

Смешанные вычисления

Узкоспециализированные, но эффективные способы решения задач традиционно противопоставляются универсальным, но менее эффективным методам. Смешанные вычисления, осуществляя совместную обработку программы и ее данных, позволяют систематически получать из универсальной программы ее эффективные специализированные версии, не утрачивая знания, заложенного в общий метод

А. П. ЕРШОВ

ОДНО ИЗ основных применений ЭВМ заключается в решении задач. Говоря о задаче, мы различаем ее условие, способ ее решения и само решение, результат. Условие задачи обычно обладает определенной степенью общности. В формулировку условия входит один или несколько параметров, называемых исходными данными, так что это условие имеет смысл для некоторого множества значений исходных данных.

Общность условия задачи стимулирует исследователя на поиск общего способа ее решения, т.е. таких правил, которые, будучи некоторым единым образом применены к исходным данным, гарантируют нахождение решения. Поиск общих, или универсальных, методов образует одну из главных парадигм современной науки. Действительно, обладание общим методом, позволяющим точно решить любую задачу из некоторого (обычно бесконечного) их разнообразия, является кульминацией исследования, надежным свидетельством разработки полной теории соответствующей предметной области. В то же время в повседневной практике обнаруживается, что универсальные методы применяются реже, чем можно ожидать.

Обратимся к собственному опыту. Все мы со школы знаем общий метод перемножения двух натуральных чисел столбиком. Но автор готов биться об заклад, что этот алгоритм применяется не более чем в одном из десяти случаев. Каждый из нас обладает целым арсеналом специализированных алгоритмов умножения, которыми старается пользоваться с максимальным предпочтением. Вот наиболее известные:

- таблица умножения однозначных чисел,
- таблица квадратов в пределах тысячи,
- для умножения на десять приписать к числу справа нуль,
- для умножения на число из и девя-

ток, приписать и нулей и вычесть множимое,

— для умножения на пять умножить на десять и разделить на два.

Если нам надо перемножить два числа, обладающих на первый взгляд хоть каким-то своеобразием, скажем 808 и 707, мы можем потратить порядочное время в попытках перемножить его каким-нибудь специальным способом, прежде чем применить универсальный алгоритм.

В чем же здесь дело? Могущество общего метода не только в его универсальности, т.е. применимости к любой задаче из некоторого класса. Он также допускает и максимальную однородность исходных данных и позволяет прийти к решению, используя наименшее число закономерностей.

Однако, когда от разработки общего метода мы переходим к его массовому практическому применению, многие достоинства метода начинают восприниматься как недостатки. Общий метод трактует исходные данные, не отдавая одному предпочтения перед другими. На практике же одни варианты задачи возникают чаще, чем другие, а их исходные данные обладают определенным своеобразием.

Налицо два взгляда на специальные способы решения задач. Если для теоретика многообразие специальных способов — это признак неразвитости общей теории, то для практика то же разнообразие свидетельствует о дополнительных возможностях решать каждую частную задачу наиболее подходящим методом.

Эти два взгляда на специальные методы формируются в разных контекстах. Если практик вынужден иметь дело с ассортиментом специальных способов при отсутствии общего метода, он постоянно рискует столкнуться с такими частными задачами, для которых у него либо нет метода решения, либо — еще хуже — он не знает, решает ли эта задача или не решается доступ-

ными специальными методами. Картина меняется радикально, когда специальные методы применяются на фоне общего метода, который для практика становится стратегическим тылом по отношению к передовой линии повседневной работы со специальными методами.

ЦЕЛЬ ЭТОЙ статьи — показать, что между общим и специальными методами существует еще более глубокая и дружественная связь. Обычно считается, что специальный метод появляется в результате догадки и не только противостоит общему методу, но и игнорирует его. Автор хочет показать, что специальные методы могут получаться из общего метода систематически, с помощью универсальной процедуры весьма общего характера. Нахождение этой процедуры, названной смешанными вычислениями, стало одним из наиболее интересных достижений современной математической теории вычислений. Оно интересно с познавательной точки зрения, так как проливает свет на глубокие закономерности взаимодействия машинных программ с их данными и на связь программирования с теорией вычислимости. Оно интересно с практической точки зрения, так как позволяет получать большое разнообразие эффективных специальных методов, не порывая со знанием, воплощенным в общей теории. Наконец, оно интересно тем, что раскрывает сущность и подводит теоретический базис под многие методы программирования, возникшие ранее и развивающиеся в виде специальных и не связанных друг с другом приемов.

Если говорить о решении задач на ЭВМ, то способ решения задачи — это программа, которая выполняется на ЭВМ, условие задачи — входные данные для программы, а решение — выходные данные, выдаваемые программой. Поскольку ЭВМ — это детерминированный автомат, то, при фиксированной программе, для любого вход-

ногданного x выходное данное z будет некоторой функцией φ , полностью определяемой программой $p : z = \varphi(x)$. Мы скажем, что функция φ реализуется программой p . Например, если задача состоит в том, чтобы уметь разлагать любое натуральное число на простые сомножители, то входным данным будет само число, допустим 819, способом решения — программа какого-нибудь алгоритма разложения числа на простые сомножители, а выходным данным — список этих сомножителей, в данном случае 3,3,7,13.

Сама программа p вместе с ее данными x выполняется посредством алгоритма Int (от слова «интерпретация»), встроенного в конструкцию ЭВМ. Этот универсальный алгоритм, будучи применен единообразным способом к любой программе и ее входным данным, реализует функцию, реализуемую программой: $Int(p; x) = \varphi(x) = z$.

Рассмотрим в этих обозначениях понятие частной задачи. Если частная задача определяется просто одним экземпляром значения x и никакие другие условия нас не интересуют, то специальный метод решения задачи находится trivialно: надо применить программу p к x , получить решение и запомнить его. Всякий раз, когда нужно решить задачу с условием x , например найти простые сомножители числа 819, надо просто взять ответ (3,3,7,13) и использовать его. Такой метод готовых решений широко применялся в виде разного рода математических таблиц во времена «докомпьютерной математики» и сохраняет свое значение и сейчас, когда таблица невелика.

В большинстве случаев, однако, частная задача носит не единичный характер, а сама обладает некоторым разнообразием входных данных. На языке функциональной зависимости это выглядит следующим образом. Предположим, что решение задачи сводится к вычислению функции двух переменных $\varphi(x,y)$. Множество входных данных образует некоторую область в плоскости координат (x,y) . Отдельные точки в этой области соответствуют единичным частным задачам. Универсальный метод решения воспринимает любую точку области исходных данных, не делая никаких различий между ними. Иначе говоря, исходные данные являются независимыми переменными. В общем случае частная задача возникает тогда, когда мы обладаем какой-то дополнительной априорной информацией об исходных данных. Например, может оказаться, что на вход программы разложения числа на множители попадают только простые числа. Тогда и способ решения задачи становится простым: само число и есть свой единственный множитель. Говоря другими словами,

СМЕШАННЫЕ ВЫЧИСЛЕНИЯ	$x = a, y = b$	$Int(p; a, b) = \varphi(a, b)$
	$x = a, y = b;$ свободных переменных нет	$Mix(p; a, b) = out(c),$ где $c = \varphi(a, b)$
	x, y — свободные, связанных переменных нет	$Mix(p; \emptyset; x, y) = p$ (\emptyset — символ пустоты)
	$x = a, y$ — свободная	$Mix(p; a; y) = p_a,$ где $Int(p_a, b) = \varphi(a, b)$
Int — процедура обычных вычислений Mix — процедура смешанных вычислений p — конкретная программа x, y — входные переменные программы $\varphi(x, y)$ — функция, реализуемая программой		

ОБЫЧНЫЕ И СМЕШАННЫЕ ВЫЧИСЛЕНИЯ связаны друг с другом. *Обычные вычисления* — это универсальная процедура Int , которая для любой программы p (первый аргумент) и заданных значений всех ее входных переменных (вторая группа аргументов, отделенная точкой с запятой) вычисляет значение функции φ , реализуемой программой. *Смешанные вычисления* — это универсальная процедура Mix , которая для любой программы (первый аргумент), заданных значений ее связанных входных переменных (вторая группа аргументов, отделенная точками с запятой) и указанных свободных входных переменных (третья группа аргументов) вычисляет другую, так называемую остаточную программу. Если все переменные связаны входными данными, остаточная программа вырождается в команду выдачи предвычисленного результата исходной программы. Если все входные переменные свободны, исходная программа целиком переходит в остаточную программу. Если часть входных переменных связана, а другая оставлена свободной, то остаточная программа p_a после задания недостающих данных вычисляет тот же результат, что и исходная программа для тех же данных.

частная задача возникает при сужении области исходных данных на некоторое ее подмножество. Максимальной информацией, соответствующей сужению области до одной точки, мы располагаем, когда объект задан конкретно. Например, если натуральное число дано, то мы знаем о нем все: простое оно или нет, оканчивается нулем или нет, равно, скажем, пяти или нет и т.д.

Важным частным случаем, с которым мы будем в основном иметь дело в этой статье, являются сужения-проекции, когда одно независимое переменное задано, или, как говорят, связано некоторым значением, а другое переменное остается полностью неизвестным, свободным. Напомним, что общая задача состоит в вычислении функции $\varphi(x,y)$. Пусть входной переменной x задано значение a , а переменная y не зафиксирована. Тогда частная задача состоит в вычислении функции $\varphi(a,y)$, т.е. такой, в которой часть аргументов связана, а другая оставлена свободной.

В этих обозначениях построение специализированного метода решения частной задачи состоит в нахождении такой программы π , которая бы реализовывала функцию $\varphi(a,y)$. Иначе говоря, если выполнить программу π для некоторого данного $y = b$ на ЭВМ с

универсальным алгоритмом выполнения $Int(\pi; b) = \varphi(a, b)$.

СМЕШАННЫЕ вычисления предлагают универсальную процедуру нахождения программы π как функции исходной общей программы, значений ее связанных входных переменных и указанных свободных переменных. Результат смешанных вычислений называется остаточной программой. Если остаточную программу выполнить, задав значения оставшимся свободным переменным, она выдаст тот же результат, что и исходная программа. Если связать заданными значениями все входные переменные, то остаточная программа выродится в команду выдачи предвычисленного результата. Если, наоборот, все входные переменные оставить свободными, то остаточная программа совпадет с исходной.

