В.Н.Пильщиков, Е.А.Бордаченкова, И.В.Горячая

РЕКУРСИВНЫЕ ФУНКЦИИ И ПРОЦЕДУРЫ, ЗАВИСЯЩИЕ ОТ ЦЕЛОЧИСЛЕННЫХ ПАРАМЕТРОВ

Предлагаемая учебно-методическая разработка предназначена для студентов первого курса факультета ВМК МГУ имени М.В.Ломоносова, может быть полезна также и преподавателям, ведущим практические занятия по курсу «Алгоритмы и алгоритмические языки». В разработке даются рекомендации по построению рекурсивных функций и процедур, зависящих от целочисленных параметров, приводятся примеры их описания на языке Паскаль и даются необходимые разъяснения.

1. РЕКУРСИВНЫЕ ФУНКЦИИ И ПРОЦЕДУРЫ

Сразу отметим, что на практике рекурсивные функции встречаются чаще, чем рекурсивные процедуры, поэтому мы в основном будем рассказывать о рекурсивных функциях. Для процедур же всё аналогично.

Пример описания рекурсивной функции

Рекурсия – это определение через себя; это сведение общего случая к аналогичным частным случаям.

Приведём классический пример рекурсивного описания понятия факториал $N!$ ($N \geq 0$):

$$N! = \begin{cases} 1, & \text{при } N = 0 \\ N*(N-1)!, & \text{при } N > 0 \end{cases}$$

Здесь факториал $N!$ определяется через факториал $(N-1)!$, т.е. через себя, поэтому данное описание $N!$ является рекурсивным.

Язык Паскаль допускает использование рекурсивных функций (и процедур). Рекурсивной при этом называется функция (процедура), в описании которой (в её теле) встречается обращение к ней же самой. Например, описание функции (дадим ей имя *F*), вычисляющей факториал, на Паскале выглядит так:

```
function F(N: integer): integer;    {N≥0}
begin
   if N=0 then F:=1 else F:=N*F(N-1)
end;
```

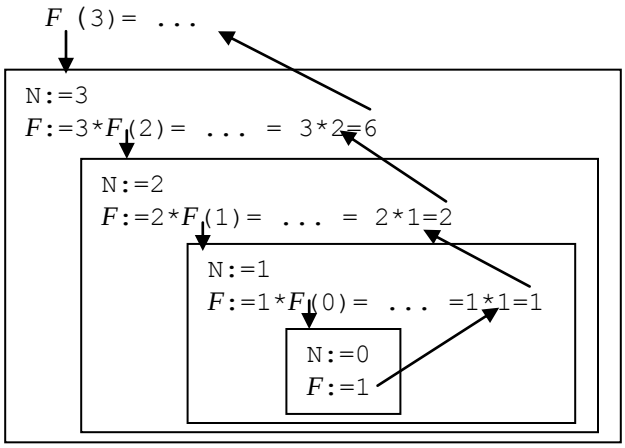
Отметим, что имя функции *F* входит как в левую, так и в правую часть оператора присваивания из ветки *else*. Так вот, вхождение в левую часть – это ещё не рекурсия: так в языке Паскаль указывается, что величина, полученная в результате вычисления правой части, объявляется значением функции. И только появление имени функции в других местах тела функции, причем с аргументами, свидетельствует о повторном обращении к функции, о том, что она рекурсивная.

Вычисление рекурсивной функции

Рассмотрим, как вычисляется *F*(3) (*см. рисунок на стр.2*). Поскольку параметр этой функции является параметром-значением, то при вызове функции заводится локальная переменная *N*, которой присваивается значение соответствующего фактического параметра, т.е. число 3, и при этом значении *N* выполняется тело функции (*см. самый внешний прямоугольник*

на рисунке). Проверка *N*=0 дает «ложь», поэтому значением функции объявляется результат умножения 3**F*(2). Но чтобы вычислить это произведение, надо предварительно вычислить значение нашей функции при аргументе 2. Поэтому вычисление функции для аргумента 3 приостанавливается и начинается вычисление *F*(2). Опять заводится локальная переменная *N*, которой присваивается значение фактического параметра, т.е. число 2, и при этом значении выполняется тело функции.

Следует понимать, что при каждом новом вызове функции *F* заводится новая локальная переменная *N*, отличная от переменных *N* из других вызовов. Таким образом, одновременно существует несколько одноименных, но различных переменных. И если, например, переменной *N* из второго вызова присвоить какое-то значение, то это никак не скажется на значении переменной *N* из первого вызова.



