{if}(y = 1)\{\ \mathbf{return}\ x;\ \} \\
\quad \mathbf{return}\ x + \mathbf{mult}(x,\ \mathbf{p}(y)\ ); \\
\}
\end{array}
$$

Для повышения читаемости, мы снова вместо номера функции пишем имя "**mult**", а вместо входных ячеек памяти m_0 и m_1 переменные x и y. Кроме этого несколько расширен синтаксис, и в один приём производится вычисление двух функций "**p**", "**mult**", и затем выполняется сложение. В исходном синтаксисе третий **return** должен был бы выглядеть так:

$$m_2 \leftarrow \mathbf{p}(m_1); \quad m_3 \leftarrow \mathbf{mult}(m_0, m_2); \quad m_2 \leftarrow m_2 + m_3; \quad \mathbf{return}\ m_2;$$

Поэтому в записи $x+\mathbf{mult}\big(x, \mathbf{p}(y)\big)$ сначала происходит определение первого аргумента функции "**mult**" (это просто число x), затем *вычисляется* функция $\mathbf{p}(y)$ и её значение помещается во второй аргумент "**mult**", и только после этого происходит её вычисление. В дальнейшем предполагается, что подобные выражения всегда вычисляются слева направо, хотя понятно, что при помощи ячеек памяти можно реализовать любой порядок вычисления аргументов функции.

Алгоритм $\mathbf{mult}(x, y)$ вызывает две функции, одна из которых снова "**mult**". Функция, которая при своём вычислении использует себя же саму, называется *рекурсивной*.

В предыдущем разделе мы определили как компьютер обходится с вызовами функций. Он останавливает текущий вычислитель, создаёт новый, обладающий собственной памятью, и запускает его перемещаться по программе вызываемой функции. Когда этот вычислитель доходит до **return** он уничтожается, а первый вычислитель, получив результат, продолжает свою работу. При этом абсолютно всё равно какая функция вызывается, поэтому рекурсия (самовызов) вполне корректная операция.

В функции $\mathbf{mult}(x, y)$ происходит следующее. Если $y = 0$, то её результат равен 0 , а если $y = 1$, то x. Если же y больше единицы, то мы пользуемся "очевидной" арифметической формулой $x{*}y = x{+}(x{*}(y{-}1))$. При каждом вызове $\mathbf{mult}(x, \mathbf{p}(y))$ значение второго аргумента уменьшается, поэтому рано или поздно сработает вторая строка алгоритма.

Например, справедлива следующая цепочки формул для $x = 4$, $y = 3$:

$$\mathbf{mult}(4,3), \text{ это } 4+\mathbf{mult}(4,2), \text{ это } 4+(4+\mathbf{mult}(4,1)), \text{ это } 4+(4+4).$$

Ответ на каждое "это" производится отдельным вычислителем. Понятно, что такие действия приведут к результату только если цепочка рекурсивных вызовов когда-либо закончится. Например, если в приведенной выше программе $\mathbf{mult}(x, y)$ убрать первые два **return**, то функция никогда не выдаст результата. При этом будет порождено, вообще говоря, конечное число вычислителей. Однако, последний из них внутри функции $\mathbf{p}(0)$ остановится не достигнув **return**, и сделает самую первую функцию не определённой.

Ещё хуже будет ситуация, если в алгоритме заменить "$\mathbf{mult}(x, \mathbf{p}(y))$", например на $\mathbf{mult}(x, y+1)$. Мы получим бесконечную цепочку:

$$\mathbf{mult}(4,3), \text{ это } 4+\mathbf{mult}(4,4), \text{ это } 4+(4+\mathbf{mult}(4,5)), \text{ это } ...$$

Теперь компьютер не только не получит значение функции, но и будет порождать всё новые вычислители и их память. Для реального компьютера, создающего цепочку таких "вычислителей", рано или поздно закончатся ресурсы, и он "умрёт". Идеальный компьютер будет работать бесконечно долго, а следовательно подобная функция не определена ни при каких $x > 1$ (при $x = 0$ и $x = 1$ сработают первые две строчки), и является частично вычислимой или просто *частичной*.

• Заканчивая расширения синтаксиса описания алгоритмов при помощи функций, введём ещё *глобальную память* $\mathcal{G} = \{g_0, g_1, g_2, ...\}$. Как и для памяти каждого вычислителя – это множество упорядоченных неотрицательных целых чисел. К ним может обращаться любой вычислитель, перемещающийся по некоторой функции. Другими словами, в тексте программ функций могут встречаться как m_i, так и g_i. После создания или уничтожения вычислителей, память $\mathcal{G}$ сохраняется. Таким образом, это *общая память* которую можно изменять из любой функции.

С помощью глобальной памяти можно реализовать не функциональное поведение. Обычная функция, при данном входе, всегда выдаёт один и тот же выход. *Автоматом* называют вариант, такого же как и функция, ящика со входом и выходом. Однако результат его вычисления может зависеть не только от данного значения входа, но и от всей предыстории вызовов этого автомата. Автоматным поведением обладают очень многие системы. Так, если Вам наступили в транспорте на ногу в первый раз – Вы вежливо поморщитесь, тоже воздействие на второй раз вызовет недовольную гримасу, ну а в третий раз...)

• Рассмотрим несколько задач, решение которых может быть найдено в приложении "*Помощь*" под соответствующим номером.

▷ Базовый язык алгоритмов может быть упрощён. На самом деле, достаточно единственной арифметической функции прибавления единицы (следующее натуральное число) и логического равенства.

($\lessdot$ H$_1$) Написать при помощи базового синтаксиса программирования функцию сложения двух чисел **add**(x, y), используя только функцию $m_i \leftarrow m_j + 1$. Рекурсию не применять.

($\lessdot$ H$_2$) Написать функцию **lt**(x, y), возвращающую 1, если $x < y$ и 0 в противном случае. Использовать только операцию сравнения $x = y$.

($\lessdot$ H$_3$) Написать функцию **mult**$(x, y) = x * y$, не используя рекурсии.

▷ Программу можно записать при помощи только натуральных чисел. Для этого необходимо воспользоваться низкоуровневым языком программирования ZIPPER. В нём есть только пять команд:

команда	код	описание
$Z(7)$	$1, 7, 0, 0$	$m_7 \leftarrow 0:$ обнулить ячеку 7
$I(7)$	$2, 7, 0, 0$	$m_7 \leftarrow m_7 + 1:$ увеличить значение ячейки 7
$P(7, 5)$	$3, 7, 5, 0$	$m_7 \leftarrow m_5:$ присвоить ячеке 7 значение ячейки 5
$E(7, 5, 1)$	$4, 7, 5, 1$	если $m_7 = m_5$ перейти к команде 1
$R(7)$	$5, 7, 0, 0$	**return** m_7

Таким образом, программа состоит из $4 \cdot n$ натуральных чисел. Каждая команда является четвёркой, первым числом которой идёт её идентификатор (число от 1 до 5, или для мнемоники буквы Z, I, P, E, R). Затем следуют три параметра. Если они не существенны, стоит 0. При переходе (команда E) метки не нужны, так как длина команд фиксирована, и достаточен только номер команды. Например программа:

$$m_2 \leftarrow 0; \quad \#1; \quad \textbf{if}(m_1 = m_2)\ \{\, \textbf{return}\ m_2; \,\}\ \ m_2 \leftarrow m_2 + 1; \quad \textbf{goto}(1);$$

выглядит так (команды нумеруются с единицы):

$$Z, 2, 0, 0,\ E, 1, 2, 7,\ I, 2, 0, 0,\ Z, 3, 0, 0,\ Z, 4, 0, 0,\ E, 3, 4, 2,\ R, 2, 0, 0$$

($\lessdot$ H$_4$) Записать программу для сложения целых чисел от 1 до числа в ячейке m_0 при помощи языка ZIPPER.

($\lessdot$ H$_5$) Придумать язык команд только с двумя параметрами, а не тремя.

($\lessdot$ H$_6$) На базовом языке BASE написать алгоритм выполнения программы на языке ZIPPER. На входе функции, в памяти содержится набор чисел – инструкций.

($\lessdot$ H$_7$) Какую возможность языка BASE, язык ZIPPER не выражает, и что надо в него добавить?

▷ Первоначально, идеальный компьютер, который придумал Алан Тьюринг имел вид очень низкоуровневого устройства не умеющего даже складывать. Машина Тьюринга является *автоматом*, который может находится в одном из m состояний $\mathcal{S}$. Например, $\mathcal{S} : \{s_1, s_2\}$ $(m = 2)$. Существует строка с входными данными, состоящая из символов некоторого алфавита с n буквами, например, $\mathcal{A} : \{a, b, \_\}$ $(n = 3)$, где "_" – это пустой символ, который окружает исходную строку слева и справа сколь угодно много раз. Задача машины – переработать данную входную строку в другую (выходную) и остановиться. Для этого она устанавливает указатель (головку машины) на фиксированный символ строки (например, первый), а своё состояние в начальное (например s_1) и затем исполняет программу, заданную в виде таблицы:

_	b	a	b	a	b	_	_
	△						

	a	b	_
s_1	bs_1R	as_1R	$\_s_2L$
s_2	bs_2L	as_2L	$\_ \bullet R$

В таблице по горизонтали перечисленны все буквы алфавита, а по вертикали – все состояния машины. Каждая ячейка $\langle s_i, a_j \rangle$ содержит инструкцию, что делать, если текущее состояние машины s_i, а её указатель стоит в строке на символе a_j. Каждая инструкция типа $a_k s_p L$ состоит из трёх действий: 1) текущий символ в строке заменить на a_k; 2) сдвинуть указатель по строке влево, если стоит буква L, вправо – если R; 3) изменить состояние на s_p, или если стоит точка остановиться. Если буквы нет остаться на месте.

При составлении программы для машины Тьюринга (т.е. таблицы переходов) необходимо выбирать свойства каждого состояния, так, чтобы оно решало некоторую элементарную задачу ("перемещаться до пробела" и т.п.). Комбинация таких подзадач-состояний и составляет алгоритм.

($\ll$ H$_8$) Пользуясь инструкциями в таблице выше, описать последовательные преобразования входной строки "*babab*".

($\ll$ H$_9$) Рассмотреть алфавит $\mathcal{A} : \{0, 1, \_\}$ и написать программу для задачи превращения нулей в единицы и наоборот.

($\ll$ H$_{10}$) Рассмотреть алфавит $\mathcal{A} : \{1, \_\}$ и написать программу сложения двух натуральных чисел, записанных в виде последовательностей единиц с пробелом между ними (т.е. пробел убрать, сдвинув второе число влево, объединив единицы чисел.)

($\ll$ H$_{11}$) Рассмотреть алфавит $\mathcal{A} : \{0, 1, \_\}$ и написать программу сортировки, переставляющей все нули в начало слова (до пробела), а единицы в конец.

($\ll$ H$_{12}$) Написать на языке BASE программу, исполняющую инструкции машины Тьюринга. Размер алфавита и число состояний заданы в ячейках m_0, m_1. Далее идут записанные в первых $3 \cdot m_0 \cdot m_1$ ячейках таблица инструкций. Затем входная строка. При попытке сдвинуться влево на первом символе строки, она должна быть сдвинута вправо. Можно использовать любые разумные расширения синтаксиса BASE.

VI Проблемы остановки и тождественности

Мы видели, что программы бывают "хорошие" и "плохие". Плохие приводят к неопределённым или т.н. *частичным функциям*. Их вызов, при некоторых входных значениях, не оканчивается получением результата. Поэтому Тьюринг поставил вопрос, – существует ли универсальный алгоритм, проверяющий достигнет ли данная функция инструкции **return** при конкретном значении её входа. Конечно, проще всего запустить программу (протестировать ее). Если она остановилась, то все нормально. Однако, если она не останавливается, то не известно – остановится ли она когда-нибудь или будет работать бесконечно долго. Для некоторых программ (например, получения предыдущего числа $\mathbf{p}(x)$ при $x = 0$) легко понять, что они никогда не остановятся, даже не запуская их. Однако мы пытаемся выяснить существует ли *универсальный* алгоритм, который способен решить проблему остановки для любой программы. Тьюринг показал, что такой алгоритм *невозможен*. Это утверждение носит название "*Теорема об остановке*".

