COMPUTER MONOGRAPHS
General Editor: Stanley Gill
Associate Editor: J. J. Florentin

RECURSIVE TECHNIQUES IN PROGRAMMING

D. W. BARRON

MACDONALD: LONDON
1969

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Д. Баррон

РЕКУРСИВНЫЕ МЕТОДЫ В ПРОГРАММИРОВАНИИ

Перевод с английского
В. В. Мартынюка

Под редакцией
Э. З. Любимского

ИЗДАТЕЛЬСТВО
«МИР»
Москва 1974

УДК 681.142.2

В книге излагаются рекурсивные методы программирования и демонстрируются возможности рекурсивного описания алгоритмов. Эти методы получают все более широкое распространение в практике программирования, и возможность их применения учитывается при разработке языков программирования и вычислительных машин. В книге показаны перспективы использования рекурсивных методов. Их удобство и эффективность демонстрируются на различных примерах.

Книга будет полезна студентам, аспирантам и специалистам, занимающимся разработкой и подготовкой алгоритмов решения задач на вычислительных машинах.

Редакция литературы по математическим наукам

Д. Баррон

РЕКУРСИВНЫЕ МЕТОДЫ В ПРОГРАММИРОВАНИИ

Редактор Л. Бабынина
Художник В. Новоселова Художественный редактор В. Шаповалов
Технический редактор Г. Алюлина. Корректор Е. Кочегарова

Сдано в набор 22/III 1974 г. Подписано к печати 8/VIII 1974 г. Бумага тип. № 2
$60\times90^{1}/_{16}$=2,5 бум. л. Усл. печ. л. 5,0. Уч.-изд. л. 4,2. Изд. № 1/7516. Цена 31 к. Зак. 155

ИЗДАТЕЛЬСТВО «МИР» Москва, 1-й Рижский пер., 2

Ордена Трудового Красного Знамени
Ленинградская типография № 2 имени Евгении Соколовой
Союзполиграфпрома при Государственном комитете Совета Министров СССР
по делам издательств, полиграфии и книжной торговли.
198052, Ленинград, Л-52, Измайловский проспект, 29

Б $\frac{20204-038}{041(01)-74}$ 38—74

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА И ПЕРЕВОДЧИКА

Широкое проникновение рекурсивных методов в практику программирования началось с распространением универсального языка программирования АЛГОЛ-60, допускающего рекурсивные обращения к процедурам. На первых порах аппарат процедур, введенный авторами АЛГОЛа, казался совершенно необычным и вызывал недоумение. Однако практика показала, что разумное применение рекурсии с последовательным сокращением количественной сложности задач, возникающих в процессе рекурсивных обращений, является весьма эффективным методом программирования, существенно упрощает запись различных сложных алгоритмов, а в ряде случаев оказывается практически незаменимым средством. Рекурсия отражает существенную характерную черту абстрактного мышления, проявляющуюся при анализе сложных алгоритмических и структурных конструкций в самых различных приложениях. Дело в том, что, пользуясь рекурсией, мы избавляемся от необходимости утомительного последовательного описания конструкции, и ограничиваемся выявлением взаимосвязей между различными уровнями этой конструкции.

В настоящее время область практического применения рекурсии весьма широка. Она включает, в частности, сложные задачи численного анализа, алгоритмы трансляции, а также различные операции над списками, являющиеся необходимым аппаратом разработки современных автоматизированных систем управления. Поэтому аппарат рекурсии предусматривается практически во всех языках программирования, появляющихся после АЛГОЛа.

Предлагаемая книга, написанная известным специалистом в области программирования Д. Барроном, содержит достаточно полный обзор рекурсивных методов в программировании и характеризуется очень удачным выбором уровня изложения, при котором сочетаются научная строгость и доступность для широкого круга читателей. Эта книга несомненно представляет большой интерес для пользователей и разработчиков современных систем математического обеспечения ЭВМ.

Э. З. Любимский
В. В. Мартынюк

ПРЕДИСЛОВИЕ

Развитие программирования на различных этапах происходило под влиянием введения новых понятий. Одним из самых ранних понятий была идея замкнутой подпрограммы, которая теперь настолько укоренилась в повседневной практике программирования, что без нее нам трудно представить себе программирование. Я убежден, что общее признание рекурсивных методов окажет в конечном счете такое же значительное влияние на программирование, как и введение подпрограмм. Появление подпрограмм сделало возможным разбиение большой программы на меньшие части. Однако развитие вычислительной математики должно привести к снятию ограничений, налагаемых теперешним уровнем наших возможностей конструирования больших программ. Возникнет необходимость разработки более сложных способов иерархического объединения частей программы и нельзя будет обойтись без рекурсивной структуры.

В настоящее время распространен взгляд на рекурсию как на интересное, но необязательное украшение системы программирования. К тому же против рекурсии существует значительное предубеждение, которое объясняется тем, что в большинстве случаев машины не предназначены для работы с рекурсивными процедурами и поэтому выполняют их неэффективно. Однако если преимущества этого способа программирования получат должную оценку, то машины будут конструироваться с таким расчетом, чтобы облегчать рекурсивное программирование. Надеюсь, что эта книга сможет способствовать признанию рекурсии и убедит читателя в том, что рекурсия не является чем-то нарочито усложненным и не предназначена для касты посвященных, а просто представляет собой еще одно средство программирования, которым можно пользоваться удачно или злоупотреблять, как и всяким другим.

Д. В. Баррон

1.1. Введение

Существует мнение [1], что «если бы в средние века были вычислительные машины, то одни программисты сжигали бы на кострах других программистов за ересь». Почти наверняка самой страшной ересью считалась бы вера (или неверие) в рекурсию. Достоинства и недостатки рекурсии как средства программирования подвергались широкому обсуждению в течение последних нескольких лет. Как нередко бывает в таких ситуациях, имеется тенденция к сохранению крайних взглядов и к рассмотрению проблемы либо в черном, либо в розовом свете. Одни утверждают, что всегда лучше пользоваться не рекурсией, а итерацией. Другие же настолько верят в рекурсию, что они не снисходят до включения очевидных методов итерации в разрабатываемые ими языки программирования. Автор придерживался при написании этой книги промежуточной позиции, избегая крайних точек зрения. По возможности предмет излагается с учетом перспектив, демонстрируются способы успешного применения рекурсии в различных ситуациях, выявляются соотношения между рекурсией и более известными итеративными методами. Автор старается показать также влияние рекурсивных методов на программное и аппаратное обеспечение вычислительных систем.

1.2. Рекурсивные функции и процедуры

В математике рекурсией называется способ описания функций или процессов через самих себя. По-видимому, наиболее известным примером рекурсивно описанной функции является следующая факториальная функция от положительного целого аргумента:

$$factorial\,(n) = n \times factorial\,(n-1), \quad \text{если} \quad n \neq 0$$
$$factorial\,(0) = 1$$

Здесь factorial (n) определяется через значение factorial ($n-1$), которое определяется через factorial($n-2$), и т. д. до сведения к значению factorial(0), которое определено явно и равно 1. Любое рекурсивное описание должно содержать явное

определение для некоторых значений аргумента (или аргументов), так как иначе процесс сведения оказался бы бесконечным. Для таких описаний удобна запись условного выражения, которую впервые ввел Маккарти [2]. Условные выражения представляются в форме

$$[b_1 \to e_1,\ b_2 \to e_2,\ \ldots\ b_{n-1} \to e_{n-1},\ e_n]$$

Здесь b_i — логическое условие, которое может быть истиной или ложью, e_i — выражение. Значение условного выражения получается проверкой значений b_i по очереди слева направо до встречи первого значения «истина»; соответствующее значение e_i берется в качестве значения всего выражения. Если все условия b_i ложны, то выражение принимает значение e_n. Таким образом описание факториальной функции имеет вид

$$factorial\,(n) = [(n = 0) \to 1,\ n \times factorial\,(n - 1)]$$

Другим примером рекурсивного описания функции может служить рекуррентное соотношение между функциями Бесселя различных порядков с одинаковыми аргументами, например:

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x)$$

Если записать $J_n(x)$ в симметричном обозначении $J(n, x)$ и заменить n на значение $n - 1$, то рекуррентное соотношение принимает вид

$$J(n,\ x) = \frac{2(n-1)}{x} J(n-1,\ x) - J(n-2,\ x)$$

Таким образом, если для данного аргумента x известны значения J_0 и J_1, то можно записать определение $J(n, x)$ в виде

$$J(n,\ x) = [(n = 0) \to J(0,\ x),\ (n-1) \to J(1,\ x),$$
$$((2(n-1)/x)\, J(n-1,\ x) - J(n-2,\ x))]$$

Еще одна возможная форма рекурсии встречается тогда, когда процесс описывается через подпроцессы, один из которых идентичен основному процессу. Например, рассмотрим двойной интеграл

$$\int_a^b \int_c^d f(x,y)\, dx\, dy$$

Один из методов вычисления состоит в представлении двойного интеграла с помощью двукратного простого интегрирования

$$\int_a^b \left(\int_c^d f(x,\ y)\, dy \right) dx$$

Для вычисления внешнего интеграла требуется знать значения подинтегрального выражения в выбранных точках, а для вычисления подинтегрального выражения требуется интегрирование, так что подпроцесс идентичен основному процессу.

Определенный интеграл

$$I = \int_a^b F(x)\,dx$$

является, разумеется, функцией от a и b, так как значение I зависит от значений a и b. Можно также сказать, что I является функцией от F, так как при изменении функции F изменяется значение интеграла. Итак, мы приходим к записи I как функции от трех переменных

$$I = I(a, b, F)$$

Мы ввели здесь понятие, которое не встречается в обычной математике, а именно переменную, представляющую *функцию*. Важно чувствовать разницу между величиной F, представляющей функцию, и величиной $F(3)$, представляющей результат применения функции к аргументу. Например, если

$$F(x) = 1 + x^2$$

то $F(2)$ — это число 5, а символ F, взятый отдельно, представляет операцию «возведение аргумента в квадрат и прибавление единицы». Обычно традиционная математика имеет дело с числами того или иного вида. При этом не встречаются функции без аргументов и нет необходимости их различать. Но программирование имеет дело с процессами, и может возникнуть желание говорить о процессе независимо от объектов, над которыми процесс выполняется. Для описания функций как абстрактных понятий можно применять «лямбда-исчисление», разработанное Черчем [4]; подробнее ознакомиться с таким подходом можно по книге Хигмана [15].

Рассматриваемое различие не отражено в большинстве языков программирования. Например, на АЛГОЛе можно написать

real procedure *куб*(*y*); **value** *y*; **real** *y*;
begin *куб*:= *y* × *y* × *y* **end**;

Эта процедура присваивает значение непосредственно переменной *куб*, несмотря на то, что обращение к процедуре содержит аргумент, например:

a:= *куб*(*b*);

Явное различие делается в языке ЛИСП, где допускаются функции, результатами применения которых являются функции.

Аналогичные возможности предусмотрены в языке CPL, допускающем переменные типа функций; поэтому можно записать

let f **be function**

$$f := b \rightarrow \text{Sin, Cos}$$

$$a := f[c]$$

Таким образом, мы можем представить интеграл

$$I_1 = \int_a^b F(x)\,dx$$

как функцию от a, b и F, но не от x, поскольку значение I_1 не изменяется, если x всюду заменяется на z. Двойной интеграл

$$I_2 = \int_a^b \int_c^d F(x, y)\,dx\,dy$$

является функцией от a, b, c, d и F (но не от x и не от y) и может быть записан как

$$I_2 = I_2(a, b, c, d, F)$$

Предположим теперь, что мы хотим описать I_2 через I_1 (т. е. получить формальное представление процесса двойного интегрирования в виде двукратного простого интегрирования). Введем вспомогательную функцию

$$G(y) = F(x, y)$$

Тогда

$$I_2 = I_1(a, b, I_1(c, d, G))$$

Теперь появилась рекурсия, поскольку функция I_1 присутствует как аргумент в обращении к I_1; это типичный пример рекурсивного использования процедуры. Следует отличать такую рекурсию от другого типа рекурсии, при котором функция присутствует в правой части своего описания.

В случае двойного интеграла вычисление одного интеграла требует вычисления другого интеграла; второй интеграл вычисляется непосредственно. Принято говорить в таких случаях, что рекурсия имеет *один уровень глубины*; это означает, что процесс обращается к себе как к подпроцессу только один раз. Для рекурсивно описанной функции глубина рекурсии — это число промежуточных вычислений функции в процессе ее вычисления для

данного аргумента или набора аргументов[1]). Например, при вычислении функции factorial(3) в соответствии с рекурсивным описанием нужно вычислить factorial(2), factorial(1) и factorial(0); в этом случае рекурсия имеет глубину в три уровня. Для факториальной функции глубина рекурсии при любом заданном значении аргумента видна сразу. Однако это — исключение, а обычно глубина рекурсии не является очевидной даже при простейших описаниях. Например, рассмотрим алгоритм Эвклида для вычисления наибольшего общего делителя (НОД) двух положительных целых чисел. Этот алгоритм может быть описан следующим образом:

$$НОД(n, m) = [m > n \to НОД(m, n),$$
$$m = 0 \to n,\ НОД(m, Ост(n, m))]$$

($Ост(n, m)$ — функция, значением которой является остаток от деления n на m.) В этом описании отражено следующее правило: если m является точным делителем n, то НОД $= m$, а в противном случае нужно брать функцию НОД от m и от остатка, получаемого при делении n на m. Первая строка в описании всего лишь позволяет писать аргументы в любом порядке. Хотя это описание и является достаточно простым, все же сразу не очевидно, какая глубина рекурсии будет достигнута при вычислении, например, значения НОД (385, 1105). В общем случае можно утверждать, что рекурсивное использование процедуры требует очевидной конечной глубины рекурсии, тогда как вычисление рекурсивно описанной функции влечет за собой неопределенную глубину рекурсии, зависящую от значений аргументов. (Если описание предназначено для практических вычислений, то глубина рекурсии должна быть конечной. Бесконечное по глубине рекурсивное описание называется регрессивным. Некоторые рекурсивные описания могут иметь конечную глубину рекурсии при одних входных данных и бесконечную при других. Теоретически невозможно определять с помощью единого алгоритма, окажется ли произвольное описание конечным в любом случае, однако часто можно показать, будет ли в отдельных случаях глубина рекурсии конечной или бесконечной.)

Оба типа рекурсии могут присутствовать одновременно. Наиболее известным примером такого сочетания является функция Аккермана:

$$A(m, n) = [m = 0 \to n + 1,$$
$$n = 0 \to A(m - 1, 1),$$
$$A(m - 1, A(m, n - 1))]$$

[1]) Такое определение глубины рекурсии справедливо при условии, что каждое начатое вычисление функции не может быть завершено до завершения начатых позднее вычислений этой функции. — *Прим. перев.*

(Напоминаем, что при вычислении условного выражения условия проверяются по очереди, так что вторая и третья строки будут достигнуты только, если $m \neq 0$.) Кажущаяся простота этой функции является обманчивой. Читатель, располагающий временем, может попытаться вычислить значение $A(2, 3)$, исходя из этого определения.

В программировании встречаются оба типа рекурсии: и рекурсивное определение, и рекурсивное использование; важно чувствовать разницу между ними. Утверждения, что рекурсия является необязательным украшением в языке программирования, обычно основываются на аргументах, касающихся рекурсивно определенных функций; при этом упускается из виду рекурсивное использование процедур. На самом деле некоторые численные процессы (такие, как повторное интегрирование) по своей природе являются рекурсивными, и если язык программирования не допускает рекурсию, то приходится предпринимать некоторые специальные меры, которые сводятся к организации рекурсивной структуры для каждого частного случая.

1.3. Обработка рекурсивных данных

Впрочем, существует еще одна ситуация, в которой выгодно применять рекурсивные методы; она возникает при обработке данных, имеющих рекурсивную структуру. Например, скобочное арифметическое выражение может быть описано в виде

$$(A \varnothing B)$$

где $\varnothing$ — один из знаков $+$, $-$, $\times$ или $/$ и A, B — либо наименования переменных, либо скобочные выражения, например:

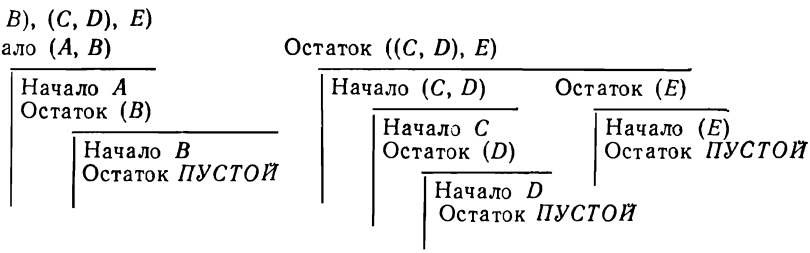
$$(a + ((b + (c \times d)) - e))$$

Очевидно, что программа, способная обрабатывать одно такое выражение, может, если она рекурсивная, обрабатывать сколь угодно сложное выражение, построенное в соответствии с этими правилами. Это достигается тем, что она рекурсивно обращается к самой себе, когда встречает открывающую скобку, и осуществляет выход, встретив закрывающую скобку.

Аналогичная ситуация возникает при обработке списков. Списковая структура может быть описана как размещение объектов, называемых *атомами*, в соответствии со следующими правилами.

Список либо ПУСТОЙ, либо состоит из начала, представляющего собой либо атом, либо список, и из остатка, который

является списком. Например:



Эта структура описана рекурсивно, и поэтому естественно применить к ней методы рекурсивной обработки. Например, для копирования списка мы пишем (см. [3] или [5])

$$Copy(z) = [Null(z) \rightarrow ПУСТО,\ Cons(Copy(Hd(z)),\ Copy(Tl(z)))]$$

Это рекурсивное описание функции[1]) в точности отражает рекурсивную структуру данных. Примеры рекурсивных функций для обработки списков будут приведены в гл. 2, а краткое описание списковых структур дается в приложении.

