Московский государственный университет
им. М.В. Ломоносова

Факультет вычислительной математики и кибернетики

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

Решение задач с использованием рекурсии

Учебно-методическое пособие
для студентов 1 курса

Москва - 2012

Введение

Рекурсия – мощный инструмент программирования, по выразительным возможностям близкий к циклам. Рекурсия широко применяется при решении игровых и переборных задач. Однако зачастую освоение рекурсии представляет существенную сложность для начинающих программистов. Данное пособие посвящено обсуждению понятия рекурсии, рассмотрению особенностей рекурсивных функций и процедур, а также приёмов их описания. Основой изложения материала является разбор большого количества учебных задач. Задачи сгруппированы по темам в соответствии с типом обрабатываемых данных. Сложность примеров возрастает по мере изложения материала.

Разделы 1 и 2 пособия были написаны В.Н.Пильщиковым и И.В.Горячей, раздел 3 – Е.А.Бордаченковой, раздел 4 – И.В.Горячей.

1. Рекурсивные функции и процедуры

В разделе вводится понятие рекурсии, разбираются особенности выполнения рекурсивных функций, обсуждается специфика программирования рекурсивных функций и процедур.

1.1. Понятие рекурсивной функции (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.

Рекурсия (от латинского *recursio* – возвращение) – это такой способ организации вычислений, при котором процедура или функция в ходе выполнения обращается сама к себе. Другими словами, рекурсия является методом определения функций (процедур), при котором определяемая функция применена в теле своего же собственного определения.[1] Понятие рекурсии используется не только в программировании, но и в математике. Интуитивно, *рекурсия* – это определение через себя; это сведение общего случая к аналогичным частным случаям.

Приведём классический пример рекурсивного определения понятия факториала $N!$ ($N \geq 0$):

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

Здесь факториал $N!$ выражается через факториал $(N–1)!$, т.е. через себя, поэтому данное определение $N!$ является рекурсивным.

Язык Паскаль допускает использование рекурсивных функций и процедур. Рекурсивной при этом называется функция (процедура), в описании которой встречается обращение к ней 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*. Вхождение в левую часть – это ещё не рекурсия: так в языке Паскаль указывается, что величина, полученная в результате вычисления правой части, объявляется значением функции. А вот вхождение *F*(*N*-1) в правую часть оператора присваивания является повторным обращением к функции и свидетельствует о том, что функция рекурсивная.

Рассмотрим, как происходит выполнение рекурсивной функции на примере вычисления *F*(3) (см. *рис.1*).

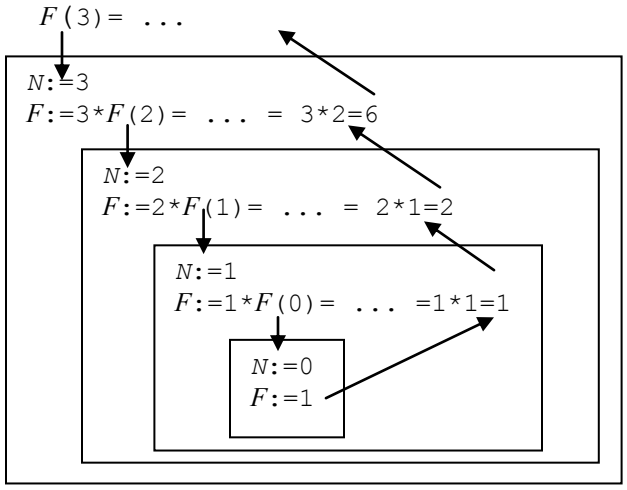

рис.1

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 (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 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.

Вычисляем теперь 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, значением функции *F* при аргументе 0 объявляется число 1, выполнение тела функции закончено. Тем самым раскручивание цепочки рекурсивных вызовов завершается, т.к. получен явный ответ.

Итак, мы добрались до аргумента, при котором ответ даётся явно (самый 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
```

При этом последовательно завершается выполнение вызовов *F*(1), затем *F*(2) и *F*(3). Видим, что в процессе такого движения назад наша цепочка рекурсивных вызовов постепенно сворачивается, что, в конечном счёте, приведёт к получению искомого ответа 6.

Отметим, что максимальное количество активизированных и незавершённых вызовов одной и той же функции для фиксированного значения аргумента называется *глубиной рекурсии*. В нашем случае, при выполнении *F*(3) глубина рекурсии равна 4, что соответствует количеству вложенных прямоугольников на приведённом рисунке. Самый большой прямоугольник соответствует первому уровню рекурсии, а самый маленький – последнему, т.е. четвёртому, уровню.

Уже на этом простом примере видна характерная особенность рекурсивной функции: при вычислении такой функции сначала мы постепенно упрощаем её аргумент, пока не дойдём до простейшего аргумента, при котором функция выдаёт явный ответ (без повторного обращения к самой 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 *нерекурсивные ветви*, где рассматриваются простейшие случаи, для которых ответ дается явно. Отметим, что хотя бы одна нерекурсивная ветвь должна быть предусмотрена обязательно, иначе при вычислении функции мы никогда не выйдем из рекурсии (формально цепочка рекурсивных вызовов окажется бесконечно длинной). Вторая часть описания – это *рекурсивные ветви*, где рассматривается общий случай решения задачи. Общий случай сводится к более простым аналогичным случаям, для решения которых рекурсивно применяется та же самая функция, а затем из полученных ответов строится окончательный ответ.

Отметим, что любую рекурсивную функцию можно определить и нерекурсивно, т.е. итеративно, 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 и используют больше памяти, 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.

1.2. 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 для решения некоторой задачи предложено рекурсивно описать функцию, но при этом 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 $N!$

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 же функции и из ответов подзадач построить ответ для исходной задачи.*

Согласно этой рекомендации, описывать действия функции при решении исходной задачи надо в предположении, что функция уже умеет решать любые подзадачи (доводы в пользу этого приведены ниже). Поэтому для решения отобранных подзадач надо смело обратиться к нашей же функции. При этом «лезть» внутрь этих рекурсивных обращений не нужно, а следует лишь понять, какие ответы будут выданы этими обращениями и как эти ответы скомбинировать, чтобы получить искомый ответ. Например, при описании *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 уже 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 упростить уже нельзя. Иначе при выполнении рекурсивной функции, когда задачи сводятся к всё более простым задачам, мы 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 обратиться к этой же функции.

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

$$P(0),\ \forall N\colon P(N\text{-}1) \rightarrow P(N) \Rightarrow \forall N \geq 0\colon P(N)$$

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

Аналогичная ситуация и с рекурсивной функцией: если функция описана так, что она верно работает в простейших случаях и верно решает исходную задачу на основе предположения корректного решения ею подзадач, тогда эта функция в целом является правильной.

Ну а теперь мы переходим к описанию конкретных рекурсивных функций согласно приведённым рекомендациям.

2. Использование рекурсии в числовых задачах

В этом разделе мы рассмотрим построение рекурсивных функций, зависящих от целочисленных аргументов.

2.1. Рекурсия по величине 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, 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 целом 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 $x^{n-1}/(n\text{-}1)!$ 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 описании функции мы должны для вычисления $x^{n-1}/(n\text{-}1)!$ рекурсивно обратиться к

$f(x,n\text{-}1)$, 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 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 $f(x,n\text{-}1)$, т.е. уменьшили второй параметр на 1. А можно ли было выбрать другую подзадачу, например, вычисление $f(x,n\text{-}2)$? 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 $x^{n-2}/(n\text{-}2)!$ 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 ($a\ge0$, $b>0$), 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 $a \bmod b$ и $a \bmod c$, где $c<b$, нет. Поэтому рекурсию будем вести по первому параметру a.

При $a \geq b$ верно равенство $a \bmod b = (a-b) \bmod b$. 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 $a-b$ выходит из области допустимых значений первого параметра функции. В этом случае ответом является само число a, т.к. $a \bmod b = a$. 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*), которая вычисляет, какой степенью числа 5 является натуральное число *N*. Если *N* не степень пяти, функция должна вернуть число **-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 с ответом **-1**).

Похожие соображения положим и 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**).