Итак, вычисляем тело функции при *N*=2. Поскольку условие *N*=0 не выполняется, то значением функции объявляется результат умножения 2**F*(1). Опять, прежде чем вычислить это произведение, надо вычислить значение нашей функции при аргументе 1. Поэтому вычисление функции для аргумента 2 приостанавливается и начинается вычисление *F*(1). Заводится новая локальная переменная *N*, ей присваивается значение фактического параметра, т.е. число 1, и затем выполняется тело функции. Снова условие *N*=0 не выполняется, поэтому значением функции объявляется величина 1**F*(0). Опять приостанавливается вычисление функции для аргумента 1 и вызывается *F*(0). Заводится новая локальная переменная *N*, ей присваивается значение 0 и при этом значении выполняется тело функции. На этот раз условие между *if* и *then* истинно, поэтому значением функции *F* при аргументе 0 объявляется число 1. На этом раскручивание цепочки рекурсивных вызовов завершается (т.к. получено частное решение).

Итак, мы добрались до аргумента, при котором ответ даётся явно (*см. внутренний прямоугольник на рисунке*). Теперь начинаем двигаться в обратную сторону в результате чего начнут выполняться в обратном порядке отложенные вызовы вплоть до самого первого:

F(0) => *F*(1)=1**F*(0)=1*1=1 => *F*(2)=2**F*(1)=2*1=2 => *F*(3)=3**F*(2)=3*2=6

Видим, что в процессе такого движения назад наша цепочка рекурсивных вызовов будет постепенно сворачиваться, что, в конечном счёте, приведёт к получению искомого ответа 6.

Особенности рекурсивных функций

Уже на этом простом примере видна характерная особенность рекурсивных функций: при вычислении рекурсивной функции сначала мы постепенно упрощаем её аргумент до тех пор, пока не дойдём до простейшего аргумента, для которого функция даёт явный ответ (без повторного обращения к себе), а затем мы начинаем двигаться в обратную сторону, вычисляя значения функции для всё более сложных аргументов.

В связи с этим описание рекурсивной функции должно состоять из двух частей. Первая часть – это т.н. нерекурсивные ветви, где рассматриваются простейшие случаи, для которых ответ дается явно. Отметим, что хотя бы одна нерекурсивная весть должна быть обязательно, иначе при вычислении функции мы зациклимся, не выйдем из рекурсии. Формально цепочка рекурсивных вызовов будет бесконечно длинной, что на практике приведёт к аварийному завершению работы программы. Вторая часть описания – это рекурсивные ветви, где рассматривается общий случай решения задачи. Этот общий случай сводится к более простым аналогичным случаям, для решения которых рекурсивно применяется та же самая функция, а затем из полученных ответов (по результатам этих повторных обращений) строится ответ для общего случая.

Рекурсия и циклы

Отметим, что любую рекурсивную функцию можно определить и нерекурсивно, т.е. через циклы (это строго доказано). Например, факториал можно определить так:

```
function F(N: integer): integer;
  var p, i: integer;
begin
  p:=1; for i:=1 to N do p:=i*p;
  F:=p
end;
```

Верно и обратное утверждение: любую нерекурсивную функцию можно определить рекурсивно (и это тоже строго доказано). Значит, рекурсивность – это не свойство самой функции, а лишь свойство её описания.

Возникает вопрос: а какое описание – рекурсивное или нерекурсивное – в программировании лучше? Однозначного ответа нет. Но в общем случае ситуация следующая: рекурсивное описание, как правило, короче, а циклическое – длиннее, зато рекурсивные функции обычно вычисляются дольше, чем циклические (потери происходят за счет повторных вызовов, введения новых локальных переменных и т.д.). Поэтому при выборе способа описания функции надо учитывать, что для нас важнее – меньше писать или быстрее вычислять.

2. РЕКОМЕНДАЦИИ ПО ОПИСАНИЮ РЕКУРСИВНЫХ ФУНКЦИЙ

Договоримся о следующем: рекурсивно описать функцию – значит не использовать в ней операторов цикла и перехода, а также не использовать глобальных (описанных вне функции) переменных, если, конечно, этого не требуется в условии задачи.

При описании рекурсивных функций типична следующая ситуация. Если уже имеется рекурсивная формула для вычисления функции (как, например, для факториала), то особых проблем с описанием этой функции на языке Паскаль нет даже у начинающих программистов. Проблемы появляются, когда предложено рекурсивно описать функцию для решения некоторой задачи, но не сказано, откуда здесь берётся рекурсия. Попробуем дать некоторые рекомендации относительно того, как надо «вылавливать» рекурсию. Сразу отметим, что это не точные, не строгие правила построения рекурсивных функций (таких правил вообще нет), а лишь подсказки, указывающие направление, в котором надо двигаться при таком построении. Сначала мы приведём эти рекомендации в общем виде, а затем покажем их применение на конкретных примерах.