1.4. Рекурсия в языках программирования

На уровне машинного языка вычисление функции или выполнение процедуры обычно осуществляется с помощью подпрограммы. Идея подпрограммы, которая сама обращается к другой подпрограмме, общеизвестна; в такой терминологии аналогом рекурсивной функции является подпрограмма, которая содержит обращение к самой себе. Разумеется для таких подпрограмм требуется специальное обеспечение, так как нужна полная информация об иерархии связей и рабочих регистров для того, чтобы по окончании текущей работы можно было восстановить состояние подпрограммы, соответствующее прерванной непосредственно предшествовавшей работе. Именно этим рекурсивные системы программирования отличаются от нерекурсивных.

Рекурсия вошла в программирование в значительной степени благодаря системам обработки списков IPL [6] и ЛИСП [7], а также языкам АЛГОЛ-58 [8] и АЛГОЛ-60 [9, 10]. Мы не хотим этим сказать, что рекурсия не применялась прежде, однако те, кто использовали ее раньше, вероятно, не удостаивали этот метод специального термина. Рекурсия естественно появилась в ЛИСПе, который является функциональным языком (см. разд. 1.5) и связан с обработкой списковых структур,

[1]) Здесь *Copy* — копия, *Null* — пустой, *Cons* — объединение, *Hd* — начало, *Tl* — остаток. — *Прим. перев.*

рекурсивных по самой своей природе. Тот факт, что АЛГОЛ допускает рекурсию, вызван разнообразными причинами, поскольку нужда в рекурсии при решении задач численного анализа не столь очевидна. Некоторые приверженцы АЛГОЛа утверждают, что наличие рекурсии является его основным преимуществом перед другими языками типа ФОРТРАНа. Это послужило поводом для контрутверждений, что «все рекурсивные связи могут быть сведены к рекуррентным или итеративным описаниям» [11]. Хотя это утверждение и справедливо для рекурсивно определенных функций, оно неверно для рекурсивного использования процедур (иногда называемого косвенной рекурсией), которое, как мы видели, имеет важное значение для численного анализа.

Различие между «рекурсивным» языком типа АЛГОЛа и «нерекурсивным» языком типа ФОРТРАНа состоит в том, что на ФОРТРАНе программист, желающий пользоваться рекурсией, должен разработать для себя соответствующий аппарат, тогда как на АЛГОЛе этот аппарат обеспечивается системой, так сказать «за кулисами». Во многих рекурсивных системах за это приходится платить тем, что автоматический аппарат, предусмотренный для обеспечения рекурсии, вступает в действие даже тогда, когда программа не является рекурсивной; это приводит к напрасным затратам машинного времени. Разумеется, существуют различные способы организации рекурсии, некоторые из них будут весьма подробно рассмотрены в следующих главах. Программист может указывать, какие функции или процедуры являются рекурсивными, а какие нет, или же достаточно развитый компилятор может сам выяснять это для себя. Однако все это неидеальные решения, так как, даже если какой-то процесс может быть представлен итеративно, его рекурсивное описание может оказаться значительно более простым. Поэтому было бы выгодно сочетать легкость написания рекурсивных описаний с высокой скоростью выполнения итеративного процесса, и лучший способ достижения этой цели состоит в изменении традиционной машинной аппаратуры. (Это реализовано по крайней мере в одной машине — см. [12, 13].) Теоретическое изучение соотношений между рекурсивными и итеративными программами показывает, что иногда возможно переводить рекурсивное описание в эквивалентную итеративную программу; такое преобразование описывается более подробно в гл. 4.

1.5. Рекурсия в функциональном программировании

Мы видели, что для некоторых задач рекурсия необходима в силу рекурсивной природы процесса или же из-за рекурсивной структуры обрабатываемых данных, тогда как в других задачах (в частности, в численных) рекурсия используется только для

удобства, позволяя более сжато описывать подлежащий выполнению алгоритм. Еще одна ситуация, когда применение рекурсии является обязательным, возникает при использовании функциональных языков программирования, которые по самой своей природе не могут обеспечить каких-либо средств для явной итерации.

Большинство языков программирования научных расчетов является «операторными» языками, в которых программа состоит из последовательности обязательных команд, например:

$$a := b + c$$

напечатать (a)

Показателем возрастания сложности языков программирования является то, что в основной «скелет» команды могут подставляться все более и более сложные выражения. Например, в одном из ранних языков (автокоде Пегас) переменные идентифицировались именами $v1$, $v2$, $v3$ и т. д. и операторы присваивания имели простую форму

$$v1 = v2 \oslash v3$$

где $\oslash$ означает одну из четырех арифметических операций $+$, $-$, $\times$ или $/$. В более развитом языке (автокоде Меркурий) допускались простые скобочные выражения, например:

$$A = (B + C)/(F + G) - G$$

а в современных языках допускаются вложенные скобки, например:

$$ИТОГ = (СУММА + СЧЕТЧИК)^{*} РАЗНОСТЬ - (АИТОГ + БИТОГ)/3 \cdot 14159$$

Язык CPL [3] доводит эту тенденцию до ее логического завершения, разрешая писать выражения слева от знака присваивания, например [1]):

$$(x = 0 \to a,\ b) := (((a + b)/c - d)e + f)$$

В рамках операторного языка повторение обеспечивается непосредственными командами, например, на ФОРТРАНе:

```
      DO 10 I=1,5
      A=A+B(I)
10    CONTINUE
```

[1]) В зависимости от значения x либо переменной a, либо переменной b присваивается значение выражения, записанного справа от знака присваивания. — *Прим. перев.*

или на АЛГОЛе:

for $x := a$ **step** b **until** c **do**

begin . . . . . end

Операторный язык требует, чтобы программист явно представил последовательность выполнения своей программы. Часто эта последовательность может оказаться несущественной; программист мог бы потребовать, чтобы выполнились некоторые действия, не касаясь порядка, в котором они должны производиться. При однопроцессорной машине эта необходимость указания последовательности только причиняет неудобство, но при многопроцессорной машине она является настоящей обузой, поскольку мешает программисту извлекать выгоду из того факта, что несколько работ могут выполняться одновременно. (По этой причине язык PL/1 включает описательные предложения, в которых указываются требуемые последовательностные соотношения между блоками программы.) Впрочем, даже в языке с командной структурой последовательность действий не всегда оговаривается явно. Например, рассмотрим следующие два варианта:

(а) $\mathrm{TEMP1} = B + C$
$\mathrm{TEMP2} = E + F$
$G = \mathrm{TEMP1} * \mathrm{TEMP2}$

(б) $G = (B + C) * (E + F)$

Если нас не интересуют значения TEMP1 и TEMP2, то это эквивалентные части программы. Однако во втором случае последовательность действий подразумевается отношением старшинства между операциями + и * (умножение), а также объединяющей способностью скобок. *Функциональное программирование* — это метод, в котором такая тенденция доведена до предела, так что последовательность действий совсем не оговаривается явно. Сама программа становится функцией, аргументами которой являются входные данные, а значением — искомый результат вычислений. Например, программа:

$$A = 0 \cdot 5$$
$$B = EXP(-A*A)$$
$$НАПЕЧАТАТЬ\ (B)$$

могла бы быть представлена функционально как функция от A

$$НАПЕЧАТАТЬ\ (EXP(-A*A))$$

применяемая к аргументу 0,5; в λ-обозначениях это записывается так:

$$\lambda A.(НАПЕЧАТАТЬ\ (EXP(-A*A)))\,[0 \cdot 5]$$

Здесь опять последовательность действий не оговорена явно. Прежде чем можно будет применять функцию НАПЕЧАТАТЬ, необходимо вычислить ее аргумент; это включает в себя вычисление присутствующей в аргументе функции EXP, и таким образом получается правильная последовательность действий.

Вычислением аргументов функции перед применением этой функции определяется простое следование действий. Однако если вся программа представляется единой функцией, то как можно добиться какого-либо повторения? Это можно сделать, применяя рекурсию: в функциональных программах рекурсия играет ту же роль, что и повторение в программах, состоящих из команд. Например, вычисление factorial(3) в соответствии с рекурсивным определением влечет за собой вычисление factorial(2), которое в свою очередь вызывает вычисление factorial(1), так что повторное употребление в описании обеспечивает желаемое повторение при выполнении.

1.6. Подсчет с помощью рекурсии

Обычная форма итерации включает известное число повторений. Как правило, это программируется с помощью простого счетчика, но можно использовать рекурсию как способ избежать явного подсчета. Например, предположим, что на некотором этапе компиляции мы хотим исключить из рассмотрения все, что находится после какой-то открывающей скобки вплоть до соответствующей закрывающей скобки (допускаются вложенные скобки). Обычно мы стали бы подсчитывать значения +1 для открывающей скобки, —1 для закрывающей скобки и игнорировали бы все остальные символы, пока в счетчик не попадет значение нуль. Если система программирования разрешает использование рекурсивных подпрограмм, то при достижении первой открывающей скобки мы можем вызвать подпрограмму со следующей спецификацией:

1: *Читать следующий символ*
Если это „(“, то вызвать ту же подпрограмму рекурсивно
Если это „)“, то выйти из подпрограммы, в противном случае уйти на метку 1.

Таким образом, подпрограмма будет выполняться до тех пор, пока не прочтет соответствующую закрывающую скобку; это приведет к заключительной «раскрутке» рекурсии и к передаче управления исходной программе.

1.7. Полезна ли рекурсия?

Ответ на этот вопрос должен зависеть от того, к какой задаче мы собираемся применять рекурсию. Если мы хотим обрабатывать информацию, структура которой описана рекурсивно,

то применение рекурсивных методов является более или менее обязательным. Пусть всякий, кто сомневается в этом, попытается написать программу копирования списковой структуры, не прибегая к применению рекурсии. Если задача касается процесса, рекурсивного по самой своей природе, то рекурсия опять-таки обязательна, хотя и может быть замаскирована специальными мерами. Поставленный вопрос приобретает смысл только тогда, когда рассматривается применение процесса, не являющегося существенно рекурсивным, к данным, не имеющим рекурсивной внутренней структуры. В таком случае ответ должен зависеть в какой-то степени от того, стремимся ли мы к машинной эффективности или к удобству для пользователя. Например, любой программист счел бы очевидным вычисление факториальной функции с помощью итеративного цикла, скажем на АЛГОЛе:

```
integer procedure Factorial (n); value n; integer n;
begin real f; integer i; f := 1;
   for i := 1 step 1 until n do f := f × i; Factorial := f
end;
```

Однако тот, кто не занимался программированием, пришел бы к выводу, что рекурсивное описание ближе к привычной традиционной математике, и вполне мог бы предпочесть такую запись:

```
integer procedure Factorial (n); value n; integer n;
  Factorial := if n = 0 then 1 else n × Factorial (n — 1);
```

(Вероятно, он предпочел бы опустить несущественную (с его точки зрения) спецификацию «**value** *n*; **integer** *n*», но мы не можем следовать здесь такой линии поведения. Впрочем, см. [3, 14].)

Чтобы нас не упрекали в искусственности этого примера, рассмотрим описанный ранее алгоритм НОД отыскания наибольшего общего делителя двух чисел. На АЛГОЛе можно написать следующие программы:

НЕРЕКУРСИВНАЯ

```
integer procedure НОД (n, m); value n, m; integer n, m;
begin integer n1, m1, p;
   if n > m then begin n1 := n; m1 := m end
            else begin n1 := m; m1 := n end;
L: if Ост(n1, m1) = 0 then НОД := m1
                      else begin
                           p := m1; m1 := Ост(n1, m1);
                           n1 := p; go to L
                           end;
end;
```

РЕКУРСИВНАЯ

integer procedure *НОД*(n, m); **value** n, m; **integer** n, m;
НОД := **if** $m > n$ **then** *НОД*(m, n) **else if** $m = 0$ **then** n
else *НОД* $(m,$ *Ост*$(n, m))$;

Рекурсивную программу можно писать почти прямо по описанию алгоритма, тогда как нерекурсивный вариант требует определенных навыков программирования. Применением рекурсии обеспечивается не только более простое написание программы, но и гораздо более легкое понимание программы пользователями, не участвовавшими в ее разработке.

Можно привести два довода против применения рекурсивной программы. Первый состоит в том, что на большинстве машин рекурсивные программы выполняются существенно медленнее, чем нерекурсивные, из-за расходов времени, связанных с организаций стека. Второй довод состоит в том, что если программа содержит ошибки, то ее отладка может оказаться весьма затруднительной, особенно если используется глубокая рекурсия. Трудность отладки рекурсивных программ всегда будет оставаться неудобством их применения. При всей очевидной важности вопроса о машинной эффективности он все же менее существен, если рассматривать его в перспективе. Принципы конструирования машин могут изменяться, и если рекурсивные методы будут признаны полезными, то, вероятно, мы увидим новое поколение вычислительных машин, на которых эффективность рекурсии будет достигаться применением специальной аппаратуры. Поэтому следует тщательно различать методы, которые являются полезными, хотя и неэффективными, с одной стороны, и бесполезные методы — с другой.

Разумеется, нельзя рекомендовать применять рекурсию повсеместно. Программист должен оценить, насколько целесообразно облегчать работу по написанию программы, подвергая себя при этом опасности усложнить отладку и резко увеличить время счета. В следующей главе мы приводим примеры как задач, для которых результаты сравнения, несомненно, в пользу рекурсивных методов, так и задач, для которых целесообразность рекурсии менее очевидна.

СПИСОК ЛИТЕРАТУРЫ

[1] Gill S., Ann. Rev. Automatic Programming, v. 1, p. 180, ed. R. Goodman, Pergamon, 1960.
[2] McCarthy J., Recursive functions of symbolic expressions and their computation by machine, *Commun. Assoc. Computing Machinery*, 3, № 4 (1960), 184.
[3] Barron D, W., Strachey C., Programming: ch. 3 of «Advances in programming and non-numerical computation», ed. L. Fox, Pergamon, 1966.
[4] Church A., The calculi of lambda conversion, Princeton University Press, 1941.

[5] Woodward P. M., List programming, ch. 2 of «Advances in programming and non-numerical computation», ed. L. Fox, Pergamon, 1966.
[6] Newell A. (ed.), Information processing language-V manual, Prentice-Hall, 1961.
[7] McCarthy J., et al., LISP 1·5 programmer's manual, M. I. T. Press, 1965.
[8] Perlis A. J., Samelson K. (ed.), Preliminary report — International algebraic language, *Commun. Assoc. Computing Machinery*, **1**, № 12 (1958), 8.
[9] Naur P. (ed.), Revised report on the algorithmic language ALGOL 60, *Commun. Assoc. Computing Machinery*, **6**, № 1 (1963), 1; *Computer J.*, **5**, № 4 (1963), 349. (Русский перевод: Наур П. (под ред.), Алгоритмический язык АЛГОЛ-60, Пересмотренное сообщение, «Мир», М., 1965.)
[10] Dijkstra E. W., A primer of ALGOL programming, Academic Press, 1962.
[11] Rice A. G., *Commun. Assoc. Computing Machinery*, **8**, № 2 (1965), 114.
[12] Burroughs Corporation, Operational characteristics of the processors for the Burroughs B5000.
[13] Barton R. S., A new approach to the functional design of a digital computer, Proc. Western Joint Computer Conf., 1961, p. 393.
[14] Wirth N., A generalisation of ALGOL, *Commun. Assoc. Computing Machinery*, **6**, № 9 (1963), 547.
[15] Higman B., A comparative study of programming languages, Macdonald, 1967. (Русский перевод: Хигман Б., Сравнительное изучение языков программирования, «Мир», М., 1974.)

2.1. Приложения к вычислениям

2.1.1. *Решение уравнений*

Вычисление квадратного корня является типичным примером общего класса итеративных процессов, описываемых уравнениями вида

$$x_{n+1} = F(x_n)$$

В таких случаях всегда определяются дискриминант d и допуск t, и решение имеет вид $S(x_0)$, где x_0 — начальное значение и

$$S(x) = [d < t \to x,\ S(F(x))]$$

В качестве конкретного примера рассмотрим вычисление квадратного корня. Итеративная последовательность для квадратного корня из y имеет вид

$$x_{n+1} = \frac{1}{2}(x_n + y/x_n)$$

Если мы рассматриваем ограниченный интервал $y < 1$, то хорошим начальным значением является $^1/_2(1+y)$ и рекурсивное определение имеет следующий вид:

Квадратный корень $(y) = S(y, {}^1/_2(1+y))$

$$S(y, x) = \left[(y/x - x) < \varepsilon \to x,\ S\left(y, \frac{1}{2}(x + y/x)\right)\right]$$

На АЛГОЛе получаем следующие программы [1]):

РЕКУРСИВНАЯ

```
real procedure Квадратный корень(y); value y;
begin real procedure S(y, x); value y, x;
      begin S := if (y/x - x) < 2 × epsilon then x
                 else S(y, (x + y/x)/2)
      end;
Квадратный корень := S(y, (1 + y)/2)
end;
```

[1]) При переводе примеров используется вариант языка АЛГОЛ-60, в котором допускаются идентификаторы, содержащие буквы русского языка и пробелы. — *Прим. перев.*

НЕРЕКУРСИВНАЯ

```
real procedure Квадратный корень(y); value y;
begin real x;
      x:=(1 + y)/2;
L: if (y/x — x) < 2 × epsilon then Квадратный корень:=x
   else begin x:=(x + y/x)/2;
              go to L
        end;
end;
```

Тем не менее применение рекурсии для такого рода задач не является естественным, поскольку это явные итеративные процедуры.

2.1.2. *Рекуррентные соотношения*

С первого взгляда кажется, что трехчленные рекуррентные соотношения (типа рекуррентного соотношения для функций Бесселя) лучше всего представлять в виде рекурсивных программ:

$$J(n, x) = [(n = 0) \to J_0, \quad (n = 1) \to J_1, \quad (A \times J(n-1, x) + B \times J(n-2, x))]$$

Однако такая программа вычисляет значение $J(n, x)$ весьма неэффективно, так как процесс ее работы включает приблизительно 2^{n-2} шагов вместо $(n-1)$ шагов, которые потребовались бы при самом эффективном методе. Например, для вычисления $J(5, x)$ при заданных $J(0, x)$ и $J(1, x)$ очевидная последовательность счета состоит в том, чтобы вычислять по очереди $J(2, x)$, $J(3, x)$, $J(4, x)$ и $J(5, x)$. Однако если пользоваться приведенным выше описанием, то мы получаем (в сокращенной записи)

$$J_5 = AJ_4 + BJ_3$$
$$J_4 = AJ_3 + BJ_2$$
$$J_3 = AJ_2 + BJ_1$$
$$J_2 = AJ_1 + BJ_0$$

Значения J_2 и J_3 вычислялись бы дважды, поэтому потребовалось бы 8 шагов.

