2. РЕКУРСИЯ

2.1. Рекурсивные определения и вычисления

Рассмотрим пример, который присутствует, по-видимому, во всех учебниках по программированию. Функция *факториал* натурального аргумента n обозначается как $n!$ и определяется соотношением

$$n!=1\cdot 2\cdot 3\ldots(n-1)\cdot n\ . \qquad (2.1)$$

Удобно доопределить $0!=1$ и считать, что n – целое неотрицательное число.

Некоторым недостатком определения (2.1) является наличие в нём многоточия «...», передающего речевой оборот «и так далее» и имеющего интуитивно понятный читателю смысл. Можно дать точное, так называемое *рекурсивное* определение функции $n!$, лишенное этого недостатка, т. е. не апеллирующее к нашей интуиции. Определим:

$$\text{а)}\quad 0! = 1\ , \qquad (2.2)$$

$$\text{б)}\quad n! = (n-1)!\cdot n \qquad \text{при } n > 0.$$

Соотношения (2.2) можно рассматривать как свойства ранее определенной функции, а можно (как в данном случае) использовать их для определения этой функции.

Далее для функции $n!$ будем использовать привычное «функциональное» (префиксное) обозначение fact (n) , указывая имя функции и за ним в скобках – аргумент. Тогда (2.2) можно записать в виде

$$\text{fact}\,(n) \;=\; \begin{cases} 1, & \text{если } n = 0; \\ \text{fact}\,(n-1)\cdot n, & \text{если } n > 0; \end{cases} \qquad (2.3)$$

или в другой форме записи

$$\text{fact}\,(n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \text{fact}\,(n-1)\cdot n, \qquad (2.4)$$

где использовано условное выражение **if** b **then** $e1$ **else** $e2$, означающее, что в том месте, где оно записано, следует читать $e1$, если выполняется условие b, и следует читать $e2$, если условие b не выполняется.

Функция, определяемая таким образом, *единственна*. Действительно, пусть есть две функции, например: fact1 (n) и fact2 (n), удовлетворяющие соотношениям (2.2) или их эквивалентам (2.3), (2.4). Рассмотрим разность dfact (n) = fact1 (n) – fact2 (n). Очевидно, что, во-первых, в силу соотношения «а» из (2.2) имеем dfact $(0) = 0$, а, во-вторых, для функции dfact (n) также справедливо соотношение «б». Действительно,

$$\text{dfact}(n) = \text{fact1}(n) - \text{fact2}(n) = \text{fact1}(n-1)\cdot n - \text{fact2}(n-1)\cdot n =$$
$$= (\text{fact1}(n-1) - \text{fact2}(n-1))\cdot n = \text{dfact}(n-1)\cdot n.$$

По индукции легко доказывается, что из соотношений $\text{dfact}(0) = 0$ и $\text{dfact}(n) = \text{dfact}(n-1)\cdot n$ следует, что $\text{dfact}(n) = 0$ для любого $n > 0$.

Как конструктивно могут быть использованы эти определения (как они «работают»)? Например, вычислим fact(4):

$$\text{fact}(4) = (\text{fact}(3)\cdot 4) = ((\text{fact}(2)\cdot 3)\cdot 4) = (((\text{fact}(1)\cdot 2)\cdot 3)\cdot 4) = ((((\text{fact}(0)\cdot 1)\cdot 2)\cdot 3)\cdot 4) =$$
$$= ((((1\cdot 1)\cdot 2)\cdot 3)\cdot 4) = (((1\cdot 2)\cdot 3)\cdot 4) = ((2\cdot 3)\cdot 4) = (6\cdot 4) = 24.$$

Здесь каждое новое использование рекурсивного определения заключается в скобки, а затем, когда ссылки на новые значения функции исчерпаны, последовательно каждое произведение двух сомножителей, заключенное в скобки, заменяется на результат умножения. Очевидно, что такое рекурсивное определение может рассматриваться как рецепт вычисления функции.