```
function degree5(N: integer): integer; {N>=1}
    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;
```

Мы рассмотрели 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. 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*(*x*,*n*) 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 перейти к положительному 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.

2.2. Рекурсия по записи числа в позиционной системе счисления

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:

```
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 вычисляет число, получаемое приписыванием слева цифры **3** к десятичной записи целого неотрицательного числа *N*. 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? 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, для числа 7 имеем: 30+7=37. 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 рекурсивной процедуры.

Пример 7. Рекурсивно описать процедуру *RevPrint*(*N*), которая печатает в обратном порядке цифры десятичной записи целого неотрицательного числа *N*. Например, *RevPrint*(12345) должна вывести текст 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. Получаем такое описание:

```
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;
```

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

2.3. Более сложные задачи

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 функцию *equal*(*N*, *S*) (где *N* и *S* – неотрицательные целые числа), которая проверяет, совпадает ли сумма цифр в 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. Простейший случай здесь имеет место для числа из одной цифры: по результату сравнения чисел *N* и *S* определяется искомый ответ.

Имеем следующее описание функции *equal*:

```
function equal(N,S: integer): boolean;   {S,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 полученного решения можно заметить один его существенный недостаток. 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, но значение *S* вышло из области допустимых значений, поэтому рекурсивно вызывать equal(1234,-1) нельзя. Есть два способа исправления решения. Первый состоит в использовании охраны перед рекурсивным вызовом:

```
function equal(N,S: integer): boolean; {S,N>=0}
var d:0..9;
begin if N<10 then equal:=N=S
      else
        begin d:= N mod 10;
          if S<d then equal:= false
                 else equal:=equal(N div 10,S-d)
        end
end;
```

Как видим, текст функции усложнился, к тому же появилась локальная переменная, что автоматически означает потери памяти при рекурсивных вызовах. Другой способ позволяет избежать этих недостатков. Он заключается в расширении области допустимых значений аргументов функции: доопределим *equal*(*N*,*S*) на область отрицательных значений *S*, полагая, что *equal*(*N*,*S*) = *false* для любого *S*<0. Получаем:

```
function equal(N,S: integer): boolean;
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;
```

Этот приём – расширение области значений и доопределение функции – довольно плодотворный приём в программировании рекурсии, он часто позволяет сделать текст функции наиболее коротким.

В заключение этого параграфа 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. Рекурсивно описать функцию *divs*(*N*) для подсчета количества всех делителей целого числа *N* (*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 *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 же 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 *N*. Нерекурсивным случаем в задаче будет пустой диапазон, в котором нет делителей числа *N*.

Но всё-таки ещё 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 *divs1*(*k*), которая занимается подсчётом делителей числа *N* среди «кандидатов» из диапазона [*k* .. *N* div 2]. Тогда ответ для исходной функции *divs*(*N*) будет совпадать с результатом вычисления функции *divs1*(2) (здесь 2 – нижняя граница проверяемого диапазона).

```
function divs(N: integer): integer;    {N>1}
   function divs1(k: integer): integer;
   {к-во делителей числа N среди кандидатов
    из  диапазона [k..N div 2]}
   begin if k>N div 2 then divs1:= 0 {диапазон пустой}
         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 удобно, т.к. таким способом в искомую сумму сразу же поставляется нужный нам ответ 0 или 1 (без привлечения условных операторов).

3. Рекурсия и последовательно организованные данные

В данном разделе рассматриваются структурированные данные – одномерные массивы, файлы и списки. Указанные структуры данных имеют последовательную природу: массив – конечная последовательность перенумерованных элементов, файл и списки также являются последовательностями элементов. При работе с такими данными естественно организовывать рекурсию в соответствии со структурой данных. Исходную задачу можно представить как комбинацию подзадач: 1) обработка первого элемента; 2) обработка оставшейся части последовательности. Простейшим (нерекурсивным) случаем при этом будет обработка пустой последовательности.

Конечно, применение рекурсии к каждой из структур – массиву, файлу, списку – имеет свои особенности, которые будут рассмотрены в соответствующих пунктах.

3.1. Рекурсивная работа с одномерными массивами

Поскольку массив состоит из наперёд заданного фиксированного количества элементов, рекурсию напрямую организовать невозможно: нельзя свести задачу обработки массива к задаче обработки этого же массива, упрощения данных не происходит. В этом случае используют следующий приём. Вместо исходной задачи обработки массива, скажем, $x[1..n]$, рассматривают задачу обработки выбранного подмассива, например, состоящего из элементов $x_k, x_{k+1},\ldots, x_n$. Новая задача решается рекурсивно, поскольку сводится к обработке элемента x_k и обработке более короткого подмассива $x_{k+1},\ldots, x_n$. Исходная задача при этом заключается в обработке подмассива $x_1,\ldots, x_n$. Рассмотрим, как работает этот приём на следующем примере.