Так как алгоритмы (функции) оперируют только с целыми числами, необходимо разобраться как они будут рассматривать тексты программ. Ранее мы договорились, что программа – это некоторая конечная строка символов из конечного алфавита. Поэтому, все возможные программы можно отсортировать сначала по длине, а при одинаковой длине – по алфавиту (лексографический порядок). Каждая получит порядковый номер. Выше это и предполагалось, так как имя каждой функции имело вид f_i, где i – некоторое натуральное число.

Соответствие "текст функции" $\rightarrow$ "целое число" можно проделать и более удобным образом. Если все символы (буквы) алфавита пронумеровать, то каждый символ текста программы можно считать некоторым целым положительным числом. Поэтому программа может быть описана конечной последовательностью чисел: $x_1,\ x_2, ..., x_n$, являющимися номерами каждого символа. При желании, эти n чисел можно свернуть в одно при помощи последовательности простых чисел: $x = 2^{x_1} \cdot 3^{x_2} \cdot ... \cdot p_n^{x_n}$. Это и будет уникальный номер нашей программы.

Такое представление удобно тем, что легко провести обратную операцию и, разложив номер программы на простые множители, получить её текст в виде последовательности $x_1,\ x_2, ..., x_n$. Естественно, подобные действия являются алгоритмическими, и существует функция $\mathbf{g}(x, i)$, дающая i-й "символ" программы под номером x. Буква $\mathbf{g}$ дана программе в честь Гёделя, который подобным образом предложил нумеровать формулы.

• Вернёмся к Проблеме Остановки и рассмотрим сначала простое рассуждение, отличное от того, которое сделал сам Тьюринг. Будем большими буквами обозначать имена программ (алгоритмов), подразумевая, что за этим именем скрывается строка символов, которая его задаёт. Маленькими буквами обозначим целочисленный номер этого алгоритма. Так, q – это номер алгоритма (строки) Q.

Пусть существует функция $T(p,i)$, которая по номерам p-той программы и её i-тых входных данных возвращает 1, если эта программа завершается, выдав результат, и ноль в противном случае:

$$T(p,i): \begin{cases} 1, & \text{если } P(i) \text{ возвращает значение} \\ 0, & \text{если } P(i) \text{ не возвращает} \end{cases}$$

Докажем теорему:

Функция $T(p,i)$ не существует.

Пусть это не так, и $T(p,i)$ существует. Определим ещё одну функцию:

$$Q(x): \begin{cases} 1, & \text{если } T(x,x)=0 \\ \infty, & \text{если } T(x,x)=1. \end{cases}$$

Символ ∞ означает, что если $T(x,x)=1$, то мы зациклим программу Q, не дав ей остановиться. Например, в этом случае будет запускаться строка "**while**(1=1){}". Вычислим теперь значение $Q(q)$, передав ей её *собственный номер*:

$$\begin{cases} 1, & \text{если } T(q,q)=0 \quad : \text{если } Q \text{ не завершается, то } Q \text{ завершается} = 1 \\ \infty, & \text{если } T(q,q)=1 \quad : \text{если } Q \text{ завершается, то } Q \text{ не завершается } (\infty) \end{cases}$$

Мы пришли к противоречию, значит теорема верна $\square$.

В доказательстве обсуждается два объекта: "неконструктивная" функция $T(p,i)$ (текста которой мы не предъявляем) и "конструктивная" $Q(x)$. Поэтому противоречие должно свидетельствовать в пользу теоремы.

Однако, на самом деле мы доказали несколько более слабое утверждение:

Не существует функции способной проанализировать свой текст и выяснить, остановится она или нет.

При этом не существенно, анализируется ли собственный текст на прямую или через несколько вызовов других функций. Действительно, противоречие получено при применении алгоритма к функции Q, которая его же и использует. Поэтому для *таких* случаев алгоритма решающего проблему остановки нет!

• Исходные рассуждения Тьюринга выглядели несколько по-другому. Рассмотрим все программы, вычисляющие функции от одного аргумента. Их бесконечно много, но они *счётны*: $F_k(x) : \{F_0(x),\ F_1(x),\ ....\}$, т.е. пронумерованы (нумерацию ведём с нуля). Пусть существует функция $T(p,i)$ определённая выше. Тогда рассмотрим функцию:

$$F(x) : \begin{cases} F_x(x)+1, & \text{если } T(x,x)=1, \text{ т.е. } F_x \text{ при } x \text{ определена} \\ 0, & \text{если } T(x,x)=0, \text{ т.е. } F_x \text{ при } x \text{ не определена.} \end{cases}$$

Она отлична от всех функций $F_k(x)$. Действительно, если значение функции $F_0(0)$ определено, то $F(0) = F_0(0)+1$ и не равно $F_0(0)$. Если же значение $F_0(0)$ не определено (она не останавливается), то $F(0)$ определено и равно 0. Поэтому $F(x)$ отлична от $F_0(x)$ при $x=0$. Аналогично $F(x)$ отлична от $F_1(x)$ при $x=1$, и т.д. Но это противоречит тому, что *все* функции были пронумерованы. Поэтому объект $T(p,i)$ использовавшийся при построении $F(x)$ невозможен, и теорема доказана $\square$.

Мы уже видели, что заканчивать доказательство от противного, такой жизнеутверждающей фразой, необходимо предельно осторожно. В сделанных рассуждениях предполагается, что алгоритм $T(p,i)$ *применим* ко *всем* функциям, включая такие, которые вызывают и его (например к $Q(x)$ описанной на предыдущей странице). Однако, это не так, и к таким функциям $T(p,i)$ не применим. Поэтому, фактически противоречия не возникает, так как отличить $F(x)$ от некоторых функций нельзя.

• Так как алгоритма решающего проблему остановки и применимого к самому себе не существует, то естественно не существует и *универсальных* алгоритмов, применимых ко *всем* программам. А это, собственно, и составляло теорему Тьюринга.

Однако, проведенное ранее доказательство не запрещает существование "*ограниченного универсального алгоритма*", решающего задачу остановки применительно к любым функциям, кроме себя самой и тем функциям, которые этот алгоритм используют (на прямую или через несколько вызовов других функций).

Таким образом, мы получили чёткий критерий области неприменимости алгоритма $T(p,i)$. Все программы которые в него *не* попадают вообще говоря *могут* быть проанализированы $T(p,i)$, и в этом случае он *может* и решать проблему остановки. Мы видим, что в таких психологических доказательствах от противного, необходимо очень осторожно выявлять объекты и утверждения приведшие к противоречию.

Рассмотрим теперь как можно выяснить используется данный алгоритм некоторой функцией или нет.

• Каждой функции можно поставить в соответствие множество имён (или номеров) других функций, которые она использует при своей работе ("*множество зависимостей*"). Для этого необходимо использовать алгоритм "поиска в ширину":

> Возьмём пустой список целых чисел. Поместим в него номера всех функций которые явным образом присутствуют в тексте программы. Затем просмотрим текст каждой из них и добавим в список *только* те функции, которых там ещё нет. Для всех *добавленных* операцию повторяем, пока добавления не прекратятся.

Понятно, что для многих функций множества зависимостей будут или пустые или конечные. Например, если:

$$f_1()\{\ \textbf{return}\ f_2();\ \}\quad f_2()\{\ \textbf{return}\ f_3();\ \}\quad f_3()\{\ \textbf{return}\ f_1();\ \},$$

то функции f_1 соответствует множество зависимостей $\{2, 3, 1\}$, т.е. вызовы второй, третьей и исходной функций. Однако, не сложно представить и бесконечную цепочку таких зависимостей. В этом случае алгоритм не остановится, добавляя в множество зависимостей всё новые номера. К слову, понятно, что не может быть и универсального алгоритма, за конечное число шагов выясняющего вычислимость подобной функции.

Тем не менее, алгоритм поиска в ширину может быть остановлен на любой заданной глубине вложенности вызовов функций. В этом случае он позволяет для данной функции всегда установить, имеет ли она конечное множество зависимостей размера не более некоторого n. Рассматривая только такие функции, мы сильно сузим класс возможных алгоритмов. Однако, для деятельности человеческих существ с ограниченной памятью этот класс важен, так как именно с такими алгоритмами мы обычно и "имеем дело". В конечном счёте, ведь мы определили, что алгоритм – это *конечная* последовательность инструкций. Поэтому можно сформулировать следующую задачу:

> *Существует ли алгоритм решающий, проблему остановки для функций имеющих множество зависимостей не длиннее некоторого n, и не использующих этого алгоритма?*

Решение даже такой ограниченной задачи сталкивается с другой неприятностью. Любую функцию можно вызвать не напрямую, а просто повторить её код в текущей функции. В этом случае алгоритм построения множества связей сможет работать только если есть решение другой проблемы, которую мы сформулируем в таком виде:

> *Существует ли универсальный алгоритм, который выясняет тождественность двух функций?*

• Понятно, что одна и та же функция (т.е. отображение $x \mapsto y$) может быть задана при помощи различных алгоритмов. Например, существующий алгоритм можно изменить, добавив к результату единицу, а после этого вычитая её. Это будет другой алгоритм этой же функции.

Алгоритм, проверяющий равенство двух функций на данном входе, *эквивалентен* алгоритму решающему проблему остановки. Если мы знаем при помощи $T(p,i)$, что две программы p_1 и p_2 останавливаются, мы просто их запускаем на входе i и сравниваем результаты. Если обе не останавливаются, то они на i тождественны. Возможно и обратное использование. Если есть алгоритм тождественности $E(p_1,p_2,i)$, то программу p_1 можно сравнить, например с функцией $\mathbf{f}(i)\{$ **while**$(1=1)\{\}$ $\}$, и если они тождественны, то p_1 не останавливается. Таким образом, мы должны признать, что существование алгоритма решающего как проблему остановки, так и проблему тождественности, даже для ограниченного класса функций, может оказаться под большим вопросом.

Тем не менее, ситуация не полностью безнадёжна. Легко указать, как минимум, один бесконечный класс достаточно нетривиальных функций, для которых алгоритм $T(p,i)$ существует:

> Функции, не использующие операции **goto** и вызывающие только *примитивно рекурсивные* функции, всегда останавливаются.

Примером примитивной рекурсии является реализация функции **mult** на стр. 20. В общем случае она определяется так:

$$
\begin{array}{l}
\mathbf{f}(x,y)\{ \\
\quad \textbf{if}(y=0)\{\ \textbf{return}\ \mathbf{g}(x);\ \} \\
\quad \mathcal{P} \\
\quad \textbf{return}\ \mathbf{h}(x,y,\mathbf{f}(x,\mathbf{p}(y)))\,; \\
\}
\end{array}
$$

где $\mathcal{P}$ – любой код, который не использует оператор **goto**, вызова функции $\mathbf{f}(x,y)$, и каких-либо функций являющихся не всюду определёнными. Кроме этого, он не должен изменять значение y. Функция $\mathbf{p}(y)$ возвращает предыдущее значение y, т.е. $y \leftarrow y-1$. Она не определена при $y=0$, но это контролируется первой строкой функции $\mathbf{f}(x,y)$.

В определении $\mathbf{f}(x,y)$ используются две *всюду определённые* вычислимые функции $\mathbf{g}(x)$ и $\mathbf{h}(x,y,z)$. То, что они всюду определены, должно устанавливаться на предыдущих итерациях алгоритма $T(p,i)$, поддерживающего список уже проанализированных функций, и тех из них, которые являются примитивно рекурсивны. При помощи примитивной рекурсии можно организовать циклические вычисления, не используя **goto**. Поэтому они позволяют задать очень широкий класс функций.

Возможны различные расширения класса всюду определённых функций. В частности, вместо аргумента x везде может стоять несколько аргументов $x_1, ..., x_n$, и тем самым будут определяться всюду вычислимые функции от $n+1$ аргументов. Можно разрешить в $\mathcal{P}$ изменения y, которые его не увеличивают и не приводят к значению $y = 0$, и т.д.

Для подобных всюду определённых функций можно написать алгоритм их *разрешения*, т.е. выяснения принадлежат ли они к данному классу или нет. Если да, то алгоритм решающий задачу остановки $T(p, i)$ вернёт 1. Если же функция не принадлежит к этому классу, то алгоритм $T(p, i)$ должен вообще говоря вернуть 2 (="не знаю" $\ddot\smile$).