Маккарти указал следующий способ повышения экономичности программы при сохранении рекурсивного описания. Записываем (по-прежнему в сокращенных обозначениях)

$$J(n, x) = JA(n, 1, J_0, J_1)$$

где $JA(n, m, a, b) = [(n = 0) \rightarrow a,$
$(m = n) \rightarrow b,$
$JA(n, m + 1, b, Ab + Ba)$

При этом $J(5, x) = JA(5, 1, J_0, J_1)$
$= JA(5, 2, J_1, J_2)$, так как $J_2 = AJ_1 + BJ_0$
$= JA(5, 3, J_2, J_3)$
$= JA(5, 4, J_3, J_4)$
$= JA(5, 5, J_4, J_5)$
$= J_5$

Заметим, что эффективность достигается удачным использованием счетчика. Это описание ближе к итеративному, чем к рекурсивному. (Мы вернемся к этому вопросу в гл. 4, где будет рассматриваться связь между итеративными и рекурсивными процессами.)

Однако существует более общий способ повышения эффективности таких процессов при сохранении истинно рекурсивной структуры описания. Этот способ состоит в использовании известного программистского приема — просмотра таблиц. Применительно к вычислению функции Бесселя мы организовали бы таблицу тех значений $J(n, x)$, которые уже вычислены, и каждый раз, когда потребовалось какое-то значение $J(n, x)$, проверяли бы сначала, не находится ли оно уже в таблице. Этим обеспечивается фактическое вычисление только нужных значений функций, причем каждое значение вычисляется только один раз. Сохраняется возможность неэффективности за счет линейного просмотра таблицы, но она не имеет прямого отношения к рекурсивности процедур и может быть устранена различными известными методами, например применением перемешанной таблицы. Пример использования этого приема приводится в разд. 2.1.4. Могло бы показаться, что легче написать нерекурсивную программу, чем организовывать просмотр таблицы, но имеется возможность написать на языке ЛИСП такую программу, которая обрабатывает рекурсивное описание любой функции на ЛИСПе, преобразуя его так, чтобы ни один аргумент не вычислялся более одного раза. (Для языка АЛГОЛ такая процедура невозможна, так как программа на АЛГОЛе не может генерировать другую программу.)

2.1.3. *Приближенное интегрирование*

Рассматриваемая здесь задача состоит в вычислении определенного интеграла на некотором интервале с соблюдением заданной точности; интервал делится на части, и формула

квадратуры применяется по очереди к каждому подинтервалу. Точность приближения контролируется путем применения формулы интегрирования к заданному интервалу с последующим делением интервала пополам, после чего формула интегрирования применяется по очереди к каждой половине интервала и полученные два результата складываются. Таким образом, если $G(a, b, f)$ — функция, являющаяся приближенным значением

$$\int_a^b f(x)\,dx$$

и мы хотим вычислить

$$\int_X^Y F(x)\,dx$$

то мы вычисляем $G(X, Y, F)$, $G(X, Z, F)$ и $G(Z, Y, F)$, где $Z = {}^1/_2(X + Y)$. Если разность

$$G(X, Z, F) + G(Z, Y, F) - G(X, Y, F)$$

меньше, чем некоторое заданное значение, то результат считается приемлемым, в противнм случае та же самая процедура применяется по очереди к каждой половине интервала. Этот алгоритм удобно выразить рекурсивно с помощью следующего описания:

$$S(x, y, f) = Mod\,((G(x, {}^1/_2(x + y), f) + G({}^1/_2(x + y), y, f) - G(x, y, f))/G(x, y, f))$$

$$I(x, y, f) = [S(x, y, f) < D \to G(x, y, f), I(x, {}^1/_2(x + y), f) + I({}^1/_2(x + y), y, f)]$$

Здесь D — максимальная допустимая ошибка; значение функции $I(x, y, f)$ является значением

$$\int_x^y F(z)\,dz$$

с заданной точностью.

При таком процессе вычисление интеграла на отдельных подинтервалах выполнялось бы более чем по одному разу, но можно избежать этого, применив уже упомянутые приемы. На практике было бы необходимо также включить в программу тест для заблаговременного выявления ситуаций, в которых процесс окажется бесконечным.

На АЛГОЛе программа интегрирования принимает следующий вид:

```
real procedure I(x, y, f);
value x, y; real x, y; real procedure f;
begin comment предполагается наличие процедуры Gauss(a,
              b, g) для вычисления интеграла g(z)dz
              в интервале от a до b по формуле ква-
              дратуры;
      real G; G:=Gauss(x, y, f);
          begin real procedure S(a, b, f);
                value a, b; real a, b; real procedure f;
                begin real z, c;
                      c:=(a+b)/2;
                      S:=abs((Gauss(a, c, f)
                        -Gauss(c, b, f)-G)/G);
                end;
      if S(x, y, f)<1_10-5 then I:=G
                           else I:=I(x, (x+y)/2, f)
                             +I((x+y)/2, y, f)
          end;
end;
```

2.1.4. *Теория чисел*

Мы приводим два примера из области теории чисел: вычисление простых множителей и отыскание разбиений.

Простые множители. Алгоритм разложения целого числа N на простые множители является очевидным:

(1) Если N — четное, то один из множителей равен двум; применяем тот же процесс к значению $N/2$.
(2) Если N делится на 3, исключаем этот множитель и повторяем тот же процесс для оставшегося значения $N/3$.
(3) Если N делится на 5, исключаем этот множитель и применяем тот же процесс к оставшемуся значению $N/5$.
(4) ...

Могло бы показаться, что необходимо выполнять шаги вида (2) и (3), используя только простые числа. На самом деле можно повторять процесс, перебирая нечетные числа, поскольку, например, кажущийся множитель 9 к моменту соответствующей проверки уже будет исключен как 3×3. Приведем теперь рекурсивную программу, реализующую этот алгоритм. Предположим, что результатом применения функции *СПМ* (n) является список простых множителей числа n, например:

$$СПМ(42) = (2, 3, 7)$$

Предположим также, что предусмотрены функция $Cons(x, y)$, которая добавляет элемент x к списку y, и функция $Список(x)$,

которая организует список, состоящий из одного элемента x. Тогда [1])

$$СПМ(n) = [(Ост(n, 2) = 0 \to Cons(2, СПМ(n/2)), ПМ1(n, 3)]$$

где $ПМ1(n, m) = [(n = 1) \to ПУСТО,$
$(n < m^2) \to Список(n),$
$Ост(n, m) = 0 \to Cons(m, ПМ1(n/m, m)),$
$ПМ1(n, m + 2)]$

Например:

$$\begin{aligned} СПМ(330) &= (2, СПМ(165)) = (2, ПМ1(165, 3)) \\ &= (2, 3, ПМ1(55, 3)) = (2, 3, ПМ1(55, 5)) \\ &= (2, 3, 5, ПМ1(11, 5)) \\ &= (2, 3, 5, 11) \end{aligned}$$

Разбиения. Разбиениями целого числа n называются способы представления числа n в виде суммы целых чисел. Например, разбиениями числа 5 являются

$$5,\ 4+1,\ 3+2,\ 3+1+1,\ 2+2+1,\ 2+1+1+1,$$
$$1+1+1+1+1$$

Представляет интерес подсчет числа различных разбиений произвольного целого n. Для этого удобно воспользоваться функцией q_{mn}, значением которой является число способов представления целого m в виде суммы, при условии, что каждое слагаемое не превосходит n. Число разбиений целого N равно q_{NN}.

Значение q_{mn} может быть определено следующим образом. Имеем

$$\begin{aligned} q_{1,n} &= 1 && \text{для всех } n \\ q_{m,1} &= 1 && \text{для всех } m \\ q_{m,n} &= q_{m,m}, && \text{если } m < n \\ q_{m,m} &= 1 + q_{m,m-1} \\ q_{m,n} &= q_{m,n-1} + q_{m-n,n}, && \text{если } m > n \end{aligned}$$

Эти соотношения объединяются в следующем рекурсивном определении:

$$q_{m,n} = [(m = 1) \vee (n = 1) \to 1,$$
$$(m \leqslant n) \to 1 + q_{m,m-1},$$
$$(q_{m,n-1} + q_{m-n,n})]$$

---

[1]) $Ост(n, m)$ — остаток от деления n на m; $ПУСТО$ — пустой список. — *Прим. перев.*

Легко написать соответствующую рекурсивную процедуру на АЛГОЛе:

```
integer procedure q(m, n); value m, n; integer m, n;
  begin q:=if m=1 ∨ n=1 then 1 else
                if m ≦ n then 1 + q(m, m — 1)
                else (q(m, n — 1) + q(m — n, n))
  end;
```

В процессе такого вычисления $q(m,n)$ некоторые другие значения q будут вычисляться многократно, что приводит к неэффективности. Можно написать нерекурсивную процедуру вычисления $q(m,n)$ симметричным способом:

```
integer procedure q(m, n); value m, n; integer m, n;
begin integer k, l; integer array Q[(1 : m), (1 : n)];
   for k:=1 step 1 until m do
   begin for l:=1 step 1 until n do
      Q[k, l]:=if k=1 ∨ l=1 then 1 else
                  if k ≦ l then 1 + Q[k, l — 1]
                  else (Q[k, l — 1] + Q[k — l, l])
   end;
                              q:=Q[m, n]
   end;
```

В процессе вычисления q_{mn} по этой процедуре ни одно значение q не вычисляется более чем один раз. Однако некоторые значения q вычисляются без надобности, хотя в отдельных случаях и неочевидно, сколько получается ненужных значений q.

Приведем теперь программу на языке ЛИСП (заимствованную у Маккарти). Эта программа организует таблицу значений q и вычисляет только те значения, которые действительно нужны. Читателям, совсем незнакомым с языком ЛИСП, следует пропустить дальнейшую часть п. 2.1.4.

Программа состоит из нескольких описаний функций. В ней используется список „известные“, в котором перечисляются все уже вычисленные значения q. Этот список имеет вид

$$((m,\ n,\ q_{mn})\ \ldots)$$

Вот описания функций:

```
(1) присутствие [m; n; известные] = ~ null [известные]
                                  ∧ [[eq [caar [известные]; m]
                                  ∧ eq [cadar [известные]; n]]
                                  ∨ присутствие [m; n; cdr
                                                 [известные]]]
```

Эта функция принимает значение **true**, если q_{mn} содержится в списке „известные“, и значение **false**, если q_{mn} отсутствует в списке или если список не существует. Описание является достаточно очевидным.

(2) $$\begin{aligned}знач\,[m;\ n;\ известные] = [eq\,[caar\,[известные];\ m]\\ \wedge\ eq\,[cadar\,[известные];\ n]\rightarrow\\ \rightarrow caddar\,[известные];\\ T\rightarrow знач\,[m;\ n;\ cdr\,[известные]]]\end{aligned}$$

Эта функция дает значение q_{mn} из списка; данное описание тоже не содержит неясностей.

(3) $$f1\,[v] = cons\,[список\,[m;\ n;\ v];\ известные]$$

Эта функция добавляет к списку „известные“ новый элемент. Заметим, что в этом определении m, n и *известные* — свободные переменные.

(4) $$f2\,[z] = знач\,[m-n;\ n;\ z] + знач\,[m;\ n-1;\ z]$$

Эта функция строит новое значение q по двум значениям q, содержащимся в списке.

(5) $$след\,[m;\ n;\ известные]$$

Эта функция служит для получения нового списка „известные“, который включает значение $q_{m,n}$ и все другие значения q, появившиеся в процессе вычисления q_{mn} и не внесенные ранее в список „известные“.

Легко получаем описание функции:

$$\begin{aligned}&след\,[m;\ n;\ известные]\\ &\quad = [присутствие\,[m;\ n;\ известные]\rightarrow известные;\\ &\qquad T\rightarrow f1\,[[m=1\vee n=1\vee m=0\rightarrow 1;\\ &\qquad\qquad m\leqslant n\rightarrow 1 + знач\,[m;\ m-1;\ след\,[m;\ m-1;\\ &\qquad\qquad\qquad известные];\\ &\qquad\qquad T\rightarrow f2\,[след\,[m;\ n-1;\ след\,[m-n;\ n;\ известные]]]]]]\end{aligned}$$

Если q_{mn} содержится в списке, то эта функция ничего не изменяет. Если q_{mn} отсутствует в списке, то происходит обращение к функции $f1$, которая добавляет новый элемент к списку известных значений. Аргументом функции $f1$ является значение q_{mn}, и поэтому аргумент для $f1$ выражается в соответствии с исходным описанием q_{mn} с той разницей, что, когда требуется какое-то значение q_{ij}, пишется функция

$$знач\ [i;\ j;\ след\ [i;\ j;\ известные]]$$

Наличие третьего аргумента *след* $[i; j;$ *известные*] означает, что значение q_{ij} будет вычисляться только в том случае, если его

еще нет в списке «известные». Окончательно получаем формулу

$$q[m; n] = знач\ [m; n; след\ [m; n; ПУСТО]]$$

Процесс начинается с применения этой формулы при пустом списке «известные».

2.1.5. *Другие примеры из численного анализа*

Вычисление кратных интегралов уже рассматривалось в гл. 1. Аналогичная ситуация возникает при вычислении полиномов от нескольких переменных. Полином от n переменных $x_1, x_2, \ldots, x_n$ может быть представлен как полином от x_n, коэффициенты которого в свою очередь являются полиномами от $(n-1)$ переменных $x_1, x_2, \ldots, x_{n-1}$. Коэффициенты этих полиномов являются полиномами от $(n-2)$ переменных $x_1, x_2, \ldots, x_{n-2}$ и т. д. Процедура вычисления такого полинома является рекурсивной по существу, так как процесс вычисления коэффициентов полинома включает вычисление других полиномов.

На этом этапе целесообразно упомянуть также получение перестановок, хотя эта задача и не совсем относится к численному анализу. Все перестановки N объектов могут быть получены, если брать по очереди каждый объект и помещать его перед всеми перестановками оставшихся $(N-1)$ объектов. Таким образом, можно описать перестановки какого-то набора объектов с помощью перестановок меньшего числа объектов. Другими словами, имеется возможность написать рекурсивную программу получения перестановок N элементов, которая явно выполняет перестановки при $N = 2$ и обращается к себе рекурсивно при $N > 2$. Этот пример еще раз иллюстрирует изящество рекурсивных описаний; любое нерекурсивное описание данного процесса оказалось бы гораздо более сложным.

2.2. Рекурсия в компиляторах

Написание компиляторов является областью широкого применения рекурсии. В сущности рекурсия в настоящее время является для разработчиков компиляторов стандартным приемом. Известно много примеров использования рекурсии в компиляторах. Мы приведем здесь лишь несколько примеров, чтобы пояснить применяемые методы.

2.2.1. *Условные операторы*

Предположим, что в языке, состоящем из командных операторов, предусмотрена конструкция

ЕСЛИ B, S1, S2

которая вызывает выполнение оператора *S*1, если условие B является истиной, или выполнение оператора *S*2 в противном случае. После компиляции эта конструкция кодируется следующим образом:

```
        Переход на L1, если не B
        S1
        Переход на L2
L1:     S2
L2:     ...
```

Компилятор, прочитав строку, начинающуюся со слова ЕСЛИ, вызвал бы подпрограмму со следующей спецификацией:

```
Компилировать код для вычисления условия не B
Завести новую метку L1
Компилировать „Переход на L1, если истина"
Компилировать оператор S1
Завести метку L2
Компилировать „Переход на L2
                L1:"
Компилировать оператор S2
Компилировать „L2:"
Выход
```

Теперь предположим, что мы хотим уметь обрабатывать вложенные условия:

*ЕСЛИ B*1, *ЕСЛИ B*2, *S*1, *S*2, *S*3

или *ЕСЛИ B*1, *S*1, *ЕСЛИ B*2, *S*2, *S*3

(эти условия означают

*ЕСЛИ B*1, (*ЕСЛИ B*2, *S*1, *S*2), *S*3

и *ЕСЛИ B*1, *S*1, (*ЕСЛИ B*2, *S*2, *S*3) соответственно).

Мы можем достичь этой цели, написав рекурсивную подпрограмму под названием COMPILE, результатом выполнения которой является компиляция одного оператора. Вот ее спецификация:

```
COMPILE: Если оператор начинается с „ЕСЛИ", то переход
         на LL
         Компилировать оператор (до запятой или до
         конца строки)
         Выход
LL:      Читать условие (до запятой)
         Компилировать код для вычисления обратного
         условия
         Завести метки L1, L2
```

Компилировать „Переход на L1, если истина“
Обратиться к COMPILE
Компилировать „Переход на L2
L1:“
Обратиться к COMPILE
Компилировать „L2:“
Выход

В случае простого условного оператора эта программа действует точно так же, как предыдущая. Но если задан оператор

ЕСЛИ B1, ЕСЛИ B2, S1, S2, S3

то после компиляции кода для условия произойдет рекурсивное обращение к COMPILE, при котором обнаружится второе слово ЕСЛИ и начнется компиляция внутреннего условного оператора *ЕСЛИ B2, S1, S2*. После завершения этой работы произойдет возврат к предыдущему выполнению подпрограммы COMPILE и будет закончена компиляция оставшейся части внешнего оператора ЕСЛИ. При рекурсивном обращении будут заведены новые метки для внутреннего оператора ЕСЛИ; в результате получится код следующего вида:

Переход на L1, если не B1
Переход на L3, если не B2
S1
Переход на L4
L3: S2
L4: Переход на L2
L1: S3
L2: ...

