Школа современного программирования

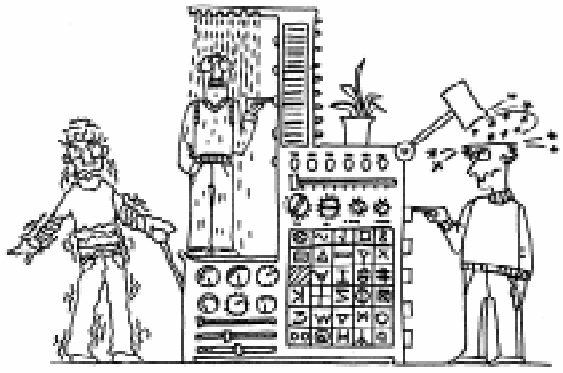
Лавров Святослав Сергеевич

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

1. ВЫЧИСЛЕНИЯ

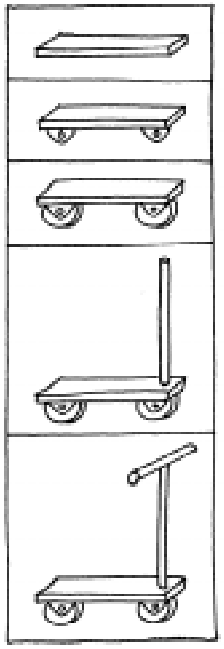
В этом разделе кратко излагается теория процессов *вычисления*, которые, отправляясь от некоторых *исходных данных*, завершаются в благоприятном случае получением соответствующего *результата*. Должен быть точно определен способ осуществления таких процессов – без этого никакую теорию построить невозможно. С этой целью описывается некоторая *абстрактная машина*, работающая с данными и преобразующая их по определенным правилам. Вид данных и правила их преобразования – это все, что надо знать о машине.

Данные могут быть чрезвычайно, даже неограниченно, разнообразны по содержанию. Но их форма должна быть определена коротко и понятно. Это нужно не столько машине, сколько человеку, который избирает некий вид данных (например, письменную речь, графические схемы, программы на алгоритмических языках и т. п.), чтобы зафиксировать свои мысли и знания о мире, событиях, фактах. В этом же виде он передает их другим людям, владеющим тем же или сходным представлением о связи подобных данных с возможными мыслями и знаниями.



Вид данных и правила их преобразования — это все, что надо знать о машине.

Примерно то же самое можно сказать и о правилах работы машины. При всем разнообразии (желательно – предельном) процессов, которые машина должна быть способна исполнить, совокупность правил должна быть обозримой. Иначе было бы невозможно ни предписать машине какой-либо образ действий, ни убедиться самому, что эти действия способны привести к желаемой цели. Для этого процесс вычисления обычно расчленяется на *шаги*. На каждом шаге машина исполняет одно *элементарное действие*, руководствуясь одним выбранным правилом. При этом используются *текущие данные* – те, которыми машина располагает к началу шага, возможно даже, лишь их небольшая часть. Те-



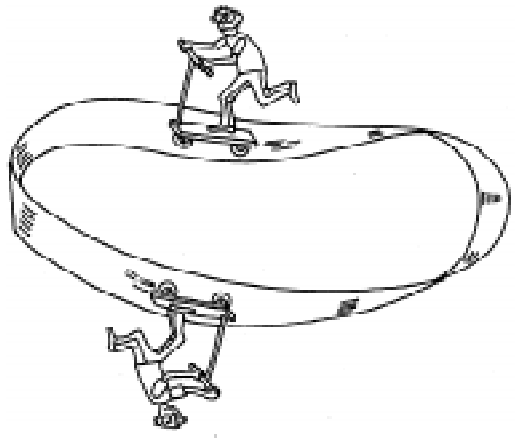
На каждом шаге машина исполняет одно элементарное действие

кущими данными для самого первого шага служат исходные данные процесса.

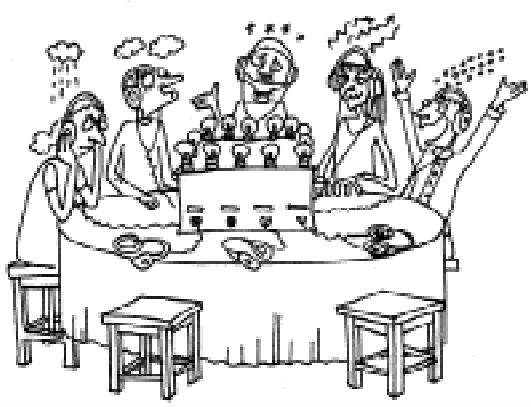
Каждый шаг исполняется по такой схеме: выбирается или окончательно уточняется правило для данного шага, изменяются в соответствии с этим правилом текущие данные или их выделенная часть, проверяется – не закончен ли процесс, если еще нет, то подготавливается следующий шаг. Если процесс закончился, то полученные текущие данные становятся результатом его исполнения.

Нет никакой гарантии, что на каком-либо шаге проверка на окончание процесса даст положительный результат (это и был бы упомянутый в самом начале благоприятный случай). В неблагоприятном случае процесс не сможет завершиться никогда. Но машина ничего «знать» об этом не может – если бы могла, то это дало бы ей повод прекратить исполнение процесса. Даже если за работой машины наблюдает человек (который, в соответствии со сказанным, не имеет права вмешиваться в работу машины, но может за ней следить), предсказать дальнейший ход процесса ему в общем случае трудно, а то и невозможно.