В этих рекомендациях ради краткости изложения будем использовать следующие термины:

исходная задача – та задача, для решения которой мы описываем функцию; например, для функции $F(N)$ исходной задачей является вычисление факториала числа N!

подзадача – исходная задача, но в более простом варианте; например, для задачи вычисления $N!$ подзадачами являются вычисление $(N-1)!$, $(N-2)!$ и т.п.

Итак, пусть требуется рекурсивно описать функцию для решения некоторой задачи. Тогда нужно придерживаться следующих рекомендаций.

Рекомендация 1. *Прежде всего следует свести исходную задачу к одной или нескольким подзадачам, из решения которых можно построить решение исходной задачи.*

Такое сведение нам нужно для того, чтобы появилась рекурсия. В самом деле, что означает рекурсивное обращение к описываемой функции? Этим обращением мы решаем подзадачу, т.е. ту же самую задачу, что и исходная, но в более простом случае. Например, определяя в функции F факториал числа N, мы рекурсивно обращается к этой же функции для вычисления более простого факториала $(N-1)!$ Поэтому, чтобы появилась рекурсия, нужно в исходной задаче хотя бы раз решить подзадачу, манипулирующую более простыми данными.

При этом важно, чтобы исходная задача сводилась не к каким угодно подзадачам, а только к таким, из ответов которых можно построить ответ для исходной задачи. Например, при вычислении $N!$ можно, конечно, рассмотреть подзадачу вычисления 0!, однако из ответа 1 этой подзадачи вряд ли можно обоснованно получить величину $N!$ при произвольном N. А вот решив подзадачу вычисления $(N-1)!$, мы сможем легко получить величину $N!$

Отметим, что сведение исходной задачи к подзадачам является наиболее сложным и важным моментом в «вылавливании» рекурсии. Но, к сожалению, каких-либо общих правил, как это делать, здесь нет; всё зависит от конкретной задачи.

Но пусть мы так или иначе сумели свести исходную задачу к подзадачам. Что делать дальше? Надо решить эти подзадачи. Но как? Об этом говорит следующая рекомендация.

Рекомендация 2. *Считая, что функция правильно решает любые подзадачи, для решения этих подзадач следует рекурсивно обратиться к нашей функции и из ответов подзадач построить ответ для исходной задачи.*

Согласно этой рекомендации, описывать действия функции при решении исходной задачи следует в предположении, что функция уже умеет решать любые подзадачи (доводы в пользу этого предположения приведены ниже). Поэтому для решения отобранных подзадач надо смело обратиться к нашей же функции. При этом «лезть» внутрь этих рекурсивных обращений не надо, а надо лишь понять, какие ответы будут выданы этими обращениями и как эти ответы скомбинировать, чтобы получить окончательный результат. Например, при описании функции $F(N)$ мы используем результат вычисления $F(N-1)$, не думая о том, как устроено это вычисление.

Как правило, такое комбинирование не вызывает особых затруднений, т.к. эта проблема решается еще на этапе сведения исходной задачи к подзадачам.

До сих пор мы рассматривали действия функции в рекурсивных ветвях. Но в функции должны быть и нерекурсивные ветви, в которых ответы даются явно, без повторного обращения к определяемой функции. Такие ветви соответствуют решению исходной задачи в простейших случаях.

А как узнать, в каких именно случаях надо давать явные ответы? Например, в функции $F(N)$ явный ответ можно дать и для $N=0$ (0!=1), и для $N=1$ (1!=1), и для $N=2$ (2!=2), и для $N=3$ (3!=6) и т.д. Когда остановиться? Ответ на этот вопрос дает следующая рекомендация.

Рекомендация 3. *Явный ответ надо давать, когда исходная задача не сводится к подзадачам.*

Например, факториал 0! нельзя свести к более простому факториалу, поэтому при N=0 в функции F надо явно указывать её значение. В то же время факториал 1! можно свести к более простому случаю, т.к. 1!=1*0!, поэтому выписывать нерекурсивную ветвь для случая N=1 уже не надо, он подпадает под общий, рекурсивный случай.

Таковы общие рекомендации, которых следует придерживаться при построении рекурсивной функции. Далее мы рассмотрим конкретные случаи их применения. Но прежде сделаем пару замечаний.