Можно также определить достаточно широкий класс алгоритмов частичных функций, которые определены не всюду, но при этом проблема остановки для них также решается. Примером таких функций была $\mathbf{p}(x)$ или вычислимая, но всюду не определённая функция $\mathbf{Inf}(x)$ { **while**$(1 = 1)$ { }}. Для этого класса существует разрешающий алгоритм выяснения принадлежности. Если функция под номером p принадлежит этому классу, то $T(p, i)$ вернёт 0, в противном случае снова 2.

С практической точки зрения компьютерных наук максимальное расширение обоих классов играет исключительно важное значение. Однако, общая проблема остановки остаётся, и всегда будут алгоритмы, которые не попадают ни в один из этих классов. Мы вынуждены признать, что существует бесконечное количество алгоритмов для которых нельзя выяснить останавливаются они когда-либо или нет. Как минимум, это алгоритмы явно или не явно использующих саму функцию, решающую задачу остановки $T(p, i)$. При этом выяснить используют они её или нет, вообще говоря, также алгоритмически нельзя.

• Прежде чем идти дальше, полезно поразмышлять над следующей проблемой. В математике нет ограничений на быстродействие компьютера. Т.е. каждая элементарная операция алгоритма может быть выполнена за сколь угодно короткое время Δt. Если функция вычислима, то её алгоритм, совершив *конечное* число действий n, останавливается и выдаёт результат. А это значит, что каким бы большим не было n, время вычисления функции $n\Delta t$ можно сделать сколь угодно малым, в пределе $\Delta t \to 0$ (ограничений нет!), в том числе и нулевым. Но тогда не должно быть и Проблемы Остановки. Любая определённая при данном x функция сразу выдаёт результат, а неопределённая не выдаёт. Между алгоритмом с большим, но *конечным* числом шагов и алгоритмом с бесконечным числом шагов существует принципиальная разница. Это два разных класса алгоритмов. Но различить их надо без компьютеров...

VII Перечислимые и разрешимые множества

• Размышления о совокупностях объектов создало математику. Множество, как и любое базовое понятие, без порочного круга синонимов (собрание объектов, совокупность, класс) определить непросто. В любом случае, мы считаем, что существуют две сущности – множества и их элементы. Элементы мы обозначаем маленькими буквами a, b, а множества – большими $\mathcal{A}$, $\mathcal{B}$.

По Кантору, элементы множества являются объектами "хорошо различимыми нашей интуицией или нашей мыслью". Пока речь идёт о конечных множествах, можно предъявить кучу камней или саблезубых тигров. В этом случае множество задаётся перечислением всех своих элементов: $\mathcal{A} = \{a, b, c\}$. При помощи *предиката*, т.е. логической функции от двух аргументов $\in (a, \mathcal{A})$, мы выражаем факт *принадлежности* элемента a множеству $\mathcal{A}$ (принадлежит, не принадлежит, и третьего не дано!). Принадлежность удобно записывать в операторном виде, ставя символ $\in$ между аргументами предиката: $a \in \mathcal{A}$.

Помимо принадлежности для множеств состоящих их одних и тех же элементов вводится предикат равенства $\mathcal{A} = \mathcal{B}$. Если же множество $\mathcal{A}$ состоит из тех же элементов, что и $\mathcal{B}$, но они не равны, говорят, что $\mathcal{A}$ является *подмножеством* $\mathcal{B}$. Для выражения этого утверждения вводится третий предикат: $\mathcal{A} \subset \mathcal{B}$.

Кроме предикатов, для образования новых множеств вводится три предметные функции:

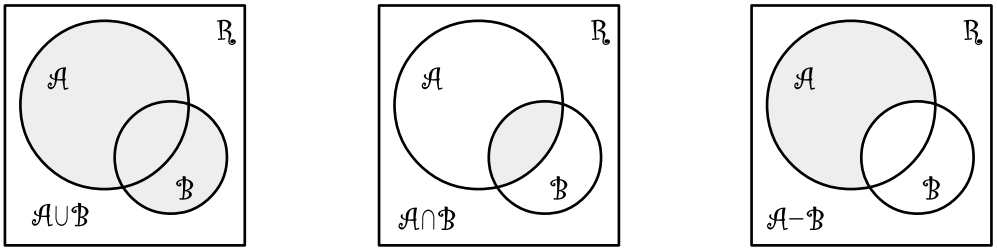
$$\begin{array}{lrcl} \text{объединение}: & \mathcal{A} \cup \mathcal{B} & = & \{x|\ \ (x \in \mathcal{A}) \vee (x \in \mathcal{B})\} \\ \text{пересечение}: & \mathcal{A} \cap \mathcal{B} & = & \{x|\ \ (x \in \mathcal{A}) \,\&\, (x \in \mathcal{B})\} \\ \text{дополнение}: & \mathcal{A} - \mathcal{B} & = & \{x|\ \ (x \in \mathcal{A}) \,\&\, (x \notin \mathcal{B})\} \end{array}$$

Задание множества при помощи логического условия: $\mathcal{A} = \{x|$ условие$\}$ иногда очень удобно. Например, натуральные числа от 1 до 5 можно задать так: $\mathcal{A} = \{1, 2, 3, 4, 5\}$, а можно и так $\mathcal{A} = \{x|\ (x > 0) \,\&\, (x < 6)\}$. То, что элемент не принадлежит множеству обозначается как $x \notin \mathcal{B}$. Это сокращение записи отрицания при помощи логической связки: $\neg(x \in \mathcal{B})$.

Таким образом, в объединённое множество попадают все элементы из обоих множеств. В пересечение только общие для множеств элементы, а в дополнение $\mathcal{A} - \mathcal{B}$ только те элементы $\mathcal{A}$, которые не принадлежат $\mathcal{B}$. Предполагается, что в множестве одинаковые элементы всегда встречаются только один раз. Для неупорядоченных множеств порядок расположения элементов не играет роли.

Удобно рассматривать понятие основного или фундаментального множества $\mathcal{R}$, содержащего *все* элементы какого-нибудь заданного типа. Например, фундаментальным будет множество всех натуральных чисел. Задав его, можно говорить о различных подмножествах натуральных чисел (чётные, простые, и т.п.). При помощи функций $\mathcal{A} \cup \mathcal{B}$, $\mathcal{A} \cap \mathcal{B}$ и $\mathcal{A} - \mathcal{B}$ можно строить новые множества.

Эти базовые функции можно визуализовать при помощи диаграмм Эйлера-Венна, в которых замкнутая область обозначает некоторое множество. Область геометрического пересечения двух множеств будет множественным пересечением $\mathcal{A} \cap \mathcal{B}$, и т.д.:



В такой форме теория множеств возвращается к своим наглядным истокам – куче камней на полу в пещере. Для того чтобы множественные функции были замкнутыми необходимо ввести объект – пустое множество $\varnothing$. Например, $(\mathcal{A} - \mathcal{B}) \cap \mathcal{B} = \varnothing$.

Чтобы определить *конкретное* множество необходимо каким-то образом выразить факт принадлежности ему некоторого элемента. Другими словами, для всех a задать логические значения функции $a \in A$. Множество может быть *бесконечным*, и нам необходим *конечный* способ его задания. Поэтому мы должны предложить *алгоритм*, который "*разрешает*" помещать число a в множество A. Например, если число p не имеет делителей кроме 1 и себя, то оно является элементом множества простых чисел $\mathcal{P}$, т.е. $p \in \mathcal{P}$. В данном случае существует функция которая на вход получает число p, раскладывает его на множители и возвращает 1 для простых чисел или 0 для составных.

Другой способ состоит в том, чтобы при помощи некоторого алгоритма "*перечислять*" элементы множества. Так множество чётных чисел можно задать в перечислительном варианте как $e_k = 2 * k$, где k все натуральные числа (т.е. выразить одно множество $\mathcal{X}$ через элементы другого $\mathcal{Y}$ в виде функции $y = f(x)$). Ещё более "перечислительно" будет выглядеть определение при помощи рекурсии: $e_1 = 0$, $e_k = e_{k-1} + 2$.

В результате мы приходим к некоторой классификации *конструктивных* множеств по способу их построения. В частности говорим, что могут быть разрешимые и перечислимые множества.

• *Разрешимым* называется множество целых чисел для которого существует алгоритм определения принадлежности ему некоторого элемента. Для разрешимого множества $\mathcal{A}$ задаётся *вычислимая* и всюду *определённая* функция $\mathbf{is}(a)$:

$$\mathbf{is}(a) = \begin{cases} 1, & \text{если } a \in \mathcal{A} \\ 0, & \text{если } a \notin \mathcal{A}. \end{cases}$$

Естественно, наличие внутри $\mathbf{is}(a)$ символа принадлежности не означает, что мы пользуемся кем то заданным предикатом $a \in A$ (получился бы порочный круг). Вместо "$a \in A$" стоит конечная последовательность символов некоторой программы, которая должна выдать не нулевое значение, тогда $\mathbf{is}(a) = 1$. Если же эта программа, или другая "$a \notin \mathcal{A}$" выдаёт ноль, то $\mathbf{is}(a) = 0$. Функция $\mathbf{is}(a)$ называется *характеристической*.

Так как существует "Проблема Остановки", будем считать, что разрешимым будет только множество, для алгоритма которого *доказано* (!), что он останавливается при любых входах.

Можно ввести также понятие "*частично разрешимое множество*" для которого функция $\mathbf{is}(a)$ является частичной, т.е. определённой не для всех a. Тогда для некоторых a будет неизвестно принадлежат они множеству или нет. Это уже не очень "хорошее" множество.

• *Перечислимым* называется подмножество натуральных чисел, *все* элементы которого можно получать (перечислять) при помощи алгоритма. Это означает что существует некоторая функция, которая в своей памяти последовательно размещает числа. Например, $\mathcal{M} : \{3, 14, 15, 92, ...\}$, т.е. после нескольких шагов алгоритма добавляется новое число e_1, затем после, вообще говоря, другого числа шагов добавляется e_2, и т.д. Приведенный в предыдущем разделе алгоритм поиска в ширину создавал подобные перечислимые множества зависимостей.

Если алгоритм перечисления существует, то его можно переписать в виде выдачи i-того элемента последовательности. Таким образом, для перечислимого множества существует *вычислимая* функция $\mathbf{get}(i)$:

$$\mathcal{E} : \{e_1, e_2, e_3, ...\} \qquad => \qquad \mathbf{get}(i)\{ \ ... \ \mathbf{return} \ e_i \ \},$$

т.е. её значение равно i-тому элементу множества.

Однако, в отличии от $\mathbf{is}(x)$, Функция $\mathbf{get}(i)$ может быть *частичной функцией*. Точнее, начиная с некоторого номера i она не определена. Это означает, что породив несколько элементов множества, алгоритм перечисления может зациклиться и перестать выдавать элементы. Неприятность состоит в том, что мы не всегда можем распознать попадание алгоритма в подобный режим.

• Важно понимать, что хотя алгоритм перечисления последовательно даёт элементы множества, это не означает, что мы способны узнать принадлежность некого, *ещё не полученного* числа к множеству. Т.е.

перечислимое множество не обязательно разрешимо

Если бы можно было доказать эквивалентность разрешимых и перечислимых множеств, смысл бы в подобном делении отпал.

Понятно, что любое разрешимое множество перечислимо. Для построения алгоритма его перечисления можно в цикле вызывать функцию $\mathbf{is}(x)$ для $x = 0, 1, 2, ...$, выдавая в память (перечисляя) только числа для которых $\mathbf{is}(x) = 1$. Этот пример ещё раз показывает, что перечислимое множество определяется, вообще говоря, частичным алгоритмом. Например, пусть разрешимое множество задано алгоритмом: "$x \in \mathcal{A}$ если $x < 4$. Тогда понятно, что после $x = 3$ алгоритм перечисления начинает крутится неограничено долго.

Область определения любой вычислимой функции $f(x)$ является перечислимым. Действительно, всегда можно реализовать алгоритм пошагового выполнения функции. Сделаем один шаг алгоритма для $f(0)$, затем по два шага алгоритма для $f(0)$ и $f(1)$, затем по три шага для $f(0)$, $f(1)$, $f(2)$, и т.д. Если на некоторой итерации этого процесса одно из вычислений $f(x)$ останавливается по **return**, то алгоритм перечисления выдаёт значение x в множество определения функции. Таким образом, если при некоторых x функция $f(x)$ не определена, то они будут в списке пошагово выполняющихся функций "висеть" бесконечно долго, но процесс перечисления не заблокируют.