Пример 1. Пусть имеются описания

```
const n = 100; type vector = array[1..n] of real;
```

Рекурсивно описать процедуру *PrNeg*(*x*), которая печатает отрицательные элементы вектора *x*.

Решение. Применим разобранный приём: рассмотрим, как напечатать отрицательные элементы подмассива $x_k, x_{k+1},\ldots, x_n$. Для решения этой задачи сначала нужно напечатать x_k, если он отрицателен, и затем напечатать все отрицательные элементы подмассива $x_{k+1},\ldots, x_n$. Нерекурсивным случаем будет ситуация, когда обработан весь массив, т.е. $k>n$, в этом случае ничего делать не нужно.

Решение вспомогательной подзадачи обработки массива оформим в виде вложенной рекурсивной процедуры от параметра *k*.

```
procedure PrNeg(var x: vector);
   procedure PrNeg1(k: integer);
   begin if k<=n then
         begin  if x[k]<0 then write (x[k]);
                PrNeg1(k+1)
         end
   end;
begin PrNeg1(1) end;
```

В качестве следующего примера рассмотрим задачу проверки, является ли некоторое предложение палиндромом. Напомним, что палиндром – это предложение, которое одинаково читается от начала к концу и от конца к началу. Примеры палиндромов: *а роза упала на лапу Азора*; *дорога за город*; *молебен о коне белом*.

Представим предложение в виде массива символов. Для простоты предположим, что в этом предложении нет знаков препинания – используются только пробелы и строчные буквы. Пробелы считаются незначащими символами. Пусть длина предложения не превосходит ста символов. Опишем соответствующую константу и тип массива:

```
const n =100; type txt = array [1..n] of char;
```

Пример 2. Рекурсивно описать логическую функцию *Pal*(*t*), которая проверяет, является ли предложение *t* типа *txt* палиндромом.

Решение. По сути, в задаче нужно проверить, является ли симметричной последовательность букв, содержащаяся в *t*. Для анализа симметричности нужно просматривать *t* с двух сторон и сравнивать соответствующие элементы *t* (см. *рис.2*).

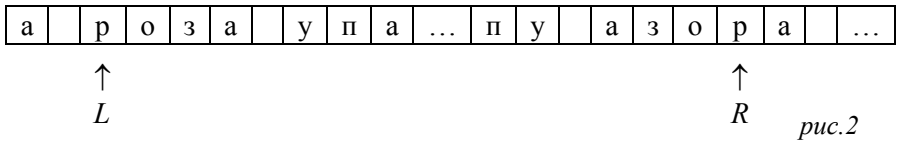

рис.2

Используем обозначения: *L* – индекс самого левого, а *R* – индекс самого правого непросмотренного символа. Пусть *t*[*L*] и *t*[*R*] – не пробелы. Если *t*[*L*] ≠ *t*[*R*], предложение не является палиндромом. Если же *t*[*L*] = *t*[*R*], предложение будет палиндромом, если последовательность символов между *t*[*L*] и *t*[*R*] – палиндром. В процессе анализа индексы *L* и *R* движутся навстречу друг другу. Анализ завершается, когда значение *L* станет больше или равно значения *R*.

Если *t*[*L*] (или *t*[*R*]) – пробел, нужно перейти к следующему символу *t*[*L*+1] (соответственно, *t*[*R*-1]).

Как и в примере 1, внутри функции *Pal*(*t*) опишем вспомогательную функцию для организации рекурсии. В качестве начала рекурсии в случае $L \geq R$ будем возвращать значение *true*, поскольку этот случай реализуется, когда просмотрены все пары соответствующих символов и ни разу не встретилось несовпадение букв.

```
function Pal(var t: txt): boolean;
   function P (L,R: integer): boolean;
   begin if L>=R then P:= true
         else if t[L]=' ' then P:= P(L+1,k)
              else if t[R]=' ' then P:= P(L,R-1)
              else {t[L],t[R] - буквы}
                   if t[L]<>t[R] then P:= false
                                 else P:= P(L+1,R-1)
   end;
begin Pal:= P(1,n) end;
```

В качестве последнего примера рекурсивной работы с массивами рассмотрим реализацию алгоритма бинарного поиска элемента в упорядоченном массиве. Пусть есть массив *X* целых чисел, упорядоченный по возрастанию элементов, и целое число *k*. Задача состоит в том, чтобы проверить, встречается ли число *k* в массиве *X*. Метод бинарного поиска состоит в следующем. Сравниваем *k* со средним элементом массива *X*. Если числа совпали, элемент найден, поиск окончен. В противном случае возможны два случая: *k* меньше среднего элемента, тогда оно может находиться только в первой половине массива; если число *k* больше среднего элемента, оно может находиться только в правой половине массива (т.к. массив упорядочен по возрастанию элементов). В любом случае, количество элементов, среди которых имеет смысл продолжать вести поиск, сокращается вдвое.

Пример 3. Пусть есть описания

```
const n=100; type vector=array[1..n] of integer;
```

Рекурсивно описать функцию *Search*(*X,k*), определяющую, входит ли целое число *k* в упорядоченный по возрастанию массив *X* типа *vector*.

Решение. Используем переменные *L* и *R* для обозначения границ подмассива, в котором ведётся поиск. Вначале *L*=1, *R*=*n*. В процессе работы границы *L* и *R* сближаются, переходя через середину подмассива, пока не будет найден искомый элемент (ответ *true*) или пока подмассив не кончится (*L*>*R*, ответ *false*).

```
function Search(var X: vector; k: integer): boolean;
 function Search1(L,R: integer): boolean;
 var m: integer;
 begin
   if L>R then Search1:=false
   else begin m:=(L+R) div 2; {индекс среднего элемента}
          if k=X[m] then Search1:=true
          else  if k<X[m] then Search1:=Search1(L,m-1)
                          else Search1:=Search1(m+1,R)
        end
 end;
begin Search:=Search1(1,N) end;
```

Следует подчеркнуть, что обычно для работы с массивами применяют циклические алгоритмы. Для рассмотренных выше задач не составит труда написать итеративные решения. В реальной программистской практике так и следует поступать. Однако поскольку цель пособия – разобрать приёмы и тонкости программирования рекурсии, мы обсудили именно рекурсивные варианты решения задач.