Можно обобщить эту схему так, чтобы допускались составные операторы. Теперь задача состоит в том, чтобы можно было заменять *S1* в операторе ЕСЛИ на

(S1, S2, S3, ..., Sn)

Например, запись

ЕСЛИ B1, (S1, S2), (S3, S4, S5)

после компиляции была бы переведена в такой вид:

Переход на L1, если не B1
S1
S2
Переход на L2
L1: S3
S4
S5
L2:

Для этого мы модифицируем подпрограмму COMPILE следующим образом:

COMPILE: *Если оператор начинается с „(“, то переход на LX*
Если оператор начинается с „ЕСЛИ“, то переход на LL
Компилировать оператор до запятой или до „)“ или до конца строки
Если встретилась запятая или кончилась строка, то установить Sw = 0, в противном случае установить Sw = 1
Выход

LX: *Обратиться к CSCOMP*
Выход

LL: *Как ранее*

CSCOMP — это рекурсивная подпрограмма со следующей структурой:

CSCOMP: *Обратиться к COMPILE*

Если Sw = 0, то переход на CSCOMP, в противном случае Выход.

Таким образом, если при чтении встретилась открывающая скобка, то управление «перехватывается» программой CSCOMP до тех пор, пока не встретится соответствующая закрывающая скобка. (Заметим, что программа CSCOMP является косвенно рекурсивной, так как она обращается к программе COMPILE, которая может в свою очередь обратиться к CSCOMP.)

Если на входе задан оператор

ЕСЛИ B, (*S*1, *S*2), *S*3

то после рекурсивного обращения к COMPILE для обработки первого ответвления выход из программы COMPILE не произойдет до тех пор, пока не встретится соответствующая закрывающая скобка. Итак, сравнительно простая рекурсивная подпрограмма справится с обработкой сколь угодно сложной суперпозиции условных и составных операторов [1]).

2.2.2. *Синтаксический анализ*

Во многих случаях синтаксис языка имеет рекурсивную структуру. Например, в английской грамматике именная группа

---

[1]) Это верно, если *Sw* — локальная величина процедуры COMPILE. — *Прим. ред.*

может быть определена как

the ⟨*существительное*⟩
или *the* ⟨*прилагательное*⟩⟨*существительное*⟩
или ⟨*именная группа*⟩⟨*именная группа*⟩⟨*глагол*⟩

Таким образом, обе конструкции «the pink elephant» и «the pink elephant the man sees» являются именными группами[1]). Мы замечаем, что третий вариант в приведенных выше определениях описывает именную группу через самое себя. Метод рекурсивного описания может успешно применяться и к синтаксису языка программирования. Общеизвестным примером является формальное описание языка АЛГОЛ-60 [2], которое дано в записи, получившей известность под названием нормальной формы Бэкуса. Эта форма описания применима к широкому классу искусственных языков, известному под названием «языки непосредственных составляющих» (см. [3]). Проиллюстрируем это на примере описания десятичного числа. Запись

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

определяет класс ⟨*цифра*⟩ как состоящий из вариантов 0, 1, 2, ..., 9. (Символ : : = можно читать как «определяется как», вертикальную черту можно читать как «или»). Теперь можно определить более сложные конструкции.

⟨*целое без знака*⟩ : : = ⟨*цифра*⟩ | ⟨*целое без знака*⟩⟨*цифра*⟩

Это рекурсивное описание целых чисел. Например, 123 — целое без знака 12, за которым следует цифра 3; 12 — целое без знака 1, за которым следует цифра 2; 1 — цифра.

⟨*правильная дробь*⟩ : : = · ⟨*целое без знака*⟩
⟨*десятичное число*⟩ : : = ⟨*целое без знака*⟩ | ⟨*правильная дробь*⟩ | ⟨*целое без знака*⟩ ⟨*правильная дробь*⟩

Другое возможное определение целого без знака могло бы быть таким:

⟨*целое без знака*⟩ : : = ⟨*цифра*⟩ | ⟨*цифра*⟩ | ⟨*целое без знака*⟩

Это определение формально эквивалентно предыдущему. На практике алгоритм, применяемый для синтаксического анализа, обычно предопределяет выбор способа записи рекурсивных описаний. Более сложным примером является описание арифметических выражений.

---

[1]) «Розовый слон», «розовый слон, которого видит человек».

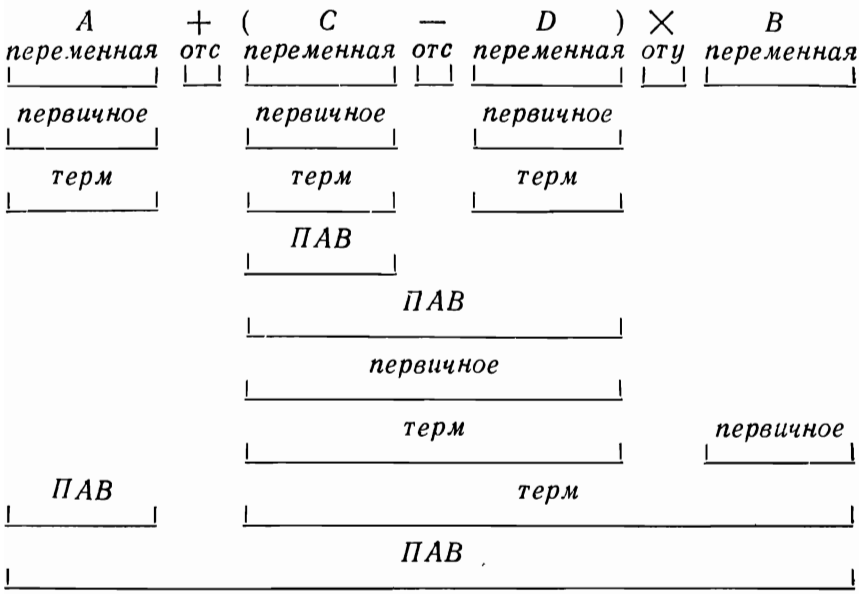
Мы определяем *простое арифметическое выражение (ПАВ)* с помощью следующих конструкций:

⟨отс⟩ : : = + | —
⟨оту⟩ : : = × | /
⟨первичное⟩ : : = ⟨целое без знака⟩ | ⟨переменная⟩ | (⟨ПАВ⟩)
⟨терм⟩ : : = ⟨первичное⟩ | ⟨терм⟩ ⟨оту⟩ ⟨первичное⟩
⟨ПАВ⟩ : : = ⟨терм⟩ | ⟨отс⟩ ⟨терм⟩ | ⟨ПАВ⟩ ⟨отс⟩ ⟨терм⟩

При компиляции возникает проблема распознавания класса, к которому принадлежит данная строка символов. Например, строка

$$A + (C - D) \times B$$

является ПАВ, так как она может быть проанализирована следующим образом:



Так как синтаксис описывается рекурсивно, то неудивительно, что рекурсивный метод хорошо подходит для такого вида анализа.

Метод, который будет сейчас описан, в основном базируется на «Дйаграмме» Айронса [4], хотя существуют и другие программы синтаксического анализа, работающие аналогичным способом (например, см. [5]). «Диаграмма» представляет собой процедуру, при входе в которую задаются указатель для исходной строки и определенная «цель». «Целью» является какой-то

синтаксический класс. Если символы, непосредственно следующие за указателем, образуют пример из этого класса, то результатом является ответ «истина» и указатель продвигается вперед; в противном случае результатом является ответ «ложь» и указатель остается в своем прежнем положении. Этапы анализа приведенного выше выражения показаны на рис. 2.1.

$A + (C - D) \times B$ *Цель: ПАВ*
↑
ПАВ может быть термом:
$A + (C - D) \times B$ *Цель: терм*
↑
Терм может быть первичным
$A + (C - D) \times B$ *Цель: первичное*
↑
„*A*“ является примером из класса „переменная“; первичное может быть переменной, поэтому получаем результат
ИСТИНА
ИСТИНА „терм“ найден
ИСТИНА „ПАВ“ найдено
Можем ли мы найти бóльшую строку, которая также является ПАВ? Пытаемся применить конструкцию ⟨*ПАВ*⟩ ::= ⟨*ПАВ*⟩ ⟨*отс*⟩ ⟨*терм*⟩
$+ (C - D) \times B$ *Цель: отс*
↑
ИСТИНА
$(C - D) \times B$ *Цель: терм*
↑
Терм может быть первичным; „(“ может служить началом первичного, поэтому пытаемся применить конструкцию
⟨*первичное*⟩ ::= (⟨*ПАВ*⟩)
$C - D) \times B$ *Цель: ПАВ*
↑
„C“ является примером из класса „переменная“; первичное может быть переменной, поэтому получаем результат
ИСТИНА

Можем ли мы найти бóльшую строку, которая также является *ПАВ*? Пытаемся применить конструкцию ⟨*ПАВ*⟩ ::= ⟨*ПАВ*⟩ ⟨*отс*⟩ ⟨*терм*⟩
$- D) \times B$ *Цель: отс*
↑
ИСТИНА
$D) \times B$ *Цель: терм*
↑
„*D*“ является примером из класса „переменная“; переменная является первичным и термом, поэтому получаем результат
ИСТИНА
ИСТИНА „*ПАВ*“ найдено

Можем ли мы найти бóльшую строку, которая также является *ПАВ*?
$) \times B$ *Цель: отс*
↑
ЛОЖЬ
Самое большое ПАВ найдено
$) \times B$ *Цель:* „)“
↑

```
    ИСТИНА
ИСТИНА „первичное“ найдено, поэтому „терм“ найден

Можем ли мы найти больший терм?
Единственной возможной конструкцией является
⟨„терм“⟩ ⟨оту⟩ ⟨первичное⟩
    ╳ В Цель: оту
    ↑
    ИСТИНА
    В Цель: первичное
    ↑
    „В“ — это переменная, а следовательно, первичное, поэтому получаем
    результат
    ИСТИНА.
ИСТИНА ПАВ найдено, и выражение полностью проанализировано.
```

Рис. 2.1. Работа «Диаграммы».

На этом рисунке каждый отступ текста представляет рекурсивный вход в «Диаграмму», причем для каждого входа справа показана входная строка.

Можно представить этот анализ в виде очень короткой программы, так как он сводится к простому процессу распознавания, применяемому с большой глубиной рекурсии. Дальнейшие подробности можно найти в статье Айронса [4]. Это очень хороший пример эффективности рекурсивной процедуры. Однако он может служить также иллюстрацией возможной расточительности при необдуманном применении рекурсии. Например, в языке АЛГОЛ-60 идентификатор определяется так:

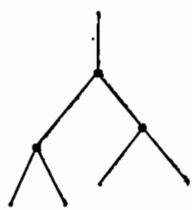
⟨*идентификатор*⟩ ::= ⟨*буква*⟩ | ⟨*идентификатор*⟩ ⟨*буква*⟩ |
⟨*идентификатор*⟩ ⟨*цифра*⟩

Это определение означает, что идентификатор состоит из одного или более символов, из которых первый должен быть буквой, а остальные могут быть буквами или цифрами. Если задано это синтаксическое определение, то «Диаграмма» будет правильно опознавать идентификаторы, но при этом возникнут большие потери при использовании процедуры рекурсивного определения для задачи опознавания, которая может быть решена простым прямым просмотром. На практике перед синтаксическим анализом следует выполнять *лексический анализ*. На этапе лексического анализа такие синтаксические объекты, как идентификаторы, числа и разделители, опознаются при прямом просмотре и передаются для синтаксического анализа как законченные величины. Аналогично, арифметические выражения, удовлетворяющие некоторым правилам, также могут анализироваться прямым просмотром (см. разд. 2.4, а также работу Флойда [10]). Если для заданного языка эти правила выполняются, то можно добиться значительного выигрыша, включив в синтаксис единое

понятие «строка операция — операнд»; после опознавания таких строк к ним применяется отдельная процедура анализа. Не следует автоматически считать преимуществом применение рекурсии, если та же задача может быть решена нерекурсивно.

2.3. Сортировка

На первый взгляд целесообразность применения рекурсивных методов для сортировки кажется несколько сомнительной. Однако существует метод *сортировки по дереву*, который очень изящно описывается в виде рекурсивной процедуры. Предположим, что подлежащие сортировке объекты имеют числовой ключ. Для них организуется дерево, состоящее из вершин; каждая вершина подчинена с помощью указателя некоторой другой вершине и может в свою очередь указывать на две вершины:



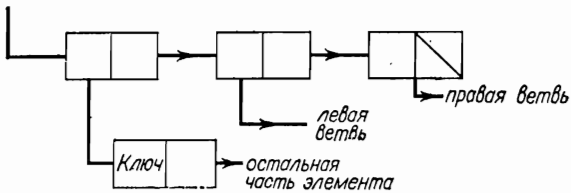
От корня дерева в каждую вершину проходит единственный путь. При сортировке по дереву элементы размещаются на вершинах дерева таким образом, чтобы на любом пути из какой-либо вершины по левой ветви встречались только такие элементы, у которых значения ключа меньше, чем значение ключа для данной вершины. И наоборот, любой путь из какой-либо вершины по правой ветви приводит только к таким элементам, у которых значения ключа больше, чем значение ключа для данной вершины.

На языке ЛИСП программа выполнения сортировки может быть составлена следующим образом. Предполагаем, что подлежащие сортировке объекты представляют собой списки, причем в каждом списке первым элементом является ключ. На вход программы подается список этих списков, т. е.

((*ключ* 1, ...), (*ключ* 2, ...), ..., (*ключ* *n*, ...))

Требуется построить список из тех же списков так, чтобы в нем ключи следовали в порядке возрастания их значений. Сначала определим функцию *добав* с двумя аргументами. Одним аргументом является элемент *x*, другим — частично построенное дерево *z*. Значением функции является дерево, получаемое путем

добавления элемента x к дереву z. Дерево состоит из вершин, каждая из которых представляет список из трех элементов:



Итак, если w — вершина, то ключ элемента из этой вершины определяется по формуле *caar* (w), левая ветвь — это *cadr* (w), правая ветвь — это *caddr* (w). Функция *добав* описывается следующим образом:

$$\begin{aligned} \textit{добав}[x;\ z] = [&\textit{null}[z] \to \textit{список}[x;\ \textit{ПУСТО};\ \textit{ПУСТО}];\\ &\textit{car}[x] < \textit{caar}[z] \to\\ &\quad \textit{список}[\textit{car}[z];\ \textit{добав}[x;\ \textit{cadr}[z]];\ \textit{caddr}[z]];\\ &T \to \textit{список}[\textit{car}[z];\ \textit{cadr}[z];\ \textit{добав}[x;\ \textit{caddr}[z]]]] \end{aligned}$$

Далее определим функцию *construct* $[w;\ v]$, которая добавляет все элементы списка w к дереву v; здесь w — список списков, аналогичный описанному ранее.

$$\begin{aligned} \textit{construct}[w;\ v] = [&\textit{null}[w] \to v;\\ &T \to \textit{construct}[\textit{cdr}[w];\ \textit{добав}[\textit{car}[w];\ v]]] \end{aligned}$$

Это формальное описание означает, что мы добавляем первый элемент списка w к дереву v, а затем применяем функцию construct к оставшейся части списка и к новому дереву. Если исходный (неупорядоченный) список называется *вхсписок*, то, применив функцию

$$\textit{construct}[\textit{вхсписок};\ \textit{ПУСТО}]$$

мы получим дерево упорядочивания элементов списка *вхсписок*.

Заключительный этап состоит в получении линейного упорядоченного списка. Это выполняется с помощью функции *лин* $[a;\ b]$, определяемой следующим образом:

$$\begin{aligned} \textit{лин}[a;\ b] = [&\textit{null}[a] \to b;\\ &T \to \textit{лин}[\textit{cadr}[a];\ \textit{cons}[\textit{car}[a];\\ &\quad \textit{лин}[\textit{caddr}[a];\ b]]]] \end{aligned}$$

В этом описании a — дерево, b — конструируемый линейный список. Если функция $\textit{null}(a)$ принимает значение «истина» (что соответствует отсутствию ответвления от вершины), то список b не изменяется, в противном случае результат выравнивания ле-

вой ветви добавляется к списку, полученному добавлением элемента из данной вершины к результату добавления выровненной правой ветви к списку b. Этот процесс начинается при пустом списке b, т. е. выполняется оператор

$$лин[t;\ ПУСТО]$$

где t — дерево. Дерево преобразуется в упорядоченный список с помощью функции

$$лин[t;\ ПУСТО]$$

Полное выполнение сортировки обеспечивается применением функции

$$sort[вхсписок] = лин[construct[вхсписок;\ ПУСТО];\ ПУСТО]$$

2.4. Обработка алгебраических выражений

Рекурсия широко применяется при обработке символов. В качестве примера мы рассмотрим здесь обработку алгебраических выражений. Эти выражения состоят из наименований переменных, а также из знаков операций и скобок. Мы будем обозначать переменные одиночными буквами. Вот примеры алгебраических выражений:

$$A + B + C*D$$
$$A + (B + C)* D$$
$$((A*B)* D — F)/(G + H)$$

Такие выражения имеют рекурсивную структуру, поскольку все операции, кроме одноместного минуса, применяются к двум операндам. Поэтому выражение можно описать так:

операнд операция операнд

Здесь операнды либо являются переменными, либо в свою очередь представляют собой выражения. Рекурсивная структура становится более очевидной, если писать операции перед операндами. Например, выражение

$$A + B + C*D$$

в таком случае приобретает вид *ПЛЮС* (*A*, *ПЛЮС* (*B*, *УМНОЖИТЬ* (*C*, *D*))).