Далее. Пусть у нас есть подмножество натуральных чисел $\mathcal{A}$ и его дополнение $\bar{\mathcal{A}}$ (т.е. все остальные натуральные числа не попавшие в $\mathcal{A}$). Если и $\mathcal{A}$, и $\bar{\mathcal{A}}$ перечислимы, то множество $\mathcal{A}$ разрешимо. Это утверждение носит название *теорема Поста*:

$$\begin{array}{rlcrl} \mathcal{A} & \text{разрешимо} & \Rightarrow & \mathcal{A}, \bar{\mathcal{A}} & \text{перечислимы} \\ \mathcal{A}, \bar{\mathcal{A}} & \text{перечислимы} & \Rightarrow & \mathcal{A} & \text{разрешимо} \end{array}$$

Алгоритм разрешения $\mathbf{is}(x)$ следующий. Для данного x запускаем *одновременно* пошагово выполняться алгоритмы порождения множеств $\mathcal{A}$, и $\bar{\mathcal{A}}$. *По определению* x находится в одном из этих множеств, поэтому рано или поздно будет порождён.

Ещё раз подчеркнём: 1) алгоритм перечисления может никогда не остановиться, хотя и перестанет выдавать новые числа; 2) порядок перечисления не регламентируется, и числа могут быть не упорядочены, например по возрастанию; 3) алгоритм перечисляет *все* элементы, но зацикливается если множество конечно.

• Таким образом, алгоритмы и вычислимые функции позволяют формализовать идею конструктивного множества, дав ей некоторые любопытные оттенки. В частности, и разрешимое и перечислимое множества задаются алгоритмом, но разрешимые множества более эффективны и являются более "правильными" множествами чем перечислимые.

Естественно, тут же возникает вопрос – "а ограничены ли мы только конструктивными множествами?" Ответ на него очень непрост и существенно зависит от философской позиции математика, точнее тех правил Игры, которые он для себя считает приемлемыми. Неконструктивные множества существуют, если допустимо работать с "неясными" понятиями, по крайней мере до момента возникновения парадоксов.

Рассмотрим пример. Пусть фундаментальным множеством $\mathcal{R}$ является множество всех алгоритмов. Это множество достаточно хорошо определено. Мы можем не только упорядочивать алгоритмы, перечисляя их, но и разрешать. В частности, всегда можно однозначным образом проверить что данная цепочка символов является синтаксически верным алгоритмом.

"Определим" теперь три подмножества $\mathcal{R}$:

$\mathcal{A}$: "множество всех алгоритмов не использующих оператор **goto**"
$\mathcal{B}$: "множество всех определённых при данном x функций $f(x)$"
$\mathcal{C}$: "множество всех алгоритмов решающих проблему остановки"

Множество $\mathcal{A}$, являющееся разрешимым (если функция не порождает бесконечной цепочки вызовов других функций). Существуют конкретные элементы подобного множества, мы можем их разрешать или порождать.

Множество $\mathcal{B}$ является уже неразрешимым, но перечислимым. Точнее для некоторых элементов предикат $b \in B$ определён, тогда как для других не определён, и мы не можем сказать принадлежит ли b к B или нет. Это уже не так хорошо. У нас нет способа применять к такому объекту множественные операции. Например, мы не можем однозначно определить дополнение до всех алгоритмов $\mathcal{R} - \mathcal{B}$.

Третье множество $\mathcal{C}$ выглядит совсем плохим. Более того, в силу теоремы Тьюринга такого множества просто не существует. Но мы же его задали, причём такой однозначной фразой! Очень часто в математике рассматриваются множества определённые не алгоритмом, а некоторым общим описанием, например "возьмём множество всех множеств". Пока противоречия не возникают рассуждать о таких объектах конечно можно. Но где проходит грань неясности между "множеством всех множеств" и "множеством тех множеств которые не являются множествами" ☺?.

• Многие математические задачи имеют алгоритмическое решение. Например, при помощи алгоритма мы всегда можем сложить два числа, или разложить любое число на простые множители. Однако, есть алгоритмически неразрешимые задачи. Примером такой задачи является "Проблема Остановки".

Ещё один хорошо известный пример связан с *диафантовыми уравнениями*. Они являются полиномами с целыми коэффициентами и несколькими неизвестными. Примеры таких уравнений:

$$x^2 + 2x + 1 = 0; \qquad x \cdot y = z; \qquad x^3 + y^3 = z^3; ...$$

В 10-проблеме Гильберта, сформулированной в 1901г. требуется найти алгоритм определяющий для любого диафантового уравнения, имеет ли оно целочисленные решения или нет. Более того, сами решения строить не обязательно, достаточно узнать что они есть в принципе. Таким образом, требуется построить *разрешающий алгоритм* на вход которого подаётся уравнение (или его номер), а на выходе получается 1, если уравнение имеет решения, и 0 – если нет.

Для некоторых классов уравнений подобные алгоритмы существуют. Например, решения уравнения $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 = 0$ находятся среди делителей свободного члена a_0. Всегда имеет решение уравнение $x \cdot y = z$. Можно доказать, что уравнение $x^3 + y^3 = z^3$ не имеет решений, и т.д. Однако, как и в "Проблеме Остановки", стоит задача построения универсального алгоритма для *любого* диафантового уравнения. В 1970г. Юрий Владимирович Матиясевич доказал неразрешимость этой задачи.

Подобная "деструктивная" деятельность очень увлекательна и обычно использует метод доказательств от обратного: "предположим, что такой алгоритм существует ....., мы пришли к противоречию, значит его нет". Понятно, что с такими противоречиями необходимо обходиться придельно осторожно. Особый интерес вызывает также и обратная задача – максимальное расширения класса разрешимых уравнений для которых можно выяснить наличие или отсутствие у них решений. Аналогично можно искать класс неразрешимых уравнений, и т.д.

Если считать проблему Гильберта решённую в таком отрицательном варианте, то понятно, что говорить о множестве $\mathcal{S}$: "всех диафантовых уравнений имеющих решения" можно, но "хорошим" его назвать трудно, хотя оно и является перечислимым. Заметим, что до 1970 г. подобные категории уточнения определения $\mathcal{S}$ были просто не доступны. Еще хуже, когда не известно, является ли множество хотя бы перечислимым. Работа с такими объектами требует незаурядного мужества.

VIII Существуют ли несчетные множества?

• Исходным методом Кантора сравнения бесконечных множеств, было установление их равномощности. В основе идеи равномощности лежит понятие *соответствия*. Если каждый элемент одного множества можно поставить во взаимооднозначное соответствие элементу другого множества, то эти множества равномощны. Соответствие это, на самом деле – некоторый *алгоритм*, который и производит такую процедуру. Если бесконечное множество равномощно множеству натуральных чисел, то говорят, что оно *счетно*. На самом деле это означает, что известен алгоритм, который позволяет в некотором порядке перебирать все элементы этого множества и тем самым их нумеровать (ставить в соответствие натуральным числам). Другими словами, это должно быть *перечислимое множество*.

То, что множества являющиеся подмножествами натурального ряда (например, чётные числа) являются счётными особенного удивления не вызывает. Неожиданнее было то, что можно пронумеровать рациональные числа m/n, которых явно "больше" чем натуральных. И алгоритм тривиален – после сокращения общих множителей необходимо перейти гёделевскому числу $g = 2^m \cdot 3^n$, поэтому множество пар (m, n) перечислимо. Это обратимый алгоритм и по g можно восстановить m и n. Аналогично, используя более длинный ряд простых чисел, можно пронумеровать и любые фиксированные наборы целых чисел.

Оригинальная нумерация Кантора состояла в расположении рациональных чисел в виде таблицы (m, n) и перечисление их по диагоналям, начиная с левого верхнего угла: $(1,1)$; $(1,2)$ $(2,1)$; $(1,3)$ $(2,2)$ $(3,1)$;... так, что $m+n$ на каждой диагонали равно 2, 3, 4,...

Другой универсальный приём нумерации – это лексографическое упорядочивание (по длине, а при одинаковой длине по алфавиту). Записав рациональное число в виде строки m/n, а целые m и n в 10-й системе счисления эти строки, очевидно, можно упорядочить, т.е. пересчитать. По этой же причине счётными оказываются *алгебраические* числа, т.е. корни уравнений $a_1\, x^n + a_2\, x^{n-1} + ... + a_n = 0$ с целочисленными коэффициентами. Часть из них (например, $\sqrt{2}$) рациональными не является. Все нерациональные числа мы называем *иррациональными*.

После установления счетности множества рациональных и алгебраических чисел, Кантор, при помощи знаменитого диагонального рассуждения показал, что множество действительных чисел несчетно. Однако, что такое действительные числа? Один из подходов определения их, это обратиться к геометрии.

• Рассмотрим отрезок прямой, длина которого принята за единицу. Действительное число – это расстояние некоторой точки внутри этого отрезка от левого края (нуля). Как мы можем измерять длину отрезков меньших чем базовый? Можно при помощи геометрических построений (требующих своей аксиоматики) разбить отрезок на n равных частей (или соединить n одинаковых отрезков, объявив результат единичным).

Если можно выяснять какая точка лежит ближе к нулю, то расстояние x выражается при помощи неравенства:

$$\frac{1}{3} < x < \frac{2}{3}$$

Взяв n очень большим, можно сколь угодно сильно зажать диапазон неточности (границы неравенства), т.е. приблизить действительное число рациональными с любой заданной ошибкой. Хотя последние определяются только парой целых m и n, они всюду плотно покрывают отрезок $0 \leqslant x < 1$. Какие бы две точки мы не взяли, *не важно* рациональные или иррациональные, между ними всегда будет находиться сколь угодно много рациональных чисел. И если натуральные числа, отложенные на прямой, очевидно "дырявые", то "между" рациональными "дырок" нет! Кавычки подчёркивают психологичность этого утверждения.

Второй способ измерения расстояния состоит в использовании иерархически уменьшающихся "линеек". Для этого будем использовать следующую кодировку для "изображения" числа. Напишем "$x = 0.$". Затем разделим отрезок на две равные части и выясним в какой из них лежит точка. Если в левой, то дальше поставим 0, а если в правой, то 1:

$$x = 0.1010...$$

В данном случае она справа, поэтому имеем "$x = 0.1$". Правую половину опять делим на две равные части и выясняем в какой из них находится x. Теперь слева, поэтому "$x = 0.10$". Повторяя эту процедуру, получим *двоичное представление* числа эквивалентное ряду:

$$x = 0.1010... = \frac{1}{2} + \frac{0}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + ...$$

Также получается десятичное представление, только делить необходимо каждый раз на 10 равных частей. Иногда x оказывается на границе деления, и процедура останавливается. Получается конечное представление в данной системе исчисления. В двоичной системе такие *конечные числа* имеют вид $m/2^n$, где m-нечётно, а в десятичной $m/10^n$, где m не делится на 10. Поэтому конечное в двоичной будет конечно и в десятичной, но, вообще говоря, не наоборот.

• Вернёмся к Георгу Кантору, и рассмотрим множество всех действительных чисел, определённых при помощи процедуры, описанной выше. Будем использовать двоичную систему счисления. Некоторые (даже рациональные) числа в ней будут бесконечной последовательностью нулей и единиц (например 1/3). Удобно считать все числа бесконечными, дописывая справа к конечным неограниченную последовательность нулей. Докажем, что множество таких чисел не счётно.

Естественно доказательство будет строиться по схеме "от противного". Пусть все действительные числа в интервале $0 \leqslant x < 1$ можно пронумеровать. Как бы мы это не сделали, получится некоторый список:

1^{e} число : 0.010010000100...
2^{e} число : 0.111110001110...
3^{e} число : 0.011101010101...
4^{e} число : 0.010001010101..., ...

Другими словами, у нас есть бинарная функция (равная 0 или 1) вида $R_n(i)$, где n-номер числа, а i-номер двоичной цифры в числе после точки. В примере выше $R_1(2) = 1$. Построим число, первая цифра после точки которого не равна первой цифре первого числа, вторая не равна второй цифре второго числа, и т.д. Другими словами, $R(i) = \neg R_i(i)$, где введена бинарная операция инвертирования: $\neg 0 = 1$ и $\neg 1 = 0$. В примере выше начало этого числа $R = 0.1001...$

Это *новое* число не входит в нашу последовательность, так как оно отличается от первого числа первой цифрой, от второго – второй, и т.д. Мы пришли к противоречию с утверждением, что все вещественные числа были пронумерованы. Следовательно, этого нельзя сделать, и множество вещественных чисел не счётно. □

Часто, на основании этого доказательства, говорят, что иррациональных чисел "больше" чем рациональных, которые счётны. Это неверно.