Замечание 1. Согласно рекомендации 1, исходную задачу надо сводить к подзадачам, т.е. к аналогичным более простым задачам. Но что такое «более простая задача»?

Общего определения этого понятия нет, в разных случаях оно понимается по-разному. Но каково бы ни было определение более простой задачи, оно должно быть таким, чтобы процесс упрощения любой из задач был конечным, т.е. через конечное число шагов мы должны обязательно прийти к задаче, которую уже нельзя упростить. Иначе при выполнении рекурсивной функции, когда задачи сводятся к всё более простым задачам, мы попросту зациклимся и никогда не выйдем из рекурсии.

Выбор подходящего определения более простой задачи – одна из основных трудностей при сведении исходной задачи к подзадачам, поэтому данному аспекту мы будем уделять особое внимание в конкретных примерах.

Замечание 2. Согласно рекомендации 2, описывать поведение рекурсивной функции в общем случае следует в предположении, что она правильно действует в более простых случаях. Поэтому для того, чтобы решить подзадачи, нужно смело обращаться к этой же функции. При этом не нужно пытаться смотреть, как это часто делают начинающие программисты, что происходит с функцией в этих рекурсивных обращениях.

Насколько законно это предположение? Здесь следует вспомнить метод математической индукции. Пусть требуется доказать некоторое утверждение $P(N)$ для всех неотрицательных целых чисел N. Тогда согласно методу математической индукции предлагается действовать так: сначала надо доказать истинность этого утверждения для простейшего N=0, а затем, предполагая истинность $P(N-1)$, надо доказать истинность $P(N)$. Отсюда следует истинность утверждения в целом:

$$P(0),\quad \forall N\colon\ P(N-1)\rightarrow P(N)\quad ==>\quad \forall N\geq 0\colon\ P(N)$$

(Обратите внимание: доказывая истинность $P(N)$, мы предполагаем только истинность $P(N-1)$, но не лезем смотреть, как доказывается истинность этого предположения.)

Аналогичная ситуация и с рекурсивной функцией: если функция описана так, что она верно работает в простейших (нерекурсивных) случаях и верно решает исходную задачу исходя из предположения верного решения ею подзадач, тогда эта функция в целом является правильной.

Ну а теперь мы переходим к описанию конкретных рекурсивных функций согласно приведённым рекомендациям. Рассмотрим построение рекурсивных функций, зависящих от целочисленных аргументов.

3. НЕОТРИЦАТЕЛЬНЫЕ ЦЕЛЫЕ ЧИСЛА

В области неотрицательных чисел более простыми, как правило, считаются меньшие по величине числа. Например, для числа N более простыми являются числа N-1, N-2 и т.д. Очевидно, что такое уменьшение числа можно сделать только конечное количество раз – до 0.

Пример 1

Требуется рекурсивно описать функцию $f(x,n)$, вычисляющую величину $x^n/n!$ при любом вещественном x и любом неотрицательном n.

Решение. Прежде всего, согласно рекомендациям, вычисление $f(x,n)$ надо свести к подзадаче – к вычислению $x^k/k!$ при некотором $k<n$. Какое конкретно k взять? Можно, например, выбрать $k=n$-1, т.к. по $x^{n-1}/(n-1)!$ легко получить $x^n/n!$ Поскольку мы описываем $f(x,n)$ в предположении, что функция правильно вычисляет $x^k/k!$ при любом $k<n$, то в функции мы должны для вычисления $x^{n-1}/(n-1)!$ рекурсивно обратиться к $f(x,n-1)$, а затем полученную величину умножить на x/n, чтобы получить значение $f(x,n)$. Нерекурсивный случай – вычисление $f(x,0)$, т.к. вычисление $x^0/0!$ нельзя свести к вычислению $x^k/k!$ при $k<0$.

Итак, получаем следующее описание нашей функции на языке Паскаль:

```
function f(x:real; n:integer): real;   {n ≥ 0}
 begin if n=0 then f:=1 else f:=x/n*f(x,n-1) end;
```

Может возникнуть вопрос: если исходную задачу можно свести к разным подзадачам, по ответу каждой из которых можно построить ответ исходной задачи, то какую из этих подзадач выбрать? В данном примере мы свели вычисление $f(x,n)$ к вычислению $f(x,n-1)$, т.е. уменьшили второй параметр на 1. А можно ли было выбрать другую подзадачу, например, вычисление $f(x,n-2)$? Да, можно, поскольку по ответу этой подзадачи легко получается ответ исходной задачи:

$$f(x,n)=\frac{x^2}{n\cdot(n-1)}\cdot f(x,n-2)$$

Но при таком выборе нам придётся указывать две нерекурсивные ветви – при n=0 и n=1, т.к. при этих значениях n выражение $x^{n-2}/(n-2)!$ бессмысленно. В итоге получаем:

```
function f(x:real; n:integer): real;   {n ≥ 0}
 begin if n=0 then f:=1 else
        if n=1 then f:= x
               else f:=x*x/n/(n-1)*f(x,n-2)
 end;
```

Этот пример показывает, что при выборе подзадачи, к которой мы сводим исходную задачу, нежелательно слишком «далеко» уходить от исходной задачи, лучше брать подзадачу, наиболее «близкую» к ней.