Правила работы абстрактной машины для решения некоторой *задачи* – получения результата, связанного определенным образом с исходными данными, называются *алгоритмом* ее решения, а достаточно полное описание этой связи – *спецификацией* задачи. На спецификацию можно смотреть как на уравнение, связывающее результат с исходными данными. Но в очень многих случаях спецификация – это просто вариант алгоритма, написанный на несколько ином языке. Стоит задуматься: как, например, поставить задачу о вычислении факториала натурального числа.



В неблагоприятном случае процесс не сможет завершиться никогда.



Одной из простейших абстрактных машин можно считать алгебру высказываний.

Одной из простейших абстрактных машин можно считать алгебру высказываний. В ней над *логическими* значениями, обозначаемыми «истина» (чаще True или T) и «ложь» (False или F), можно выполнять хорошо известные *операции* с логическими аргументами и результатом: *отрицание* (знак операции not или $\neg$), *импликацию* (imp или $\Rightarrow$), *конъюнкцию* (and или $\wedge$), *дизъюнкцию* (or или $\vee$) и *эквивалентность* (equ или $\Leftrightarrow$).

Задание 1. Основные и производные логические операции.

Известно, что две первые из названных операций можно считать основными (кстати, они и употребляются чаще), а остальные рассматривать как сокращения для некоторых их комбинаций. Выпишите соответствующие правила.

Выполнение задания 1.

Пусть знак $\cong$ означает «равно по определению». Тогда:

$$x \vee y \cong \neg x \Rightarrow y,$$

$$x \wedge y \cong \neg (x \Rightarrow \neg y),$$

$$x \Leftrightarrow y \cong (x \Rightarrow y) \wedge (y \Rightarrow x)$$

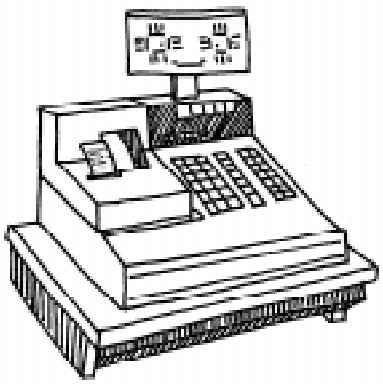
или, избавляясь от промежуточной операции $\wedge$,

$$x \Leftrightarrow y \cong \neg ((x \Rightarrow y) \Rightarrow \neg (y \Rightarrow x)).$$

Конец задания.

В качестве более сложного примера абстрактной машины рассмотрим трактовку простейших *арифметических* функций с числовыми аргументами и результатом в *теоретической арифметике* (на самом деле – в подступах к ней).

Арифметика начинается с умения *считать* – двигаться вдоль последовательности: нуль, один, два, три, ..., называемой *натуральным рядом* чисел. Как, – скажут некоторые из читателей в изумлении, – всем известно, что натуральный ряд начинается с единицы, а не с нуля! Да, детей в школах, как, по-видимому, и их педагогов в вузах, учат именно так, забывая при этом сказать, что есть и другое возможное определение, ничем не уступающее первому, а кое в чем более удобное. Если даже изгнать 0 из начала натурального ряда, через десять шагов он появится вновь в качестве полноправной, «значащей», то есть изображающей число, десятичной цифры из последовательности 0, 1, ..., 9. Так не лучше ли оставить за ним все права с самого начала? Все определения в математике и других естественных науках не вполне естественны – они придуманы людьми, а не взяты из природы.



В качестве более сложного примера такой машины рассмотрим трактовку простейших арифметических функций...

Один шаг процесса *счёта* – это переход от натурального числа n к следующему за ним числу Prim(n), обычно обозначаемому n'.

Вопрос. Почему n', а не $n + 1$?

Ответ будет дан, подождите немного.

Счет – это достаточно определенный процесс. Если мы с вами пересчитали баранов в стаде (и не ошиблись при этом!), то получили одно и то же число – ваш результат равен моему. *Равенство* чисел – это не арифметическая и не логическая функция, а так называемое *отношение* или арифметический *предикат* – операция с двумя числовыми аргументами и логическим результатом. Свойства равенства определяются двумя аксиомами:

E1. $x = x$ (*рефлексивность* равенства).

E2. $(x = y) \Rightarrow (\mathtt{A}(x, x) \Rightarrow \mathtt{A}(x, y))$ (его частичная *подстановочность*).

Аксиома E2 означает, что в одной из двух групп вхождений переменной x в формулу $\mathtt{A}$ ее можно заменить равной ей по значению переменной y. Из этих аксиом вытекают другие фундаментальные свойства равенства: его *симметричность* $x = y \Rightarrow y = x$ и *транзитивность* $x = y \Rightarrow (y = z \Rightarrow x = z)$. Во всех следующих аксиомах равенство участвует совместно с другими операциями. Это позволяет (а в аксиоме S1 – запрещает), подставляя, вместо x и y, различные *термы* (выражения с числовыми значениями), строить новые равенства (здесь в смысле: формулы, утверждающие равенство двух значений). Некоторые аксиомы будут даны в двух вариантах – под оба определения натурального ряда.