В обычной записи алгебраических выражений подразумевается старшинство операций умножения по сравнению со сложением. Запись с вынесенными вперед операциями в точности соответствует полной скобочной записи:

$$(A + (B + (C*D)))$$

Если требуется обрабатывать такие выражения, то выгодно выбрать представление в машинной памяти, которое отражает

рекурсивность структуры. Одно из таких представлений сводится к применению списков: выражение представляется в виде списка из трех элементов, соответствующих операции и двум ее операндам. Если операнды являются выражениями, то они представляются как подсписки. Таким образом, выражение

$$A + (B + C)^* D$$

представляется списком

(*ПЛЮС*, *A*, (*УМНОЖИТЬ*, (*ПЛЮС*, *B*, *C*), *D*))

а выражение

$$((A^*B) \uparrow D - F)/(G + H)$$

представляется списком

(*РАЗДЕЛИТЬ*, (*МИНУС*, (*СТЕПЕНЬ*, (*УМНОЖИТЬ*, *A*, *B*), *D*), *F*), (*ПЛЮС*, *G*, *H*))

Предположим, что теперь мы хотим продифференцировать такое выражение. Заметим, что операция дифференцирования является рекурсивной, поскольку

$$\frac{d}{dx}(a + b) = \frac{d}{dx}(a) + \frac{d}{dx}(b)$$

$$\frac{d}{dx}(u^*v) = u^* \frac{d}{dx}(v) + v^* \frac{d}{dx}(u)$$

Если предположить на время, что единственными операциями являются + и *, то программа дифференцирования становится очень простой. Следующая функция на языке ЛИСП порождает производную от выражения *e* относительно переменной *x*:

```
diff[e; x] = [atom[e] ∧ eq[e; x] → 1;
              eq[car[e]; ПЛЮС] → список[ПЛЮС;
                                        diff[cadr[e]]; diff[caddr[e]]];
              eq[car[e]; УМНОЖ] →
                 список[ПЛЮС;
                        список[УМНОЖ; cadr[e]; diff[caddr[e]];
                        список[УМНОЖ; diff[cadr[e]]; caddr[e]]]]
```

Очевидно, что легко обобщить эту программу так, чтобы она обрабатывала и другие операции.

Так как полная скобочная форма арифметического выражения имеет простую рекурсивную структуру, то она может быть преобразована в форму с вынесенными вперед операциями следующим образом. Описываем две рекурсивные подпрограммы OPERATOR и OPERAND.

OPERATOR: *Если очередной символ не является* +, —, × *или* /, *то записать ОШИБКА; в противном*

случае выдать в качестве результата очередной символ и продвинуть на один шаг указатель во входной строке.

OPERAND: Если очередной символ (элемент) является наименованием переменной, то выдать его в качестве результата. В противном случае, если очередной символ не „(", то записать ОШИБКА; если „(", то

Вызвать OPERAND: обозначить результат через R_1

Вызвать OPERATOR: обозначить результат через R_2

Вызвать OPERAND: обозначить результат через R_3

Построить список (R_2, R_1, R_3)

Если следующий символ „)", то выйти из подпрограммы с выдачей полученного списка в качестве результата, в противном случае записать ОШИБКА.

Если все выражение заключено в скобки, то одно обращение к подпрограмме OPERAND приведет к получению результата, представляющего собой список в форме с вынесенными вперед операциями. Программа такого типа описана в работе [6]. Однако полная скобочная запись выражений обычно не применяется. Поэтому мы опишем теперь способ, который пригоден и для бесскобочных выражений.

Представление выражения в виде списковой структуры является, по сути дела, «прямой польской записью», в которой операция записывается перед своими операндами, т. е.

$A + B*C$ записывается в виде $+\ A*BC$

и $(A + B)*C$ записывается в виде $*\ +\ ABC$

Дийкстра [7] предложил простой способ преобразования выражения в «обратную польскую запись» за один последовательный просмотр. Каждой операции ставится в соответствие некоторый приоритет, а именно:

$+, -$	2
$*, /$	3
$\uparrow$	4

(Эти приоритеты отражают старшинство операций умножения и деления по сравнению со сложением и вычитанием и старшинство возведения в степень по сравнению со всеми остальными операциями.) Алгоритм преобразования основан на использовании стека и состоит в следующем:

(1) Проверяется очередной символ во входной строке.
(2) Если это операнд, то он передается в выходную строку.
(3) Если это открывающая скобка, то она заносится в стек с приоритетом нуль.
(4) Если это операция, то ее приоритет сравнивается с приоритетом операции, находящейся на вершине стека. Если приоритет новой операции больше, то эта новая операция заносится в стек. В противном случае берется операция с вершины стека и помещается в выходную строку, после этого повторяется сравнение с новыми верхними элементами стека до тех пор, пока на вершине стека не окажется операция с приоритетом, меньшим, чем у текущей операции, или пока стек не станет пустым. После этого текущая операция заносится в стек.
(5) Если текущий символ во входной строке является закрывающей скобкой, то операции из стека последовательно переносятся в выходную строку до тех пор, пока на вершине стека не появится открывающая скобка; эта открывающая скобка отбрасывается.
(6) Если выражение закончилось, то из стека последовательно переносятся в выходную строку все оставшиеся в нем операции.

Этот процесс является примером синтаксического анализа. Соответствующая грамматика относится к классу грамматик с предшествованием, для которых такой способ нерекурсивного анализа является удобным. Проиллюстрируем это на примере анализа выражения

$$A + B*C + (D + E)* F$$

Входная строка	Выходная строка	Стек (вершина стека справа)
$A + B*C + (D + E)* F$	*пустая*	*пустой*
$+ B*C + (D + E)* F$	A	*пустой*
$B*C + (D + E)* F$	A	$+$
$*C + (D + E)* F$	AB	$+$
$C + (D + E)* F$	AB	$+ *$
$+ (D + E)* F$	ABC	$+ *$
$(D + E)* F$	$ABC* +$	$+$
$D + E)* F$	$ABC* +$	$+ ($
$+ E)* F$	$ABC* + D$	$+ ($
$E)* F$	$ABC* + D$	$+ (+$
$)* F$	$ABC* + DE$	$+ (+$
$* F$	$ABC* + DE +$	$+$
F	$ABC* + DE +$	$+ *$
пустая	$ABC* + DE + F$	$+ *$
	$ABC* + DE + F* +$	*пустой*

Выходная строка представляет собой обратную польскую запись исходного выражения. Однако если мы перепишем ее в обратном порядке, то она станет прямой польской записью, с той разницей, что для вычитания и для возведения в степень операнды будут следовать в обратном порядке. Если задана строка в прямой польской записи, то соответствующая списковая структура генерируется с помощью простой рекурсивной программы.

Опишем подпрограмму «Получение операнда» со следующей спецификацией:

Если очередной элемент во входной строке является наименованием переменной, то он выдается в качестве операнда и указатель во входной строке продвигается вперед. В противном случае очередной элемент должен оказаться операцией. Организуется список из трех элементов, операция заносится в первый элемент списка, и указатель во входной строке продвигается. Затем два раза выполняется «Получение операнда», и полученные два операнда заносятся в список.

Таким образом, если задана входная строка

$$* + ABC$$

то подпрограмма «Получение операнда» работает так:

(1) Организуется список (*, *ПУСТО, ПУСТО*).
(2) Производится обращение к «Получению операнда». Следующий элемент является операцией, поэтому организуется список (+, *ПУСТО, ПУСТО*). Обращение к «Получению операнда». Следующий элемент — это *A*, поэтому он заносится в качестве операнда для +, получаем (+, *A*, *ПУСТО*). Снова обращение к «Получению операнда», в результате имеем (+, *A*, *B*).
(3) Теперь производится возврат на предыдущий уровень рекурсии с получением первого операнда для *; в результате имеем (*, (+, *A*, *B*), *ПУСТО*).
(4) Обращение к «Получению операнда» за вторым операндом для *; в результате имеем (*, (+, *A*, *B*), *C*).

Опять мы имеем дело с очень простой рекурсивной подпрограммой; эта простота является, разумеется, следствием рекурсивной структуры, выбранной нами для представления выражений.

2.5. Системы решения проблем

Рекурсивные методы играют большую роль в системах, предназначенных для моделирования человеческого мышления, т. е. в области, которую иногда называют «искусственным

интеллектом». Для иллюстрации этого утверждения мы рассмотрим здесь в качестве примера «Универсальный решатель задач», авторами которого являются Ньюэл, Шоу и Саймон [8]. С другими примерами можно ознакомиться по книге [9].

Общая программа решения проблем, которую мы будем называть далее GPS (сокращение английского названия «General Problem Solver»), представляет собой пример эвристической программы. Это означает, что GPS в процессе своей работы пытается найти нужный подход к решению проблемы путем пробного применения многих различных методов и сравнения получаемых при этом результатов. Вообще говоря, если задана цель, которую нужно достигнуть, то GPS разлагает ее на подцели и сначала пытается достичь отдельных подцелей. Процесс достижения какой-то подцели может включать дальнейшее подразделение, так что нам видна существенно рекурсивная структура этой системы: подпроцессы идентичны основному процессу, частями которого они являются. Система GPS оперирует символьными объектами, с помощью которых описываются или характеризуются ситуации. Эта система работает также с символами, представляющими различие между парами объектов, и с символами, представляющими операторы, которые применяются к объектам и могут вызывать изменения в этих объектах.

Система GPS может выполнять процессы, связанные с достижением целей трех типов:

(а) *Преобразование*: преобразовать объект a в объект b.
(б) *Сокращение различия*: уменьшить или исключить различие между объектами a и b.
(в) *Применение оператора*: применить оператор q к объекту a.

Эти процессы рекурсивно взаимосвязаны. Процесс преобразования объекта a в объект b состоит в следующем:

(1) Установить различие d между a и b.
(2) Подцель: уменьшить d.
(3) Попытаться достичь подцели, в случае успеха найти новое различие и, если оно не нуль, повторить тот же процесс.

Процесс сокращения различия d состоит в следующем:

(1) Найти оператор q, который подходит для различий типа d.
(2) Подцель: применить q к d.
(3) В случае успеха завершить процесс, в противном случае совершить новую попытку с другим оператором.

Процесс применения оператора q к различию d состоит в следующем:

(1) Проверить выполнение условий применимости q к d.
(2) Если условия выполняются, то применить q; в противном случае
(3) Подцель: преобразовать d в такой объект, который удовлетворяет условиям.
(4) В случае успеха применить q.
(5) В любом случае после применения q завершить процесс, выдав в качестве результата измененное различие d.

Очевидно, что такая совсем простая проблема, как доказательство тождества

$$\sin^2 x + \cos^2 x = \operatorname{tg} x \cdot \operatorname{ctg} x$$

приведет к сложному рекурсивному переплетению описанных процессов, и было бы немыслимо применять GPS без использования рекурсии.

СПИСОК ЛИТЕРАТУРЫ

[1] McCarthy J., Memorandum 32, Artificial Intelligence Project, MIT Computation Center.
[2] Naur P. (ed.), Revised report on the algorithmic language ALGOL 60, *Commun. Assoc. Computing Machinery*, **6**, № 1 (1963), 1; *Computer J.*, **5**, № 4 (1963), 349. (Русский перевод: Наур П. (под ред.), Алгоритмический язык АЛГОЛ-60, Пересмотренное сообщение, «Мир», М., 1965.)
[3] Chomsky A. N., On certain formal properties of grammars, *Information and Control*, **2** (1959), 137. (Русский перевод: Хомский А., О некоторых формальных свойствах грамматик, Кибернетический сборник, вып. 5, ИЛ, 1962, 279—312.)
[4] Irons E. T., The structure and use of the syntax directed compiler, Ann. Rev. Automatic Programming, v. 4, p. 207, ed. R. Goodman, Pergamon, 1963.
[5] Brooker R. A., et al, The compiler-compiler, *Ann. Rev. Automatic Programming*, **4**, p. 229. Ed. R. Goodman, Pergamon, 1963.
[6] Barron D. W., Strachey C., Programming: ch. 2 of «Advances in programming and non-numerical computation», ed. L. Fox, Pergamon, 1966.
[7] Dijkstra E. W., Making an ALGOL translator for the XI. Reprinted, *in* Annu. Rev. Automatic Programming, v. 4, ed. R. Goodman, Pergamon, 1963.
[8] Newell A., Shaw C. J., Simon H. A., Report on a general problem solving program, *in* Information processing, Proceedings of the UNESCO Conference, Paris, 1959, p. 256, UNESCO (Paris), 1960.
[9] Feigenbaum E. A., Feldman J. (ed.), Computers and thought, McGraw-Hill, 1964.
[10] Floyd R. W., On syntax analysis and operator precedence. Report CA-62-2, Massachusetts Computer Associates, 1962.

1.1. Постановка задачи

Реализация рекурсивных процедур сводится в основном к проблеме работы с подпрограммой, содержащей обращение к самой себе. Это означает, что каждому новому обращению к такой подпрограмме должны соответствовать другой набор рабочих регистров и другое место в памяти для запоминания адреса возврата. При входе в подпрограмму нужно отводить для нее новый набор рабочих регистров, а при выходе, прежде чем передавать управление по адресу возврата, следует восстанавливать то состояние рабочих регистров и других ресурсов, которое существовало к моменту соответствующего обращения к подпрограмме. Эти операции выполняются автоматически при «встроенных» рекурсивных системах, однако их приходится программировать явно, если рекурсия реализуется с помощью системы подпрограмм или с помощью метода макропрограммирования. Может оказаться также, что рекурсия реализуется с помощью специальной аппаратуры, но, с точки зрения программиста, нет существенной разницы между аппаратной реализацией и «встроенным» программным обеспечением; и в том и в другом случаях он может программировать рекурсивно, не уделяя никакого внимания проблемам запоминания и восстановления рабочих ячеек, ссылок и т. д.

3.2. Специальные методы

Рассмотрим в качестве типичного примера вычисление определенного интеграла. Соответствующая программа обычно получает три элемента начальных данных: верхний и нижний пределы интегрирования и адрес той подпрограммы, которая вычисляет подынтегральную функцию. Эта последняя подпрограмма часто называется *вспомогательной подпрограммой* и обычно выбирает из сумматора то значение аргумента, для которого требуется вычислить функцию; при выходе из этой подпрограммы значение функции для данного аргумента остается на сумматоре. Для подпрограммы интегрирования потребуются

2 или 3 рабочих регистра, а также нужно запоминать где-то адрес возврата. Предположим теперь, что мы хотим вычислять двойной интеграл повторным интегрированием. Вспомогательная подпрограмма для внешнего интеграла должна обращаться к подпрограмме интегрирования для вычисления внутреннего интеграла, и если не предусмотреть заранее эту ситуацию, то возникнет хаос.

Самое грубое решение этой проблемы состоит в том, чтобы хранить в памяти две различные копии подпрограммы интегрирования. Каждая копия пользуется своими собственными рабочими регистрами, и поскольку адреса возврата запоминаются отдельно для каждого варианта, то не может быть путаницы. Однако этот метод требует слишком большого расхода памяти и поэтому не применяется на практике. На следующем этапе усложнения метода память содержит только одну копию подпрограммы, но разные области рабочих ячеек и разные ссылки (адреса возврата). При этом подпрограмма имеет дополнительный входной параметр, являющийся адресом области рабочих ячеек (рабочего поля); адрес возврата запоминается на рабочем поле, и при всех обращениях из подпрограммы к ее рабочему полю используется индекс-регистр или косвенный адрес, который формируется по входному параметру. (Разумеется, это и есть метод «чистых процедур» или «реентерабельного кода», используемый в системах мультипрограммирования). Этот метод применим всякий раз, когда глубина рекурсии явно известна (и постоянна); он не годится, если глубина рекурсии не определена во время написания программы и становится известной только в процессе выполнения программы. На практике этот метод применяется только при малой глубине рекурсии.

3.3. Стеки

Почти все системы рекурсивного программирования основываются на идее *стека* (называемого иногда магазинной памятью или «нанизанной» памятью). Одно из самых ранних опубликованных описаний этого метода содержится в статье [1].

Магазинная память — это память, действующая по принципу LIFO[1]). Всякий раз, когда элемент информации заносится в такую память, мы говорим, что он поступает на вершину па-

---

[1]) Начальные буквы английского оборота «last-in-first-out», означающего «последним пришел — первым ушел». — *Прим. перев.*

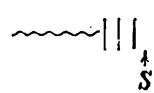
мяти, «утапливая» все элементы, уже содержащиеся в памяти. Всякий раз, когда элемент выбирается из памяти, он берется с вершины и все оставшиеся элементы поднимаются вверх по стеку. Поэтому всегда первым удаляется именно тот элемент, который появился позже всех остальных имеющихся. Термин «стек» обычно применяется к такой структуре памяти, в которой, помимо доступа к верхнему элементу, имеется возможность обращаться к внутренним элементам. Более точно, стек — это память, работающая по магазинному[1]) принципу, поскольку дело касается добавления или удаления элементов, причем можно обращаться к любому из нижних элементов, ссылаясь на его положение относительно одного из нескольких указателей.

Рассмотрим сначала простой стек, в котором каждый элемент занимает одно слово машинной памяти. Стек состоит из некоторого числа последовательных регистров, а также из указателя стека S, который может храниться в регистре памяти или в индекс-регистре, в зависимости от конкретной машины. Указатель S всегда показывает очередную свободную ячейку в стеке. Пусть $C[S]$ — содержимое ячейки, на которую указывает S (т. е. адрес которой содержится в S). Тогда существуют следующие основные операции над стеком:

Поместить элемент A в стек: $C[S] := A$
$S := S + 1$

Извлечь элемент из стека: $S := S - 1$
$A := C[S]$

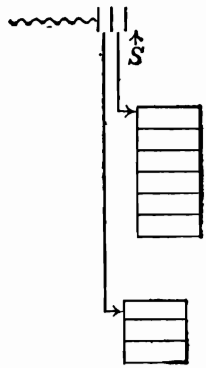
Далее в этой главе стеки будут изображаться горизонтально, с ростом в правую сторону, таким образом:



Не обязательно, чтобы стек состоял из однословных элементов. Если все элементы состоят из одинакового числа слов, то обобщение описанной структуры является очевидным. Если же элементы имеют различные размеры, то открываются две возможности. Элементы могут храниться вне стека, состоящего из

[1]) Подразумевается аналогия с магазинным устройством огнестрельного оружия. — *Прим. перев.*

указателей, которые соответствуют элементам; например:



Другая возможность состоит в том, чтобы помещать в стек информацию о размере элемента. Например, можно поместить в стек элемент из n машинных слов, а сразу за ним поместить управляющее слово, содержащее значение n; например:

При этом основные операции над стеком приобретают следующий вид:

Добавить к стеку элемент из n слов $A[1], \ldots, A[n]$

$$
\begin{aligned}
C[S] &:= A[1] \\
&\vdots \\
C[S+n-1] &:= A[n] \\
C[S+n] &:= n \\
S &:= S+n+1
\end{aligned}
$$

Извлечь элемент из стека:

$$
\begin{aligned}
n' &:= C[S-1] \\
A[n'] &:= C[S-2] \\
&\vdots \\
A[1] &:= C[S-n'-1] \\
S &:= S-n'-1
\end{aligned}
$$

Различные варианты этого метода описаны, например, в статьях [2, 3].