Однако бывает и так, что по ответам «близких» подзадач не удается построить ответ исходной задачи, и потому приходится использовать достаточно «далёкую» подзадачу. Рассмотрим пару соответствующих примеров.

Пример 2

Не используя операции умножения и деления, рекурсивно описать функцию $M(a,b)$ от целых чисел a и b ($a\geq 0$, $b>0$), которая вычисляет остаток от деления a на b, т.е. $M(a,b)=a \bmod b$.

Решение. Здесь прежде всего надо определиться с тем, по какому параметру функции M мы будем вести рекурсию, т.е. какой параметр будем упрощать. Вести рекурсию по второму

параметру *b* нельзя, поскольку никакой хорошей зависимости между *a* mod *b* и *a* mod *c*, где *c*<*b*, нет. Поэтому рекурсию будем вести по первому параметру *a*.

При *a*≥*b* верно: *a* mod *b* = (*a*-*b*) mod *b*. Поэтому если мы сможем вычислить *M*(*a*-*b*,*b*), то сможем вычислить и *M*(*a*,*b*) – эти величины совпадают; значит, в данном случае исходную задачу надо сводить к подзадаче, где величина *a* уменьшена не на 1, а на *b*.

Что касается нерекурсивного случая, то он возникает при *a*<*b*, поскольку разность *a*-*b* выводит из области допустимых значений первого параметра функции. В этом случае ответом является само число *a*: *a* mod *b* = *a*.

Итак, получаем следующее описание нашей функции:

```
function M(a, b:integer): integer;   {a≥0, b>0}
begin
      if a<b then M:=a
             else M:=M(a-b,b)
end;
```

Пример 3

Описать рекурсивную функцию *degree*5(*N*), определяющую, является ли натуральное число *N* степенью числа 5. Если *N* – степень пятёрки, то функция возвращает показатель этой степени, иначе функция выдаёт ответ **-1**.

Например, *degree5*(50)= -1, *degree5*(125)=3, *degree5*(5)=1, *degree5*(1)=0.

Решение. Идея циклического решения задачи понятна: последовательно делить наше число на 5, пока это возможно. В результате серии таких делений мы либо дойдём до единицы (это положительный исход, при котором искомый показатель соответствует числу выполненных делений), либо в какой-то момент выяснится, что делить нацело на 5 дальше невозможно (отрицательный исход, соответствующий ответу **-1**).

Похожие соображения положим в основу рекурсивного решения задачи. При этом заметим, что если величина *N* div 5 – степень пятёрки с показателем *k*, то очевидно, что и число *N* – тоже степень пятёрки, но с показателем *k+1*. Также заметим, что единица – это наименьшая степень пятёрки с показателем 0.

Обратим внимание, что исходную задачу для числа *N* следует сводить к подзадаче для числа *N* div 5 лишь тогда, когда наше число *N* делится нацело на 5 (т.к. в этом случае остаётся надежда на положительный исход решения задачи), иначе ответ уже ясен (-1). Упрощение задачи здесь идёт в направлении перехода к степени с меньшим показателем и попытки дойти до степени с минимальным показателем.

Нерекурсивных случаев здесь два: первый возникает при *N*=1 (положительный исход с ответом 0), второй – если *N* не удалось поделить нацело на 5 (отрицательный исход с ответом **-1**).

Тем самым получаем такое описание функции *degree5*:

```
function degree5(N: integer): integer;
 var k: integer;
 begin
     if N=1 then degree5:=0 else
        if N mod 5 <> 0 then degree5:= -1 else
               begin
                    k:=degree5(N div 5);
                    if k=-1 then degree5:=-1 else degree5:=k + 1
               end
 end;
```

4. ЛЮБЫЕ ЦЕЛЫЕ ЧИСЛА

Мы рассматривали случаи, где значением параметров были неотрицательные целые числа, а теперь рассмотрим случай любых целых чисел.

Пример 4

Рекурсивно описать функцию *pow*(*x*,*n*), вычисляющую x^n для любого вещественного x ($\neq 0$) и любого целого n.