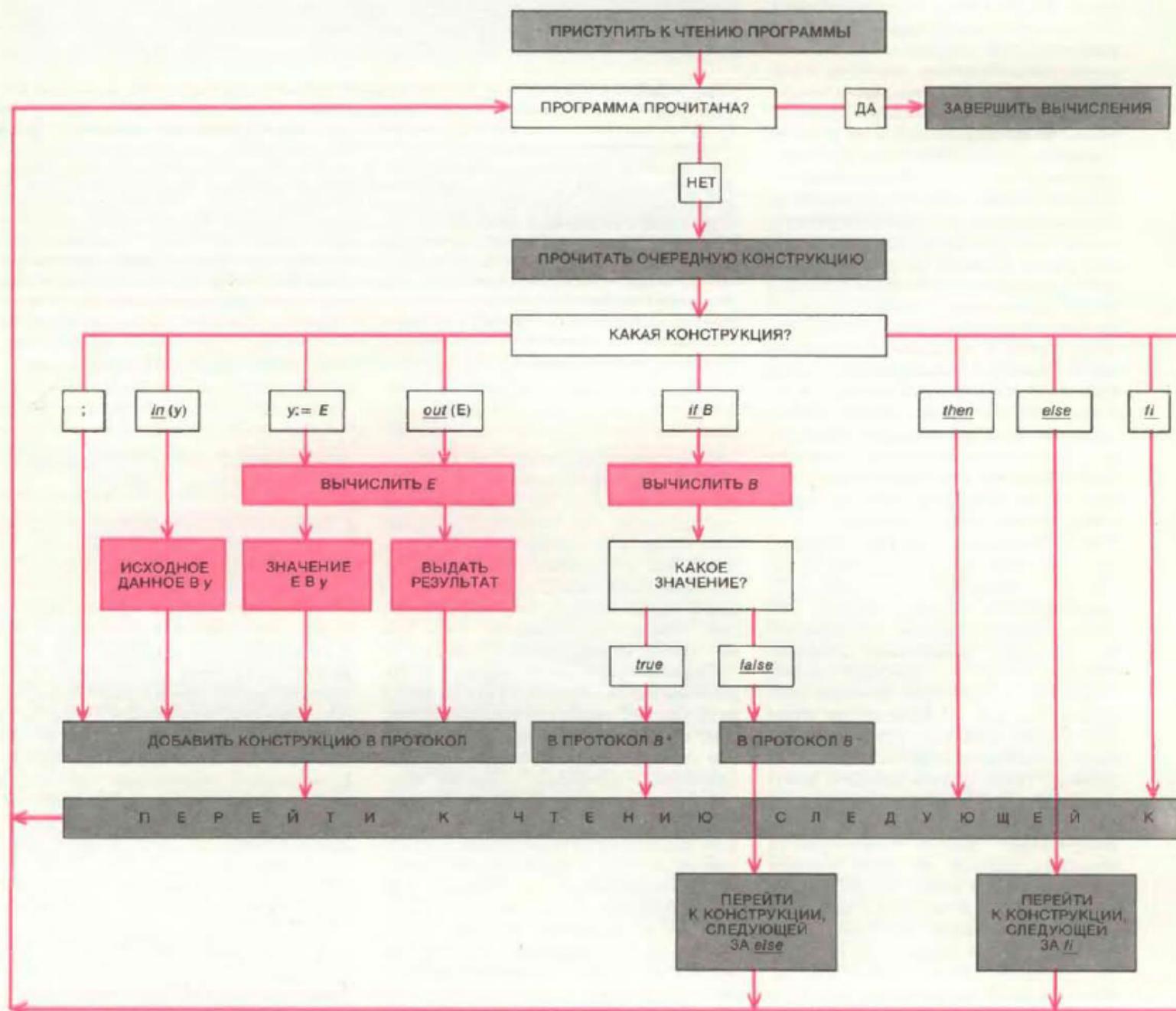
Сказанное выше можно рассматривать как функциональное определение смешанных вычислений. Кроме этого, нужно учитывать и некоторые прагматические соображения, которые должны оправдывать применение такой универсальной процедуры. Для начала уместно объяснить, почему среди других способов сужения области исходных данных проекция заслуживает особого рассмотрения. Дело в том, что ис-

ходные данные редко входят в условие задачи симметрично, как, например, переменные x и y в формуле длины вектора на плоскости: $d = \sqrt{x^2 + y^2}$. Чаще бывает так, что одна независимая переменная выступает в качестве параметра, а другая является «истинным» аргументом. Например, в степенной функции $y = x^n$ принято считать вещественное x аргументом функции, а целое n — ее параметром, в том смысле, что степенную функцию, как правило, надо вычислять, когда на одно значение параметра n приходятся

разнообразные значения аргумента x . В данном случае, если бы имелся специальный способ вычисления x в заданной степени n , то его применение вместо общего метода позволяло бы решать задачу быстрее. Например, x^{16} можно вычислить, сделав только четыре умножения вместо пятнадцати последовательных умножений на x , а именно $x^{16} = (((x^2)^2)^2)^2$. Очевидно, что фиксация параметра и варьирование остальных аргументов есть проекция области исходных данных на значение параметра.

ДЛЯ ТОГО чтобы понять, как можно систематически получать из программы общего метода эффективные программы специальных методов, нужно более подробно узнать, как устроены и выполняются машинные программы.

Все способы записи и выполнения программ опираются на некоторые общие предпосылки. Обрабатываемая информация представляет собой совокупность элементарных данных (чисел или символов). Для элементарных данных существуют их буквальные об-



ПРАВИЛА ОБЫЧНЫХ ВЫЧИСЛЕНИЙ в языке МИЛАН представлены в виде его операционной семантики, т.е. универсального алгоритма выполнения любой программы в этом языке. Вычисление выглядит как чтение программного текста и извлечение из него последовательности базовых команд (протокола), при выполнении которых происходит

непосредственная обработка информации и проверка ее свойств. Особенностью выполнения программы является то, что она сама направляет порядок своего чтения. Если при этом игнорировать шаги выполнения базовых команд (цветные блоки), а вместо проверки значений условий выбирать *true* или *false* произвольно, то операционная семанти-

значения, называемые константами. Кроме этого в программе могут фигурировать переменные, значениями которых являются элементарные данные. Эти значения задаются переменным в виде исходных данных или могут быть присвоены этим переменным во время вычислений. Значение переменной остается неизменным до тех пор, пока ей не будет присвоено новое значение. Любая обработка информации сводится к последовательности выполнения небольшого количества базовых операций над элементарными

данными, в частности сложения или умножения двух чисел. Не меньшее значение имеют базовые операции проверки некоторых свойств данных, например, равны ли элементарные данные, положительно ли число, четно оно или нет и т.д. Операции, проверяющие свойства данных, называются предикатами — функциями, определенными над данными и принимающими два значения, обычно называемые *истина* и *ложь* (*true* и *false*), одно из которых соответствует наличию свойства, а другое — его отсутствию.

Программа представляет собой текст, имеющий фразовую структуру и записанный на некотором языке программирования. В этот текст в качестве слов входят константы, переменные, имена базовых операций и предикатов, названия разных частей программы или сгруппированных данных, а также служебные слова и «знаки препинания», которые позволяют делить программный текст на осмысленные части. Описание языка программирования состоит из грамматики (правил записи программ) и операционной семантики, которая интерпретирует программный текст, задавая правила выполнения программы.

Опишем грамматику простого императивного языка программирования МИЛАН (MiLi LANguage), обычно используемого в демонстрационных целях. Его фразами являются команды выполнить то или иное действие. Основной командой является присваивание вида $x := E$, где x — переменная, а E — выражение, образованное базовыми операциями, константами и переменными. Эта команда предписывает вычислить значение выражения E при текущих значениях входящих в него переменных и полученное значение сделать значением переменной x . Для задания переменным начальных значений — из исходных данных программы — используется команда ввода $in(x)$, где in — служебное слово, а x — переменная, принимающая значение исходного данного. Для того чтобы прочитать результат выполнения программы, используется команда вывода $out(E)$, где E — выражение. Значение выражения при текущих значениях переменных выдается в качестве результата.

Последовательность команд, разделенных точками с запятой, образует серию, в которой команды выполняются в порядке написания.

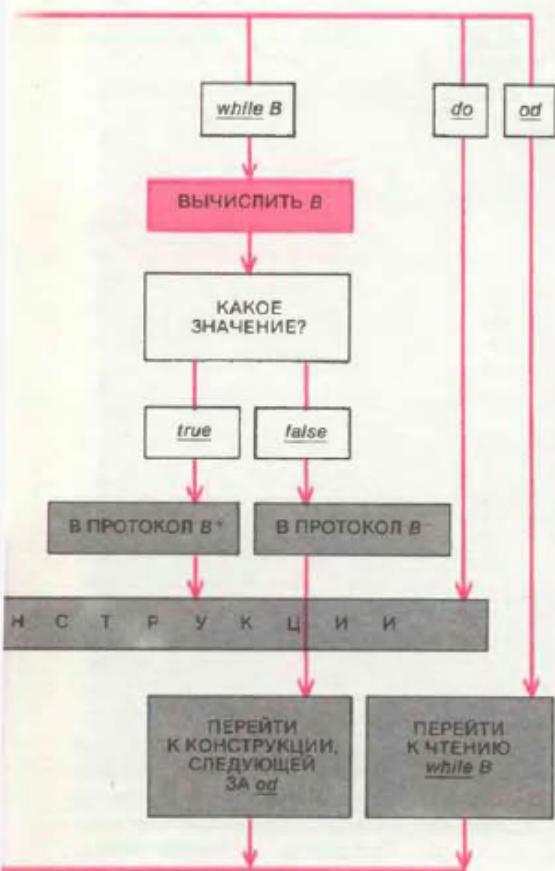
Программы, однако, не могли бы делать много, если бы состояли только из серий базовых команд. Примером составной команды является альтернатива, имеющая вид $if B then S1 else S2 fi$, где выражение B — условие, образованное комбинацией базовых предикатов. Альтернатива является сокращенной записью следующего правила: «ес-

ли свойство данных, выраженное условием B , имеет место, то выполнить серию команд $S1$, в противном случае выполнить серию команд $S2$ ». Служебное слово fi — это перевернутое if , вместе они символизируют пару скобок, ограничивающих альтернативу.