Во-первых, между любыми иррациональными числами всегда расположено бесконечно много рациональных чисел. У двух близких, но не равных иррациональных чисел $a < b$ первые n цифр двоичного разложения совпадут. Мы всегда можем выбрать рациональное число в виде конечной рациональной дроби у которой $n + 1$ цифр совпадают с первыми $n + 1$ цифрами числа "b" и дальше идут нули. Очевидно, что это рациональное число будет больше "a" и меньше "b". Если бы иррациональных чисел было "больше" чем рациональных, то мы получили бы довольно неприятную ситуацию: между *любыми* двумя иррациональными числами всегда путается бесконечное число рациональных, которых меньше!? Хотя, конечно, это слишком "психологичный" аргумент.

Во-вторых, достаточно трудно ответить на простой вопрос - а какие же действительные числа оказались несчетными? Множество рациональных и алгебраических чисел счетно. Поэтому, традиционный ответ – несчетными являются "все остальные" вещественные числа, а именно *трансцендентные*. К трансцендентным числам, в частности, относятся "e" и "π". В такой формулировке это утверждение тоже не верно.

Как появляются (определяются) в математике те или иные *конкретные* числа? Прежде всего – как решения каких либо уравнений. Например, $\sqrt{2}$ это обозначение (имя!) числа, являющегося решением уравнения $x^2 = 2$. Числа можно определить при помощи бесконечных рядов и пределов. Так "$e : (1+x)^{1/x},\ x \to 0$", или $e :\ 1/1!+1/2!+1/3!+...$. Не смотря на то, что эти формулы выражают некоторые бесконечные действия, само определение числа "e" – конечно. Так, многоточие в определении ряда лишь обозначает, что у нас есть конечный алгоритм, позволяющий получить любой, наперед заданный, его член. Еще одним способом определения числа является предъявление алгоритма вычисления любой цифры в его десятичном представлении. Таким является трансцендентное число Лиувилля: 0.11000100000000000000000010..., где n-я единица стоит в позиции $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$.

Во всех этих случаях существует алгоритм или определение, задающее число. Определение не обязательно должно быть конструктивным (не обязательно есть алгоритм вычисляющий это число). Например, рассмотрим десятичное представление числа $\pi = 3.1415926535897932.....$ Назовём $\bar{\pi}_0$ число, которое равно максимальному количеству идущих подряд нулей в десятичной записи числа π (для миллиона цифр $\bar{\pi}_0 = 5$). Если максимальной последовательности не существует, то $\bar{\pi}_0 = 0$. Такое неконструктивное число определено "не очень хорошо", но оно определено. По крайней мере, если мы *верим*, что максимальная последовательность или существует, или нет, и третьего не дано.

Каким бы ни было построение конкретного числа (конструктивным или неконструктивным),

> *все определения, всех чисел, с которыми мы имеем и когда-либо будем иметь дело в математике, задаются конечной последовательностью слов состоящих из букв конечного алфавита.*

Такие определения и, следовательно, числа счетны по той же причине, по которой счетны формулы или алгоритмы.

Таким образом, все, хоть как-то определяемые вещественные числа, являются счетными. *Что же тогда, мы не смогли посчитать?*

• Математическое рассуждение корректно, когда его логическая структура истинна в не зависимости от содержательного смысла участвующих в нем объектов. Вернёмся к Проблеме Остановки и представим такое "доказательство" несчётности вычислимых функций:

> Рассмотрим множество всех вычислимых функций одной переменной. Пусть это множество счетно, т.е. каждая функция имеет номер n: $F_n(i)$. Построим функцию $F(i) = F_i(i) + 1$. Она тоже определяется алгоритмом. Она не совпадает ни с одной функцией из $F_n(i)$. Мы пришли к противоречию. Поэтому, множество таких функций несчетно.

Мы знаем, что это "доказательство" заведомо неверное. Точнее оно доказывает не несчетность вычислимых функций а невозможность построения универсального алгоритма, выясняющего определённость функции. Его отсутствие не позволяет определить значение $F_i(i)$ при некоторых i. Поэтому и сравнение $F(i)$ и $F_i(i) + 1$ иногда просто некорректно.

Не сложно видеть, что структура "доказательств" несчётности функций и действительных чисел эквивалентны. В случае функций мы твердо знали, что их множество счетно, так как получили этот результат прямым а не косвенным рассуждением. Счетны ли вещественные числа мы не знаем, поэтому, получив противоречие, с радостью делаем вывод что они несчетны. Однако, счётность это не единственное предположение. Мы верим, что для любого числа можно получить i-ю цифру, а благодаря этому *можно построить* число отличное от всех других в уже упорядоченной последовательности.

Понятно, что каждая вычислимая и всюду определённая функция задаёт некоторое вещественное число. Такими функциями в доказательстве Кантора были $R_n(i)$. Можно расширить это определение вещественного числа на все возможные алгоритмы. Но тогда мы получим множество "не совсем" определённых вещественных чисел. Простейший пример такого числа следующий. Пусть i-тая цифра двоичного представления числа равна 1, если i-тый алгоритм всюду определён, и 0 в противном случае: 0.1011?1... Мы *принципиально* не можем предложить алгоритма, всегда определяющего остановку данной программы, поэтому для некоторых чисел, в некоторых позициях будет стоять неопределенная цифра "?". Следовательно, мы не можем сравнить это число с другим действительным числом (что и пытаемся делать в доказательстве Кантора). Вопрос: является ли таким образом заданный объект вообще числом? Например, к нему неприменимо то, что делает число собственно числом: арифметические действия и операции сравнения.

• Конечно, можно рассуждать по другому. Пусть множество конструктивных вещественных чисел задается только при помощи всюду определенных вычислимых функций $R_n(i)$. И пусть еще существуют числа, которые не задаются ни одним алгоритмом, не являются решением ни одного уравнения или предела произвольного ряда. Двоичное представление этих чисел есть "просто" произвольное (случайное?) бесконечное множество цифр 0 и 1. Именно этот довесок и оказывается несчетным.

Выше мы уже договорились, что счетность (соответствие элементов множества) разумно понимать только в алгоритмическом смысле. Множество счетно, если можно построить алгоритм, позволяющий "пересчитать" элементы такого множества (произвести 1-1 соответствие). Однако, было бы странно требовать от алгоритма оперировать с актуально бесконечным объектом, который мы не можем не только определить, но и назвать. Но, в таком случае, несчетность становится фактом тривиальным. Нельзя применять алгоритм к неалгоритмическим объектам, поэтому нет такого алгоритма и, следовательно, такие "вещественные числа" несчетны по определению. Однако существуют ли они? Первый же пример (определение) такого числа сделает его счетным. Ситуация напоминает невидимого суслика, который "на самом деле" есть...

• Двоичное представление вещественного числа тесно связано с другой задачей. Рассмотрим множество $\mathbb{N} : \{0, 1, 2, ...\}$ всех натуральных чисел. "Возьмём" множество всех его подмножеств, обозначив его $2^{\mathbb{N}}$. Каждое из таких подмножеств можно задать в виде двоичной последовательности 010100..., где 1 на i-том месте означает, что i – принадлежит этому множеству (например, если после "..." идут нули, то $010100... = \{1, 3\}$). Происхождение названия $2^{\mathbb{N}}$ связано с тем, что для конечного набора из n элементов количество всех его подмножеств (включая пустое $\varnothing = 00...0$) равно 2^n. Используя диагональное доказательство Кантора, не сложно показать, что множество $2^{\mathbb{N}}$ всех подмножеств $\mathbb{N}$ несчётно.

Однако, что такое $2^{\mathbb{N}}$? Говоря о любом множестве, мы считаем, что оно состоит из объектов "хорошо различимых нашей мыслью". Понятно, что в $2^{\mathbb{N}}$ есть конечные множества: $\{3\}$, $\{1, 8\}$, ... Их множество счётно. Оно также состоит их бесконечных множеств ($\mathcal{P}$: "все простые числа", $\mathcal{E}$: "чётные числа", и т.п.). *Бесконечное* множество мы можем задать только при помощи *конечного* алгоритма, и все такие множества счётны. Никак не определив множество, мы не можем его *сравнивать* т.е. *различать* с другими множествами. А ведь различимость элементов – это основа построения теории множеств! Содержит ли $2^{\mathbb{N}}$ в качестве своих элементов такие "неясно определённые объекты"?

IX Формулы арифметики их номера

• Прежде чем обсуждать теорему Гёделя и связанные с ней проблемы неполноты математики, необходимо несколько подробнее остановиться на построении формальных теорий.

Формальная теория – это множество утверждений относительно некоторых предметных сущностей, которые записаны на специальном, формальном языке. В арифметике этими сущностями являются натуральные числа $0, 1, 2, ...$ Через x, y, ... обозначаются некоторые не определённые числа (предметные *переменные*), а при помощи цифр их конкретные реализации (предметные *константы*). Мы считаем, что есть предметные *функции* от чисел, которые снова дают числа. Например $a(x, y) = x + y$ или $n(x) = x + 1$. Такие предметные функции мы обозначаем маленькими буквами со скобками. Кроме этого, вводятся логические утверждения относительно сущностей – *предикаты*. Это тоже функции, аргументами которых являются числа, но их значения равны "истина" или "ложь". Предикаты обозначаются большими буквами. Например, $E(x, y)$ может значить утверждение "x равно y", и чаще кратко обозначается, как $x = y$, а $L(x, y)$ – "x меньше y". Если предикат зависит от переменной, то говорят, что она *свободна*. При одних её значениях предикат истинен, а при других – ложен. Например, $E(0, 0)$ это истина, а $E(0, 1)$ – ложь.

Так как предметная функция даёт число, то она может находиться в аргументе предиката: $E(x+y, y+x)$ (т.е. $x+y = y+x$). В отличие от этого, в рамках арифметики, предикаты не могут быть аргументами других предикатов, так как чисел не возвращают. Так, формула $E(L(x, y), z)$, т.е. $(x < y) = z$ считается бессмысленной. Поэтому говорят, что арифметика – это *теория предикатов первого порядка* (т.е. предикатов не зависящих от других предикатов).

Кроме предикатов вводятся логические связки, заменители слов "не" ($\neg$), "или" ($\vee$), "и" ($\&$), "если $\mathcal{A}$, то $\mathcal{B}$" ($\rightarrow$), эквивалентно ($\leftrightarrow$):

$$\neg\mathcal{A}, \qquad \mathcal{A} \vee \mathcal{B}, \qquad \mathcal{A} \,\&\, \mathcal{B}, \qquad \mathcal{A} \rightarrow \mathcal{B}, \qquad \mathcal{A} \leftrightarrow \mathcal{B}.$$

На месте рукописных заглавных букв могут находится либо предикаты, либо составные формулы, состоящие из предикатов и логических связок. На самом деле связки – это функции, принимающие на вход значение истина "И" или "Л", и их же сообщающие на выходе. Так,

¬И это Л, ¬Л это И, Л ∨ И это И, Л ∨ Л это Л, и т.д.

Такие определения *смысла* логических связок называются *таблицами истинности*.

Логические связки позволяют из элементарных предикатов (базовых высказываний) создавать более сложные. Например, предикат неравенства *определяется* так $x \neq y : \neg(x = y)$. Естественно, если составные высказывания содержат свободную переменную, то при одних её значениях они истинны, а при других ложны:

$$P(x): \quad (x < 2)\,\&\,(x > 3), \qquad\qquad Q(x): \quad x + 2 = 5.$$

Так, $P(x)$ ложно всегда, а $Q(x)$ истинно только если $x = 3$. Если мы смогли приписать элементарным предикатам истинностные значения (например, $E(0,0)$ это "И", а $E(0,1)$ это "Л"), то составные высказывания также приобретают значение "И" или "Л". В результате, мы можем говорить об истинности данной формулы.