S1. $\neg (0 = x')$, S1a. $\neg (1 = x')$

(счет начинается с нуля или единицы – ни одно число не предшествует выбранному началу ряда),

S2. $x = y \Rightarrow x' = y'$

(если мы с вами не сбились в счете, дойдя, соответственно, до x и y, то и следующий шаг должен быть успешным),

S3. $x' = y' \Rightarrow x = y$

(и наоборот, шаг может быть успешен, если сбоя не было перед этим).

Две фундаментальные арифметические операции: сложение и умножение, определяются с помощью аксиом:

A1. $x + 0 = x$, A1a. $x + 1 = x'$

(вот и ответ на возникший вопрос: не x' определяется через сложение числа с единицей, а наоборот),

A2. $x + y' = (x + y)'$

(предполагается, что сложить x с y мы уже умеем, тогда прибавить к x число, следующее за y, – это то же самое, что взять число, следующее за $x + y$),

M1. $x * 0 = 0$, M1a. $x * 1 = x$,

M2. $x * y' = (x * y) + x$

(тот же прием, с опорой на уже определенную операцию сложения).

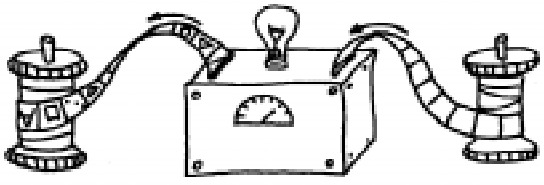
Эти аксиомы дополняет принцип *математической индукции*: если некоторое свойство $\mathtt{A}(x)$ (имеется в виду некоторая формула со свободной переменной x, вместо которой можно подставлять любые термы, например, 0 или x') верно для числа 0 (*база* индукции), а из того, что оно верно для числа x, следует, что оно верно для x' (индукционный *шаг*), то оно верно для любого натурального числа x. Соответствующая схема аксиом:

I1. $\mathtt{A}(0) \Rightarrow (\forall x(\mathtt{A}(x) \Rightarrow \mathtt{A}(x')) \Rightarrow \forall x\mathtt{A}(x))$.

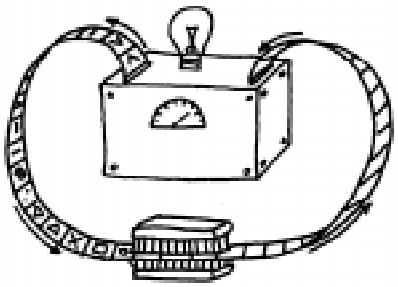
Последним примером нам послужит абстрактная *машина Тьюринга*. Данные, с которыми она работает, представлены на бесконечной *ленте*, разбитой на *ячейки*. В каждой ячейке может быть записана *буква* из заранее фиксированного *алфавита* этой машины. Над одной из ячеек находится *головка* машины. Задан также конечный набор возможных *состояний* машины. Эти состояния никак не связываются с содержимым ленты.

Содержимое ленты (записанное на ней слово), состояние машины и положение головки вместе составляют то, что было названо текущими данными.

Правила работы машины определяются *программой*, состоящей из конечного набора *команд*. Выбор команды для исполнения зависит от текущего состояния машины и от буквы, записанной в ячейке, над которой находится головка



Последним примером нам послужит абстрактная машина Тьюринга.

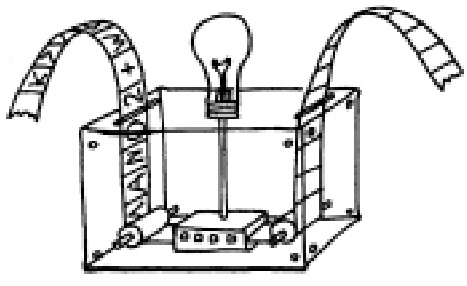


...понятие бесконечной ленты неконструктивно.

(ячейка может оказаться и пустой – пробел на ленте считается одной из букв алфавита). Для каждой такой пары в программе должно содержаться не более одной команды. Если команды не оказалось, то работа машины на этом *завершается*. Если команда нашлась, то в ней должны быть указаны: буква, *записываемая* в ячейку под головкой (содержимое остальных ячеек не меняется), *перемещение* головки (оставаться над той же ячейкой или сдвинуться на одну ячейку вправо или влево), состояние, в которое *переходит* машина.

Перед началом работы машина находится в выделенном *начальном* состоянии, головка расположена над самой левой из непустых ячеек ленты, на ленте, начиная с этой ячейки, записано *исходное слово*, остальные ячейки пусты. Слово, записанное на ленте в момент завершения работы машины (все ячейки слева и справа от него должны быть пусты), считается *результатом* работы машины по данной программе над данным исходным словом.

Это неформальное определение не лишено недостатков, которые довольно легко устранить. Например, понятие бесконечной ленты неконструктивно. Хуже того, на такой ленте невозможно найти ни один из концов слова-результата – за любой группой пустых ячеек могут снова начаться непустые, тогда они вместе с упомянутыми пустыми принадлежат этому слову. От этого понятия можно избавиться, считая, что по обоим концам исходного слова на конечной ленте находятся (не входящие в основной алфавит) символы конца слова. Если при сдвиге головки она



Устройство машины Тьюринга ... примитивно.