Наиболее мощной составной командой является итерация, имеющая вид $while B do S od$. Это сокращенная запись следующего правила: «Проверить соблюдение свойства данных, выраженного условием B . Если оно имеет место, выполнить серию S , после чего снова приступить к проверке B и т.д. При первом же (включая и самую первую проверку) невыполнении условия B завершить выполнение итерации». Таким образом серия S при выполнении итерации либо не выполнится ни разу, либо выполнится конечное число раз, либо будет повторяться бесконечно. Второй случай возможен только тогда, когда B зависит от переменных, значения которых изменяются при выполнении серии S . Именно итерации позволяют с помощью сравнительно коротких программных текстов диктовать машине выполнение огромных по длине последовательностей команд, поспевая тем самым за ее электронной скоростью. Принципиальной трудностью в употреблении итераций является отсутствие общего метода установления того, будет ли данная итерация при данных значениях переменных выполняться конечное число раз. Выяснение этого вопроса требует каждый раз отдельного исследования.

ТРАДИЦИОННЫЙ способ описания работы программ заключается в систематическом чтении программного текста и извлечении из него последовательности базовых команд проверки свойств данных (условий), присваиваний, ввода и вывода, которые выполняют фактическую обработку информации. Эту последовательность базовых команд называют протоколом, или историей вычислений. Особенностью выполнения программы является то, что она сама направляет порядок своего чтения. Этот порядок предписывается отчасти текстом программы (текстуальный порядок выполнения команд в серии), отчасти значениями перерабатываемых данных (зависимость выбора команд от значений условий в альтернативе и итерации). Обычно ЭВМ, снабженная программой, ведет себя как полностью детерминированное устройство. Это значит, что исходные данные и сама программа полностью и единственным образом определяют соответствующий протокол вычислений и благодаря этому обеспечивают единственность результата.

Протокол по своей записи сам является выполнимой программой. Он имеет вид последовательности команд



ка сопоставит каждой программе некоторое множество протоколов, называемое детерминантом программы. Если программа содержит хотя бы одну итерацию, ее детерминант будет бесконечным.

ПЕРЕМЕННЫЕ: x, y, n

$\underline{in}(x); \underline{in}(n); y := 1; \text{while } n > 0 \text{ do while even}(n) \text{ do } n := n/2; x = x^2 \underline{od}; n := n - 1; y := y \times x \underline{od}; \underline{out}(y)$

ПРОГРАММА ВЫЧИСЛЕНИЯ $y = x^n$ на языке МИЛАН использует то свойство, что для вычисления, например, x^{16} достаточно сделать только четыре умножения вместо пятнадцати: $x^{16} = ((x^2)^2)^2$. Программа организует следующий процесс над переменной n . Проверяется четность положительного n . Если оно нечетно, из него вычитается 1, после

чего оно становится четным. Если n четно, оно делится пополам, после чего проверка повторяется и все завершается, когда n станет равным нулю. С каждым вычитанием единицы из n связывается умножение y (первоначально равного 1) на текущее значение x . При редукции n к нулю y становится равным x в степени исходного значения n .

присваивания, ввода и вывода. Правда, если трактовать протокол как программу, остается не вполне ясной роль выражений условий вместе с приписанными им значениями («+» или «-»). Их наличие, однако, не мешает выполнять протокол как программу. Если протокол вычисления $y = x^n$, построенный для входных данных, скажем $x = 10$ и $n = 2$, выполнить для этих же данных, то в результате мы получим

$y = 10^2 = 100$, а значения всех предикатов, вычисленных в момент их чтения для соответствующих значений переменных, будут совпадать с приписанными им значениями, указанными в протоколе.

Это наблюдение тривиально. Рассмотрим теперь эффект применения этого протокола к каким-либо данным, отличным от тех, для которых он был построен. Если, например,proto-

кол для $x = 10$ и $n = 2$ применить к входным данным, скажем $x = 20$ и $n = 2$, то мы опять получим правильный результат $y = 20^2 = 400$, а значения, приписанные предикатам-условиям, будут соответствовать значениям, вычисляемым при выполнении протокола. Попробуем теперь протокол для $x = 10$ и $n = 2$ применить к $x = 10$ и $n = 3$. Сначала все будет в порядке, но при попытке вычислить предикат

ПРОГРАММА:

$\underline{in}(x); \underline{in}(n); y := 1; \text{while } n > 0 \text{ do while even}(n) \text{ do } n := n/2; x := x^2 \underline{od}; n := n - 1; y := y \times x \underline{od}; \underline{out}(y)$

ПРОТОКОЛЫ:

$x = 10, n = 0$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^-; \underline{out}(y)$

$x = 10, n = 1$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^-; \underline{out}(y)$

$x = 10, n = 2$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^+ n := n/2; x := x^2 \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^-; \underline{out}(y)$

$x = 20, n = 2$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^+ n := n/2; x := x^2 \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^-; \underline{out}(y)$

$x = 10, n = 3$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^+ \underline{even}(n)^+ n := n/2; x := x^2 \underline{even}(n)^-; n := n - 1;$

$y := y \times x \ n > 0^-; \underline{out}(y)$

ВЫЧИСЛЕНИЕ ПО ПРОТОКОЛУ, СООТВЕТСТВУЮЩЕМУ $x = 10, n = 2$:

для $x = 10, n = 2$

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^+ n := n/2; x := x^2 \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^-; \underline{out}(y)$

x	10	10	10	10	100	100	100	100
n	—	2	2	2 > 0 ⁺ $\underline{even}(2)^+$	1	1 $\underline{even}(1)^-$	0	0 $0 > 0^-$
y	—	—	1	1	1	1	100	100

для $x = 10, n = 3$

x	10	10	10	10	100	100	100	100
n	—	3	3	3 > 0 ⁺ $\underline{even}(3)^+$	1	1 $\underline{even}(1)^-$	0	0 $0 > 0^-$
y	—	—	1	1	1	1	100	100

ПРОТОКОЛ, ПРОТИВОРЕЧАЩИЙ ЛЮБОМУ n :

$\underline{in}(x); \underline{in}(n); y := 1; n > 0^+ \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^+ \underline{even}(n)^-; n := n - 1; y := y \times x \ n > 0^-; \underline{out}(y)$

ПРОТОКОЛ имеет вид последовательности команд присваивания, ввода, вывода и предикатных выражений с приписанными им значениями. Каждому набору исходных данных программы соответствует один протокол — тот самый, который возникает при выполнении программы для этих данных. Протокол сам имеет вид выполнимой программы и может применяться к некоторому разнообразию входных данных. При этом вычисляемые значения предикатных выражений сравниваются со значениями (“+” для $true$, “-” для $false$), приписанными им при построении протокола. Ес-

ли эти значения не совпадают, то протокол противоречит взятым исходным данным. Протокол, порожденный какими-то исходными данными, противоречит любым данным, порождающим некоторый другой протокол. В детерминанте программы могут существовать протоколы, которые противоречат любому набору исходных данных. Таким, в частности, является самый нижний протокол, поскольку любое положительное натуральное число при вычитании из него единицы обязательно изменяет свою четность.

even(n)⁺ (означающий, что *n* должно быть четным) для *n* = 3 мы обнаружим несовпадение приписанного и вычисленного значений предиката. Это неудивительно, поскольку мы знаем, откуда взялся протокол. Если выполнить программу возвведения в степень для *x* = 10 и *n* = 3, проверка условия четности *even(n)* даст значение ложь и тогда мы будем обязаны поместить в протокол условие в виде *even(n)*⁻ и перейти не к выполнению внутренней итерации, а к вычитанию 1 из *n*. Таким образом, становится понятной роль предикатных выражений вместе с приписанными им значениями в протоколе. Они являются стражами, отличающими свои данные, т.е. те, к которым для получения результата можно применить построенный единичный протокол, от чужих данных, к которым должны применяться их протоколы. Если программа вычисляет однозначную функцию, то для каждой комбинации входных данных существует в точности один свой протокол. Однако один протокол может обслуживать несколько входных данных или даже их бесконечное множество (как в случае *xⁿ*, который отвечает бесконечному разнообразию значений *x* при фиксированном *n*).

С логической точки зрения можно рассматривать выполнение программ как двухэтапный процесс декомпозиции, т.е. извлечения базовых команд, образующих протокол, и прямых вычислений, т.е. выполнения команд протокола.

Таким образом, протоколы (истории вычислений) являются конструкциями, вскрывающими механизм работы программы. Именно последовательность команд протокола определяет результат и объем работы, затраченной на его получение. Когда программисты говорят, что две программы работают одинаково, то обычно имеют в виду, что эти две программы, как бы они ни отличались, обладают одним и тем же множеством протоколов. Так постепенно среди теоретиков программирования сложилось представление, что множество протоколов лучше характеризует программу, нежели сам исходный программный текст. Этот взгляд станет еще более справедливым, если удостовериться, что, хотя каждый протокол может быть вычислен только для своих исходных данных, получение всего множества протоколов для данной программы не требует никаких вычислений с данными и может быть достигнуто простым комбинаторным процессом чтения программы.

БУДЕМ теперь читать текст программы в порядке, предписываемом алгоритмом, и печатать команды протокола, пропуская шаги выполнения команд. Попадая на шаг проверки условия, вместо проверки значения (кото-

рое мы не имеем) будем делать произвольный выбор дальнейшего продвижения по программе. Дойдя до конца программы, мы получим некоторый протокол. Процесс может оказаться бесконечным, что говорит о наличии бесконечных протоколов. Перебирая все возможные варианты произвольного выбора на шагах проверки условий, мы получим и все возможные протоколы. Это множество протоколов называется детерминантом программы. В отличие от отдельных конечных протоколов и текста программы детерминант является абстрактной мысленной конструкцией, как и большинство других бесконечных математических объектов. Однако, чтобы доказать, что детерминант, пусть даже и в теоретическом анализе, заменяет программный текст, следует показать, как можно вычислять по детерминанту. Представим, например, такую схему. Исходные данные подаются на вход всех протоколов детерминанта. Как только при вычислении по какому-то протоколу возникает противоречие, т.е. значение предиката отличается от ему предписанного, так этот протокол выбывает из игры. Здесь возможны четыре исхода: все протоколы оказались противоречивыми — результат не определен, непротиворечивые протоколы оказались бесконечными — результат не определен, несколько конечных протоколов не противоречивы — функция многозначна, нашелся один-единственный непротиворечивый протокол — он и выдает результат вычислений.