Решение. Поскольку $x^n = x \cdot x^{n-1}$, то, казалось бы, вычисление x^n нужно сводить к вычислению x^{n-1}. Однако это не так.

Особенность этой задачи в том, что второй параметр (n) функции *pow*, по которому будем вести рекурсию, может быть любым целым числом, а в области целых чисел процесс упрощения числа n путем вычитания из него 1 бесконечен: n, n-1, …, 1, 0, -1, -2, … Поэтому мы никогда не дойдём до такого значения n, при котором рекурсия остановится. С учётом этого, в области целых чисел надо как-то по-другому определять понятие более простого числа. Как? Сделать это можно по-разному.

Возможный вариант: рассмотреть в области целых чисел две подобласти – положительные и отрицательные числа, и в каждой из них использовать своё понимание более простого числа, считая более простым числом то, которое ближе к 0:

`n>0:  n, n–1, n–2, …,  2,  1` (здесь n–1 проще n)

`n<0:  n, n+1, n+2, …, -2, -1` (здесь n+1 проще n)

А для n=0 можно дать явный ответ: *pow*(x,0)=1.

Если так и сделать, то получим следующую рекурсивную формулу:

$$x^n = \begin{cases} 1, & n=0 \\ x \cdot x^{n-1}, & n>0 \\ x^{n+1}/x, & n<0 \end{cases}$$

В этом случае описание функции *pow* на языке Паскаль выглядит так:

```
function pow(x:real; n:integer): real;   {x≠0}
 begin if n=0 then pow:=1 else
        if n>0 then pow:=x*pow(x,n-1)
               else pow:=pow(x,n+1)/x
 end;
```

Другой вариант: поскольку верна формула $x^{-n}=1/x^n$ (n>0), то можно при отрицательном показателе степени перейти к положительному показателю согласно этой формуле, а затем традиционно упрощать положительное число, вычитая из него 1. Здесь для отрицательного числа более простым считается его модуль, а для положительного – число, на 1 меньшее его.

Это даёт следующую рекурсивную формулу:

$$x^n = \begin{cases} 1, & n=0 \\ 1/x^{|n|}, & n<0 \\ x \cdot x^{n-1}, & n>0 \end{cases}$$

И тогда описание функции *pow* выглядит так:

```
function pow(x:real; n:integer): real;   {x≠0}
 begin if n=0 then pow:=1 else
        if n<0 then pow:=1/pow(x,abs(n))
               else pow:=x*pow(x,n-1)
 end;
```

Можно придумать и какие-то другие определения более простого числа в области целых чисел, но в любом случае надо внимательно следить за тем, чтобы процесс упрощения был конечным.

5. ОБРАБОТКА ЦИФР В ЗАПИСИ ЧИСЛА

Теперь рассмотрим класс задач обработки целых чисел, где важна не величина чисел, а набор цифр, из которых составлены записи этих чисел. Такова, например, задача нахождения суммы цифр в десятичной записи заданного неотрицательного целого числа. Ясно, что величина числа не играет здесь главную роль, поскольку сумма цифр, скажем, в 7-ричной записи того же самого числа будет иной.

В такого рода задачах более простым следует считать не число, меньшее исходного числа по величине, а число, запись которого получается отбрасыванием одной цифры (например, последней, самой правой) из записи исходного числа: 123 проще 1234. Ясно, что полученное таким способом число проще (в нём меньше цифр) и что подобное упрощение чисел возможно только конечное количество раз.

Именно с этим более простым числом должна работать подзадача, к которой сводим исходную задачу. Нерекурсивный случай возникает для числа из одной цифры, т.к. отбрасывание цифры из однозначного числа приводит к пустой записи, но не к записи числа.

Рассмотрим несколько примеров на обработку цифр в записи чисел.

Пример 5

Рекурсивно описать функцию *maxdig*(*N*), которая находит наибольшую цифру в десятичной записи неотрицательного целого числа *N*. Например, *maxdig*(27306)=7.

Решение. Согласно предложенной рекурсивной схеме, в качестве подзадачи берём нахождение наибольшей цифры в числе, полученном из исходного числа (скажем, из 27306) отбрасыванием последней цифры (в числе 2730). Далее, согласно общей рекомендации, предполагаем, что наша функция правильно решает любую подзадачу, поэтому для решения нашей подзадачи рекурсивно применяем функцию к числу 2730, в результате чего она выдаст в ответе цифру 7. Теперь надо построить ответ для исходной задачи: сравниваем полученную цифру 7 с последней цифрой 6 исходного числа и наибольшую из них выдаём как ответ исходной задачи. Для однозначного же числа выдаём само это число как явный ответ.