попадает на такой символ, то лента пополняется пустой ячейкой между этим символом и той частью ленты, на которой записано текущее слово. Над этой пустой ячейкой и устанавливается головка. В каждый момент работы машины лента конечна, сохраняется лишь возможность неограниченно наращивать ее длину (*потенциальная бесконечность* ленты).

Устройство машины Тьюринга, как и многих других абстрактных машин, чрезвычайно примитивно. Поэтому составление для этой машины программ, решающих даже очень простые задачи, оказывается довольно сложным делом. Не видно общего способа строить из таких программ программы решения более сложных задач. Тем не менее, эта модель абстрактной машины весьма популярна в традиционных монографиях, статьях и руководствах по теории вычислимости.

В разделе «Язык Лисп», который предполагается опубликовать в следующем выпуске журнала, будет рассмотрена абстрактная машина с языком более «высокого» уровня. Язык частично основан на идеях и обозначениях, предложенных А. Черчем, которому практически одновременно с А. Тьюрингом и некоторыми другими авторами удалось формализовать понятие алгоритма.

Вскоре выяснилось (было математически строго доказано), что все предложенные схемы абстрактных машин обладают одинаковыми возможностями. Было обнаружено также, что на практике не удается указать вычислительный процесс, который не мог бы быть описан любым из предложенных способов. Это дало повод высказать так называемый *тезис Черча* о том, что чисто интуитивное понятие алгоритма обрело и строгую математическую формулировку. Сам тезис Черча не может быть строго доказан, поскольку это интуитивное понятие в нем сохраняется.

Среди предложенных формализаций упомянем понятие частично рекурсивной функции, возникшее как результат попыток не только формализовать арифметику, но и *арифметизировать* логику – представить натуральными числами и действиями над ними как утверждения из области теоретической арифметики, так и связанные с ними логические построения.

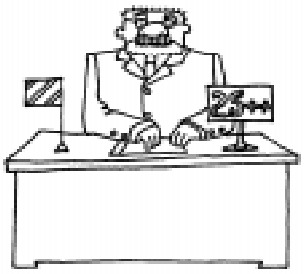
2. О ФУНКЦИОНАЛЬНОМ СТИЛЕ ПРОГРАММИРОВАНИЯ

Функция – это одно из главнейших и известнейших математических понятий. Как и все в мире, оно существует во многих видах. Подойдем к нему с позиций программирования для *реальных* машин (компьютеров) как к способу вычислить новое значение – *результат* по уже существующим – *аргументам*. Их может быть как один, так и больше. Функция с n аргументами называется *n-местной*. Значения могут различаться по *типам*. Выше уже встречались функции с *логическими* и/или *числовыми* аргументами и результатом. Именно функции, но не в их классическом теоретико-множественном, а в программистском понимании – как способ вычислить (когда это удается) значение

...как способ вычислить ... значение функции при заданном значении аргументов

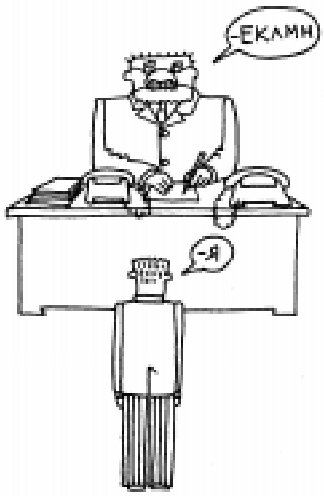
функции при заданном значении аргументов, фактически служат одним из вариантов понятия алгоритма.

Функцию можно вычислить только тогда, когда она *определена*, то есть когда задан способ (правила) ее вычисления. В этих правилах сама функция и ее аргументы получают имена – *буквенно-цифровые* (идентификаторы) или *символьные*, например, ‘+’ или ‘⇒’. *Идентификатор* – это слово, составленное из букв и цифр, но начинающееся обязательно с буквы. В разделе «Вычисления» для каждой логической операции были приведены оба варианта имени. С практической точки зрения идентификаторы удобнее: они легче набираются.



...имена — буквенно-цифровые (идентификаторы)...

Если мы хотим вычислить функцию, надо написать *обращение* к ней (ее *вызов*). Отдельно взятое имя аргумента или вызов функции называется *выражением*, лучше – *термом*. В приложениях тексты обычно имеют подразумеваемую интерпретацию, имеющую некоторую *область* возможных значений. Например, в арифметике область – это числа, а логические значения относятся к более фундаментальной области (Дети начинают говорить «да» и «нет» раньше, чем многое другое, и заведомо раньше, чем считать). Выражения со значениями, принадлежащими области интерпретации, называются термами, а выражения с логическими значениями – формулами. Неинтерпретируемые выражения часто называют просто *словами*.



Отдельно взятое имя аргумента или вызов функции называется выражением...

Имя аргумента функции в правилах ее вычисления обозначает его значение, которое обычно оказывается разным в различных обращениях к этой функции. Встречаются три главные вида обращений – в *префиксной*, *постфиксной* или в *инфиксной* записи. Например, $f(1)$ – это префиксная запись обращения к функции f с аргументом 1, $\neg\ x$ – префиксное же обращение к функции ‘¬’ с аргументом x, но без скобок, $n!$ – постфиксное обращение к функции «факториал» со знаком операции ‘!’ и аргументом n, а $x + y$ – инфиксное обращение к функции ‘+’ с аргументами x и y.