Теперь мы в состоянии объяснить природу смешанных вычислений. Примем за основу выполнение программы в два этапа — декомпозицию, формирующую протокол, и прямые вычисления по протоколу. Протокол — это линейная программа и единожды выполненная команда больше не потребуется. Будем сопровождать выполнение протокола его редукцией: выполненная команда или вычисленное условие из протокола убираются, а вычисленные значения переменных подставляются туда, где они используются. Сделаем исключение только для команды выдачи результата, вычислив, однако, стоящее в нем выражение. Тогда при вычислении по протоколу программы возвведения в степень для *x* = 10, *n* = 2 весь протокол редуцируется к команде *out(100)*. Это и есть готовое решение для единичной задачи: *y* = 10². Попытка применить к входным данным *x* = 10, *n* = 2 какой бы то ни было другой протокол в случае нашей программы приведет к противоречию.

Обратимся теперь к опыту школьной алгебры. Нам хорошо понятна задача: вычислить значение $(b + \sqrt{b^2 - 4ac})/2a$ при *a* = 3, *b* = 8 и *c* = 4. Подставив указанные числа в формулу, мы получим выражение

$(8 + \sqrt{8^2 - 4 \cdot 3 \cdot 4})/2$, которое можно подвернуть редукциям, состоящим в замене каждого терма, образованного операцией и ее заданными аргументами, константой, обозначающей результат. (Например 8² заменяется на 64.). Такая полная редукция арифметического выражения соответствует полной редукции протокола при его применении к своим данным.

Так же хорошо нам понятна и другая задача: упростить выражение $(b + \sqrt{b^2 - 4ac})/2a$ для случая *a* = 1 и *b* = 2*d*. Подставив указанные числа в формулу, мы получим выражение $(2d + \sqrt{4d^2 - 4c})/2$, которое на этот раз уже не полностью редуцируется, а через цепочку частичных редукций и простых алгебраических преобразований приводится к более простому виду *d + √d² - c*. В контексте нашей статьи можно сказать, что мы получили эффективную специализацию общей формулы на сужение входных данных, а именно для любого *c*, четного *b* и *a* = 1.

ПОПРОБУЕМ мысленно обобщить эту ситуацию на случай программ. Вместо одной формулы мы имеем детерминант программы, состоящий из бесконечного множества протоколов. Определим для каждого протокола процесс частичной редукции (частичной из-за того, что значения некоторых переменных не заданы и поэтому вычисления с ними невозможны). Сначала устанавливаем, какие аргументы даны, а какие свободны. Затем читаем протокол и пытаемся выполнять его команды. Те команды, которые можно выполнить, из протокола убираются, а те, что выполнить нельзя, задерживаются в протоколе, чтение которого продолжается, пока не дойдет до конца. При чтении команды *in(z)* выполняем команду, если входное данное доступно, и делаем значение *z* равным этому данному. Если входное данное отсутствует, *z* признается задержанным, т.е. недоступным, и команда пропускается. При чтении присваивания *z := E* производится упрощение выражения *E* за счет возможной доступности его аргументов. Если *E* редуцировалось до константы (т.е. вычислилось), его значение присваивается *z*, переменная объявляется доступной и команда удаляется из протокола. Если же *E* редуцировалось неполностью, то *z* объявляется недоступной и команда пропускается. Команда *out(E)*, где *E* по возможности редуцировано, остается в протоколе всегда.

Предикатные выражения подвергаются такому же упрощению. Если они редуцируются неполностью, то остаются в протоколе с предписанными им значениями («+» или «-»). Если же условие вычислено, то вычисленный результат сравнивается с предписан-

РЕДУКЦИЯ ФОРМУЛЫ $(b + \sqrt{b^2 - 4ac})/2a$ для $a = 3$, $b = 8$, $c = 4$:

$(8 + \sqrt{8^2 - 4 \cdot 3 \cdot 4})/2 \cdot 3$
 $(8 + \sqrt{64 - 4 \cdot 12})/36$
 $(8 + \sqrt{64 - 48})/6$
 $(8 + \sqrt{16})/6$
 $(8 + 4)/6$
 $12/6$
 2

РЕДУКЦИЯ ПРОТОКОЛА ВОЗВЕДЕНИЯ x В СТЕПЕНЬ n для $x = 10$, $n = 2$:

```

in(x); ln(n); y := 1; n > 0+ even(n)+ n := n/2; x := x2 even(n)-; n := n - 1; y := y × x    n > 0-; out(y)
ln(n); y := 1; n > 0+ even(n)+ n := n/2; x := 102 even(n)-; n := n - 1; y := y × x    n > 0-; out(y)
y := 1; 2 > 0+ even(2)+ n := 2/2; x := 102 even(n)-; n := n - 1; y := y × x    n > 0-; out(y)
n := 1; x := 100 even(n)-; n := n - 1; y := 1 × x    n > 0-; out(y)
x := 100 even(1)-; n := 1 - 1; y := 1 × x    n > 0-; out(y)
n := 0; y := 1 × 100 n > 0-; out(y)
y := 100 0 > 0-; out(y)
out(100)

```

ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКОГО ВЫРАЖЕНИЯ (аверху) сводится к замене символов переменных их значениями, а затем к выполнению цепочки редукций, которые состоят в замене операции и ее аргументов на результат применения операции к этим аргументам. Вычисление завершается, когда формула редуцируется к одной константе — значению выражения. Поскольку в протоколе каждая команда выпол-

няется однократно, вычислению по нему можно придать аналогичный характер (внизу). Выполняемая команда или вычисляемое условие устраняются из протокола, а вычисленные значения переменных подставляются туда, где они используются. Результат вычисления читается в аргументе команды выдачи out(y).

ным. При совпадении значений предикатное выражение удаляется из протокола. Несовпадение означает противоречие, и в этом случае весь протокол бракуется и удаляется из детерминанта.

Представим себе, что такая процедура проделана с каждым протоколом детерминанта программы. В результате часть протоколов исчезла (из-за противоречий), а другие упростились. При редукции детерминанта были проделаны все вычисления, ставшие возможными благодаря доступным данным, и отброшены все варианты вычислений, противоречие этим данным. Легко сообразить, что по существу мы максимально использовали дополнительную информацию, возникшую благодаря сужению области входных данных. Если бы мы могли построить программный текст, для которого редуцированный детерминант был бы ее нормальным детерминантом, мы бы сочли такую программу удовлетворительной версией специализированного метода решения частной задачи.

На пути к конструктивной реализации только что описанного общего метода лежит, однако, одно принципиальное препятствие. Каждой программе соответствует некоторое множество протоколов (ее детерминант), но не каждое множество протоколов может быть детерминантом какой-либо программы. Характеристическим свойством детерминантов является их принадлежность специальному классу, так называемых автоматных мно-

жеств. Описанная выше операция редукции в общем случае неконструктивна и выводит получающееся множество протоколов за пределы этого класса. Поэтому, вооружившись общей идеей метода, мы должны организовать процесс частичных редукций так, чтобы в любой момент находиться в пределах некоторого программного текста, который по завершению процесса будет объявлен результатом специализации универсальной программы.

Опишем такой процесс в виде обобщения схемы выполнения программ на языке МИЛАН в двух взаимосвязанных направлениях. Во-первых, определим процесс выполнения программы для неполностью заданных значений переменных — так называемые частичные вычисления. Во-вторых, вместо печати выполнимых команд будем печатать те команды, которые нельзя выполнить из-за отсутствия информации. В отличие от протоколов это будут не только нередуцируемые присваивания, но и нередуцируемые альтернативы и нераскрытие итерации. Эти невыполнимые команды будут печататься в такой логической последовательности, чтобы образовывать в совокупности осмысленную остаточную программу. Сам процесс выполнения частичных вычислений и генерации остаточной программы потому и получил название смешанных вычислений, что при его выполнении шаги обычных вычислений перемежаются шагами формирования остаточной программы. Детерминант остаточной программы —

это некоторое непротиворечивое расширение описанной выше редукции детерминанта исходной программы.

Изложенный алгоритм смешанных вычислений в языке МИЛАН соответствует традиционному способу описания операционной семантики языков программирования. Выполнение программы по этому способу напоминает работу ЭВМ, управляемой программой на машинном языке. Обладая сходной организацией, алгоритмы обычных и смешанных вычислений существенно отличаются друг от друга. Получается так, что язык имеет две операционные семантики: одну для обычных вычислений, а другую — для смешанных. Существует, однако, совершенно иной подход к описанию вычислений, который, хотя и восходит к математическим работам 30-х годов, в литературу по программированию вошел совсем недавно. Достоинством этого так называемого трансформационного подхода является то, что в нем обычные и смешанные вычисления выглядят совершенно одинаково.

ИДЕЯ трансформационного подхода состоит в следующем. Мы просматриваем программу и ищем команды, которые могут быть в данный момент выполнены. Выполненную команду редуцируем — убираем из программы. Однако, благодаря итерациям в тексте программы есть команды, которые выполняются многократно и поэтому не могут быть редуцированы после первого выполнения. Выход со-

стоит в том, что в число правил преобразования программного текста кроме редукций включаются так называемые раскрытия, которые «отщепляют» от итерации один ее экземпляр и приносят его в жертву очередной редукции, оставляя саму итерацию в качестве источника последующих экземпляров.

Трансформационная семантика языка МИЛАН состоит из списка правил преобразования текста программы, включая значения переменных. Для каждой конструкции языка есть одно относящееся к ней правило. Правило итерации является раскрытием и составлено так, что раскрывать можно только ту итерацию, которая в программе должна обязательно выполниться и при этом раньше других итераций.

Все остальные правила сводятся к редукциям, их применение приводит к устраниению данной конструкции или ее замене более простой конструкцией. Вхождение переменной величины в виде аргумента выражения заменяется на константу только тогда, когда есть уверенность, что должно быть использовано именно то значение переменной, которое она имеет в момент выполнения редукции. Присваивание ве-

личине x редуцируется только тогда, когда есть уверенность, что старое значение x , если оно существует, уже не может быть использовано, а само присваивание обязательно должно выполниться. Редукция команды ввода показывает, что ее выполнение эквивалентно присваиванию переменной соответствующего значения исходного данного.