3.2. Рекурсивная работа с файлами

При рекурсивной обработке файлов необходимо обратить внимание на то, что процедуры, устанавливающие режим работы с файлом (*reset*, *rewrite*), нужно применять только один раз. Если мы поместим *reset* в рекурсивную функцию:

```
procedure P(var f: text);
begin reset(f); {***} P(f){***}  end;
```

при каждом рекурсивном вызове будет заново выполняться *reset*(*f*), текущая позиция будет сдвигаться в начало файла *f*, файл будет просматриваться каждый раз с самого начала, таким образом, рекурсия зациклится. От этой проблемы легко избавиться. Достаточно рекурсивно оформить обработку не всего файла, а только его части, начиная с текущей позиции и до конца. Подготовку же файла к работе следует проделать в основной процедуре (функции).

Пример 4. Пусть имеется описание:

```
type numbers= file of real;
```

Рекурсивно описать процедуру *PrNeg*(*f*), которая печатает отрицательные элементы файла чисел *f*.

Решение. Сначала будем считать, что *f* уже открыт на чтение. Если файл *f* пустой, с ним ничего делать не нужно. Если же *f* не пуст, надо

прочитать один элемент, напечатать его, если элемент отрицателен, и обработать аналогично оставшуюся часть файла. Эти действия опишем в виде вспомогательной процедуры без параметров *PrNeg1*. В основной процедуре остаётся открыть *f* на чтение и вызвать процедуру *PrNeg1*.

```
procedure PrNeg(var f: numbers);
var a: real;
  procedure PrNeg1;
  begin
   if not eof(f) then
    begin read(f,a); if a<0 then write(a); PrNeg1 end
  end;
begin reset(f); PrNeg1 end;
```

Для экономии памяти рекурсивная процедура *PrNeg1* работает с глобальной переменной *a*. Каждый рекурсивный вызов изменяет значение *a*, так что в процессе обработки файла переменная *a* принимает последовательно значения всех элементов файла и после завершения рекурсии содержит последний элемент файла *f*. Однако это не мешает работе процедуры *PrNeg1*, поскольку прочитанное из *f* значение сразу же обрабатывается, его не нужно хранить до завершения обработки оставшейся части файла.

Пример 5. Пусть имеется описание

```
type numbers = file of real;
```

Написать процедуру *Inverse*(*f1*,*f2*), которая в обратном порядке записывает в файл *f2* элементы файла вещественных чисел *f1*.

Решение. Как и в предыдущем примере, рекурсивную часть опишем во вложенной вспомогательной процедуре *Inv*. Основная процедура *Inverse* будет устанавливать режимы работы для файлов *f1* и *f2* и вызывать процедуру *Inv*.

Простейший, нерекурсивный случай имеем, если файл *f1* пустой; тогда нужно вернуть пустой файл *f2*, процедура *Inv* никаких действий выполнить не должна. Пусть *f1* не пуст. Как переставить его элементы в обратном порядке, если считать, что мы умеем делать такую перестановку для любого более короткого файла? Надо прочитать и запомнить первый элемент (при этом количество необработанных элементов *f1* сократится на 1), применить *Inv* для записи в *f2* в обратном порядке оставшейся более короткой части *f1*, затем записать в *f2* запомненный элемент.

```
procedure Inverse (var f1,f2: numbers);
   procedure Inv;
```

```
    var a: real;
    begin if not eof(f) then
            begin read(f1,a); Inv; write(f2,a) end
    end;
 begin reset(f1); rewrite(f2); Inv end;
```

Заметим, что использование локальной переменной – принципиальный момент в решении данной задачи. По мере обработки файла *f1*, его элементы записываются в память в качестве значений локальной переменной *a* последовательных рекурсивных вызовов. В конце концов, все элементы *f1* оказываются в оперативной памяти. Когда достигнут конец файла *f1*, процесс рекурсивных вызовов заканчивается и начинается закрытие в обратном порядке вызванных процедур с записью значений локальных переменных последовательных вызовов в файл *f2*. Таким образом в f2 оказываются элементы файла f1, переставленные в обратном порядке.

Необходимо сделать следующее замечание. Нет никакой гарантии, что наша процедура сумеет успешно обработать файл любой длины, памяти может попросту не хватить, и тогда программа аварийно завершит свою работу. Файл с точки зрения языка Паскаль – структура данных, изображающая внешнее запоминающее устройство на магнитной ленте. Вся работа с файлами строится на том, что файл целиком нельзя поместить в память и на том, что работать с файлом необходимо поэлементно. Рассмотренная задача является чисто учебным примером, иллюстрацией использования локальных переменных в рекурсии.

Пример 6. Рекурсивно описать функцию *Empty*(*t*), определяющую, со скольких пустых строк начинается текстовый файл *t*.

Решение. Решение состоит в просмотре и подсчёте пустых строк в начале файла, пока файл не кончится или пока не встретится непустая строка. Нерекурсивными случаями являются ситуации, когда файл пустой или файл начинается с непустой строки. В этих случаях следует вернуть 0. Рекурсивный случай: прочитать первую пустую строку файла и прибавить 1 к числу пустых строк в оставшейся части файла.

```
function Empty(var t: text): integer;
  function Empty1: integer;
  begin if eof(t) then Empty1:= 0
        else if not eoln(t) then Empty1:= 0
             else begin readln(t); Empty1:= 1+Empty1 end
  end;
begin reset(t); Empty:= Empty1 end;
```

Итак, если требуется оформить рекурсивно обработку массива или файла, нужно описывать вложенные рекурсивные функции и процедуры. Дело обстоит иначе при работе со списками, что мы увидим в следующем пункте.

3.3. Рекурсивная работа со списками

Будем рассматривать линейные однонаправленные списки без заглавного звена (см. *рис.3*). [1, 2] Изображённый на рисунке 3 список будем записывать в виде (e_1, e_2, …, e_n).

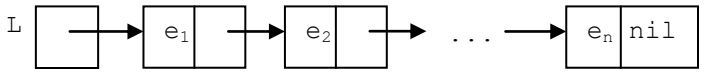

рис.3

Рекурсивное оформление обработки списка, как правило, не представляет трудностей. Приведем необходимые описания:

```
type TE = …;        {тип элементов списка}
   list = ↑chain;
  chain = record  elem: TE;  next: list  end;
```

Пример 7. Описать рекурсивную процедуру *PrNeg*(*L*), которая печатает отрицательные элементы списка вещественных чисел *L* (*TE=real*).