Упражнение 1. В каком контексте допустимо это последнее обращение?

Ответ. Там, где имеют смысл обозначения x и y, то есть внутри определения функции, где они служат именами аргументов.

Запись с использованием символьных имен часто связывается с заменой термина «*функция*» на «*операция*» обычно для одноместных функций в префиксной или постфиксной записи и двухместных функций – в инфиксной записи. Порядок действий при вычислении значений выражений всегда может быть задан скобками, но часто он определяется установленным *старшинством* операций. Самыми старшими, выполняемыми в первую очередь, обычно считаются одноместные операции. Например, – $x + y$ – это то же самое, что $(–x) + y$. Применяются также правила, разрешающие опускать в некоторых случаях знаки операций или заменять их позиционированием (подъемом вверх) некоторых аргументов.

Упражнение 2. Восстановите всё недостающее в выражении $2x^2–5x+3$.

Решение. $((2 * (x\text{^}2)) – (5 * x)) + 3$, где $*$ – знак операции умножения (старше сложения или вычитания), а ^ – знак операции возведения в степень (старше умно-

жения). Операции сложения и вычитания, имеющие одинаковое старшинство, выполняются слева направо.

Аксиомы A1, A2, M1 и M2, приведенные в разделе «Вычисления», могут служить примерами описания операций сложения и умножения в *функциональном стиле*. Главная черта этого стиля – выделить частные случаи (вроде сложения числа с нулем), в которых операция выполняется тривиальным образом, а для общего случая дать правило или правила его пошагового сведения к более простому варианту, пока не появится возможность завершить вычисления по одной из этих тривиальных схем. Так описанные функции называют еще *рекурсивными* (от латинского глагола «бежать назад» или просто «возвращаться»). О рекурсивных функциях и алгоритмах популярно рассказано в [1] и [2].

Задание 2.

В программировании, функциональном или каком-либо ином, написать программу – это даже не полдела. Главное – убедиться самому и убедить тех, кто будет пользоваться вашей программой, что программа решает именно ту задачу, для которой она была составлена. В этом задании вам предлагается доказать простейшие свойства программ сложения и умножения.

a) Докажите справедливость переставновочного закона для сложения.

b) Докажите, что $2 * 2 = 4$.

Выполнение задания 2.

Перестановочный закон: $x + y = y + x$ – это утверждение общего характера о свойстве операции сложения применительно к произвольным натуральным числам x и y. Подобные утверждения доказываются по индукции (см. раздел «Вычисления»). Заслуживает внимания родство принципа индукции с функциональным стилем: база – это выделенный простейший случай, а шаг (шаги) – сведение общего случая к этому частному.

a) Справедливость равенства $x + y = y + x$ начинаем доказывать индукцией по переменной x при фиксированном, хотя и произвольном, значении переменной y.

База индукции: надо установить, что $0 + y = y + 0$. Здесь значение y произвольно, так что текущее доказательство прерываем и снова начинаем с базы: $0 + 0 = 0 + 0$. Эта формула – следствие тривиального свойства равенства: его рефлексивности. Шаг индукции: надо убедиться, что из условия $0 + y = y + 0$ следует $0 + y' = y' + 0$. Строим цепочку равенств:

$0 + y' = (0 + y)' =$	(по A2)
$= (y + 0)' =$	(по условию)
$= y' =$	(по A1)
$= y' + 0$	(по A1).

Индукция по y завершена, осталось завершить индукцию по x, выполнив индукционный шаг. Пусть $x + y = y + x$. Необходимо установить, что $x' + y = y + x'$. Построение цепочки начинаем с правой части:

$y + x' = (y + x)' =$	(по A2)
$= (x + y)' =$	(по допущению)
$= x + y'$	(по A2, ... а куда это нас занесло?).

Очередное продвижение по цепочке, казалось бы – естественное, заводит в тупик. Поэтому обрываем цепочку на предыдущем звене и попытаемся доказать, что $(x + y)' = x' + y$ для любого y.

База индукции: $(x + 0)' = x' + 0$. Ее от последних двух звеньев из предпоследней цепочки отличают лишь обозначения. Пусть $(x + y)' = x' + y$, тогда $(x + y')' = ((x + y)')' = (x' + y)' = x' + y'$ (по A2, допущению и снова A2). Итак, завершен шаг индукции по y, а вместе с ним – и многострадальная индукция по x.

b) Прежде всего, следует догадаться, что 1 – это $0'$ (здесь 1 не имеет ничего общего с тем же символом в аксиоме S1a), $2 = 1' = 0''$, $3 = 2' = 0'''$, $4 = 3' = 0''''$ и т. д. Если раньше вы не ценили десятичную систему счисления, то сейчас самое время признать ее достоинства (впрочем, сразу за этим признанием придется составить рекурсивные программы выполнения основных арифметических действий над десятичными числами). Дальше уже просто – выписываем цепочку равенств, в которой равенство каждой

пары соседних выражений вытекает из предыдущих равенств:

$2 * 2 = 2 * 1' = 2 * 1 + 2 = 2 * 0' + 2 =$
$= (2 * 0 + 2) + 2 = (0 + 2) + 2 =$
$= (2 + 0) + 2 = 2 + 2 = 2 + 1' =$
$= (2 + 0')' = ((2 + 0)')' = (2')' = 3' = 4.$

Здесь каждое звено цепочки основано на одной из аксиом арифметики, кроме звена, связывающего первую строчку со второй, где, заменяя 0 + 2 на 2 + 0, мы не словчили, а использовали часть а) в качестве леммы – обычный прием в математике.