ОРГАНИЗАЦИЯ вычислений в языке программирования, заданном трансформационной семантикой, имеет ряд существенных отличий от операционной семантики. Главное из них — это изменение модальности предписаний о выполнении команд. Если операционная семантика в каждый момент чтения программы указывает на единственную команду, которая должна выполниться, то трансформационная семантика в каждый момент работы с программой показывает, какие команды могут «выполниться», т.е. раскрыться или редуцироваться. При этом вполне возможно, что в данный момент в программе существует несколько конструкций, к которым можно применить соответствующие преобразова-

ния. Семантика языка не подсказывает, какое из них должно быть выбрано, т.е. допускает в ходе выполнения программы акты произвольного выбора. Среди всех возможных состояний вычислительного процесса естественно выделяется такое, когда ни одно из преобразований не может быть применено ни к одной из оставшихся конструкций. Такое состояние программы и ее данных, т.е. текущих значений переменных, называется неподвижной точкой. Трансформационная семантика определена так, что если все исходные данные в программе заданы и программа для этих данных останавливается и выдает некоторый результат, скажем 78, в качестве значения команды вывода, то неподвижной точкой для этих же исходных данных будет программа, которая редуцируется к одной-единственной команде out(78). Если же часть входных данных окажется не заданной и соответствующие переменные останутся свободными, то неподвижной точкой будет некоторая остаточная программа, та же самая, которая получилась бы в результате смешанных вычислений согласно их операционному определению.

ЧАСТИЧНАЯ РЕДУКЦИЯ ФОРМУЛЫ $(b + \sqrt{b^2 - 4ac})/2a$ для $a = 1, b = 2d, c$ и d СВОБОДНЫХ:

$$\begin{aligned} & (2d + \sqrt{(2d)^2 - 4 \cdot 1 \cdot c})/2 \cdot 1 \\ & (2d + \sqrt{4d^2 - 4c})/2 \\ & (2d + \sqrt{4(d^2 - c)})/2 \\ & (2d + 2\sqrt{d^2 - c})/2 \\ & 2(d + \sqrt{d^2 - c})/2 \\ & d + \sqrt{d^2 - c} \end{aligned}$$

ЧАСТИЧНАЯ РЕДУКЦИЯ ПРОТОКОЛА ВОЗВЕДЕНИЯ x В СТЕПЕНЬ n для $n = 2$ и x СВОБОДНОГО:

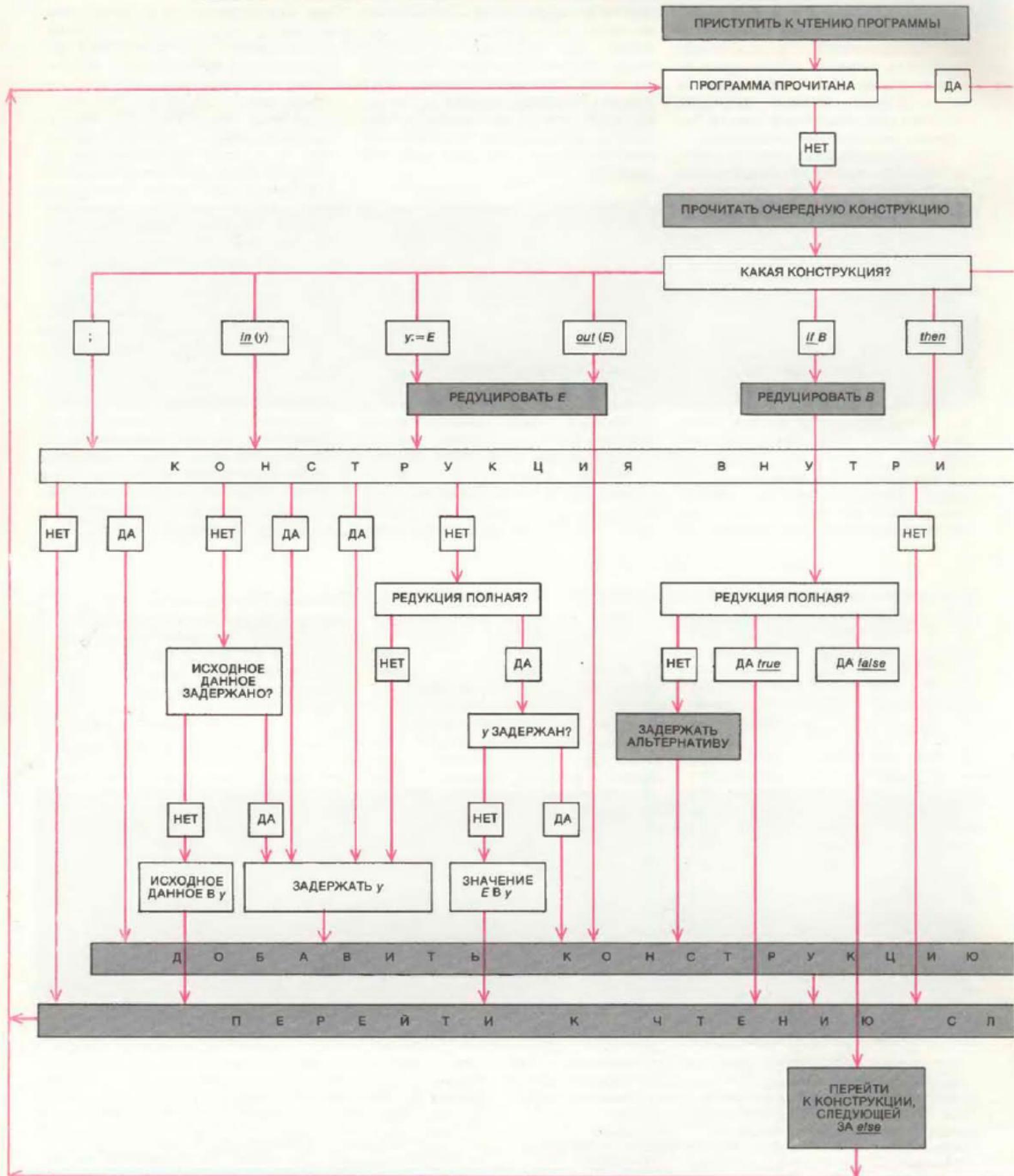
$$\begin{aligned} & \underline{\text{in}}(x); \underline{\text{in}}(n); y := 1; n > 0^+ \underline{\text{even}}(n)^+ n := n/2; x := x^2 \underline{\text{even}}(n)^-; n := n - 1; y := y \times x \quad n > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad y := 1; 2 > 0^+ \underline{\text{even}}(2)^+ n := 2/2; x := x^2 \underline{\text{even}}(n)^-; n := n - 1; y := y \times x \quad n > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad \quad \quad n := 1; \quad x := x^2 \underline{\text{even}}(n)^-; n := n - 1; y := 1 \times x \quad n > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad \quad \quad \quad \quad x := x^2 \underline{\text{even}}(1)^-; n := 1 - 1; y := 1 \times x \quad n > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad \quad \quad \quad \quad x := x^2; \quad \quad \quad n := 0; \quad \quad \quad y := 1 \times x \quad n > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad \quad \quad \quad \quad x := x^2; \quad \quad \quad \quad \quad y := 1 \times x \quad 0 > 0^-; \underline{\text{out}}(y) \\ & \underline{\text{in}}(x); \quad \quad \quad \quad \quad x := x^2; \quad \quad \quad \quad \quad y := 1 \times x; \quad \quad \quad \underline{\text{out}}(y) \end{aligned}$$

СПЕЦИАЛИЗИРОВАННАЯ ПРОГРАММА ВЫЧИСЛЕНИЯ $y = x^2$:

$$\underline{\text{in}}(x); x := x^2; y := 1 \times x; \underline{\text{out}}(y)$$

ЧАСТИЧНАЯ РЕДУКЦИЯ арифметического выражения (вверху) происходит в том случае, когда не все значения переменных заданы. При этом замена операции и ее аргументов на результат может дополняться алгебраическими преобразованиями, отражающими свойства операций и величин, входящих в выражение. Результатом частичной редукции является остаточное выражение, получившееся специализацией исходной формулы на суженное разнообразие своих аргументов. Точно так же смешанные вычисления могут трактоваться как совместная частичная редукция всех протоколов, входящих в детерминант общей программы (внизу). При этом часть протоколов сокращается, а часть других отбрасывается как противоречащая доступным данным. Найти остаточную программу — значит найти такой