3.4. Основа для рекурсии

Мы можем теперь рассмотреть способ организации рекурсии на базе стека. Такая система включает стек, а также две подпрограммы, которые мы будем называть ОБРАЩЕНИЕ и ВОЗВРАТ.

Подпрограмма ОБРАЩЕНИЕ имеет три аргумента:

(а) Адрес подпрограммы, в которую нужно войти.
(б) Адрес первого регистра подлежащего запоминанию участка рабочего поля.
(в) Число регистров, подлежащих запоминанию.

Работа подпрограммы ОБРАЩЕНИЕ состоит в том, чтобы запомнить в стеке адрес возврата и указанные регистры, а затем обычным образом передать управление на ту подпрограмму, к которой требуется обратиться.

Подпрограмма ВОЗВРАТ имеет два аргумента:

(а) Адрес первого из подлежащих восстановлению регистров рабочего поля.
(б) Число регистров, подлежащих восстановлению.

Работа подпрограммы ВОЗВРАТ состоит в том, чтобы восстановить по содержимому стека состояние указанных регистров, а затем передать управление по адресу возврата, который хранится в стеке.

Если ОБРАЩЕНИЕ и ВОЗВРАТ описаны как макроэперации, то подпрограмма вычисления факториальной функции, выбирающая аргумент из сумматора и помещающая результат в сумматор, имела бы следующий вид:

```
FAC,  Перейти на ZERO, если сумматор содержит нуль
      Занести содержимое n сумматора в ячейку N
      Вычесть 1, занести n — 1 в сумматор
      Рекурсивно обратиться к подпрограмме FAC с помощью оператора

                  ОБРАЩЕНИЕ FAC, N, 1

      Умножить содержимое сумматора на содержимое ячейки N
      Выйти по оператору ВОЗВРАТ, N, 1
ZERO, Занести в сумматор константу 1
      Выйти по оператору ВОЗВРАТ, N, 1
```

Аналогичный прием для ФОРТРАНа описан Эйром [4]. Он несколько усложнен из-за того, что ФОРТРАН, как правило, не допускает обращения из подпрограммы к ней же самой и не предоставляет программисту доступа к связующей информации (адресу возврата). Нам понадобятся две подпрограммы, назы-

ваемые STORE и RSTOR. Подпрограмма STORE запоминает в стеке параметры, зависящие от глубины рекурсии, и информацию об адресах возврата, а подпрограмма RSTOR восстанавливает эти величины по содержимому стека. При использовании этих подпрограмм подпрограмма вычисления факториальной функции имеет следующий вид:

```
 1 SUBROUTINE NFAC (N, NFACT, DUMMY)
 2 IF (N) 3, 4, 6
 3 CALL EXIT (6)
 4 NFACT = 1
 5 GO TO 11
 6 CALL STORE (N)
 7 N = N - 1
 8 CALL DUMMY (N, NFACT, DUMMY)
 9 CALL RSTOR (N)
10 NFACT = NFACT*N
11 RETURN
```

Параметр *DUMMY* служит для того, чтобы подпрограмма могла обращаться к себе самой. Из основной программы производится такое обращение:

```
CALL NFAC (N, NFACT, NFAC)
```

и поэтому на время выполнения *DUMMY* заменяется на *NFAC*, и подпрограмма обращается к себе самой.

Подпрограмма *STORE* должна запомнить в стеке текущее значение *N* и связующую информацию; *RSTOR* должна восстановить эти данные по состоянию стека. Доступ из этих подпрограмм к связующей информации обеспечивается следующим приемом [1]), который его изобретатель сам расценивает как «беспринципную сделку». Если применяется стандартная процедура реакции на ошибки из системы ФОРТРАН/FAP (см. [5]), то обращение к подпрограмме дополняется следующими двумя словами:

```
NTR *  + 2, 0, A
PZE C,      0, B
```

Первое слово соответствует обычному выходу из подпрограммы. Второе слово содержит связующую информацию: *B* — это адрес, откуда происходит обращение, *C* — адрес *связующего звена* в подпрограмме, содержащего адрес возврата. «Беспринципная сделка» состоит в том, что запись подпрограмм STORE и RSTOR предусматривает наличие трех аргументов *N*, *NTR* и *LD*, а обращение к подпрограмме производится только с одним

[1]) Этот прием рассчитан на конкретный транслятор [5]. — *Прим. ред.*

аргументом *N*. Поэтому обращение из подпрограммы к величине *LD* обеспечит доступ к связующему звену через дополнительные слова, добавленные к обращению.

Приводим теперь программы STORE и RSTOR

```
 1 SUBROUTINE STORE (N, NTR, LD)
 2 COMMON LIST
 3 DIMENSION LIST (20)
 4 LIST (1) = LIST (1) + 1
 5 I = LIST (1)
 6 LIST (I) = LD
 7 LIST (1) = LIST (1) + 1
 8 I = LIST (1)
 9 LIST (I) = N
10 RETURN
```

```
 1 SUBROUTINE RSTOR (N, NTR, LD)
 2 COMMON LIST
 3 DIMENSION LIST (20)
 4 I = LIST (1)
 5 N = LIST (I)
 6 LIST (1) = LIST (1) — 1
 7 I = LIST (1)
 8 LD = LIST (I)
 9 LIST (1) = LIST (1) — 1
10 RETURN
```

(В обеих программах выход осуществляется посредством *TRA* 4, 4, а не *TRA* 2, 4.)

3.5. Система IPL-V

Не обязательно, чтобы в системе был только один стек. Пожалуй, самая развитая организация работы со многими стеками используется в системе IPL-V [6]. В ней имеется один стек для связей подпрограмм, а также отдельные стеки, соответствующие каждой переменной. В системе обработки списков наименование переменной обозначает списковую структуру и обычно соответствует регистру, который содержит указатель на первый элемент этой структуры. В системе IPL-V наименование соответствует стеку, причем каждая ячейка из стека содержит указатель на списковую структуру или элемент числовых данных. В системе предусмотрены в качестве основных операций Запоминание («утапливание» в магазине) и Восстановление (выборка из магазина). От программиста требуется, чтобы он запоминал все нужные элементы, прежде чем обращаться к другой подпрограмме. Это правило не распространяется на одну

особую ячейку, которая называется *ячейкой связи* и для которой магазинные операции запоминания и восстановления выполняются автоматически.

При использовании многих стеков возникает необходимость организовать различные стеки из одного стека. Не так легко отвести для каждого стека свой блок последовательных регистров, избежав при этом больших избыточных затрат памяти. Поэтому, особенно в системах обработки списков, обычно принято применять в памяти цепочки ссылок и организовывать списковую структуру одного уровня. (Читатели, не знакомые со списковыми структурами, найдут краткую информацию в приложении.)

3.6. Обобщение понятия стека

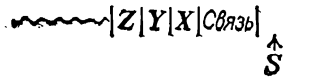
Опишем теперь метод, с помощью которого сохранение параметров, зависящих от глубины рекурсии, может выполняться автоматически. В любой системе программирования существуют правила, относящиеся к способу обращения к подпрограммам, т. е. к способу передачи подпрограмме ее аргументов. Например, в системе *FAP* адреса аргументов появляются в регистрах, следующих за командой передачи управления на подпрограмму:

```
TSX   SUB, 4
PZE   аргумент 1
PZE   аргумент 2
и т. д.
```

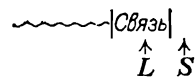
Приемлемой основой для рекурсии может служить соглашение, что подпрограммы берут свои аргументы из стека и оставляют в стеке любые результаты. Таким образом, если мы имеем подпрограмму с тремя аргументами X, Y и Z, то последовательность обращения к подпрограмме будет такой:

занесение Z в стек
занесение Y в стек
занесение X в стек
занесение адреса возврата в стек
вход в подпрограмму

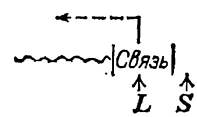
При входе в подпрограмму состояние стека будет таким:



Мы ставим в соответствие стеку еще один указатель L, который показывает на связь (адрес возврата) текущей подпрограммы; таким образом, имеем

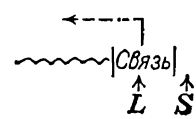


Итак, подпрограмма может обращаться к своим аргументам, ссылаясь на L, так как i-й аргумент должен находиться в стеке в регистре $L-i$. В качестве рабочего поля подпрограмма использует часть стека, находящуюся за связью. Для того чтобы эта система стала работоспособной, требуется следующее добавление: связь должна содержать не только адрес возврата, но также и указатель на связь для самой последней из ранее активизированных подпрограмм, т. е.

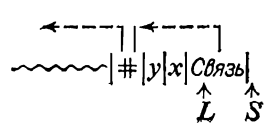


Теперь работа по выходу из подпрограммы состоит в передаче управления по адресу возврата и в перемещении указателя L. В результате восстанавливается то состояние стека, которое имело место при входе в подпрограмму. Если подпрограмма выработала какие-то результаты, то операция выхода должна включать в себя копирование результатов на место аргументов. Например, предположим, что внутри подпрограммы $S1$ имеется обращение к подпрограмме $S2$, которая использует аргументы x и y и вырабатывает результат z. Тогда на различных этапах будут возникать следующие состояния стека:

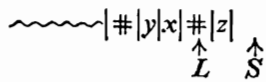
(1) При входе в $S1$



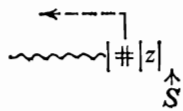
(2) При входе в $S2$



(3) Во время работы $S2$ (непосредственно перед выходом)



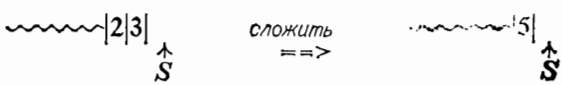
(4) После выхода из $S2$



Теперь результат z занимает ту ячейку стека, которую первоначально занимал аргумент y. Заметим, что с точки зрения подпрограммы $S1$ как бы выполняется операция «Занести z в стек», поскольку не остается других следов от работы подпрограммы $S2$.

Эта система гарантирует, что каждый раз, когда производится вход в подпрограмму, ее рабочее поле обязательно отделено от рабочих полей любых других программ (включая предыдущие еще не завершенные активизации самой этой подпрограммы), так что необходимые для рекурсии условия удовлетворяются.

При использовании этого метода можно обеспечить изящное единообразие, если выполнять в стеке также и арифметические операции, выбирая из стека два операнда и оставляя в стеке один результат, например:



Нужна также тестовая операция, которая сравнивает два верхних элемента стека и записывает в стек результат **true** (T) или **false** (F) в зависимости от того, одинаковы или различны эти элементы. Кроме того, требуется операция ППИ условного перехода при истине, которая осуществляет передачу управления, если верхний элемент стека имеет значение **true** и исключает из стека верхний элемент независимо от того, происходит или нет передача управления.

При наличии этих операций программа вычисления factorial (n) в соответствии с определением

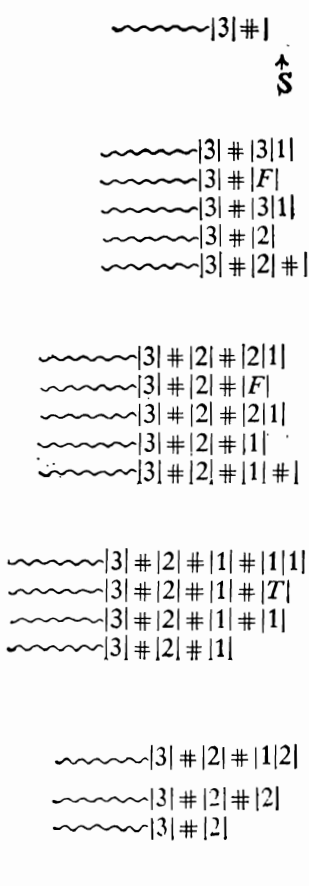
$$factorial\,(n) = [n = 1 \rightarrow 1,\ n \times factorial\,(n - 1)]$$

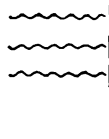
имеет следующий вид:

Занести первый аргумент в стек
Занести константу 1 *в стек*

```
    Применить тестовую операцию
    Перейти на 10, если вершина стека = true
    Занести первый аргумент
    Занести константу 1
    Вычесть
    Обратиться к программе вычисления факториала
    Занести первый аргумент
    Умножить
    Выйти из программы, оставив в стеке один результат
10  Занести константу 1
    Выйти из программы, оставив в стеке один результат
```

На рис. 3.1 показаны состояния стека на различных этапах вычисления factorial (3)





умножение
factorial (3) = 6
выход, копирование результата в стеке.

Рис. 3.1. Вычисление factorial (3) с использованием стека.

Если арифметическое выражение представлено в обратной польской записи (в которой каждой операции предшествуют ее операнды), то можно сразу указать последовательность операций над стеком для вычисления этого выражения. Например, выражение

$$(A + B) \times C$$

в обратной польской записи приобретает вид

$$CBA + \times$$

и ему соответствуют следующие операции над стеком:

занести C	C	
занести B	CB	
занести A	CBA	
сложить	$C\alpha$	$(\alpha = A + B)$
умножить	β	$(\beta = C \times \alpha)$

Аналогично, если мы расширяем возможности работы со стеком и включаем в них применение функций, то вычислению $f(a1, a2, \ldots, an)$ соответствует следующая последовательность операций:

занести	a_n
занести	a_{n-1}
. . .	
занести	a_1
применить	f

Здесь операция «*применить* f» означает запись в стек связи (адреса возврата) и обращение к программе вычисления f. Таким образом, компилирование для такой системы сводится, в сущности, к преобразованию выражений в обратную польскую запись.

3.7. Способы повышения эффективности

Такая система, как описанная в предыдущем разделе, обеспечивает очень удобный способ обработки рекурсивных процедур, но для нерекурсивных процедур ее применение оказывается неэффективным, поскольку в этом случае нет необходимости организовывать управление стеком. Если программа составлена в основном из рекурсивных процедур (что типично для программ на языке ЛИСП), то неэффективность применения этой системы к нерекурсивным процедурам не имеет значения. Однако если рекурсивные процедуры образуют только малую часть программы (что типично для программ на АЛГОЛе), то неэффективность применения стека для нерекурсивных процедур может оказаться дорогой платой за удобство рекурсии. Поэтому некоторые системы (например, PL/1, CPL) требуют от программиста явных указаний относительно того, какие процедуры должны обрабатываться как рекурсивные. Были предложены методы автоматического выявления рекурсивности, и мы коснемся двух из них.

Первый способ предложен Айронсом и Ферцайгом [7]. Рабочие ячейки и адрес возврата для процедуры хранятся в виде единого блока памяти. При первом обращении к процедуре вход в нее осуществляется обычным (нерекурсивным) способом, но имеются средства обнаружения рекурсивных обращений к процедурам в процессе счета, и если процедура вызывается рекурсивно, то ее блок рабочих ячеек запоминается в магазинной памяти.

Второй способ применяется в компиляторе с языка АЛГОЛ для машины KDF9; он описан Хокинсом и Хакстейблом [8]. Этот способ основан на том, что во время компиляции выясняется, возможны ли для той или иной процедуры рекурсивные обращения; после этого рекурсивные и нерекурсивные процедуры компилируются по-разному[1]). Так как во время компиляции удается только отличить процедуры, которые заведомо не могут вызываться рекурсивно, от тех процедур, к которым как будто бы возможны рекурсивные обращения, то такая система будет в некоторых случаях генерировать рекурсивное обращение к процедуре, которое никогда не понадобится. Например, если тело процедуры P1 не содержит никаких обращений к процедурам, то очевидно, что эта процедура никогда не будет вызы-

---

[1]) Метод предварительного формального выделения рекурсивно используемых процедур получил дальнейшее развитие в статье В. И. Собельмана «Выделение рекурсивно используемых процедур в АЛГОЛ-60», опубликованной в сб. «Цифровая вычислительная техника и программирование», вып. 2 (изд-во «Советское радио», М., 1967). — *Прим. перев.*

ваться рекурсивно. С другой стороны, рассмотрим процедуру P2, которая содержит оператор вида

if *b* **then** *P*2 **else** *P*3

Очевидно, что к процедуре P2 возможно рекурсивное обращение, хотя может оказаться, что во время счета переменная *b* всегда принимает значение **false** и поэтому рекурсивное обращение никогда не произойдет.

Если процедура описана рекурсивно, то легко обнаружить это во время компиляции. Более интересен случай «косвенной» рекурсии, примером которого является следующая последовательность обращений:

*P*1 *вызывает* *P*2 *вызывает* *P*3 *вызывает* *P*1

Для того чтобы обнаружить такой вид рекурсии, необходимо проследить структуру обращений в программе. Метод, примененный Хокинсом и Хакстейблом, состоит в том, что сначала организуется булевская матрица, в которой строки и столбцы соответствуют процедурам таким образом, что элемент (i, j) принимает значение 1, если процедура i может обратиться к процедуре j. Единицы в исходном состоянии матрицы соответствуют непосредственным обращениям. Например, предположим, что имеются пять процедур *P*1, *P*2, ..., *P*5 и

*P*1 *содержит обращения к* *P*2, *P*3, *P*4
*P*2 *содержит обращения к* *P*5
*P*3 *не содержит обращений к процедурам*
*P*4 *содержит обращения к* *P*1, *P*5
*P*5 *содержит обращения к* *P*3, *P*4

Исходная матрица имеет вид

$$\begin{vmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{vmatrix}$$

Далее заметим, что если процедура P_i содержит обращение к P_j, то P_i имеет «доступ» ко всем обращениям, содержащимся в процедуре P_j; поэтому полная структура обращений может быть получена применением к исходной матрице следующего алгоритма.