Конец задания.

Замечания.

1. Доказательства свойств программ оказываются во много раз длиннее самих программ, да и строить их сложнее. В математической практике встречается нечто похожее. Простые по формулировке теоремы иногда доказываются чрезвычайно сложно (классический пример – великая теорема Ферма), а математический аппарат заимствуется из гораздо более абстрактных разделов математики. Но и при этом мало-мальски обозримое доказательство можно написать, лишь уйдя весьма далеко от правил и приемов формальных логических выводов. Примерно то же самое происходит и в программировании.

2. Каждому программисту, желающему писать правильные, то есть обладающие всеми желательными свойствами, программы можно дать совет: создавайте для себя мастерскую с удобным для вас инструментарием и оборудованием (наборы проверенных программ, доказанные теоремы и леммы, эффективные приемы доказательства и т. п.).

Функции будем описывать в виде, который объясним на примере описания функции возведения целого числа n в степень с натуральным показателем m:

function ^(n, m: integer): integer; infix;
$n \wedge 0 = 1$;
$n \wedge (m') = (n \wedge m) * n$.

В первой строчке – *заголовке* этого описания – задано имя функции (символ ^), имена n и m ее аргументов, их тип и тип (также integer) результата. Указано также, что обращение к функции записывается в инфиксном виде. Две следующие строчки очень похожи на пары аксиом A1 и A2, M1 и M2, определявшие свойства (правила вычисления) сумм и произведений натуральных чисел.

Задание 3. Опишите две функции из «джентльменского набора»: вычисление факториалов и чисел Фибоначчи.

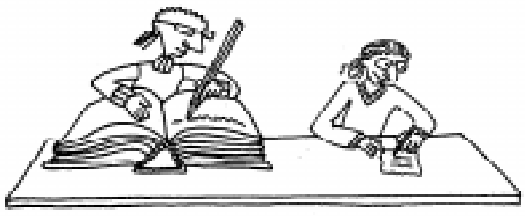
Выполнение задания 3.

function !(x: integer): integer; postfix;
$0! = 1$;
$x'! = x! * (x')$.
function Fibonacci(x: integer): integer;
Fibonacci(0) = 1; Fibonacci(1) = 1;
$\text{Fibonacci}(x'') = \text{Fibonacci}(x) +$
$+ \text{Fibonacci}(x')$.

Можно надеяться, что факториал особых сюрпризов читателю не преподнес. Последнюю строчку описания функции Fibonacci хотелось бы заменить на $\text{Fibonacci}(x') = \text{Fibonacci}(x) + \text{Fibonacci}(x - 1)$. Но «отнимать и делить» мы еще не обучены. Вычитание – это операция, обратная по отношению к сложению. Пока мы ее не ввели, ограничимся операцией, обратной по отношению к prim (или ′). Обозначим через $\sim x$ число, предшествующее x, так что $(\sim x)' = \sim(x') = x$. Это дает право написать

$\text{Fibonacci}(x') = \text{Fibonacci}(x) +$
$+ \text{Fibonacci}(\sim x)$.

Но программист должен стремиться писать не только правильные, но и экономные программы. Пусть NFib(x) обозначает число сложений, требуемых для вычисления чисел Фибоначчи по предложенной схеме. В ней Fibonacci(x) и Fibonacci($\sim x$)



Но программист должен стремиться писать не только правильные, но и экономные программы.

вычисляются независимо друг от друга, поэтому

$$NFib(x') = NFib(x) + NFib(\sim x) + 1$$

или

$$NFib(x') + 1 = (NFib(x) + 1) + (NFib(\sim x) + 1).$$

Сравнивая две последние формулы и учитывая, что $NFib(0) = NFib(1) = 0$, получаем

$$NFib(x) = Fibonacci(x) - 1.$$

Но достаточно очевидно, что вычислить $Fibonacci(x)$ можно, выполнив всего $x - 1$ сложение. Нужна схема вычислений, в которой к началу вычисления $Fibonacci(x')$ доступны значения $Fibonacci(x)$ и $Fibonacci(\sim x)$, то есть имена для них. В рамках функционального программирования других имен, кроме имен аргументов, у нас нет (имена функций едва ли могут помочь). Необходимый прием достаточно прост – описываем вспомогательную функцию Fib с достаточным запасом аргументов:

function Fibonacci(x: integer): integer;
Fibonacci(x) = Fib(x, 1, 1, 1).
function Fib(x, y, u, v: integer): integer;
Fib(x, x, u, v) = u;
Fib(x, y, u, v) = Fib(x, y', $u + v$, u).

Здесь у функции Fib заданное при обращении к ней из Fibonacci значение аргумента x в обращениях из Fib не меняется, аргумент y пробегает при этих обращениях последовательные натуральные значения, а аргументы u и v – значения $Fibonacci(y)$ и $Fibonacci(\sim y)$. Как только значения x и y сравняются, вычисление Fib завершается с результатом, равным текущему значению u, то есть $Fibonacci(x)$.

Конец задания.