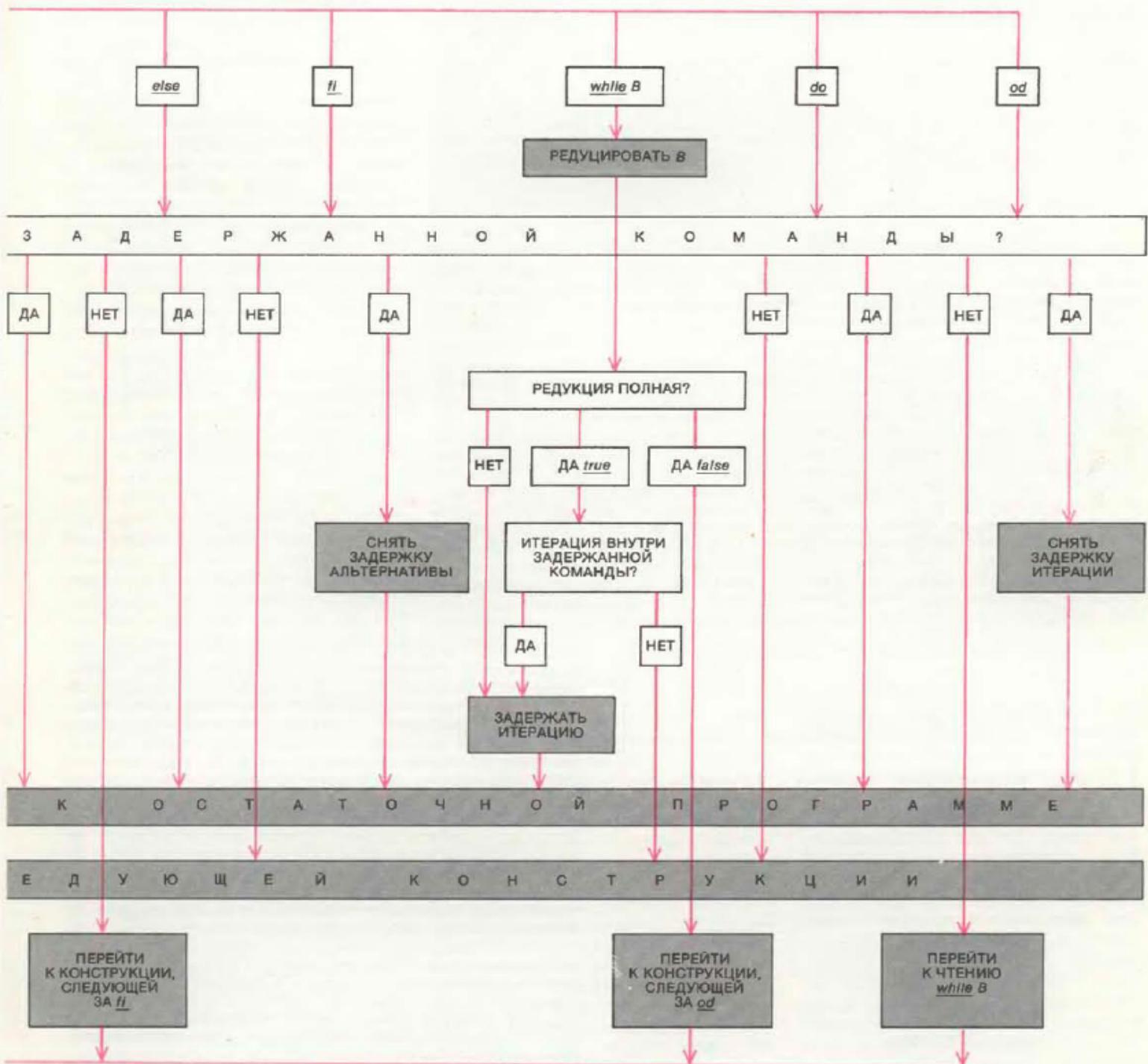
программный текст, чей детерминант был бы как можно ближе к редуцированному детерминанту исходной программы. В случае программы вычисления $y = x^n$ мы, конечно, не можем буквально перебрать все протоколы детерминанта, но зато без особого труда можем доказать, что для $n = 2$ все протоколы, кроме одного (соответствующего этому значению n), будут противоречивы. В результате частичной редукции этого протокола в качестве своеобразного остатка получаем простую программу, очень близкую к идеальной программе для вычисления $y = x^2$. Остаточная программа содержит избыточное умножение на 1, но выполнение соответствующего очевидного упрощения требует, понятно, иного знания, нежели то, что используется в редукциях, основанных на доступности значений переменных.



ОРГАНИЗАЦИЯ СМЕШАННЫХ ВЫЧИСЛЕНИЙ в языке МИЛАН напоминает его операционную семантику. Шаги обычных вычислений перемежаются с шагами формирования остаточной программы. Каждая прочитываемая команда либо выполняется, либо задерживается и добавляется к

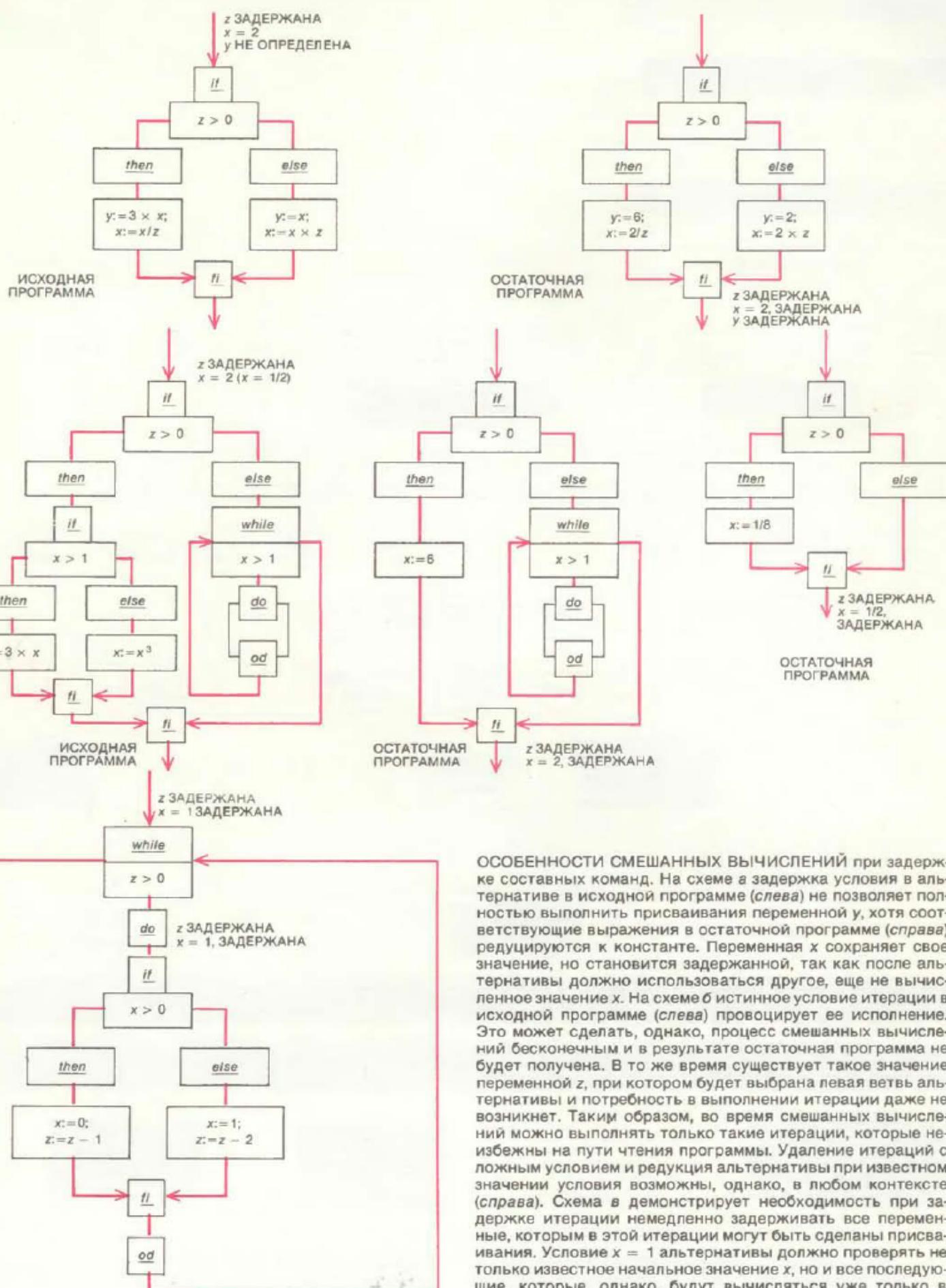
остаточной программе. Причиной задержки команд ввода является незданность входных переменных, которые при этом объявляются задержанными. Задержанная переменная недоступна для дальнейшего использования. Неполная редукция выражения или условия является причиной за-

ЗАВЕРШИТЬ СМЕШАННЫЕ ВЫЧИСЛЕНИЯ



держки присваивания, альтернативы или итерации. Задержка присваивания некоторой переменной, естественно, задерживает эту переменную. Задержка альтернативы и итерации автоматически задерживает все находящиеся внутри них базовые команды и итерации, однако не пре-

деляет присваивания, альтернатив и итераций со сложными условиями. Кроме этого, при задержке итерации немедленно задерживаются все переменные, которым возможны присваивания внутри этой итерации.



ОСОБЕННОСТИ СМЕШАННЫХ ВЫЧИСЛЕНИЙ при задержке составных команд. На схеме *a* задержка условия в альтернативе в исходной программе (*слева*) не позволяет полностью выполнить присваивания переменной *y*, хотя соответствующие выражения в остаточной программе (*справа*) редуцируются к константе. Переменная *x* сохраняет свое значение, но становится задержанной, так как после альтернативы должно использоваться другое, еще не вычисленное значение *x*. На схеме *b* истинное условие итерации в исходной программе (*слева*) провоцирует ее исполнение. Это может сделать, однако, процесс смешанных вычислений бесконечным и в результате остаточная программа не будет получена. В то же время существует такое значение переменной *z*, при котором будет выбрана левая ветвь альтернативы и потребность в выполнении итерации даже не возникнет. Таким образом, во время смешанных вычислений можно выполнять только такие итерации, которые неизбежны на пути чтения программы. Удаление итераций с ложным условием и редукция альтернативы при известном значении условия возможны, однако, в любом контексте (*справа*). Схема *a* демонстрирует необходимость при задержке итерации немедленно задерживать все переменные, которым в этой итерации могут быть сделаны присваивания. Условие $x = 1$ альтернативы должно проверять не только известное начальное значение *x*, но и все последующие, которые, однако, будут вычисляться уже только в остаточной программе. Недопустимая редукция альтернативы может быть предотвращена немедленной задержкой

Система преобразований любой текстовой информации, которая приводит к одной и той же неподвижной точке независимо от порядка применения допустимых преобразований, называется системой Чёрча—Россера по имени известного американского математика А. Чёрча и его в то время молодого ученика Дж. Россера, которые в 30-е годы впервые исследовали подобного рода системы под названием λ -исчислений. Свойство Чёрча—Россера в применении к трансформационной семантике языка МИЛАН гарантирует однозначность вычислений, не взирая на возможные акты свободного выбора применяемых преобразований. Система преобразований, однако, не гарантирует, что какая-бы то ни было неподвижная точка будет обязательно достигнута. Раскрытие цикла может предлагать свои услуги бесконечно, если возникающее при этом условие альтернативы никогда не получит значение ложь, что привело бы к устраниению раскрываемой итерации из программы.

Естественно, что применение «в лоб» трансформационной семантики для обработки программы путем комбинаторного перебора всех возможностей применения преобразований к конструкциям программы очень незэффективно. Достоинства такого способа описания языков программирования лежат в другом. Базовые трансформации программного текста, предписываемые трансформационной семантикой, могут быть основой для достоверного построения разнообразных программных процессоров, существенно более широких, нежели жесткие схемы обычных и смешанных вычислений, задаваемых операционной семантикой. Еще одним достоинством трансформационной семантики является то, что она легко допускает присоединение дополнительных преобразований, сопровождающих улучшению остаточной программы. Например, присоединение к трансформационной семантике языка МИЛАН преобразования вида $E \cdot 1 - E$, где E — произвольное арифметическое выражение, позволяет получить идеальную остаточную программу для вычисления x^2 (см. рисунок на с. 35).