Пусть матрица — это **boolean array** $B[1:n, 1:n]$

```
begin integer i, j, k, p;
for p:=1, 2 do
   for j:=1 step 1 until n do
      for i:=1 step 1 until n do
         begin if B[i, j] then
            begin for k:=1 step 1 until n do
                  B[i, k]:=B[i, k] ∨ B[j, k]
            end;
         end;
end;
```

Воршелл [9] показал, что для построения матрицы всех связей [1]) достаточно двух проходов по матрице, как в приведенной выше программе на АЛГОЛе. Если применить этот алгоритм к рассмотренному выше примеру, то получится матрица:

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{vmatrix}$$

Возможность рекурсивного обращения указывается значением 1 на главной диагонали полученной матрицы.

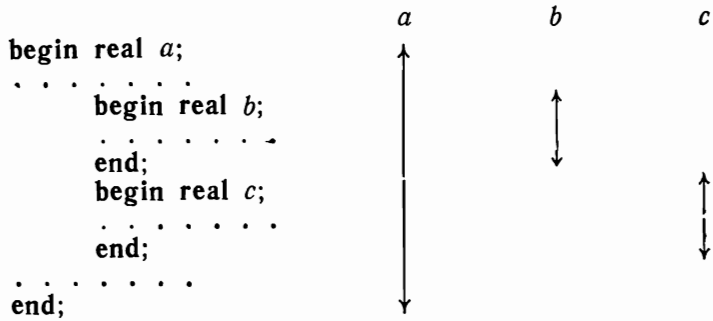
При анализе программ на языке АЛГОЛ возникают дополнительные сложности, связанные с тем, что какая-то процедура может оказаться формальным параметром другой процедуры. Поэтому необходимо исследовать структуру соответствий между формальными и фактическими параметрами. Если читатель интересуется этой проблемой, то он может подробно ознакомиться с ней в статье Хокинса и Хакстейбла.

3.8. Стековая система для АЛГОЛа

Большинство компиляторов с языка АЛГОЛ вырабатывают такие программы, которые используют стек в процессе счета (см., например, у Ренделла и Рассела [10]). Применяемая при этом стековая система в основном аналогична системе, описанной в разд. 3.6; однако в ней имеется одна дополнительная особенность, связанная с областями действия описаний. В языке АЛГОЛ область действия идентификатора, т. е. та часть программы, в которой этот идентификатор имеет смысл, определяется лексикографически как блок, в котором этот идентификатор описан вместе с любыми внутренними блоками, если в них не описан такой же идентификатор. Поэтому могла бы встретиться такая структура:

[1]) То есть прямых и косвенных обращений. — *Прим. перев.*

Области действия идентификаторов



Такую блочную структуру очень удобно реализовывать в рамках стековой системы: блок рассматривается как процедура без параметров, и при входе в блок регистры стека, непосредственно следующие за связью, отводятся для переменных, описанных в начале блока. Таким образом, аналогично тому как для процедуры i-й аргумент находится в ячейке $L - i$, точно так же i-я по счету описанная в блоке переменная находится в ячейке $L + i$. При этом легко решается проблема путаницы имен, возникающая при повторном описании переменной с таким же именем, как у описанной ранее переменной. Область памяти, соответствующая локальным переменным какого-то блока, возвращается системе при выходе из этого блока. Кроме того, если блок является частью рекурсивной процедуры, то при каждой активизации (т. е. при каждом рекурсивном обращении) образуется новый набор локальных переменных, а при каждом соответствующем выходе восстанавливается предыдущий набор локальных переменных.

Положение усложняется, если включить в рассмотрение описания процедур. Рассмотрим следующий фрагмент программы:

```
begin real a;
      procedure Q;
      begin real b;
      . . . . . .
      end;
      begin real c;
            Q;
      . . . . . .
      end;
      Q;
      . . .
end;
```

Поскольку области действия переменных определяются лексикографически, то в пределах процедуры Q определены только переменные a и b. Непосредственно перед первым обращением к Q действуют описания для a и c; непосредственно перед вторым обращением к Q действует только описание для a. Теперь нужно руководствоваться таким правилом: если переменная не описана в начале того блока, в котором она используется, то нужно проверить описания в начале непосредственно объемлющего ее блока и т. д. Поэтому необходимо сохранять в стеке «статическую цепь», соединяющую «связи» для тех блоков, описания из которых в текущий момент доступны в указанном выше смысле. Эта цепь является добавлением к «динамической цепи», которая уже существует в стеке и соединяет связи тех процедур (блоков), в которые был осуществлен вход, а соответствующего выхода еще не было. (Указатели из связей в стеке из разд. 3.6 образуют динамическую цепь.) Для повышения эффективности статическая цепь обычно дублируется вне стека; если читатель интересуется дальнейшими подробностями, то ему следует обратиться к статье Дийкстра [11] или книге Ренделла и Рассела [10].

3.9. Система рекурсии для языка ЛИСП

Язык ЛИСП допускает (и в сущности подразумевает) рекурсивность всех функций. Система программирования на языке ЛИСП может быть реализована с использованием стековой системы, аналогичной той, которая описана в разд. 3.6. Первоначальная система ЛИСП 1.5, реализованная впервые на машинах IBM 7090/94, использует стек, элементами которого являются блоки регистров, причем каждый блок содержит информацию о своей длине и о соответствующем месте в памяти, откуда он появился, а также содержит имя той процедуры, которой он принадлежит. Эти имена процедур позволяют организовать полезную систему диагностики ошибок, так как путем исследования стека можно проследить последовательность тех функций, в которые был осуществлен вход. Эта «обратная последовательность» обычно печатается после любой ошибки.

3.10. Аппаратные стеки

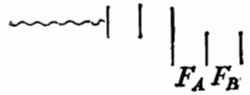
Стековые системы, описанные в предыдущих разделах, реализуются с помощью программного обеспечения. Однако можно иметь аппаратные стеки. Такие стеки предусмотрены, в частности, в английской машине KDF9 [12] и в машине Burroughs B5000 [13, 14].

Стек машины KDF9. В машине KDF9 имеются два аппаратных стека: «хранилище переходов для вложенных подпрограмм» и «магазинный сумматор». Каждый из них состоит из 16 ячеек. Как следует из его названия, хранилище переходов для вложенных подпрограмм используется для хранения адресов возврата из подпрограмм. При входе в подпрограмму адрес возврата заносится на вершину стека; операция выхода включает в себя передачу управления по адресу, находящемуся на вершине стека, и продвижение содержимого стека вверх. Магазинный сумматор является стеком, в котором предусмотрена возможность выполнения арифметических операций над верхними двумя элементами; поэтому машинные арифметические команды приобретают вид строки в обратной польской записи. Кроме того, предусмотрены некоторые операции перемещения, которые выполняются над верхними двумя, тремя или четырьмя ячейками стека: например, обмен местами двух верхних элементов, дублирование верхнего элемента, циклическая перестановка верхних четырех элементов и т. д. Магазинный сумматор удобен для вычисления арифметических выражений именно тем, что он допускает строку команд в обратной польской записи. (Это экономично, так как адреса требуются только для команд выборки операндов.) Однако стековая система такого вида только в очень ограниченных случаях облегчает рекурсивное программирование. Например, поскольку хранилище переходов для вложенных подпрограмм содержит только шестнадцать регистров, то, даже если бы удалось преодолеть остальные трудности, максимальное значение возможной глубины рекурсии не превышало бы шестнадцати.

Burroughs B5000. Машина B5000 представляет особый интерес. Ее аппаратура обеспечивает стековую систему, которая в основном соответствует системе, описанной в разд. 3.6. Стек хранится в ферритовой памяти и может иметь неограниченную длину; поэтому он действительно обеспечивает все возможности для рекурсивных процедур. (Аппаратный стек является только одной из многих интересных особенностей этой машины. Например, операндные ссылки на стек называются *дескрипторами*, и в зависимости от содержимого определенных управляющих разрядов из машинного слова остальная часть содержимого поля операндов может интерпретироваться либо как операнд, либо как указатель операнда, либо как адрес подпрограммы, которая вычислит указатель операнда.)

Недостатком стека в ферритовой памяти является возможность лишних циклов обращения к памяти в процессе выполнения арифметических операций. Для того чтобы сократить число обращений к памяти, в машине B5000 верхние два регистра стека размещаются в триггерных регистрах, причем каждому из

этих регистров соответствует флажок, указывающий, занят ли данный регистр. Такой стек может быть представлен следующим образом:



Здесь F_A и F_B — флажки занятости для аппаратных регистров A и B. Условно мы будем обозначать свободное состояние через 0, а занятое через 1. На схеме 3.2 показана последователь-

Строка в польской записи: $ab + cd + \times$

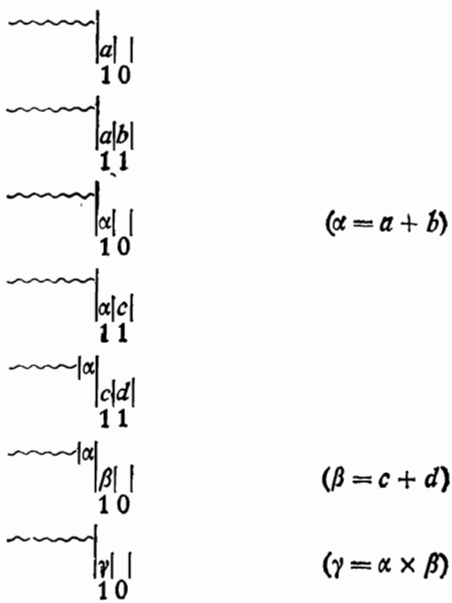


Рис. 3.2. Действие стека машины Burroughs B5000.

ность событий при вычислении выражения $(a + b) \times (c + d)$ в предположении, что в начальный момент стек пуст.

При выполнении обращения к подпрограмме нужно запомнить в стеке текущее содержимое аппаратных регистров. Поэтому если $F_A = 1$ или $F_B = 1$, то перед началом выполнения стандартного входа в подпрограмму содержимое соответствующего регистра переносится в стек. Таким образом, вход в подпрограмму производится при пустых аппаратных регистрах.

СПИСОК ЛИТЕРАТУРЫ

[1] Dijkstra E. W., Recursive programming, *Numerische Math.*, **2**, № 5, p. 312.
[2] Baecker H. D., Gibbens B. J., A commercial use of stacks, *Ann. Rev. Automatic Programming*, **4**, p. 183. Ed. R. Goodman, Pergamon, 1964.
[3] Strachey C., A general purpose macro generator, *Computer. J.*, **8**, № 3 (1965), 225.
[4] Ayer A. J., *Commun. Assoc. Computing Machinery*, **6**, № 11 (1963), 667.
[5] FAP Reference Manual, IBM Publication No. C28—6235.
[6] Newell A. (ed.), Information processing language-V manual, Prentice-Hall, 1961.
[7] Irons E. T., Feuerzig W., *Commun. Assoc. Computing Machinery*, **4**, № 1 (1961), 65.
[8] Hawkins E. N., Huxtable D. H. R., A multi-pass translation scheme for ALGOL 60, *Ann. Rev. Automatic Programming*, **3**, p. 316. Ed. R. Goodman, Pergamon, 1963.
[9] Warshall S., A theorem on Boolean matrices, *J. Assoc. Computing Machinery*, **9** (1962), 279.
[10] Randall B., Russell L. J., ALGOL 60 implementation, Academic Press, 1964. (Русский перевод: Рендел Б., Рассел Л., Реализация АЛГОЛа-60, «Мир», М., 1967.)
[11] Dijkstra E. W., Making an ALGOL translator for the XI, Reprinted in *Annu. Rev. Automatic Programming*, **4**. Ed. R. Goodman, Pergamon, 1963.
[12] Davis G. M., The English Electric KDF9 computer system, *Computer Bull.*, **4**, № 3 (1960), 119.
[13] Burroughs Corporation, Operational characteristics of the processors for the Burroughs B5000.
[14] Barton R. S., A new approach to the functional design of a digital computer, *Proc. Western Joint Computer Conf.*, 1961, p. 393.

4 РЕКУРСИЯ И ИТЕРАЦИЯ

4.1. Введение

В этой главе мы изучаем формальное соотношение между рекурсией и итерацией. Утверждение, что «все рекурсивные связи могут быть сведены к рекуррентным или итеративным описаниям», уже обсуждалось в гл. 1. Данная глава начинается с наброска возможного способа формального доказательства истинности этого утверждения для широкого класса числовых функций. Установив, что определенные функции могут быть описаны с помощью рекурсивной или итеративной схемы, мы затем рассматриваем такой подход, с точки зрения которого эти описания могут считаться эквивалентными, и исследуем возможность автоматического преобразования из одной формы в другую. Используемые идеи в основном заимствованы у Маккарти [1, 2] и развиты Купером [3]. У Маккарти интерес к этой отрасли возник в связи с его попытками создать основы теории вычислений. Возможность доказательства эквивалентности программ интересует его как средство формального доказательства правильности программы вместо того, чтобы демонстрировать ее правильную работу для отдельных наборов тестовых данных.

4.2. Вычислимые функции

Этот раздел содержит краткий обзор теории вычислимых функций. С полным изложением этой теории читатель может познакомиться [1]) по книге Дэвиса [4]. Отправной точкой является определение примитивно рекурсивной функции. (Заметим, что слово «рекурсивная» несет в данном контексте несколько специализированную смысловую нагрузку: оно относится не к самим функциям, а к тому методу, с помощью которого определяется класс функций.) Можно определить примитивно ре-

---

[1]) На русском языке теория вычислимых функций систематически изложена в монографии В. А. Успенского «Лекции о вычислимых функциях» (Физматгиз, М., 1960). — *Прим. перев.*

курсивные функции как такие функции, которые могут быть построены с помощью конечного числа операций суперпозиции и примитивной рекурсии, исходя из функций:

(1) $S(x) = x + 1$

(2) $N(x) = 0$

(3) $U_i^n(x_1, \ldots, x_n) = x_i \; (1 \leqslant i \leqslant n)$

Можно показать, что класс примитивно рекурсивных функций включает в себя все числовые функции, которые обычно встречаются на практике. Мы ограничимся здесь приведением некоторых примеров в подтверждение этого утверждения, причем будем рассматривать только функции от положительных целых аргументов.

(а) Функция $x + y$ является примитивно рекурсивной, так как

$$x + 0 = U_1^1(x)$$
$$x + (y + 1) = S(x + y)$$

(б) Функция $x \times y$ является примитивно рекурсивной, так как

$$x \times 0 = N(x)$$
$$x \times (y + 1) = (x + y) + U_1^2(x, y)$$

(в) Функция x^y является примитивно рекурсивной, так как

$$x^0 = S(N(x))$$
$$x^{y+1} = x^y \times U_1^2(x, y)$$

(г) Из предыдущих трех результатов следует, что любой полином от x с положительными коэффициентами является примитивно рекурсивной функцией.

(д) Функция $n!$ является примитивно рекурсивной, так как

$$0! = S(N(x))$$
$$(n + 1)! = n! \times S(n)$$

(е) Функция предшествования $P(x)$, где $P(0) = 0$ и $P(x) = x - 1$, если $x > 0$, является примитивно рекурсивной, так как

$$P(0) = N(x)$$
$$P(x + 1) = U_1^1(x)$$

Действуя аналогичным образом, можно показать, что все обычно встречающиеся числовые функции, в том числе логарифм, показательная функция, квадратный корень и т. д., принадлежат к классу примитивно рекурсивных функций.

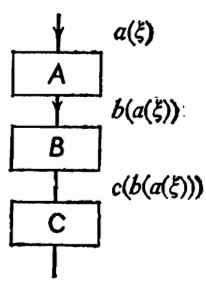
Интуитивно ясно, что функции, описанные таким способом, могут быть вычислены либо с помощью рекурсивной, либо с помощью итеративной программы. Райс [5] приводит программу на языке ФОРТРАН IV, которая может в принципе вычислить любую примитивно рекурсивную функцию, используя итеративный цикл.