О комбинаторике.

Так называется полуприкладной раздел математики, изучающий методы и приемы подсчета числа элементов различных множеств и накапливающий результаты этих подсчетов. Для программиста все это небезразлично, ведь подсчет числа действий, затрачиваемых на выполнение программы, – это из той же оперы. Пример: полученная выше формула для NFib. Достаточно оценить лишь порядок величины, но сознание, что можешь сделать это точно, греет душу.

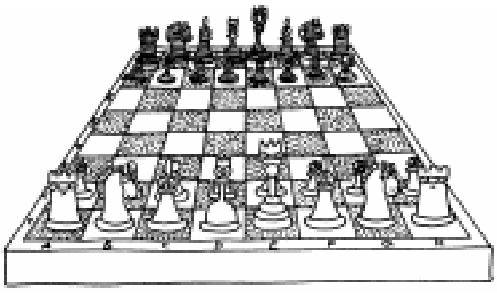
Примеры.

1. *«Запас слов».* Сколько различных n-буквенных слов в m-буквенном алфавите? Примем каждую букву алфавита за цифру m-ичной системы счисления. Тогда каждое слово длины n – это n-значное целое неотрицательное число в этой системе, а их всего m^n: например, 100 при $m = 10$, $n = 2$ – от 00 до 99. Но и прямой подсчет не сложнее. Первая буква слова дает m возможностей для выбора, каждая следующая – m возможностей для продолжения. Перемножая, приходим к тому же результату.

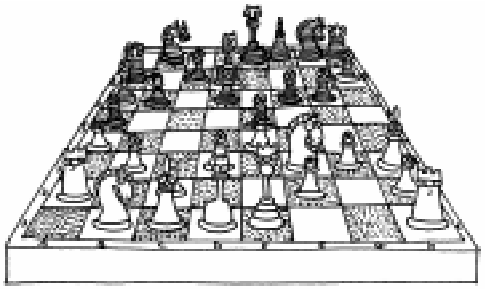
2. *Размещения.* Подсчитаем число $A(n, m)$ способов размещения m из n имеющихся различных предметов в ряд. Здесь число возможностей для продолжения убывает на 1 с каждым шагом: часть предметов уже размещена. Число способов выбрать первый элемент в ряду равно n, после построения ряда для выбора остается $n - m$ возможностей, так что $A(n, m) = n(n-1)...(n-m+1) = n!/(n-m)!$

3. *Перестановки.* Пусть теперь требуется разместить в ряд все n имеющихся предметов. Число $P(n)$ всех таких размещений равно $P(n) = A(n, n) = n!$

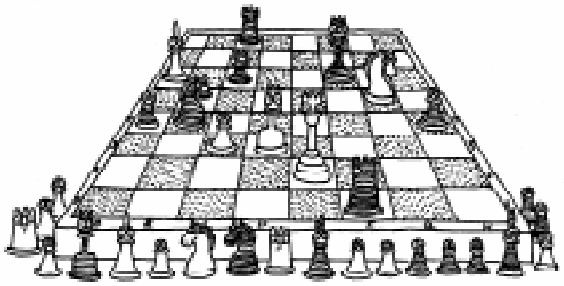
4. *Сочетания.* Число $C(n, m)$ способов выделить m мест в ряду из n позиций – это едва ли не самый известный и распространённый комбинаторный объект. Связь между числами $A(n, m)$, $C(n, m)$ и $P(m)$ дается формулой: $A(n, m) = C(n, m)P(m)$. Действительно, $C(n, m)$ задает число способов



Размещения.



Перестановки.



Сочетания.

выделить, не нарушая их порядка, те m из общего числа n предметов, которые войдут в размещение. Далее, $P(m)$ – число способов изменить этот порядок (при переходе от $C(n, m)$ к $A(n, m)$ важно указать не только позицию, но и объект, который может ее занять). Перемножая, получаем полное число A из n по m возможных размещений. Аналогично говорят: C из n по m и P из n или, как в данной формуле, из m).

Таким образом, $C(n, m) = A(n, m)/P(m) = n!/(m!(n - m)!)$. Частные случаи: $C(n, 0) = C(n, n) = n!/(0!n!) = 1$. Отсюда же следует симметрия формулы для C: $C(n, m) = C(n, n - m)$, позволяющая свести диапазон значений m к $0 \le m \le n/2$. Не следует путать это с симметричностью функции: симметрична функция $C'(m, k) = C(m + k, m)$, для которой $C'(m, k) = C'(k, m)$.

В теоретико-множественной терминологии $P(n)$ – это число способов упорядочивания n-элементного множества, $A(n, m)$ – число упорядоченных m-элементных подмножеств упорядоченного n-элементного множества, а $C(n, m)$ – число его же неупорядоченных m-элементных подмножеств.

К понятию сочетания можно подойти и так. Пусть n и m – натуральные числа, $k = n - m$. Рассмотрим всевозможные пути из начала координат в точку (m, k) целочисленной сетки, составленные из единичных отрезков вдоль одной из осей координат (в ее положительном направлении). Число путей равно $C(n, m) = C(m + k, m)$. Действительно, путь может быть задан последовательностью из m единиц и k нулей, символизирующих направление участка пути. Она же задает выбор m мест из n в ряду. Придти в точку (m, k) можно либо из точки $(m - 1, k)$ (число путей равно $C(m + k - 1, m - 1)$, либо из $(m, k - 1)$ (число путей равно $C(m + k - 1, m)$. Отсюда еще одна формула: $C(n, m) = C(n - 1, m - 1) + C(n - 1, m)$ для числа сочетаний.