Решение. Если список *L* пуст (*L = nil*), процедура ничего делать не должна. Если же *L* не пустой, надо напечатать первый элемент, если он отрицателен, а затем обработать оставшуюся часть списка.

```
procedure PrNeg(L: list);
begin if L<> nil then
      begin if L↑.elem <0 then write(L↑.elem);
            PrNeg(L↑.next)
      end
end;
```

Как видим, никакие дополнительные вложенные процедуры, подобные тем, что использовались при работе с массивами и с файлами, при обработке списков не нужны.

Пример 8. Описать рекурсивную функцию *Increase*(*L*), проверяющую, упорядочены ли элементы списка чисел *L* по неубыванию (*TE= integer*).

Решение. Сформулируем идею рекурсивного решения. Пустой список и список из одного элемента, как обычно, считаются упорядоченными. Если же в списке же два и более элементов, он будет упорядоченным, если первый элемент меньше второго и список, начинающийся со второго элемента, упорядочен по возрастанию.

```
function Increase(L: list): boolean;
begin if L=nil then Increase:= true
      else if L↑.next=nil then Increase:=true
           else if L↑.elem>L↑.next↑.elem
                  then Increase:=false
                  else Increase:=Increase (L↑.next)
end;
```

Обратим внимание на важную особенность решения. В рекурсивной ветви можно было использовать оператор присваивания

`Increase:=(L↑.elem<=L↑.next↑.elem) and Increase(L↑.next),`

явно соответствующий сформулированной идее решения. Однако из соображений эффективности была использована конструкция с полным условным оператором, обеспечивающая для неупорядоченных списков меньшее число операций. В самом деле, если при проверке первой пары элементов выяснилось, что первый элемент больше второго, ответ известен – список не упорядочен – и нет смысла делать рекурсивные вызовы для остальной части списка.

Рассмотрим теперь задачу не на просмотр, а на изменение списка. Решим задачу включения элемента в упорядоченный список. Важной особенностью подобных задач является необходимость передачи в процедуру параметра-переменной, а не параметра-значения. Ссылка на первое звено может измениться, например, если элемент включается в пустой список или перед первым звеном списка.

Пример 9. Описать рекурсивную процедуру *Insert*(*L*,*k*), которая включает число *k* в упорядоченный по неубыванию список *L* так, чтобы сохранилась исходная упорядоченность (*TE*=*integer*).

Решение. Если решать задачу без рекурсии, с помощью циклов, решение получится таким: найти место в списке, куда нужно вставить элемент *k*, создать звено, записать в него число *k* и подцепить звено в нужное место списка. Причём довольно большая часть решения приходится на поиск первого элемента, большего или равного *k*.

В рекурсивном решении этой проблемы не возникает: попытаемся вставить число *k* перед первым элементом списка, а если первый элемент меньше *k*, применим те же действия к списку, начинающемуся со второго элемента и т.д. Таким образом, раскручивание рекурсии приводит к тому, что элемент, перед которым нужно вставить *k*, определяется автоматически.

Отдельного рассмотрения требуют нерекурсивные случаи: ситуация включения в пустой список и включение перед первым элементом списка. Действия, которые нужно выполнить в этих случаях, одни и те же, а именно, создать звено, записать в него число *k* и сделать новое звено первым звеном списка. Чтобы не выписывать одну и ту же последовательность операторов два раза, воспользуемся логической переменной *before*, объединяющей эти случаи.

```
procedure Insert(var L: list; k: integer);
var t: list; before: boolean;
begin
  if L=nil then before:= true else before:= L↑.elem>=k;
  if before then
    begin new(t); t↑.elem:= k; t↑.next:= L; L:= t end
  else Insert(L↑.next, k)
end;
```

Рассмотрим подробнее, как происходит включение элемента в середину списка. Пусть мы имеем список *M* = (100, 300). Выполнение процедуры *Insert*(*M*, 200) происходит следующим образом. Имя параметра *L* ассоциируется с именем *M*, в локальную переменную *k* записывается число 200. В результате выполнения первого условного оператора переменная *before* получает значение *false* (т.к. 100<200). Второй условный оператор вызывает рекурсивную ветвь *Insert*(*L*↑.*next*, *k*) для *L*=*M* и *k*=200. В начале работы процедуры имя *L* ассоциируется с переменной *M*↑.*next* и в созданную локальную переменную *k* записы-вается 200. Теперь имя *L* указывает на звено, содержащее число 300. При выполнении первого условного оператора выполняется сравнение 300>=200 и в *before* записывается значение *true*. Выполнение второго условного оператора создаёт новую запись, в поле *elem* записывает 200, в поле *next* – значение L, т.е. ссылку на звено с элементом 300. Далее в *L* (которая изображает переменную *M*↑.*next*, т.е. поле *next* первого звена списка *M*) записывается ссылка на созданную запись (200). Новое звено оказывается прикреплённым к предыдущему звену.

Читателю рекомендуется самостоятельно разобрать работу процедуры в случаях *Insert*(*M*, 400) и *Insert*(*M*, 50).

4. Рекурсивная обработка двоичных деревьев

При решении задач этого раздела нам будет удобно опираться на следующее *рекурсивное определение* двоичного дерева. *Двоичное дерево* – это конечное множество элементов, которое либо *пусто*, либо содержит один элемент, называемый *корнем,* а остальные элементы делятся на два непересекающихся подмножества, каждое из которых само является двоичным деревом. Эти подмножества называются *левым* и *правым поддеревьями* исходного дерева. Каждый элемент двоичного дерева называется *вершиной* этого дерева. Рекурсия здесь состоит в том, что исходное двоичное дерево определяется через двоичные деревья с *меньшим* числом вершин (и в конечном счёте сводится к *пустым* деревьям).

Итак, каждое поддерево двоичного дерева – это тоже двоичное дерево. Поэтому исходную задачу на обработку двоичного дерева будем сводить к более простым аналогичным задачам на обработку его поддеревьев. В конечном счёте такое сведение даст нам простейшие (нерекурсивные) случаи – дерево из одной вершины и/или пустое дерево (в зависимости от конкретной задачи будут использоваться либо оба случая, либо какой-то один). Возможны и другие частные случаи, соответствующие явным ответам, что во многом зависит от решаемой задачи (например, выяснилось, что нарушено проверяемое свойство, а значит, нет смысла в дальнейших проверках и действиях, т.к. ответ известен).