4.3. Функции и блок-схемы

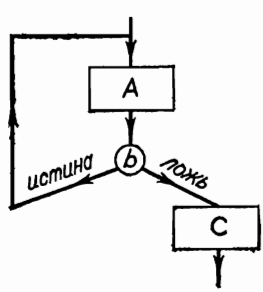
Характерным признаком итеративного процесса является то, что его можно представить с помощью блок-схемы. Поэтому мы рассмотрим сначала проблему перевода итеративного процесса, представленного с помощью блок-схемы, в форму рекурсивной функции. Каждый этап итеративного процесса может рассматриваться как функция, оперирующая над входными данными этого этапа и вырабатывающая выходные данные. Поэтому, если процесс состоит из последовательности подпроцессов A, B, C, то мы можем представить его выполнение в виде

$$c(b(a(\xi)))$$

где ξ — вектор, представляющий состояние машины, a — функция, эквивалентная процессу A и т. д. Поэтому имеем



Теперь предположим, что процесс содержит условный переход:



Пусть блок A представляется функцией $a(\xi)$, блок C — функцией $c(\xi)$, а вся программа — функцией $f(\xi)$. Мы можем представить всю программу в виде

$$f(\xi) = s(a(\xi))$$

где

$$s(\xi) = [b \to f(\xi),\ c(\xi)]$$

или, более кратко,

$$f(\xi) = \{\lambda\xi . [b \to f(\xi),\ c(\xi)]\}[a(\xi)]$$

Теперь рассмотрим проблему перехода от рекурсивного описания функции к блок-схеме. Возьмем в качестве примера описание факториальной функции:

$$Factorial(n) = \phi(n,\ 1)$$

где

$$\phi(n,\ s) = [(n = 0) \to s,\ \phi(n - 1,\ n \times s)]$$

Вектор состояния для этого вычисления состоит из значений n и s, т. е.

$$\xi = \{n,\ s\}$$

Предположим, что мы описали две функции:

$$f_1(n,\ s) = \{n,\ n \times s\}$$
$$f_2(n,\ s) = \{n - 1,\ s\}$$

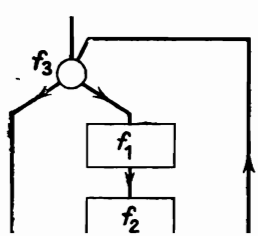
Тогда можно записать $\phi(n-1, n\times s)$ в виде $\phi(f_2(f_1(n, s)))$ и описание функции ϕ принимает следующий вид:

$$\phi(n,\ s) = [(n = 0) \to s,\ \phi(f_2(f_1(n,\ s)))]$$

Если предикат $n = 0$ представляется функцией $f_3(n,\ s)$, то мы получаем запись

$$\phi(\xi) = [f_3(\xi) \to \xi,\ \phi(f_2(f_1(\xi)))]$$

которая сразу переводится в блок-схему:



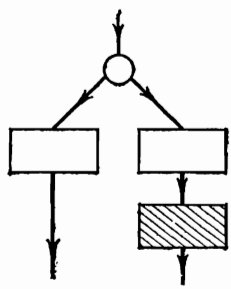
Читатель может убедиться, что эта блок-схема эквивалентна программе:

```
a: if n = 0 then go to b;
   s := n × s;
   n := n — 1;
   go to a;
b:
```

Если теперь мы попытаемся применить тот же процесс к другому описанию факториальной функции:

$$f(n) = [(n = 0) \to 1,\ n \times f(n - 1)]$$

то встретимся с трудностями. Полученная при этом блок-схема имеет следующий вид:



Здесь заштрихованный прямоугольник представляет весь процесс вычисления факториальной функции. Это эквивалентно процедуре на языке АЛГОЛ, которая рекурсивно обращается к себе. Таким образом, существуют некоторые рекурсивные описания, которые не могут быть прямо переведены этим способом в итеративные процедуры. (Из этого не следует, что существуют математические функции, которые могут быть вычислены только вычислительными системами, допускающими рекурсию. Мы должны четко уяснить разницу между математическим описанием функции и программой, вычисляющей эту функцию.)

Маккарти формально определил различие между теми рекурсивными описаниями, которые переводятся прямо в итеративные процедуры, и теми, которые не переводятся.

В общем случае если

$$f(x_1, \ldots, x_n) = \varepsilon(f, x_1, \ldots, x_n, g_1, \ldots, g_m)$$

где ε — какое-то выражение (обычно условное), и $g_1, \ldots, g_m$ — функции, то f называется функцией *итеративного типа*, если f никогда не встречается в качестве аргумента какой-либо из функций g_i. Если это условие выполняется, то можно начертить блок, представляющий f, и каждое появление f в правой части

описания переводится в возврат к началу процесса. Но если f встречается в качестве аргумента одной из функций g_i, то появление f в правой части описания будет переводиться в блок, представляющий весь процесс.

Согласно утверждению из разд. 4.2, для любой числовой функции описание всегда может быть приведено к итеративному типу.

4.4. Эквивалентность описаний

Мы показали, что если задано рекурсивное описание в «итеративной» форме, то его можно перевести в итеративную программу. Следующим этапом является поиск способа доказательства эквивалентности двух описаний одной и той же функции. Такое формальное доказательство явилось бы существенным шагом на пути к автоматическому переводу неитеративного описания функции в итеративное описание. Рассмотрим два описания факториальной функции в несколько различных формах:

$$Factorial(n) = [(n = 0) \to 1,\ P(n,\ Factorial(\delta(n)))] \qquad (1)$$

$$Factorial(n) = \phi(n,\ 1) \qquad (2)$$

$$\phi(n,\ m) = [(n = 0) \to m,\ \phi(\delta(n),\ P(n,\ m))]$$

В этих описаниях использованы вспомогательные функции

$$\delta(n) = n - 1$$

$$P(n,\ m) = n \times m$$

Из описания (1) следует, что

$$Factorial(n) = P(n,\ P(\delta(n),\ P(\delta^2(n),\ \ldots,\ P(\delta^{\lambda-1}(n),\ 1)\ \ldots)) \qquad (3)$$

а из описания (2) получаем

$$Factorial(n) = P(\delta^{\lambda-1}(n),\ \ldots,\ P(\delta^2(n),\ P(\delta(n),\ P(n,\ 1))\ \ldots)) \qquad (4)$$

где $\delta^2(n) = \delta(\delta(n))$ и т. д., λ — целое число, такое, что $\delta^{\lambda}(n) = 0$.

Функция P обладает тем свойством, что $P(\alpha, P(\beta, \gamma)) = P(\beta, P(\alpha, \gamma))$. Последовательно применяя это соотношение, мы можем переместить самое глубокое вхождение P в формуле (4) в начало выражения. Затем тот же процесс может быть повторен для нового самого глубокого вхождения P и так далее до тех пор, пока мы не получим, наконец, равенство (3). Таким образом, будет доказана эквивалентность двух рассматриваемых описаний факториальной функции.

В частном случае имеем

$$\begin{aligned} Factorial(3) &= \phi(3, 1) \\ \phi(3, 1) &= \phi(2, P(3, 1)) \\ &= \phi(1, P(2, P(3, 1))) \\ &= \phi(0, P(1, P(2, P(3, 1)))) \\ &= P(1, P(2, P(3, 1))) \end{aligned}$$

Последнее выражение имеет вид (4).

Однако

$$\begin{aligned} P(1, P(2, P(3, 1))) &= P(1, P(3, P(2, 1))) \\ &= P(3, P(1, P(2, 1))) \\ &= P(3, P(2, P(1, 1))) \end{aligned}$$

т. е. мы получили вид (3).

Можно обобщить этот принцип.

Предположим, что заданы описания

$$F1(n) = [(n = L) \rightarrow B, H(n, F1(\delta(n)))] \tag{5}$$

$$F2(n, A) = [(n = L) \rightarrow A, G(\delta(n), E(n, A))] \tag{6}$$

Они идентичны нашим предыдущим описаниям (1) и (2) с той разницей, что числа 0 и 1 заменены соответственно на L и B и произведение P заменено на функцию H в (5) и на функцию E в (6). Купер [3] показал, что эквивалентность этих двух функций может быть доказана, если при некотором значении константы B

$$H(\alpha, B) = E(\alpha, B) \tag{7}$$

и

$$H(\alpha, E(\beta, \gamma)) = E(\beta, H(\alpha, \gamma)) \tag{8}$$

Доказательство проводится таким же способом, как и в предыдущем случае. Из исходных описаний получаем

$$Factorial(n) = H(n, H(\delta(n), H(\delta^2(n), \ldots, H(\delta^{\lambda-1}(n), B), \ldots)) \tag{9}$$

$$Factorial(n) = E(\delta^{\lambda-1}(n), \ldots, E(\delta^2(n), E(\delta(n), E(n, B)), \ldots)) \tag{10}$$

Применив соотношение (7), можно заменить в формуле (10) самое глубокое вхождение E на H. Затем, последовательно применяя соотношение (8), можно переместить это вхождение H в начало выражения. Такой процесс повторяется до тех пор, пока не исчезнут все вхождения E; полученный при этом результат будет идентичен формуле (9).

Предположим теперь, что N — список, $\delta[N]$ — операция исключения первого элемента списка, $H[N, A]$ — список, получае-

мый добавлением первого элемента списка N в конец списка A, B и L — пустые списки. На языке ЛИСП это записывается так:

$$\delta[N] = Cdr[N]$$
$$H[N;\ A] = Append[Car[N];\ A]$$
$$B = L = ПУСТО$$

Тогда

$$F1(N) = [Null[N] \to ПУСТО;$$
$$Append[Car[N];\ F1[Cdr[N]]]]$$

Это рекурсивная функция переворачивания списка[1]). Аналогично

$$F2(N) = G[N;\ ПУСТО]$$

где

$$G[N;\ A] = [Null[N] \to A;$$
$$G[Cdr[N];\ E[n;\ A]]]$$

Если мы сумеем найти функцию E, удовлетворяющую условиям (7) и (8), то получим эквивалентное описание в итеративной форме.

Итак,

$$H[N;\ A] = Append[Car[N];\ A]$$

Поэтому

$$H[a;\ E[b;\ c]] = Append[Car[a];\ E[b;\ c]]$$

В результате применения последней формулы получаем список $(E[b;\ c],\ Car[a])$.

Если $E[b;\ c] = Cons[Car[b];\ c]$, то выражение $H[a;\ E[b;\ c]]$ даст нам список $(Car[b],\ c,\ Car[a])$. С другой стороны,

$$E[b;\ H[a;\ c]] = Cons[Car[b];\ H[a;\ c]]$$
$$= Cons[Car[b];\ Append[Car[a];\ c]]$$

что также дает список $(Car\ [b],\ c,\ Car\ [a])$.

Итак, $E[N,\ ПУСТО] = H[N,\ ПУСТО]$, так что условия (7) и (8) удовлетворяются.

Таким образом, мы располагаем методом преобразования любого описания типа (1) в эквивалентное итеративное описание, если умеем найти подходящую функцию E.

Можно описывать такие методы в виде некоторых подробных доказательств теорем относительно какого-то ограниченного класса программ. Непосредственной задачей дальнейших исследований в этой области должно быть получение менее глубоких

---

[1]) То есть изменения порядка элементов в списке на противоположный. — *Прим. перев.*

результатов для более широкого класса программ. Неясно, приведет ли работа в этом направлении к практически полезным результатам, т. е. возникнет ли возможность автоматического преобразования рекурсивной процедуры в эквивалентную нерекурсивную процедуру. Скорее можно ожидать, что изменения в аппаратуре вычислительных машин исключат ту неэффективность, которая в настоящее время присуща рекурсивным процедурам. Однако не вызывает сомнений следующее. Если будут написаны программы для выполнения этих преобразований, то сами эти программы обязательно будут рекурсивными.

СПИСОК ЛИТЕРАТУРЫ

[1] McCarthy J., Towards a mathematical science of computation, *in* Information processing 1962, Proceedings of the IFIP Congress, 1962, ed. C. M. Popplewell, North Holland, 1963.
[2] McCarthy J., A basis for a mathematical theory of computation, *in* Computer programming and formal systems, ed. B. Braffort and D. Hirschberg, North Holland, 1963.
[3] Cooper D. C., The equivalence of certain computations, *Computer J.*, **9**, № 1 (1966), 45.
[4] Davis M., Computability and unsolvability, McGraw-Hill, 1958.
[5] Rice H. G,. *Commun. Assoc. Computing Machinery*, 8, № 2 (1965), 114.

ПРИЛОЖЕНИЕ

Обработка списков

Это приложение содержит очень краткий обзор методов обработки списков и предназначено для читателей, которые незнакомы с этим предметом. С дальнейшими подробностями читатель может ознакомиться по статье Маккарти [1] и по книге Фостера [2].

Списковая структура представляет собой некоторое размещение *атомов*. Атомы могут являться любыми комбинациями символов. Для наших целей необходимо выполнение одного существенного условия, что атомы — это неделимые объекты. Например, «ВСЕЧТОВАМУГОДНО» — единый атом. В дальнейшем мы будем обозначать атомы прописными буквами.

Простой список представляет собой линейную последовательность атомов, например:

(A, B, C, D)

(A)

Отметим разницу между атомом A и списком (A), состоящим из одного элемента, причем этот элемент является атомом A.

Можно определить списковую структуру как список, элементами которого являются либо атомы, либо списки. Таким образом, структура

(A, (B, C), D)

представляет собой список из трех элементов, причем вторым элементом является список (B, C). Структура

((A, (B, C)), (D, E))

представляет собой список из двух элементов; первый элемент, в свою очередь, является списком, составленным из атома A и списка (B, C). Можно получить список вида

(((A)))

Это список из одного элемента, являющегося списком из одного элемента, который представляет собой список из одного элемента, атома A. Занимаясь списковыми структурами, мы имеем

дело с такой ситуацией, при которой размещение элементов сообщает нам не меньше, а то и больше информации, чем содержится информации в самих элементах.

Основными функциями для обработки списковых структур являются следующие:

(а) *Car* [*z*] или *Hd* (*z*)
Значением этой функции является первый элемент списка *z*.

Таким образом:

z	*Hd* (*z*)
(A, B, C)	A
(A, (B, C))	A
((A, B), C)	(A, B)
((A))	(A)
(A)	A

(б) *Cdr* [*z*] или *Tl* [*z*]
Значением этой функции является та часть списка *z*, которая остается после удаления из списка первого элемента [1]). Если за первым элементом ничего не оказалось, то значением *Tl* [*z*] является пустой список ПУСТО [2]). (ПУСТО, T (истина) и F (ложь) являются универсальными константами системы.) Таким образом:

z	*Tl* (*z*)
(A, B, C)	(B, C)
(A, (B, C))	((B, C))
((A, B), C)	(C)
((A))	ПУСТО
(A)	ПУСТО

(в) *Cons* [*x*; *y*]. Значение этой функции — новый список, в котором началом является *x*, а остальной частью — *y*. Таким образом:

x	*y*	*Cons* [*x*; *y*]
A	(B, C)	(A, B, C)
(A, B)	(C)	((A, B), C)
(A)	(B)	((A), B)
A	ПУСТО	(A)
(A)	ПУСТО	((A))

Заметим, что *Cons* [*Hd* [*x*]; *Tl* [*x*]] = *x*.

---

[1]) Аналогично *Caar* [*z*] — первый элемент первого элемента списка *z*, *Cadr* [*z*] — первый элемент остатка списка *z* и т. д. — *Прим. перев.*

[2]) В оригинале NIL. — *Прим. перев.*

(г) *Atom* [x] принимает значение „истина“ тогда и только тогда, когда x является атомом.
Null [x] принимает значение „истина“ тогда и только тогда, когда x является атомом ПУСТО.
Eq [x; y] принимает значение „истина“ тогда и только тогда, когда x и y — атомы, причем $x = y$. Значение этой функции не определено, если x или y не является атомом.

Остальные функции описываются через *Hd*, *Tl* и *Cons* с помощью условных выражений, в которых используются предикаты *Atom*, *Null* и *Eq*. Например:

(а) *Equal* [x; y] принимает значение „истина“ тогда и только тогда, когда x и y — одинаковые атомы или x и y — идентичные списковые структуры.

$$Equal[x;\ y] = [Atom[x] \to Atom[y] \wedge Eq[x,\ y],$$

$$Atom[y] \to F,$$

$$Equal[Hd[x];\ Hd[y]] \wedge Equal[Tl[x];\ Tl[y]]]$$

(б) *Rev* [x]. Значением этой функции является список, который содержит такие же элементы, как и список x, но эти элементы располагаются в обратном порядке. Например:

$$Rev[(A,\ B,\ (C,\ D))] = ((C,\ D),\ B,\ A)$$

Для получения функции *Rev* используем формулу:

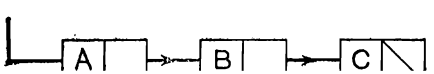
$$Rev[x] = Reva[x;\ ПУСТО]$$

где

$$Reva[a;\ b] = [Null[a] \to b,$$

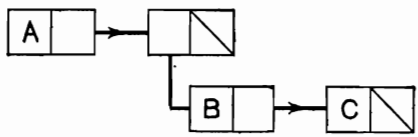
$$Reva[Tl[a];\ Cons[Hd[a];\ b]]]$$

Списковая структура обычно представляется в вычислительной машине последовательностью ячеек, каждая из которых разделяется на две части. Одна часть содержит очередной элемент списка, а другая — указатель на следующую ячейку. Это означает, что для хранения списков не требуются идущие подряд ячейки. Список (A, B, C) был бы представлен следующим образом:

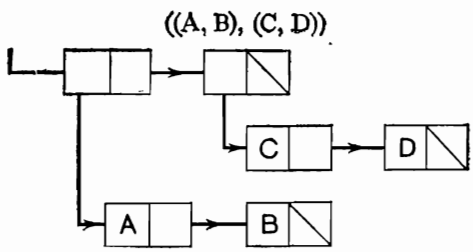


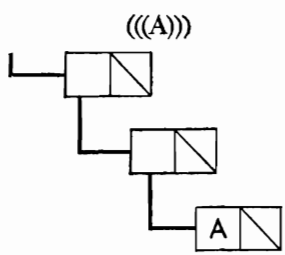
Каждая стрелка означает, что соответствующая ячейка содержит указатель на следующий элемент. Символ

означает конец списка: его можно интерпретировать как атом ПУСТО. Если какой-то список содержит в качестве элеменга другой список, то соответствующая ячейка первого списка содержит указатель на подсписок. Например, списковая структура (A, (B, C)) была бы представлена следующим образом:



Более сложными являются следующие списковые структуры:





Итак, основные функции *Hd* и *Tl* выделяют первую или вторую из двух частей ячейки. (Отсюда происходят другие названия тех же функций *Car* = Contents of Adress Register, т. е. содержимое регистра адреса, и Cdr = Contents of Decrement Register, т. е. содержимое регистра декремента. Эти названия отражают способ хранения списков при первоначальной реализации системы ЛИСП на машине IBM 7090.)

Для выполнения функции $Cons(x, y)$ берется новая ячейка; x помещается в начальную часть этой ячейки, ссылка на y помещается в оставшуюся часть.

Следует заметить, что, хотя мы и говорим о двух частях ячейки, вовсе не обязательно, чтобы это были части одного и того же регистра. Например, если заданы две линейные последовательности НАЧАЛО и ОСТАТОК, то мы можем рассматривать в качестве i-й ячейки пару физических регистров НАЧАЛО (i) и ОСТАТОК (i). Этот прием удобно использовать, если обработка списков включается в систему трансляции с языка АЛГОЛ.

СПИСОК ЛИТЕРАТУРЫ

[1] McCarthy J., Recursive functions of symbolic expressions and their computation by machine, *Commun. Assoc. Computing Machinery*, **3**, № 4 (1960), 104.
[2] Foster J. M., List processing, Macdonald, 1967. (Русский перевод: Фостер Дж., Обработка списков, «Мир», М., 1974.)

ОГЛАВЛЕНИЕ