Задание 4. Число сочетаний.

Напиите программу вычисления числа $C(n, m)$, воспользовавшись формулами:

a) $C(n, m) = n!/(m!(n - m)!)$,

b) $C(n, 0) = C(n, n) = 1$,

$C(n, m) = C(n - 1, m - 1) + C(n - 1, m)$ – для других значений m.

Выполнение задания 4.

a) function C (n, m: integer): integer;

C (n, 0) = 1;

C (n, m) = C (n, m – 1)*(n – m+1)/m.

Функция C – рекурсивная. Первое правило задает ее начальное значение для $m = 0$, второе – добавляет очередные сомножители в числитель и знаменатель при переходе от $m - 1$ к m.

b) Язык, применявшийся нами для записи программ, недостаточно богат для выполнения этой части задания: значения, используемые для вычисления $C(n, m)$, надо хранить достаточно долго. Пополнив язык, напишем следующий вариант программы.

```
type ArrType = array[0.. ] of integer;
const init: ArrType = (0); const init1: ArrType = (1);
```

Тип ArrType – это массив целых (фактически – натуральных) чисел с переменной верхней границей, ее текущее значение на единицу меньше числа элементов массива. В частности, у массивов *init* и *init*1 верхняя граница совпадает с нижней, то есть равна нулю.

function Combinations(*n*0,*m*0: integer): integer;
if *m*0 < *n*0–*m*0 then Combinations(*n*0,*m*0) = Comb(*n*0,*m*0,0,0,*init*,*init*1)
else Combinations(*n*0,*m*0) = Comb(*n*0,*n*0–*m*0,0,0,*init*,*init*1).

В паре Combinations-Comb функции взаимодействуют примерно так же, как в паре Fibonacci-Fib. Аргументы *n*0 и *m*0 функции Comb хранят исходные значения одноименных аргументов обращения к Combinations.

function Comb(*n*0,*m*0,*n*,*m*: integer, *prev*, *curr*: ArrType): integer;
Comb(*n*0,*m*0,*n*0,*m*0,*prev*,*curr*) = *curr*[*m*0];
Comb(*n*0,*m*0,*n*,*m*0,*prev*,*curr*) = Comb(*n*0,*m*0,*n*+1,0,AddA(*m*0+1,*prev*,*curr*),*init*);
Comb(*n*0,*m*0,*n*,*m*,*prev*,*curr*) = Comb(*n*0,*m*0,*n*,*m*+1,*prev*,AddA(*m*+1,*prev*,*curr*)).

Аргумент *n* рекурсивной функции Comb при последовательных обращениях к ней пробегает значения от 0 до *n*0, а аргумент *m* – от 0 до *m*0 для каждого значения *n*. Аргумент *curr* – это последовательность вычисляемых значений C(*n*, *m*) для текущего значения *n*, а *prev* – это накопленная последовательность тех же значений для предыдущего значения *n*. Функция [ ] (*a*: ArrType, *m*: integer): integer с двучленным (инфиксно-постфиксным) символом [ ] обеспечивает доступ к значению из *m*-й ячейки массива *a*. Первое правило обеспечивает, как обычно, завершение вычисления Comb, второе и третье организуют последовательность рекурсивных обращений.

function AddA(*m*: integer, *prev*,*curr*: ArrType): ArrType;
curr[*m*]:=*prev*[*m*–1]+*prev*[*m*];
AddA(*m*,*prev*,*curr*) = *curr*.

Функция AddA заносит в массив *curr* по индексу *m* значение C(*n*, *m*), вычисляемое по указанной в задании формуле, и возвращает полученный массив в качестве результата. Первое правило предписывает извлечь нужные значения из массива *prev* и занести их сумму в массив *curr*. При этом в массиве появляется новый элемент, то есть верхняя граница индексов возрастает. Второе правило указывает, что значением функции становится преобразованный массив *curr*.

В разд. «Язык Лисп» дадим другое решение этой задачи.

Конец задания.

Замечание (для любителей «Паскаля» и подобных ему языков). Функции AddA и [ ] и обозначение (*a*0,...,*am*) для массива с элементами *a*0,...,*am* – почти всё, что требуется для работы с массивами в функциональном стиле. Не предусмотрено изменение содержимого существующих в массиве ячеек. В отличие от «Паскаля» допускаются массивы с переменной верхней границей по внешнему измерению. Чтобы предписать неизменность границы, требуется явно задать её значение в описании типа массива. Скромность требований к запасу операций над массивами будет принята во внимание в следующем разделе.

Литература.

1. *Павлова М.В.* Рекурсивные алгоритмы и их построение // Компьютерные инструменты в образовании. СПб, 2000. № 1.

2. *Павлова М.В., Паньгина Н.Н.* Примеры и задачи на тему «Рекурсивные алгоритмы и их построение» // Компьютерные инструменты в образовании. СПб, 2000. № 1.

© Наши авторы, 2002.
Our authors, 2002.

Лавров Святослав Сергеевич,
доктор технических наук.
профессор.