Однако, приведенных выше логических связок не достаточно для формулировки некоторых математических утверждений. Иногда требуется формальное выражение слов "для любого" и "существует". Поэтому, вводится понятие кванторов *всеобщности* ($\forall$) и *существования* ($\exists$):

$$\begin{aligned} \forall x A(x): &\quad A(0)\,\&\,A(1)\,\&\,A(2)\,\&\,... \\ \exists x A(x): &\quad A(0) \vee A(1) \vee A(2) \vee ... \end{aligned}$$

Фраза "утверждение $A(x)$ справедливо для любого x" обозначает, что истинно и $A(0)$, и $A(1)$, и $A(2)$, и, так далее, т.е. $A(x)$ выполняется для всех натуральных чисел. Аналогично существующий x делает справедливым (истинным) высказывание $A(x)$ хотя бы для одного (или нескольких x), что выражается бесконечной цепочкой логических "или". Например, Великую Теорему Ферма можно записать следующим образом:

$$\neg\exists n\,\exists x\,\exists y\,\exists z\;\,(n > 2)\,\&\,(x^n + y^n = z^n),$$

где конечно, предполагается, что арифметическая функция возведения в степень определена через другие (например, сложение).

Переменная x привязанная к квантору называется *связанной*, так как она играет роль индекса суммирования, пробегающего по всем значениям предметных сущностей (чисел). Вместо неё можно подставить любую не используемую в выражении переменную, но нельзя ставить константу. Так разрешено записать $\forall y A(y)$ вместо $\forall x A(x)$, а $\forall 0 A(0)$ – нет.

В отличие от обычных логических связок, для кванторов нет таблиц истинности, так как для их "вычисления" потребовалось бы перебрать бесконечное число логических утверждений $A(0), A(1), A(2), ...$ Поэтому, формулам, содержащим кванторы, истинность приписывается априори (тогда эти формулы становятся аксиомами), либо они доказываются при помощи *формального вывода*, и *затем* объявляются истинными.

• *Формальный вывод* – это последовательность формул, в которой каждая следующая получается из предыдущих при помощи фиксированных правил (алгоритмов) преобразования формул. Например, если есть формула с квантором всеобщности, то выводима формула с константой:

$$\forall x\, A(x) \qquad => \qquad A(a),$$

где a – это любая конкретная предметная константа (например, 0). Стрелочка "=>" означает что из одной формулы $\forall x\, A(x)$ получается (выводится) другая, например $A(0)$.

Ещё одно правило вывода:

$$\mathcal{A}, \quad \neg\mathcal{A} \vee \mathcal{B} \qquad => \qquad \mathcal{B}$$

означает, что если есть формула $\mathcal{A}$, которая считается истинной, и формула $\neg\mathcal{A} \vee \mathcal{B}$ в которую входит отрицание $\mathcal{A}$, то можно вывести формулу $\mathcal{B}$ (так как Л $\vee$ И это И, а Л $\vee$ Л это Л, то Л $\vee\, \mathcal{B}$ это просто $\mathcal{B}$).

Третье правило вывода *двойного отрицания* позволяет перед любым предикатом поставить $\neg\neg$ или убрать их. Наконец, в четвёртом разрешено переставлять формулы вокруг логического "или" $\mathcal{A} \vee \mathcal{B} \quad => \mathcal{B} \vee \mathcal{A}$.

Рассмотрим теперь систему аксиом Пеано для арифметики:

$$\mathbf{P_1}: \quad \forall x\,[n(x) \neq 0]; \qquad\qquad \mathbf{P_2}: \quad \forall x\,\forall y\,[\,n(x) \neq n(y) \;\vee\; x = y\,]$$

которые *определяют* свойства предметной функции $n(x)$, имеющей смысл получения следующего числа [в более привычной записи $n(x) = x + 1$]. При этом предполагается, что есть только одна явно представленная предметная константа "0". Все остальные получаются (определяются) при помощи 0 и функции $n(x)$. Так, $1:\ n(0), \quad 2:\ n(n(0))$, и т.д.

Аксиома $\mathbf{P_1}$ утверждает, что не существует чисел "предшествующих" нулю, а аксиома $\mathbf{P_2}$ наделяет арифметику свойством бесконечности предметных констант. Докажем, например, что $2 \neq 1$:

$$\begin{array}{lll} \forall x\,[n(x) \neq 0] & => & \underline{\neg(n(0) = 0)}, \\ \forall x\,\forall y\,[\,n(x) \neq n(y) \;\vee\; x = y\,] & => & n(n(0)) \neq n(0) \;\vee\; \underline{\neg\neg(n(0) = 0)} \end{array}$$

Первый вывод использует правило подстановки в аксиому $\mathbf{P_1}$ вместо x константы 0 и определение предиката неравенства $x \neq y : \neg(x = y)$. Во втором выводе проводится подстановка в $\mathbf{P_2}$ вместо x: $n(0)$, а вместо y: 0. Затем, перед $n(0) = 0$ ставится двойное отрицание. Эти две формулы содержат подформулу $\neg(n(0) = 0)$ и её отрицание. Значит её можно выбросить, и вывести $n(n(0)) \neq n(0)$, что учитывая сокращения для констант имеет вид $2 \neq 1$.

• Таким образом, арифметика (и математика в целом) – это *конечные* последовательности символов формального языка, выражающие некоторое утверждение или его доказательство. Важным результатом Гёделя было описание алгоритма, который по входной строке символов, составляющей формулу $\mathcal{F}$, даёт её уникальный номер $f = \gamma(\mathcal{F})$. Таким образом, мы можем говорить не о символах, а о последовательностях натуральных чисел.

Для этого символы (буквы) формального языка определённым образом кодируются при помощи целых чисел. Обычно алфавит языка выбирается конечным. Для этого любые переменные задаются при помощи только одного, при необходимости повторяющегося символа x. Тогда y – это xx, z – это xxx, и т.д. Аналогично, для выражения бесконечного числа предметных констант достаточно символа "0" и предметной функции $n(x)$ дающей следующее после x число. Ещё более кратко мы можем писать $1 : 0'$, $2 : 0''$, $3 : 0'''$, т.е. при помощи символа штриха и константы 0 выражать любое натуральное число.

Далее поступим также, как и при нумерации программ. Присвоив каждому символу целочисленный номер (написанный под символом):

$$\begin{array}{cccccccccc} 0 & ' & x & = & (&) & \forall & \vee & \neg & ... \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & ..., \end{array}$$

мы можем выразить любую формулу в виде последовательности чисел, которые, при помощи набора простых чисел, можно превратить в одно число, равное гёделевскому номеру формулы. Для формулы с n символами берём n последовательных простых чисел, возводим их в степени, соответствующие коду символа и перемножаем:

$$\gamma(\text{"}\forall x\,(x = 0)\text{"}) = 2^7 \cdot 3^3 \cdot 5^5 \cdot 7^3 \cdot 11^4 \cdot 13^1 \cdot 17^6 = 17018665279814998800000.$$

Функция $f = \gamma(\mathcal{F})$ получает на вход строку символов, выражающих данную формулу $\mathcal{F}$, и на выходе даёт число, равное гёделевскому номеру f. Если термин "символы" смущает для арифметической функции, можно считать, что она сразу получает набор целочисленных кодов. Выше это $\gamma(7, 3, 5, 3, 4, 1, 6, 0, 0, ...)$, где 0 зарезервирован для конца строки.

Функция $f = \gamma(\mathcal{F})$ всюду определена, и более того – обратима. Т.е. существует $\Gamma(f)$, которая по номеру формулы, разложив его на простые множители, записывает в памяти набор чисел эквивалентных символьной записи формулы.

Также можно пронумеровать доказательства $\mathcal{D}$, которые как, как и формулы $\mathcal{F}$, являются некоторыми строками символов $\mathcal{D} : \mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}$.

X Теорема Гёделя о неполноте

Как ни странно, но существование формул, которые невозможно ни доказать ни опровергнуть, на самом деле, достаточно естественно. В математике нет ограничения на длину доказательства. Мы можем записать сколь угодно длинную формулу. Её формальное доказательство также является формулой, которая будет ещё длиннее. Более того, весь опыт математики говорит о том, что минимально возможная длина доказательства данной формулы напрямую не связана с её длиной. Многие очень простые формулы имеют очень сложные (=длинные) доказательства (теорема Ферма, задача о 4-х красках, и т.п.).

Отсутствие ограничений на длину доказательства наводит на мысль, что некоторые теоремы (в том числе и достаточно простые в своей формулировке) никогда не будут доказаны или опровергнуты, хотя бы потому, что самое короткое из возможных их доказательств превышает все ресурсы памяти и времени жизни человека и даже всего человечества. Так как не существует "самого длинного доказательства", то не сложно сделать еще один шаг и допустить существование формул, которые требуют *бесконечно длинных доказательств*. Такие формулы не доказуемы принципиально.

Гёдель в 1931 г. предложил формальную демонстрацию этого факта. Введем логическую функцию "$\mathbf{Proof}(A)$", обозначающую то, что существует доказательство формулы "A". Если доказательство "A" существует, то "$\mathbf{Proof}(A) = $ И $=$ истина", а если нет то "$\mathbf{Proof}(A) = $ Л $=$ ложь". Теперь рассмотрим высказывание $\mathbf{G}$, определяемое как утверждение о том, что оно недоказуемо:

$$\mathbf{G} : \neg\mathbf{Proof}(\mathbf{G}).$$

Сформулируем теорему Гёделя:

> *"Если теория непротиворечива, то формула* $\mathbf{G}$ *истинна, но ее нельзя доказать:* $\mathbf{Proof}(\mathbf{G}) = $ Л*".*

Непротиворечивость означает, что ложную формулу нельзя доказать $\mathbf{Proof}(\text{Л}) = \text{Л}$, а если формула доказуема $\mathbf{Proof}(\mathcal{F}) = $ И, то она истинна: $\mathcal{F} = $ И. Поэтому строится такое доказательство от противного:

$\Diamond_1$: пусть $\mathbf{Proof}(\mathbf{G}) = \text{И} \quad => \quad \mathbf{G} = \text{И},$ но $\mathbf{G} = \neg\mathbf{Proof}(\mathbf{G}) = \neg\text{И} = \text{Л}.$

Мы получили противоречие, значит исходная посылка "$\mathbf{Proof}(\mathbf{G}) = $ И" не верна, следовательно, доказательства утверждения $\mathbf{G}$ не существует. Но об этом и *утверждает* $\mathbf{G}$, поэтому мы построили высказывание, которое не доказуемо, но при этом истинно *по своему смыслу.*

Конечно можно было построить и прямое доказательство теоремы:

$$\Diamond_2\text{: пусть } \mathbf{Proof}(\mathbf{G}) = \text{Л} \quad => \quad \mathbf{G} = \text{Л}, \text{ но } \mathbf{G} = \neg\mathbf{Proof}(\mathbf{G}) = \neg\text{Л} = \text{И}$$

и опять получить противоречие. Однако, так не делают. Почему? Потому, что в первом случае вывод из "$\mathbf{Proof}(\mathbf{G}) = $ И" того, что "$\mathbf{G} = $ И" основывается на определении истинности (раз доказательство утверждения существует, то утверждение истинно). Во втором же случае, учитывая общие соображения, приведенные в начале этого раздела, можно усомниться в том, что из отсутствия доказательства утверждения $\mathbf{G}$, т.е. "$\mathbf{Proof}(\mathbf{G}) = $ Л" следует его ложность "$\mathbf{G} = $ Л" (как в прочем, и истинность). Поэтому, рассуждения $\Diamond_2$ считаются неправомерными.

Формулировки и доказательство, приведенные выше, очень *психологичны*. Они используют никак не определённое понятие **Proof** ("существует доказательство"), а также объект $\mathbf{G}$, который, подобно парадоксу лжеца, вводится со ссылкой на самого себя. Поэтому, в такой форме теорема о неполноте убедительна только при изрядной харизме доказывающего. Заслуга Гёделя состоит в том, что он придал и **Proof** и $\mathbf{G}$ видимость конструктивных объектов в рамках арифметики натуральных чисел.

Введём предикат $D(p, f)$, обозначающий, что формула под номером f имеет доказательство под номером p, т.е. $\mathcal{P}$ доказывает $\mathcal{F}$. Понятно, что когда доказательство $\mathcal{P}$ *уже построено*, в силу его формальности, мы всегда можем проверить, что оно соответствует принятыми правилам, и в его конце находится доказываемая формула $\mathcal{F}$. Подобный алгоритм вполне реализуем, поэтому предикат $D(p, f)$ можно также рассматривать как всюду определённую вычислимую на компьютере функцию, возвращающую 1, если $\mathcal{P}$ доказывает $\mathcal{F}$ или 0, если нет.