МЫ ОПИСАЛИ смешанные вычисления как общий метод специализации программ на сужение области их исходных данных. В качестве особого случая сужения мы выделили проекцию области на заданные значения части независимых переменных. Остаточная программа, соответствующая этому сужению, называется проекцией исходной программы на заданные значения переменных.

Пожалуй, наиболее интересным открытием, сделанным на основе понятия смешанных вычислений, было вы-

РЕДУКЦИЯ АРГУМЕНТА	x в передовой позиции аргумента выражения, $x = a$	a
РЕДУКЦИЯ ПЕРЕМЕННОЙ	x находится в списке переменных, но не входит в текст программы	\emptyset
РЕДУКЦИЯ ОПЕРАЦИИ	$f(a_1, \dots, a_n)$, где f — операция, a_1, \dots, a_n — константы, $f(a_1, \dots, a_n) = b$	b
РЕДУКЦИЯ ВВОДА	$in(x)$, входное значение равно a	$x := a$
РЕДУКЦИЯ ПРИСВАИВАНИЯ	$x := a$; x в передовой позиции, a — константа	$\emptyset, x = a$
РЕДУКЦИЯ АЛЬТЕРНАТИВЫ	$if\ true\ then\ S1\ else\ S2\ fi$ $if\ false\ then\ S1\ else\ S2\ fi$	$S1$ $S2$
РАСКРЫТИЕ ИТЕРАЦИИ	$while\ B\ do\ S\ od$, в передовой позиции	$if\ B\ then\ S;\ while\ B\ do\ S\ od\ else\ fi$

ТРАНСФОРМАЦИОННАЯ СЕМАНТИКА языка МИЛАН задается набором правил преобразования программного текста и значений переменных. Все правила, кроме правила для итерации, являются редукциями, приводящими к устранению конструкции из программы или замене ее на более простую. В левой колонке таблицы помещено название конструкции, в средней — описывается контекст, в котором преобразование возможно, в правой — указывается, на что заменяется вхождение конструкции, к которому применено правило. Правило для итерации является раскрытием: оно вычленяет один экземпляр условия итерации и ее внутренних команд, к которым могут теперь применяться редукции, сохраняя саму итерацию для возможных повторных раскрытий. Переменная x находится в передовой позиции аргумента выражения, если между ним и началом программы нет присваивания этой переменной и выражение не является условием итерации. Присваивание переменной x находится в передовой позиции, если оно не находится внутри составной команды и между ним и началом программы нет ни одного вхождения x . Итерация находится в передовой позиции, если она не находится внутри составной команды и между ней и началом программы нет других редуцируемых итераций. Выполнение программы в рамках трансформационного подхода состоит в попытках применения к программе любого из правил. При полностью заданных входных переменных программа редуцируется к команде вывода, в аргументе которой читается результат. В общем случае, при наличии свободных входных переменных, исходная программа редуцируется к некоторой остаточной программе.

ЯСНЕНИЕ сущности трансляции — одно из наиболее фундаментальных и часто встречающихся в работе ЭВМ процессов. Способ записи программ, которые могут непосредственно выполняться на ЭВМ, — так называемый машинный язык — «удобен» для машины, но очень не удобен для человека, который придумывает эти программы. Без особого преувеличения можно сказать, что все развитие программирования шло в сторону приспособления языка программирования к удобству человека. При этом основным стилем было желание составлять программу в тех же обозначениях, в которых человек формулирует задачу. Так появлялись языки программирования все более и более высокого уровня. Проблема внедрения этих языков могла считаться, однако, решенной только в том случае, если перевод их на ма-

шинный язык мог быть выполнен с помощью автоматической процедуры, т.е. общей программы, получившей название транслятора или компилятора. Транслятор по тексту программы на входном языке строит программу на машинном языке так, что эти программы реализуют одну и ту же функцию. Построение трансляторов в конце 50-х годов стало весьма специфической и очень трудоемкой областью программного обеспечения. Существует, конечно, и более прямой путь восприятия машинной программы на входном языке высокого уровня. Надо взять операционную семантику входного языка и запрограммировать ее на языке машины. Такие универсальные программы получили название интерпретаторов. Если подать на вход интерпретатора текст программы на входном языке и ее данные, то машина про-

изведет такие действия, как если бы она непосредственно выполняла входную программу. Но, хотя такая техника применяется часто, особенно на персональных компьютерах, в общем случае исполнение входных программ в режиме прямой интерпретации языка высокого уровня очень не экономично, особенно если программа должна выполняться многократно.

Предположим, однако, что на языке машины запрограммированы смешанные вычисления. Соответствующую программу назовем смешанным вычислителем. Он реализует функцию трех переменных: первый аргумент — исходная программа, второй аргумент — ее известные входные данные и третий аргумент — ее свободные переменные. Результат смешанного вычисления — проекция исходной программы на ее известные данные, а сама проекция реализует функцию свободных переменных. Возьмем теперь смешанный вычислитель и применим его к интерпретатору входного языка при некоторой заданной входной программе, но при ее данных, оставленных свободными. В результате мы получим

проекцию интерпретатора на входную программу, при этом проекция — это программа на машинном языке. По свойствам остаточной программы, если на вход этой машинной программы подать исходные данные входной программы на языке высокого уровня, она вычислит то же значение, что и входная программа. Другими словами, проекция интерпретатора входного языка на программу в этом языке представляет собой перевод этой программы на машинный язык.

Можно пойти дальше. Рассмотрим смешанный вычислитель как программу, которая сама может быть подвержена смешанным вычислениям — с заданным интерпретатором входного языка, но с его аргументами, оставленными свободными. В результате мы получим проекцию смешанного вычислителя на интерпретатор. Если на вход этой остаточной программы подать текст какой-либо входной программы, то по свойствам смешанных вычислений в качестве результата будет получен ее перевод на машинный язык. Другими словами, проекция смешанного вычислителя на интерпретатор вход-

ного языка — это программа транслятора с этого языка на машинный.

Эти замечательные соотношения автор называет проекциями Футамуры по имени молодого японского программиста, который впервые нашел их в 1971 г. Они были опубликованы в малоизвестном японском техническом журнале и с тех пор по крайней мере два раза переоткрывались, прежде чем стали общезвестным фактом.

СМЕШАННЫЕ вычисления как конструктивное направление в теории и практике программирования обозначились только в последние несколько лет. В качестве же абстрактной математической концепции они имеют более чем 50-летнюю историю и, мало того, принадлежат к глубинным понятиям оснований математики.

В 20-е годы ученые усиленно размышляли над природой математического языка, одним из главных компонентов которого всегда были функциональные обозначения. Немецкий математик М. Шёнфinkel сделал следующее наблюдение, касающееся функций нескольких переменных. Пусть у нас

ИСХОДНАЯ ПРОГРАММА:

```
in(x); in(n); y = 1; while n > 0 do while even(n) do n := n/2; x := x2 od; n := n - 1; y := y × x od; out(y)
```

dod *DOD*

ЦЕПОЧКА РЕДУКЦИЙ И РАСКРЫТИЙ (вход для *n* равен 2):

```
x, y, n
in(x); in(n); y = 1; while DOD; out(y) —
x, y = 1, n
in(x); n := 2; if n > 0 then while dod; n := n - 1; y := y × x; while DOD else fi; out(y) —
x, y = 1, n = 2
in(x); if 2 > 0+ then while dod; n := n - 1; y := 1 × x; while DOD else fi; out(y) —
x, y = 1, n = 2
in(x); if even(2)+ then n := 2/2; x := x2; while dod else fi; n := n - 1; y := 1 × x; while DOD; out(y) —
x, y = 1, n = 2
in(x); n := 1; x := x2; if even(n) then n := n/2; x := x2; while dod else fi; n := n - 1; y := 1 × x; while DOD; out(y) —
x, y = 1, n = 1
in(x); x := x2; if even(1)- then n := 1/2; x := x2; while dod else fi; n := n - 1; y := 1 × x; while DOD; out(y) —
x, y = 1, n = 1
in(x); x := x2; n := 1 - 1; y := 1 × x; while DOD; out(y) —
x, y = 1, n = 1
in(x); x := x2; n := 0; y := 1 × x; if n > 0 then while dod; n := n - 1; y := y × x; while DOD else fi; out(y) —
x, y = 1, n = 0
in(x); x := x2; y := 1 × x; if 0 > 0- then while dod; n := n - 1; y := y × x; while DOD else fi; out(y) —
x, y = 1
in(x); x := x2; y := 1 × x; out(y) — неподвижная точка
```

ОСТАТОЧНАЯ ПРОГРАММА, построенная согласно трансформационной семантики языка МИЛАН для программы $y = x^n$, конкретизируемой на случай $n = 2$. Редуцируемые команды и выражения напечатаны синим, раскрываемые итерации показаны красным. *DOD* — сокращение итерации по положительности *n*, *dod* — сокращение внутренней итерации по четности *n*. Раскрытие итераций поставляет остальным редукциям «материал для работы». Редукция альтернатив отсекает ветви вычислений, противоречащие

доступным данным. «Рабочими редукциями» являются редукции аргументов, операций и присваиваний. В отличие от операционной семантики, предписывающей единственную последовательность вычислений, трансформационная семантика допускает для этой программы несколько сот вариантов смешанных вычислений, которые в силу свойства Чёрча-Россера приводят к одной и той же остаточной программе, показанной в последней строке схемы.

есть функция двух переменных $f(x, y)$. Подстановка вместо одной переменной некоторого значения, скажем $x = 10$, делает из функции двух переменных $f(x, y)$ новую функцию одной переменной $\varphi(y)$, связанную с f следующим соотношением: для любого y $\varphi(y) = f(10, y)$. Можно сказать, что связывание некоторого аргумента конкретным значением создает функциональный оператор, сопоставляющий функции двух переменных функцию одной переменной. Этот оператор, изученный в начале 30-х годов американским математиком Х. Карри, получил название карриева оператора и, очевидно, является абстрактным прообразом процедуры смешанных вычислений.