Итак, имеем следующее описание функции на языке Паскаль:

```
function maxdig(N: integer): integer;  {N≥0}
  var m, last: integer;
begin
  last:=N mod 10;  {последняя цифра}
  m:=maxdig(N div 10);  {наибольшая цифра среди остальных}
  if last>m then maxdig:=last else maxdig:=m
end;
```

Пример 6

Рекурсивно описать функцию *Head3*(*N*), которая приписывает слева к десятичной записи неотрицательного целого числа *N* цифру 3 и возвращает в качестве ответа это новое число.

Например: *Head3*(1592)=31592

Решение. И здесь исходную задачу, скажем, с числом 1592, сводим к подзадаче, где рассматривается число 159, полученное из исходного числа отбрасыванием последней цифры. Предполагая, что функция правильно решает подзадачи, мы рекурсивно обращаемся к функции и получаем для нашей подзадачи ответ 3159. Как из этого числа построить ответ для исходного числа? А надо просто приписать справа к этому числу последнюю цифру исходного числа, т.е. вычислить 3159*10+2. В нерекурсивном случае, т.е. при однозначном числе, приписывание слева цифры 3 реализуется прибавлением 30 к этому числу, например: 7 → 30+7=37.

Итак, получаем следующее описание функции:

```
function Head3(N: integer): integer;  {N≥0}
begin
  if N<10 then Head3:=30+N
          else Head3:=Head3(N div 10)*10+N mod 10
end;
```

Если во всех предыдущих примерах мы описывали рекурсивные функции, то теперь приведём пример на рекурсивную процедуру.

Пример 7

Рекурсивно описать процедуру *RevPrint*(*N*), которая выводит неотрицательное целое число *N* в обратном порядке, точнее, выводит в обратном порядке цифры из его десятичной записи.

Например, по *RevPrint*(12345) должно быть выведено 54321

Решение. Снова рассматриваем подзадачу для числа, полученного из исходного числа, скажем, 12345, удалением последней цифры, т.е. для числа 1234. Решение этой подзадачи означает вывод 4321. Как получить правильный вывод для исходного числа? Для этого надо сначала вывести последнюю цифру 5 исходного числа, а уж затем рекурсивно обратиться к функции для решения данной подзадачи, чтобы вывести в обратном порядке остальные цифры. Для однозначного числа переворачивание означает просто вывод этого числа.

С учётом сказанного, получаем следующее описание процедуры:

```
procedure RevPrint(N: integer);  {N≥0}
begin
  if N<10 then write(N)
          else begin write(N mod 10); RevPrint(N div 10) end
end;
```

Обратите внимание, каким коротким оказалось рекурсивное описание этой процедуры; циклическое описание было бы значительно длиннее.

В приведённых выше примерах рекурсия велась только по одному параметру. Рассмотрим теперь задачу, в которой рекурсия необходима сразу по нескольким параметрам.

Пример 8

Описать рекурсивную булевскую функцию *equal*(*N, S*) (где *N*≥0 и *S* –целые числа), которая проверяет, совпадает ли сумма цифр десятичной записи числа *N* со значением *S*.

Например: *equal*(12345,15)=*true*, *equal*(24,7)=*false*, *equal*(100,1)=*true*

Решение. Здесь при переходе от исходной задачи к подзадаче упрощаем сразу два параметра: у числа *N* отбрасываем его младшую цифру, а значение *S* уменьшаем при этом на величину потерянной цифры. Тем самым, решение исходной задачи полностью зависит от решения полученной более простой подзадачи. Нерекурсивный случай имеет место для числа из одной цифры: по результату совпадения чисел *N* и *S* определяется искомый ответ.

С учётом сказанного, имеем следующее описание функции *equal*:

```
function equal(N,S: integer): boolean;   {N>=0}
begin
     if N<10 then equal:=N=S
             else equal:=equal(N div 10,S-N mod 10)

end;
```