При помощи $D(p, f)$ мы можем определить утверждение, которое себя не может доказать. Рассмотрим сначала следующий простой вариант:

$$\mathbf{G} : \neg\exists p\, D(p, g), \qquad \text{где} \qquad g = \gamma(\mathbf{G}) = \gamma(\text{"}\neg\exists p\, D(p, g)\text{"}).$$

В этом определении, символ g – это *конкретное* число (гёделевский номер формулы), получаемое из решения уравнения: $g = \gamma(\text{"}\neg\exists p\, D(p, g)\text{"})$.

Теперь можно "смело" проводить доказательство теоремы о не полноте. Например, пусть $\mathbf{G}$ доказуемо, тогда существует некоторый номер доказательства p_0, при котором $D(p_0, g)$ истинно. Следовательно, можно ввести квантор всеобщности и записать $\exists p\, D(p, g)$, а это не что иное, как $\neg\mathbf{G}$. Мы приходим к противоречию, следовательно $\mathbf{G}$ не доказуемо. Аналогично проверяется недоказуемость $\neg\mathbf{G}$.

• Вернёмся к уравнению $g = \gamma(\text{"}\neg\exists p\, D(p,g)\text{"})$. Как оно должно решаться? В предикате $D(p,x)$ (или в соответствующем алгоритме для компьютера) кодируются все символы, и в тех местах, где встречается x, ставится сначала "0", вычисляется гёделевский номер формулы, и проверяется не равен ли он 0. Если нет, подставляется $0'$ (т.е. 1) и сравнивается результат с единицей, затем $0''$, и т.д, пока решение найдено не будет. Увы, но, по крайней мере в гёделевской нумерации, это уравнение *не имеет решения*! Действительно, если код штриха равен числу $m > 0$, то g штрихов в $0''''$ вместо x на k-той позиции приведут в γ к произведению простых чисел в степени m: $p_{k+1}^m \cdot \ldots \cdot p_{k+1+g}^m$, что существенно больше чем число g (которому должно равняться γ). Взять для штриха код $m = 0$ нельзя, так как тогда формулы $\gamma(\text{"}0 = 0\text{"})$ и $\gamma(\text{" } 0 = 0''\text{"})$ будут неотличимы. Таким образом, такой прямолинейный вариант построения $\mathbf{G}$ не проходит, и необходимы дополнительные хитрости.

• Предикат $D(p,f)$ получает номер любой формулы f. Однако, её доказательство p может существовать только если f замкнута (без свободных переменных). Так, "$x > 2$" не доказуема и не опровергаема. Рассмотрим, тем не менее, класс формул, которые содержат одну свободную переменную вида $F(x)$ и будем их доказывать при *данном* x:

$$Q(x): \quad x > 2, \qquad\qquad F(x): \quad \exists a\, \exists b\, \exists c\, (a^x + b^x = c^x), \qquad\qquad \ldots.$$

При любом конкретном x, первая формула имеет простое доказательство, и $Q(3)$, например, будет истинной. Вторая формула, для $F(3)$ потребует уже достаточно нетривиальных рассуждений и будет ложной, и т.д. Все такие формулы имеют номер $f = \gamma(\text{"}x > 0\text{"})$. Если же на место x подставлено конкретное число, номер будет *другой*, так как это другая строка $f_1 = \gamma(\text{"}0''' > 0\text{"})$.

Для подстановки "данного числа" введём функцию:

$$s(f) = \gamma\left(\,\{x/f\}\;\Gamma(f)\,\right).$$

Она вычисляется по следующему алгоритму. Берётся формула с произвольным номером f. По этому номеру "раскручивается" её строковое представление $\Gamma(f)$. В нём вместо всех свободных вхождений x подставляется число f [в форме $0'''''$], и снова вычисляется гёделевский номер. В принципе, может возникнуть несколько свободных переменных, поэтому будем считать, что $\{x/f\}$ означает, что все они замещаются на f. Чтобы $s(f)$ была всюду определена, необходимо модифицировать алгоритм нумерации так, чтобы для всех $f = 0, 1, 2, \ldots$ при применении $\Gamma(f)$ получались *правильно построенные* формулы, иначе нельзя будет идентифицировать свободна переменная или нет. *Пусть* это сделано.

Определим теперь высказывание (формулу):

$$G(x): \quad \neg\exists p\, D(p, s(x)).$$

Здесь содержится утверждение о том, что не существует доказательства некоторой формулы под номером x, в которую вместо всех свободных переменных подставлен номер этой формулы, т.е. конкретное число x. Пусть гёделевский номер $G(x)$ (формула зависящая от одной свободной переменной) равен g. Тогда определим:

$$\mathbf{G}: \quad \neg\exists p\, D(p, s(g)), \qquad \text{где } g = \gamma(\text{"}\neg\exists p\, D(p, s(x))\text{"}).$$

Подчеркнём, что g – это *конкретное число*, вычисленное при помощи гёделевской нумерологии по строке $\gamma(\text{"}\neg\exists p\, D(p, s(x))\text{"})$, где x *свободная переменная.*

Не сложно видеть, что формула $\mathbf{G}$ имеет номер равный $s(g)$:

$$\gamma(\mathbf{G}) = \gamma(''\neg\exists p\, D(p, s(g))'') = s(g).$$

Действительно, функция $s(g)$ восстанавливает формулу $G(x)$ и подставляет в неё число g. Таким образом, $\mathbf{G}$ утверждает свою недоказуемость.

• Подобное построение уже не содержит явных просчётов, но тем не менее предполагает доказанными ряд "очевидных" посылок. Так, мы считаем, что можно вычислять номер формул, содержащих "мета"- арифметическую функцию $s(x)$, которую не просто записать в рамках только арифметических операций, так как она использует поиск и замену свободных переменных. Поэтому, лучше $D(p, f)$ и $s(x)$ воспринимать как программы для идеального компьютера, например, на языке BASE. Правда, переходя к программированию, прийдётся выразить символы существования и единственности в виде бесконечных циклов, а не использовать их как самостоятельные математические объекты. Тем не менее, будем считать, что объект $\mathbf{G}$ "сконструирован".

Получая противоречие при доказательстве $\Diamond_1$ теоремы Гёделя, необходимо помнить, что возможны два варианта: (1) теорема верна; (2) объекты, участвующие в её доказательстве, не существуют, так как некорректно определены. Не всегда существование противоречия доказывает то, что мы хотим доказать!

Даже если при построении $\mathbf{G}$ удаётся формально устранить все сомнения, есть ещё один вопрос, связанный с квантором существования $\exists$, и определением предиката $\mathbf{Proof}(f) = \exists p\, D(p, f)$. Вспомним, что при доказательстве теоремы Тьюринга, противоречие свидетельствовало об отсутствии универсальной функции контроля остановки $T(p, i)$. Не возникает ли той же проблемы с функцией $\mathbf{Proof}(f)$?

XI Что такое истина?

• Утверждение о том, что "существует некоторое доказательство формулы $\mathcal{F}$ с номером f" в логических терминах можно выразить так:

$$\mathbf{Proof}(\mathcal{F}) \ : \ \exists p\, D(p,f) \ \leftrightarrow \ D(0,f) \vee D(1,f) \vee D(2,f) \vee ...$$

(справедливо или нулевое доказательство, или первое, или второе, и т.д.). Можно даже написать программу:

$\mathbf{Proof}(f)\{$
  $p \leftarrow 0;$
  $\mathbf{while}(1=1)\{$: бесконечный цикл
    $\mathbf{if}(D(p,f)=1) \quad \{\mathbf{return}\ 1;\}$: доказуемо $\mathcal{F}$
    $\mathbf{if}(D(p,\mathbf{not}(f))=1)\{\mathbf{return}\ 0;\}$: доказуемо $\neg\mathcal{F}$
    $p \leftarrow p+1;$: следующее док $-$ во
  $\}$
  : сюда никогда не попадём, если утверждение недоказуемо
$\}$

Мы чуть улучшили исходное определение, и для входного *номера* замкнутой формулы f проверяется, что p-тое доказательство доказывает формулу $\mathcal{F}$ либо её отрицание $\neg\mathcal{F}$. Проверка доказательства проводится при помощи функции "$D(p,f)$". Функция **not** добавляет перед строкой формулы символ "$\neg$", и вычисляет номер формулы с отрицанием.

Если *верить* в доказуемость любого утверждения, то $\mathbf{Proof}(f)$ рано или поздно остановится. Если на веру не полагаться, то для некоторых f функция $\mathbf{Proof}(f)$ может быть *неопределена*. Иногда она не останавливается и не выдаёт результата, когда цикл **while** крутится бесконечно долго. И распознать этого мы не можем! Конечно, скорее всего, можно построить более "хитрый", эффективный и всюду определённый алгоритм '**Proof**", но беда в том, что *его несуществование мы и пытаемся доказать*.

Действительно, в теореме Гёделя мы стремимся продемонстрировать, что бывают не доказуемые утверждения. Поэтому, аналогично теореме Тьюринга об остановке, мы "догадываемся", что не существует *универсального* алгоритма, проверяющего доказуемость любой формулы f. Поэтому, получение противоречия в теореме Гёделя "$\mathbf{Proof}(\mathbf{G}) = Л$" может свидетельствовать не об её истинности, а о не существовании предиката "**Proof**", применимого к любым высказываниям. Но тогда говорить об истинности $\mathbf{G}$ несколько странно, так как она определена при помощи не существующего универсального алгоритма. Дракон снова начинает хватать свой собственный хвост.

• Что же такое истина? В конкретной теории мы *договариваемся* о том, что некоторые исходные утверждения считаются истинными. Они объявляются аксиомами, а их отрицание – ложью. Например, пусть есть две аксиомы "W: *снег белый*", "C: *снег холодный*". Тогда мы легко можем *вычислить* истинности утверждений $W \,\&\, C$, $\neg W$, $\neg\neg C$, и т.д. Подобным образом можно сгенерить сколь угодно много истинных или ложных утверждений. "Сколь угодно много" означает *потенциальную бесконечность*, которую мы не предъявляем при помощи конечных символов, а просто верим в неограниченность некоторого определённого построения.

Проблемы появляются, когда в теории необходимо использовать кванторы всеобщности и существования, охватывающие все объекты теории. Для конечных предметных областей никаких проблем нет. Например, пусть истинно, что "*каждая блондинка умна*". Тогда очевидно истинно и утверждение "*не существует не умных блондинок*".

Однако, в бесконечной предметной области кванторы $\exists$, $\forall$ оказываются *актуально бесконечными* конструкциями. Они в замкнутой форме говорят обо всех сущностях, которых *бесконечно* много. В этом случае формулы, которые содержат $\exists$, $\forall$, не могут быть вычислены так же, как это делается для остальных логических операторов. Единственный путь проверки истинности утверждения, это построить его формальный вывод из исходных аксиом, а только *после этого* объявить утверждение истинным или ложным. В этом смысле, истинность или ложность отходят на задний план, а вперед выходит понятие доказуемости. Мы можем при помощи аксиом и правил вывода получать новые формулы. Все выведенные таким образом формулы называются истинными. Если в их множество попадает формула $\neg F$, тогда формулу F мы называем ложной. Если теория непротиворечива, то формула не может быть одновременно ложной и истинной, т.е. в множество выводимых формул не может попасть F вместе с $\neg F$.

Поэтому, учитывая рассуждения, сделанные в самом начале предыдущего раздела, можно сформулировать следующее достаточно правдоподобное *предположение*:

> Некоторое математическое утверждение F *может оказаться* принципиально недоказуемым, и во множество выводимых из аксиом формул не попадет ни F ни $\neg F$. Утверждать, что такое утверждение ложно или истинно мы не имеем никакого права. Оно просто *не обладает этим свойством*.

• Вообще сейчас мы вступаем на скользкую дорогу философии. Для платонически настроенного математика мир математических объектов реально существует – а он его только изучает. Поэтому, истинность или ложность любого утверждения является объективным, реально существующим фактом. Такой математик отождествляет себя с физиком, изучающим внешний по отношению к его сознанию объективный мир. Он верит, что этот мир непротиворечив и познаваем.