При рассмотрении задач этого параграфа мы воспользуемся представлением дерева в виде совокупности однотипных динамических объектов комбинированного типа (записей), связанных между собой посредством ссылок. Каждая запись представляет некоторую вершину дерева и хранит *элемент*, а также *ссылки* на корни левого и правого поддеревьев этой вершины. Опишем соответствующие типы:

```
type TE = …;     {тип элемента дерева}
   tree = ↑node;
   node = record elem: TE;  left, right: tree end;
```

Отметим некоторые особенности. При таком представлении во все вершины, кроме корня, входит ровно одна ветвь. Из каждой вершины *могут* исходить *2 ветви*: левая и правая. Причем, левая ветвь ведёт в *корень левого поддерева*, а правая ветвь – в *корень правого поддерева* этой вершины. Вершина, из которой не исходит ни одной ветви, называется *листом*.

Теперь приступим к рассмотрению конкретных задач.

Пример 1. Написать рекурсивную функцию *Member*(*T,E*), определяющую, входит ли элемент *E* в дерево *T* (*TE=integer*).

Решение. Простейший случай здесь относится к пустому дереву, для которого даём ответ *false* (элемента *E* нет, т.к. в пустом дереве вообще вершин нет). Если же дерево непустое, то через указатель *T* нам доступна корневая вершина, а значит мы сможем проверить, хранится ли в ней искомый элемент *E*. Если да, то ответ готов (это нерекурсивная ветвь, соответствующая ответу *true*). Если нет, то исходная задача распадается на две подзадачи по проверке содержимого левого и правого поддеревьев. Если поиск элемента *E* в левом поддереве увенчался успехом, то на этом завершаем работу функции с ответом *true* (исследовать правое поддерево в этом случае уже нет смысла – только потеряем время). Если же в левом поддереве элемент *E* не найден, то окончательный ответ будет полностью зависеть от результатов поиска в правом поддереве. Получаем следующее описание нашей функции:

```
function Member(T: tree;  E: TE): boolean;
begin
   if T=nil then Member:=false else {проверка корня:}
      if T↑.elem=E then Member:=true else {общий случай:}
          if  Member(T↑.left,E) then Member:=true else
                   Member:=Member(T↑.right,E)
end;
```

Пример 2. Написать рекурсивную функцию *Leaves*(*T*) для подсчёта числа листьев в дереве *T* .

Решение. В общем случае исходная задача по подсчёту листьев в заданном дереве сводится к подзадачам по подсчёту листьев в левом и правом поддеревьях (рекурсия!) и суммированию полученных ответов. Нерекурсивных ветвей здесь две. Первая соответствует ответу **0** – для пустого дерева (в нём листьев нет, потому что нет вершин). Вторая соответствует ответу **1** – для дерева из одной вершины (она является и корнем и листом одновременно). Приходим к следующему решению:

```
function Leaves(T: tree): integer;
begin
  if T=nil then Leaves:=0 else {проверка на лист:}
    if (T↑.left=nil)and(T↑.right=nil) then Leaves:=1 else
    {рекурсия:} Leaves:=Leaves(T↑.left)+ Leaves(T↑.right)
end;
```

Пример 3. Написать рекурсивную функцию *Height*(*T*) для нахождения высоты дерева *T* (считать, что *высота* непустого дерева – это число

вершин на самом длинном из путей от корня дерева до листьев, *высота* пустого дерева – **0**).

Решение. Очевидно, что для непустого дерева T высота $h(T) = max(h(T\uparrow.left), h(T\uparrow.right)) + 1$. Поэтому в общем случае первоначальную задачу можно свести к двум подзадачам по нахождению высот левого и правого поддеревьев. Тогда искомый ответ – это наибольшая из двух полученных высот, увеличенная на 1. Простейший случай здесь соответствует пустому дереву, высота которого, по определению, **0**.

```
function Height(T: tree): integer;
var hL,hR: integer;{высоты левого и правого поддеревьев}
begin if T=nil then Height:=0 else {общий случай:}
      begin
        hL:=Height(T↑.left); hR:= Height (T↑.right);
        if hL>hR then Height:=hL+1 else Height:=hR+1
      end
end;
```

Пример 4. Написать рекурсивную функцию *Level*(*T,n*) для подсчёта числа вершин на *n*-м ($n \geq 1$) уровне дерева T (считать, что 1-ый уровень соответствует корню дерева, а уровень любой другой вершины – на единицу больше уровня её родительской вершины).

Решение. При упрощении задачи и переходе от исходного дерева к его поддеревьям мы должны учитывать наличие у нашей функции второго параметра, задающего номер (n) уровня. Поэтому подсчёт числа вершин на уровне n в дереве T (общий случай) будем сводить к подсчёту числа вершин на уровне *n-1* в левом и правом его поддеревьях (рекурсия!), а затем – к суммированию полученных ответов. Простейших случаев здесь два. Первый соответствует пустому дереву, в котором нет вершин (независимо от заданного значения n даём ответ **0**). Второй – соответствует наименьшему (**1**) значению n при непустом дереве: на *1*-ом уровне в непустом дереве имеется ровно одна вершина – корень (даём ответ **1**). Приходим к такому решению:

```
function Level(T: tree; n: integer): integer; {n>=1}
begin if T=nil then Level:=0 else {непустое дерево:}
        if n=1 then Level:=1 else {общий случай:}
          Level:= Level(T↑.left,n-1)+ Level(T↑.right,n-1)
end;
```

Пример 5. Описать рекурсивную функцию *IsPos*(*T*), проверяющую, есть ли в дереве T хотя бы один путь (от корня до листа включительно),

во всех вершинах которого находятся только положительные элементы (*TE=real*).

Решение. Если дерево *T* непустое (общий случай), то проверив элемент из корневой вершины, мы сразу же определим, есть ли дальнейший смысл искать в этом дереве заданный путь. Действительно, если элемент в корне – положителен (удачное начало пути), то тогда всё будет зависеть от обстановки в поддеревьях. Если при этом искомый путь найдётся при прохождении через левое поддерево, то правое рассматривать нет смысла. Иначе поиск следует продолжить в правом поддереве. По отношению к поддеревьям применимы аналогичные рассуждения, тем самым мы «поймали» рекурсию.