Однако при внимательном рассмотрении можно обратить внимание на один существенный недостаток полученного решения. Пусть, например, произошло обращение к нашей функции с аргументами *N*=12345 и *S*=4.Тогда решение нашей задачи для этих аргументов будет сведено к решению подзадачи для аргументов *N*=1234 и *S*=-1. Следующей будет решаться подзадача для аргументов *N*=123 и *S*=-5, затем – для аргументов *N*=12 и *S*=-8, наконец – для аргументов *N*=1 и *S*=-10. И только теперь будет получен явный ответ. Ясно при этом, что искомый ответ был понятен намного раньше, на предыдущих этапах решения, точнее, в момент первого обращения к нашей функции с отрицательным значением аргумента *S* (ведь сумма цифр числа – величина заведомо неотрицательная). Такой момент в решении следует сразу же «отловить», иначе цепочка соответствующих рекурсивных вызовов будет раскручиваться впустую (и тем самым замедляться вычисление нашей функции). Исправляем решение с учётом высказанного замечания:

```
function equal(N,S: integer): boolean;   {N,S >= 0}
begin
    if S<0 then equal:=false else
        if N<10 then equal:=N=S else
                        equal:=equal(N div 10,S-N mod 10)
end;
```

Из приведённого примера видно, как важно бывает в решении обращать внимание на все возможные частные случаи, возникающие при упрощении параметров, и тем самым повышать эффективность работы нашей функции.

6. ИСПОЛЬЗОВАНИЕ ДОПОЛНИТЕЛЬНЫХ ПАРАМЕТРОВ

В заключение приведём одну нестандартную задачу, показывающую, как вводить дополнительные параметры, если они нужны для проведения рекурсии, но у требуемой функции эти параметры, увы, отсутствуют.

Пример 9

Не используя глобальные переменные и операторы цикла и перехода, описать целочисленную функцию *divs*(*N*) для подсчета количества всех делителей целого числа *N* (>1), без учета делителей 1 и *N*.

Например: *divs*(5)=0, *divs*(18)=4.

Решение. Ясно, что решать задачу (в силу ограничений в её условии) нужно рекурсивно. Но как здесь можно «выловить» рекурсию? Попытка как-либо упростить параметр *N* вряд ли сможет привести нас к какой-нибудь осмысленной подзадаче, из решения которой можно построить решение исходной задачи. Здесь нужно искать другой подход. В этой связи обратим внимание на то, что искать делители числа *N* нужно только среди чисел из диапазона [2 .. *N* div 2]. Вместо исходной рассмотрим вспомогательную задачу: подсчитать количество делителей числа *N* (фиксированного), лежащих в диапазоне [*k* .. *N* div 2]. Решение исходной задачи получается как решение вспомогательной задачи для *k*=2.

Вспомогательную задачу можно решать рекурсивно. Задача по подсчёту делителей из диапазона [*k* .. *N* div 2] сводится к подзадаче подсчёта делителей из более узкого диапазона [*k*+1 .. *N* div 2] и одновременной проверке, является ли при этом выброшенное из диапазона число *k* делителем нашего числа *N*. Нерекурсивным случаем в задаче будет пустой диапазон, в котором нет делителей числа *N*.

Но всё-таки не ясно, за счёт какого параметра проводить рекурсию, сужая диапазон (двигая вправо его нижнюю границу)? Ведь требуемая функция *divs*(*N*) должна иметь только один

параметр *N*, которого, как мы поняли, нам явно мало. Пойдём тогда на следующую «хитрость». В теле функции *divs*(*N*) опишем вспомогательную рекурсивную функцию *divs*1(*k*), которая занимается подсчётом делителей у числа *N* среди «кандидатов» из диапазона [*k* .. *N* div 2]. Тогда ответ для исходной функции *divs*(*N*) будет полностью зависеть от ответа для функции *divs*1(2) (здесь 2 – нижняя граница для исходного проверяемого диапазона).

В результате получаем такое решение:

```
function divs(N: integer): integer;     {N>1}
   function divs1(k: integer): integer;
   {к-во делителей числа N среди чисел из диапазона [k..N div 2]}
      begin
          if k > N div 2 then divs1 := 0 {диапазон исчерпан}
                         else divs1 := ord(N mod k = 0) + divs1(k+1)
      end;
begin
       divs :=  divs1(2)
end;
```

Обратим внимание, что результат проверки отбрасываемого числа *k* оформлен в виде выражения ord(*N* mod *k* = 0), что в данном случае очень удобно, т.к. таким способом в искомую сумму сразу же поставляется нужный нам ответ 0 или 1 (без привлечения условных операторов).

СПИСОК ЛИТЕРАТУРЫ

1. В.Г.Абрамов, Н.П.Трифонов, Г.Н.Трифонова. Введение в язык паскаль: учебное пособие – М.:КНОРУС, 2011

2. К.Йенсен, Н.Вирт. ПАСКАЛЬ: руководство для пользователя и описание языка. М.:Компьютер, 1993

3. В.Н.Пильщиков Язык Паскаль: упражнения и задачи. М.:Научный мир, 2003