В начале 30-х годов Чёрч разработал формальное исчисление функций в надежде получить язык конструктивного манипулирования функциональными обозначениями предельной простоты и максимальной общности. Это исчисление, получившее название λ -исчисления, сыграло большую роль в нахождении точного определения эффективно вычислимых функций. Именно в рамках λ -исчисления в 1936 г. Чёрчом был найден пример задачи, которую можно точно определить математически, но для которой не существует единой процедуры ее эффективного решения. В терминах языка МИЛАН эта задача звучит так: «Для любой программы, содержащей итерации, определить, может ли она войти в бесконечный цикл или нет». В λ -исчислении «программы» и «данные» изображались объектами одной природы. Вычислительный процесс задавался правилами, которые можно считать прототипом трансформационной семантики языков программирования. В λ -исчислении тоже были правила раскрытия и редукции, удовлетворявшие свойству Чёрча—Россера, а «смешанные вычисления» использовались в качестве основного метода вычисления функций от нескольких переменных согласно правилу Шёнфинкеля—Карри.

В 40-х годах в трудах известного американского математика С. Клини получила большое развитие теория рекурсивных функций как теоретической модели эффективной вычислимости. В книге «Введение в метаматематику», вышедшей в свет в 1952 г., Клини привел доказательство так называемой $s\text{-}t\text{-}n$ -теоремы, которая устанавливала существование процедуры смешанных вычислений для рекурсивных функций. По ходу развития теории рекурсивных функций стало ясно, что $s\text{-}t\text{-}n$ -теорема играет центральную роль в теории вычислимости.

В конце 50-х годов московский математик В. Успенский в своих «Лекциях о вычислимых функциях» доказал теорему, которая в понятиях этой статьи звучит, примерно, так: «Для того что-

ОБОЗНАЧЕНИЯ

$Int(P; X)$	— интерпретатор входного языка
$Tran(P; X)$	— транслятор с входного языка
P	— входная программа
X	— исходные данные входной программы
$Mix(P; A; Y)$	— смешанный вычислитель
P	— общая программа
A	— заданные значения входных переменных
Y	— свободные входные переменные
p	— конкретная программа
$p(x)$	— программа p с указанными входными переменными x
$p(a)$	— результат выполнения программы p для исходных данных a
$ob_p(x)$	— объектная программа, перевод входной программы p на машинный язык
$p_a(y)$	— остаточная программа, проекция общей программы $p(x, y)$ на данные значения a части x ее входных переменных

СВОЙСТВА

свойство интерпретатора:

$$Int(p; a) = p(a)$$

свойство объектной программы:

$$p(a) = ob_p(a)$$

свойство транслятора:

$$Tran(p; x) = ob_p(x)$$

свойство проекции:

$$p_b(b) = p(a, b)$$

свойство смешанного вычислителя:

$$Mix(p; a; y) = p_a(y)$$

ПРОЕКЦИИ ФУТАМУРЫ

$$Int_p(x) = ob_p(x) \quad (1)$$

$$Mix_{Int}(P; X) = Tran(P; X) \quad (2)$$

ДОКАЗАТЕЛЬСТВА

1-й проекции:

$$ob_p(a) = Int(p; a) = p(a)$$

2-й проекции:

$$Tran(p; x) = Mix(Int; p; x) = Int_p(x) = ob_p(x)$$

ФОРМУЛИРОВАНИЕ ЗАКОНОМЕРНОСТЕЙ в виде математических выражений зачастую требует большой подготовительной работы в поиске формальных определений и обозначений. Эта работа, однако, вознаграждается общностью, краткостью и точностью формулировки, выразительно раскрывающей природу вещей и явлений. Именно так обстоит дело с проекциями Футамуры, демонстрирующими органическую связь трансляции и смешанных вычислений. Первая проекция гласит, что перевод входной программы на машинный язык может быть получен путем смешанного вычисления интерпретатора входного языка при заданной программе, но ее входных переменных оставленных свободными. В доказательстве справедливости первой проекции первое равенство имеет место в силу свойства проекции, а второе равенство — в силу свойства интерпретатора. Вторая проекция гласит, что сам транслятор может быть получен как проекция смешанного вычислителя на интерпретатор с оставлением аргументов интерпретатора свободными. В доказательстве справедливости второй проекции первое равенство имеет место в силу свойства проекции, второе равенство — в силу свойства смешанного вычислителя и третье равенство — в силу 1-й проекции Футамуры. Важность этих формулировок не в том, что они предлагают еще один подход к получению объектных программ и построению трансляторов, а в том, что они показывают, каким знанием нужно обладать для реализации этих важнейших процедур программирования.

бы на данный машинный язык можно было бы перевести любой другой язык программирования, необходимо и достаточно, чтобы в машинном языке были определены смешанные вычисления». Это утверждение, по существу, эквивалентно проекциям Футаму-

ры, однако лишь в конце 70-х годов эти два утверждения стали рядом.

Смешанные вычисления в применении к программному обеспечению восходят к итальянскому специалисту по вычислительной науке Л. Ломбарди, который назвал свой подход к обра-

ботке программ пошаговыми вычислениями с неполной информацией. В основу его идеи была положена критика основного положения традиционного программирования: использовать ЭВМ для выполнения программы только при полностью определенных данных. Этому требованию полной определенности Ломбарди противопоставил реальные условия, в которых человек постоянно должен принимать решения в условиях неполноты информации. Для того чтобы приблизиться к этим условиям, общая организация вычислений на машинах должна быть изменена. В несколько метафорическом стиле идея Ломбарди выглядит так: двигаясь на встречу постепенно поступающей информации, машина должна редуцировать универсальную программу на основе доступных данных, сохраняя запас фрагментов общей программы, подготовленные к восприятию временно отсутствующей информации. Как только появляется порция новой информации, соответствующий фрагмент редуцируется, обеспечивая общий прогресс вычислений.

Ломбарди не развел математической теории вычислений с неполной информацией, хотя и провел эксперимент с разработкой частичного вычислителя для языка программирования Лисп. Его работы дали толчок серии докторских работ, выполненных шведскими аспирантами А. Харалдсоном и П. Эмануэльсоном, которые построили усовершенствованный смешанный вычислитель для языка Лисп. Коллектив московских специалистов Г.Х. Бабича, Л.Ф. Штернберга и Т.И. Югановой, основываясь на идеях Ломбарди, разработал специальный язык ИНКОЛ для организации вычислений с неполной информацией и применил его для процедуры принятия экономических решений в условиях неопределенности. Наконец, работы Ломбарди привели Футамуру к его проекциям.

СИСТЕМАТИЧЕСКАЯ работа над теорией и применением смешанных вычислений и ее интеграция с другими примыкающими исследованиями началась в 1977 г. с работ автора этой статьи и его сотрудников. Это направление быстро развивается и сейчас литература по смешанным вычислениям уже перевалила за полусятню наименований. Теоретические модели смешанных вычислений, реализация смешанных вычислений для главных языков программирования, системы преобразований и редукций программ, аппаратная реализации смешанных вычислений, применения к теории компиляции, рассмотрение аналогичных явлений в процессах логического доказательства, приложения к параллельным вычислениям, построение эффективных специализированных алгоритмов,

адаптация и генерация пакетов прикладных программ, рассмотрение сужений, более общих, нежели проекции, — все это заслуживает подробного обзора, но уже в специальной литературе.

Автор хотел бы завершить это изложение двумя методологическими замечаниями. Смешанные вычисления дают яркий пример того, насколько важно в зрелой науке уметь сочетать увлеченность практической работой с широким кругозором и знанием фундаментальных дисциплин, как важно теоретику уметь обнаруживать проблески своих теоретических концепций в пестроте реального мира. Широкая практика программирования, которая сделала актуальными проблемы трансляции и взаимодействия универсальных и специализированных методов, сложилась к концу 50-х годов. К этому времени в теории вычислимости уже существовали теоретические основы смешанных вычислений в виде λ -исчислений, $s\text{-}t\text{-}n$ -теоремы, теоремы о транслируемости. Многие из теоретиков 60-х годов перешли в сферу программирования. Многие из тех, кто сейчас с таким энтузиазмом работают над смешанными вычислениями, в свое время при подготовке к аспирантским экзаменам читали литературу по теории вычислимости. Соответствующие потоки информации проходили рядом или даже пересекались и существовали подолгу в головах исследователей и практиков. Потребовалось, однако, немало лет, чтобы искра догадки осветила все разнообразие практических и теоретических предпосылок этого нового раздела программирования.

Второе замечание — более деловое. Уже начальный опыт использования смешанных вычислений показал, что универсальные алгоритмы нужно создавать с учетом их двухступенчатого применения: получения специализированной версии и ее утилизации в суженной области. Понимание устройства смешанных вычислений позволяет значительно повысить эффективность остаточных программ и упростить их получение. Заметим, что этот подход более удобен для программиста. Как правило, для смешанных вычислений подходит такой универсальный алгоритм, который непосредственно вытекает из общей теории. Усложненная кодировка исходных данных, излишняя ориентация на особенности ЭВМ, компактификация — все это противопоказано хорошему универсальному алгоритму. Выражаясь метафорически, универсальный алгоритм вместо рабочей лошадки становится элитным производителем остаточных программ, которые, принимая на себя груз рабочих вычислений, объединяют приобретенную эффективность с достоверностью общей теории, воплощенной в универсальном методе.

Издательство МИР предлагает:

Л. Маргелис РОЛЬ СИМБИОЗА В ЭВОЛЮЦИИ КЛЕТКИ

Перевод с английского

Монография известного американского специалиста по происхождению жизни посвящена интересной проблеме — происхождению клеток эукариот в результате симбиоза. В монографии детально излагаются суть теории симбиоза и те фактические данные, на которых она основана, геологические и физические предпосылки возникновения жизни, ранние этапы формирования обмена веществ, история возникновения различных клеточных органелл.

Предназначена для эволюционистов, цитологов, молекулярных биологов, зоологов, ботаников, палеонтологов.

1983, 40 л. Цена 4р. 30к.

Дж. Симпсон ВЕЛИКОЛЕПНАЯ ИЗОЛЯЦИЯ История млекопитающих Южной Америки

Перевод с английского

Книга известного американского ученого посвящена эволюции млекопитающих Южной Америки, развивавшихся в условиях изоляции на протяжении большей части кайнозойской эры. Рассмотрены общеэволюционные аспекты — влияние изоляции, факторы биоценотических сукцессий, влияние животных иммигрантов на перестройку флоры и фауны континента, явления параллельного и конвергентного развития.

Предназначена для зоологов, палеонтологов, биогеографов, а также биологов других специальностей.

1983, 16 л. Цена 1р. 50к.