Выявим теперь возможные частные случаи. Во-первых, это пустое дерево – в нём нет вершин, а, значит, нет и какого-либо пути (даём ответ *false*). Во-вторых, если в корне хранится элемент ≤ 0, то искомого пути точно нет (т.к. все возможные пути исходят только из корня). В этом случае даём сразу же ответ *false* и не продолжаем дальнейшего поиска. В-третьих, корень (с положительным элементом) может оказаться единственной вершиной дерева, а значит, искомый путь тем самым найден, т.е. добрались до листа - конечной вершины пути (даём ответ *true*). Получаем следующее описание функции:

```
function IsPos(T: tree): boolean;
begin
  if T=nil then IsPos:=false
    else {проверка содержимого корневой вершины:}
      if T↑.elem<=0 then IsPos:=false {пути нет}
         else {корень положителен:}
           if (T↑.left=nil)and(T↑.right=nil) then
               IsPos:=true {это лист, т.е. путь найден}
              else {поиск пути в левом поддереве:}
                 if IsPos(T↑.left) then IsPos:=true
                    else {поиск пути в правом поддереве:}
                       IsPos:=IsPos(T↑.right)
end;
```

Отметим, что в процессе работы нашей рекурсивной функции обращение к очередному поддереву соответствует переходу на следующий (более глубокий) уровень рекурсии. Если поддерево не подошло (его корневой элемент ≤ 0 или поддерево – пустое), то происходит возврат к предыдущему уровню рекурсии. Рекурсия здесь разворачивается сложным каскадом: спуск по дереву (путём перехода к его поддеревьям) чередуется с подъёмом к предыдущим (ранее пройденным) верши-

нам. Рано или поздно произойдёт одно из двух: либо на очередном уровне рекурсии мы окажемся в положительном листе (искомый путь построить удалось), либо мы проверим все возможные варианты и ни один из них не подойдёт (путь не найден).

Пример 6. Написать рекурсивную процедуру *Copy*(*T,T1*), которая строит дерево *T1* – копию дерева *T*.

Решение. Если дерево *T* – пустое, то, очевидно, что и его копия *T1* – пустое дерево (это простейший случай, задающий нерекурсивную ветвь). Для копирования непустого дерева воспользуемся рекурсией: сначала выделим память под корень формируемого дерева *T1* и запишем туда элемент из корня дерева *T*; а далее (т.е. с использованием процедуры *Copy*) построим копии левого и правого поддеревьев, записав ссылки на построенные поддеревья в поля *left* и *right* созданного корня. Приходим к такому описанию нашей процедуры:

```
procedure Copy(T: tree; var T1: tree);
begin
  if T=nil then {случай пустого дерева:} T1:=nil else
    begin {копирование непустого дерева:}
    new(T1); {выделение памяти под корень дерева T1}
    T1↑.elem:=T↑.elem; {копирование элемента из корня}
    Copy(T↑.left,T1↑.left);{копирование левого поддерева}
    Copy(T↑.right,T1↑.right){копирование правого поддерева}
   end
end;
```

Обратим внимание, поскольку дерево *T1* является результатом работы процедуры, параметр *T1* необходимо передать по ссылке. Заметим также, что за счёт указанного способа передачи данного параметра ссылки на построенные копии левого и правого поддеревьев будут автоматически записаны в поля *left* и *right* корневой вершины дерева *T1* (т.к. указанные поля передаются в качестве фактических параметров при рекурсивном обращении к процедуре *Copy*). Наконец, благодаря передаче этого параметра по ссылке в поля *left* и *right* концевых вершин (листьев) будут автоматически записаны пустые ссылки (см. ветку *then*).

Пример 7. Написать рекурсивную процедуру *Leaves*(*T*), которая удаляет (с освобождением памяти) все листья дерева *T*.

Решение. Начнём с рассмотрения частных случаев, которых здесь два. Во-первых, это пустое дерево, в котором нет вершин, а значит нет и листьев; над таким деревом не надо производить каких-либо действий. Во-вторых, это дерево из единственной вершины (она – одновременно

лист и корень); данную вершину следует удалить. В общем же случае, когда в дереве больше одной вершины, исходную задачу сводим к двум подзадачам по удалению всех листьев в левом и правом его поддеревьях. Получаем следующее решение:

```
procedure Leaves(var T: tree);
begin {действия производить только с непустым деревом!}
  if T<>nil then {дерево непустое, в нём одна вершина?}
    if (T↑.left=nil)and(T↑.right=nil)then {да, одна}
      begin {удаление единственной вершины, т.е. листа:}
        dispose(T);{высвобождение памяти, занятой под лист}
        T:=nil {после удаления листа дерево стало пустым}
      end else {в дереве имеется более одной вершины}
             begin {рекурсивная обработка поддеревьев}
                    Leaves(T↑.left); Leaves(T↑.right)
             end
end;
```

Отметим, что параметр нашей процедуры важно передавать по ссылке (а не по значению!). Это обеспечит сохранность изменений, выполненных над параметром *T* в теле процедуры. В частности, для дерева из более чем одной вершины при удалении его листьев значения *nil* будут автоматически записываться в поля *left* (если лист был слева – см. *рис.4*) или *right* (если лист был справа – см. *рис.5*) родительских вершин (т.к. эти поля указываются как фактические параметры при рекурсивном обращении к процедуре *Leaves*).

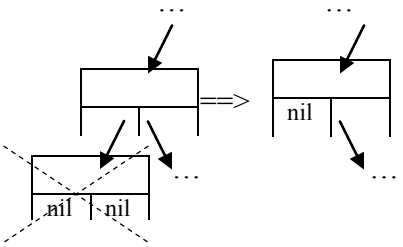

рис.4 (*удаление листа* ***слева***)

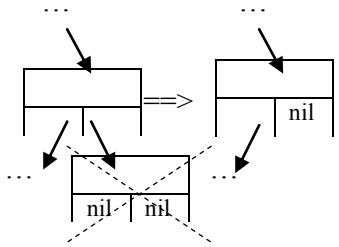

рис.5 (*удаление листа* ***справа***)

Пример 8. Рекурсивно описать функцию *Same*(*T*), определяющую, есть ли в дереве символов *T* хотя бы два одинаковых символа (*TE=char*).