Такое отождествление сомнительно. В реальном мире нет даже отрицательных чисел, а уж тем более множества всех множеств. Придуманные правила Игры позволяют эффективно применять математику к окружающему миру для его познания. Однако, вера в то, что *порождение человеческого разума* должно обязательно оказаться полным и непротиворечивым, а истинность утверждения не требовать доказательств, является проявлением повышенного самомнения. То, что придумал человек, по определению, не может быть объективно.

Платонист, мотивируя абсолютный характер истинности или ложности, обычно приводит следующий аргумент. Математик, изучая структуру выдуманной (изобретенной) им теории, открывает результаты, которые он не закладывал в её аксиоматические основы. *Изобретатели* комплексных чисел Кардано и Бомбелли и не подозревали, какие удивительные *открытия* в этой теории в последствии будут сделаны.

Тот факт, что простые исходные правила Игры могут порождать очень сложные и красивые структуры и их свойства, действительно вызывает изумление. Благодаря этому, нам удается относительно простыми методами познавать бесконечно разнообразный окружающий мир. Однако, совершенно очевидно, что в выдуманной человеком системе, такая же сложность может оказаться и парадоксально противоречивой при неудачных исходных изобретениях. Пример тому – парадокс лжеца.

В любом случае, факт богатства выводов, лежащих в основе той или иной изобретенной модели, вряд ли может быть весомым доводом её объективности. А следовательно, то, что некоторое утверждение в такой теории объективно обладает свойством истинности или ложности, вызывает большие сомнения. Особенно, подозрительно объявлять истинным "неясно" определённый объект.

Рассуждать об истинности недоказуемого утверждения, которое себя не может доказать, безусловно забавно. А как быть, например, если окажется, что утверждения о бесконечности или конечности простых близнецов ($p_k - p_{k-1} = 2$: {2,3; 3,5; 5,7; 11,13; 17,19; 29,31; ...}) не выводимы из аксиом арифметики? Будет ли этот факт истинен или ложен?

Вопрос об истинности недоказуемых утверждений является, в известном смысле, терминологическим. В зависимости от того, как мы определяем "истинность формулы", и какую философскую позицию занимаем, возможно то или иное понимание результатов Гёделя. Однако, в любом случае, математическое мышление не может существовать без точных определений исходных "метапонятий", особенно таких важных, как доказуемость и истинность.

После последних разделов посвящённых Кантору и Гёделю, могут возникнуть вопросы:

- ▷ "А всё же, существуют ли несчётные множества, трансфинитные числа и подобные им объекты?"
- ▷ "Можно ли считать гёделевское утверждение **G** корректно определённым, а следовательно истинным и недоказуемым?"
- ▷ "Можно ли работать с неконструктивными объектами, для описания которых нет конечного алгоритма?"
- ▷ "Корректно ли рассуждать об объектах, определённых при помощи бесконечной рекурсии?"

Возможно два ответа. Один принадлежит Пуанкаре: "*В математике нет символов для неясных мыслей*". Второй ответ – эти вопросы поставлены *не верно*. Подобные объекты конечно существуют. Хотя бы в форме определённого возбуждения нейронной сети в голове математиков. И если есть часть математиков, которым нравится о них размышлять, убеждая друг друга, то это их право. Конечно, они могут быть однажды не приятно удивлены появлением новых парадоксов. Но это ведь тоже, в конечном счёте, интересно. Поэтому, СМОТРИ:

XII Интуиция искусственного разума

Замечательный математик и физик Роджер Пенроуз в книге "*Новый Ум Короля*" (1989 г.), в связи с самореферирующим доказательством теоремы Геделя, пишет (выделение Пенроуза):

> "Если ум математика работает полностью алгоритмически, то алгоритм (или формальная система), которые он обычно использует для построения своих суждений $\mathbf{Proof}(k, k)$, оказываются не в состоянии справиться с утверждением полученным с помощью его собственного алгоритма. Тем не менее, *мы* можем (в принципе) понять что $\mathbf{Proof}(k, k)$ на самом деле *истинно*! Этот факт, по всей видимости, должен был бы указать *ему* на противоречие, поскольку он, как и мы, не может не заметить его. А это, в свою очередь, может свидетельствовать о *неалгоритмическом* характере его рассуждений!"

Т.е. речь идет о том, что хотя мы не способны алгоритмически доказать некоторое утверждение, однако способны "интуитивно" понять, что оно истинно. На это компьютер, работающий строго алгоритмично, не способен в принципе. А если это так, то всегда будут оставаться области познания, в которых человек имеет преимущество перед машиной.

Стоит вспомнить, что говорил по этому поводу сам Тьюринг, который кое-что понимал в алгоритмах и теореме Гёделя:

> "Ответ на это возражение вкратце состоит в следующем. Установлено, что возможности любой конкретной машины ограничены, однако в разбираемом возражении содержится голословное, без какого бы то ни было доказательства, утверждение, что подобные ограничения не применимы к разуму человека. Я не думаю, чтобы можно было так легко игнорировать эту сторону дела. Когда какой-либо из такого рода машин задают соответствующий критический вопрос и она дает определенный ответ, мы заранее знаем, что ответ будет неверным, и это дает нам чувство известного превосходства. Не является ли это чувство иллюзорным? Несомненно, оно бывает довольно искренним, но я не думаю, чтобы ему следовало придавать слишком большое значение. Мы сами слишком часто даем неверные ответы на вопросы, чтобы то чувство удовлетворения, которое возникает у нас при виде погрешимости машин, имело оправдание." ["Могут ли машины мыслить?" (1950)]

Ответ Тьюринга прозвучал на сорок лет раньше вопроса Пенроуза.

Как мы видели, истинность недоказуемых утверждений не очевидна для интуиции части человечества, по крайней мере, автора этой критики. Следовательно, интуиция не может быть надежным инструментом для установления истинности. И все же, проблема реализации интуиции для системы искусственного интеллекта имеет место. Для того же Пенроуза истинность недоказуемого утверждения интуитивно очевидна. Имеют ли машины право на такое же разнообразие интуитивных умозаключений?

Кроме этого, интуиция играет важную роль, как при выборе "интересной" теоремы, так и при поиске её доказательства. Достаточно привести решительное высказывание Анри Пуанкаре:

> Доказывают при помощи логики, изобретают при помощи интуиции. Хорошо уметь критиковать, ещё лучше уметь творить. Вы способны распознать, правильна ли данная комбинация, и это недурно, раз вы не обладаете искусством сделать выбор между всеми возможными комбинациями. Логика нам говорит, что на таком-то пути мы можем быть уверены, что не встретим препятствий; она не говорит, какой путь ведёт к цели. Для этого необходимо видеть цель издалека, и интуиция есть та способность, которая этому нас учит. ["*Наука и метод*"].

Несомненно, человек обладает интуицией. Она является отражением вполне объективных процессов, происходящих "в бессознательной" части нашего разума. И только то, что сознание человека обычно не способно ощущать детали этих процессов, проявления интуиции часто воспринимается как ощущение некоторого чуда. По этой же причине широко распространено заблуждение о том, что интуиция не является проявлением алгоритмической активности, и компьютер в принципе не способен ею обладать. Поэтому, искусственный разум всегда будет уступать человеку в некоторых интеллектуальных задачах.

Интуиция в компьютерных науках имеет прямое отношение к проблеме распознавания. Человек, рассматривая различные доказательства, как в формальных теориях, так и в разговорных доводах, делает обобщения и учится распознавать то, что является истинным, не проводя детального доказательства. Таким образом, интуицию можно рассматривать как сложное проявление аналога способности живых существ распознавать зрительные и слуховые образы.

Интеллектуальные компьютерные системы подобное распознавание делают пока хуже человека. Однако делают! И прогресс в этой области постоянный. Рано или поздно к зрительному и слуховому распознаванию добавится распознание истинности математических утверждений.

• Учитывая "очевидность" для многих неалгоритмичности интуиции, остановимся на этом подробнее.

Распознавание образов, проведенное как человеком, так и компьютером, не является алгоритмичным в том смысле, что не использует формально определенных методов решения задачи распознавания. Кошка похожа на кошку, и все тут. Когда строится тот или иной *алгоритм* распознавания, обычно в нем не перечисляются строго заданные правила типа: "Проверь есть ли у объекта хвост. Если да – то это кошка, если нет – математик". Компьютеру, как и ребёнку, показывают некоторое "множество кошек" и для правильно построенной системы этого оказывается достаточным, чтобы "понять", что есть кошка.

При построении "интеллектуальных алгоритмов" явным образом прослеживаются две противоположные тенденции.

В первой происходит поиск алгоритмов, *решающих конкретную задачу*. Например, шахматная программа, пытаясь найти оптимальный ход, последовательно перебирает узлы дерева вариантов развития партии. Методы типа "альфа – бета отсечений" этот перебор делают достаточно эффективным. Этот алгоритм приводит к более сильной игре компьютера по сравнению с игрой большинства представителей человечества. Однако, шахматная программа в покер играть не умеет. Аналогично, при помощи хорошо структурированной базы знаний (экспертной системы), можно достаточно неплохо решать некоторые задачи принятия решений в узких предметных областях, для которых эта база и создавалась.

Второй, противоположный, подход заключается в алгоритмическом задании *правил функционирования* некоторых систем. Причем, эти правила могут быть очень простыми а поведение систем очень сложным. Классический пример такой ситуации – игра "Жизнь". Три простых правила поведения клеточного автомата приводят к поразительно сложным структурам, которые в этих правилах явным образом заложены не были.

Некоторые, таким образом определенные системы, обладают не только сложным поведением, но и способностью ассоциативно запоминать информацию и проводить обобщения. В этом случае они становятся универсальными машинами, способными решать, например, задачи распознавания. Алгоритм "как это делать" в эти системы не закладывается. Например, правила работы компьютерных нейронных сетей задаются простыми алгоритмами описывающими "физику" поведения нейрона. В них ничего не говорится, как решать ту или иную задачу распознавания. Одна и та же, по разному обученная сеть может распознавать как спектрограммы химических элементов, так и лица симпатичных девушек.

Являясь по своей сути алгоритмическим объектом, нейронная сеть обладает элементами "неалгоритмичности" в своём поведении. По крайней мере, её создатель часто не способен точно предсказать результат работы такой системы. Она же может выдавать и нечёткие выводы (с вероятностью 70% – это углерод, или это красавица, и т.д.) За невысокими горами и развитие на этой же основе математической интуиции. Естественно, если две системы обучаются на различных выборках, они будут иметь и несколько различную интуицию, хотя в большинстве случаев их выводы должны совпадать. Только благодаря этому, математики, обычно, способны убедить друг друга в достоверности своих выводов.

В связи с реализацией математической интуиции, в искусственных системах возникает множество вопросов. Какое пространство признаков необходимо использовать при обучении компьютера интуитивному мышлению? Понятно, что доказательства из обучающей выборки будут иметь различную длину (размерность). Какой мерой близости необходимо наделить две формулы или два математических рассуждения? Их список достаточно длинный, и задача выглядит исключительно сложной.

Проблема несчетности физического мира также имеет определенное отношение к счётным алгоритмам и искусственному интеллекту. Однако, даже если не принимать во внимание сомнения, приведенные выше, проблема несчетности, по-видимому, в равной степени относится как к компьютеру, так и к человеку. Хотя человеческий мозг является аналоговым природным компьютером, для его высокоуровневого поведения важность непрерывности, нижнего физического уровня представляется сомнительным. В конечном счете, мозг – это колония клеток, а интеллектуальное поведение – лишь отходы их жизнедеятельности. Компьютер также является физическим устройством, хотя его поведение, в большинстве случаев, достаточно идеально. Вполне вероятно, что в будущем потребуется дополнить классическую архитектуру компьютера некоторыми аналоговыми устройствами, например, физическим генератором случайных чисел и т.п. Однако, нет ни одного факта, доказывающего принципиальную функциональную несводимость естественных интеллектуальных систем к искусственным.

В любом случае, результаты Геделя или Кантора не имеют никакого отношения к проблеме построения искусственного интеллекта. Скорее всего, эта задача является типично конструктивной. Возможность построения искусственного интеллекта, превосходящего человеческий, будет доказана в момент его возникновения. Тогда как доказать его невозможность по-видимому невозможно...