Решение. Обратим внимание на то, что, по условию задачи, в вершинах дерева хранятся элементы символьного типа. Вспомним также, что в языке Паскаль на базе значений типа *char* можно строить множества. Воспользуемся вспомогательным множеством следующим обра-

зом. Организуем обход дерева (посещая каждую вершину только один раз), и в процессе обхода будем заносить в наше множество элементы из просмотренных вершин. При этом будем действовать по принципу: если элемент из рассматриваемой вершины ранее нам не встречался (т.е. этого элемента сейчас в множестве нет), то добавим его в множество, а иначе (элемент уже присутствует в множестве) – завершим просмотр дерева, т.к. обнаружен повтор. Оставшиеся элементы (из непросмотренных вершин) учитывать нет смысла.

Ясно, что множество перед началом обхода дерева должно быть пустым. Причем, начальную инициализацию множества следует выполнить *только один раз* (в разделе операторов функции *Same*), а уже затем запускать рекурсивный обход. Поэтому в теле основной функции *Same* потребуется описать вспомогательную рекурсивную функцию *Same1*, которая и будет выполнять такой обход (используя глобальное по отношению к ней множество). Результат работы *Same1* будет принят за ответ для основной функции *Same*.

Приведём сначала описание функции *Same*, а затем сделаем несколько дополнительных замечаний.

```
function Same(T: tree): boolean;
var S: set of char; {элементы из просмотренных вершин}
  function Same1(T: tree): boolean;
  begin {обход дерева до первого совпадения}
    if T=nil the Same1:=false else {проверка корня:}
      if T↑.elem in S then Same1:=true else
        begin {элемент из корня ранее не встречался}
           S:=S+[T↑.elem]; {добавить новый элемент в S}
           if Same1(T↑.left) then Same1:=true
                             else Same1:= Same1(T↑.right)
        end
   end;
begin   S:=[]; Same:=Same1(T) end;
```

По мере обхода заданного дерева будут рассматриваться (за счёт рекурсивных обращений) всевозможные его поддеревья, в результате чего в множество *S* будут заноситься всё новые и новые элементы из пройденных вершин. Пусть произошло очередное обращение к функции *Same1*, и пусть в корне очередного рассматриваемого поддерева встретился элемент, не принадлежащий множеству *S*. Тогда этот новый элемент добавляется в *S*, а далее – аналогично обрабатывается сначала левое поддерево. Если в процессе обхода этого левого поддерева нашлась вершина с элементом, который уже находится в множестве *S*, то даём

ответ *true* и на этом завершаем работу функции. Иначе – ответ будет зависеть от результатов аналогичного просмотра правого поддерева. В конце концов, сработает одно из двух. Либо в корне какого-то очередного поддерева встретится элемент, ранее включённый в множество *S* (ответ *true*), либо в процессе полного обхода дерева все его элементы окажутся различными (ответ *false*). Отметим, что частный случай *T=nil* очень важен, и не только для изначального пустого дерева *T* (в котором повторяющихся элементов, очевидно, нет), но и для пустых поддеревьев, к которым могут происходить обращения в процессе полного обхода *T*. Заметим также, что функция *Same*(*T*) получает результат *false* по окончании полного просмотра вершин дерева *T*, когда её рекурсивный вызов выполняется для поля *right* (= *nil*) у самого правого листа.

Итак, рассмотренные в этом параграфе примеры показывают, насколько удобно работать с деревьями рекурсивно (в силу рекурсивной природы этих структур данных). Используемые в примерах рекурсивные функции и процедуры при этом отличаются не только большой лаконичностью и наглядностью, но и высокой эффективностью. Конечно, существуют и циклические алгоритмы обработки деревьев, но они, как правило, моделируют рекурсию.

Заключение

В заключение, обсудим некоторые достоинства и недостатки рекурсии. Большим плюсом рекурсии является то, что она позволяет кратко и логически ясно сформулировать алгоритм, что наглядно демонстрируют рассмотренные примеры. Итеративные алгоритмы решения тех же задач, как правило, более длинные.

Недостатком рекурсии является её затратность, вытекающая из особенностей выполнения рекурсивных функций и процедур. Вспомним, что при каждом рекурсивном вызове создаются новые локальные переменные, к тому же, необходимо запоминать точку возврата (место, куда нужно передать управление после завершения выполнения процедуры). Причём, оценить заранее по тексту процедуры или функции, сколько будет сделано рекурсивных вызовов нельзя – глубина рекурсии зависит от значения параметра. При большой глубине рекурсии памяти компьютера может не хватить, что приведёт к аварийному завершению выполняемой программы. Нельзя упускать из виду и временны*е* затраты: каждый вызов процедуры или функции требует время на передачу

параметров (создать локальную переменную, присвоить ей соответствующее значение) и время на передачу управления процедуре, а также на возврат из неё.

Рекурсия – мощный и изящный механизм, однако, затратный по времени и по памяти. При решении конкретных задач программист должен сделать выбор в пользу рекурсии или циклов, исходя из особенностей задачи. Если задача несложно решается итеративно, рекурсию использовать не стоит. Однако если задача заключается в работе с рекурсивными структурами данных или сама природа задачи рекурсивна, нет смысла отказываться от рекурсии. Но всегда следует иметь в виду затраты на рекурсивные вызовы и стараться избегать их, если возможно – используя дополнительные переменные для сохранения уже полученного результата или путём своевременного выхода из процедуры (функции), когда ответ уже получен.

В заключение отметим, что в пособии были рассмотрены учебные примеры на программирование рекурсивных решений задач, которые имеют относительно несложные итеративные решения, либо задач обработки деревьев, где рекурсия естественно проистекает из устройства рассматриваемых данных. Напомним, что нашей целью являлась демонстрация понятия рекурсии и отработка первоначальных приёмов программирования рекурсивных решений. Обсуждение более сложных задач, для решения которых наиболее подходящим инструментом является рекурсия, можно найти в книге [4].

На этом мы закончим рассказ о рекурсивных функциях и процедурах.

Литература

1. Рекурсия. Материал сайта Академик.
URL: http://dic.academic.ru/dic.nsf/ruwiki/6791

2. В.Г.Абрамов., Н.П.Трифонов., Г.Н.Трифонова. Введение в язык Паскаль. – М.: КНОРУС, 2011. – 384 с.

3. В.Н.Пильщиков. Язык Паскаль: упражнения и задачи. – М.: Научный мир, 2003. – 224 с.

4. Н.Вирт. Алгоритмы и структуры данных.: Пер. с англ. – 2-е изд., испр. – СПб.: Невский Диалект, 2001. – 352 с.: ил.

Содержание