Приведём примеры других известных функций, которые также могут быть определены рекурсивно. *Наибольший общий делитель (**g**reatest **c**ommon **d**iviser)* натуральных a и b:

$$\gcd(a,b) \equiv \textbf{if } a = b \textbf{ then } a \textbf{ else if } a > b \textbf{ then } \gcd(a-b,b) \textbf{ else } \gcd(a,b-a),$$

или при $a > b \geq 0$

$$\gcd(a,b) \equiv \textbf{if } b = 0 \textbf{ then } a \textbf{ else } \gcd(b, a \textbf{ mod } b).$$

Степенная функция $f(a,n)=a^n$ (основание степени a – например, вещественное число, а показатель степени n – целое неотрицательное число):

$$f(a,n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } f(a,n-1)\ a\ , \qquad (2.5)$$

или другой вариант

$$f(a,n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } (f^2(a,\ n \textbf{ div } 2))\cdot f(a,\ n \textbf{ mod } 2)\ . \qquad (2.6)$$

Ещё один вариант можно получить из (2.6) с учётом того, что n **mod** 2 может принимать только значения 0 или 1, а $f(a,0) = 1$ и $f(a,1) = a$. Таким образом получаем

$$f(a,n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else}$$
$$\textbf{if } \text{Even}(n) \textbf{ then } (f^2(a,\ n \textbf{ div } 2)) \textbf{ else } (f^2(a,\ n \textbf{ div } 2))\ a, \qquad (2.7)$$

где $\text{Even}(n) = \textbf{not } \text{Odd}(n)$. Заметим, что соотношения (2.5) и (2.7) определяют одну и ту же функцию, но вычисления значения функции при заданном аргументе, порождаемые этими соотношениями, различны.

Рекомендуется самостоятельно рассмотреть пример: $f(a,11)$.

2.2. Рекурсивные функции и процедуры

Рекурсивное *определение* можно превратить в *описание* рекурсивной процедуры или функции. Например:

```
function fact(n: Nat0): Nat;
begin
    if n=0 then fact:=1 else  fact:=fact(n–1)*n
end {fact}
```

В языке Паскаль (и Турбо Паскаль) можно использовать рекурсивные процедуры и функции. В нашем примере в теле функции fact в операторе присваивания fact:=fact(n–1)*n находится вызов этой же функции, а именно fact(n–1). Как происходит выполнение таких функций и процедур? Рассмотрим следующую наглядную *модель* исполнения рекурсивных алгоритмов вычислительной машиной на примере функции fact. Пусть в программе имеется вызов описанной ранее рекурсивной функции, например z:=fact(k) с фактическим параметром (аргументом) k>0. Будем считать, что этот вызов запускает первый (стартовый) *экземпляр* функции fact. Каждый очередной рекурсивный вызов этой же функции (в данном примере это fact(k–1), fact(k–2) и т. д.) приводит к запуску нового экземпляра функции. При этом предыдущий экземпляр *откладывается,* и так происходит до тех пор, пока не будет вызван последний (*терминальный*) экземпляр функции, т. е. экземпляр, не содержащий рекурсивного вызова (в данном примере fact(0)). Этот терминальный экземпляр порождает значение функции (fact(0)=1) и завершает свою работу. Затем *восстанавливается* предыдущий (отложенный последним) экземпляр. Он, в свою очередь, порождает новое значение функции и завершает работу. Затем восстанавливается предыдущий экземпляр и т. д., до тех пор, пока не будет восстановлен и завершен экземпляр, соответствующий стартовому запуску (и тем самым закончится процесс вычисления fact(k)). Очевидно, память машины, используемая для хранения отложенных экземпляров, должна быть устроена таким образом, чтобы обеспечить восстановление первым того экземпляра, который был отложен последним.

Такой способ организации и функционирования памяти известен как *стек (Stack),* или *магазин* (по аналогии с магазином огнестрельного оружия), или *очередь типа LIFO*, т. е. Last In First Out (последним пришёл – первым ушёл). Опираясь на описанную модель, рассмотрим процесс вычисления fact(4), представленный в табл. 2.1.

Таблица 2.1

Вызов функции и результат	Экземпляр функции и фаза исполнения
fact(4)=fact(3)·4	экз1 (вызов)
fact(3)=fact(2)·3	экз2 (вызов)
fact(2)=fact(1)·2	экз3 (вызов)
fact(1)=fact(0)·1	экз4 (вызов)
fact(0)=1	экз5 (вызов и завершение)
fact(1)=1·1=1	экз4 (восстановление и завершение)
fact(2)=1·2=2	экз3 (восстановление и завершение)
fact(3)=2·3=6	экз2 (восстановление и завершение)
fact(4)=6·4=24	экз1 (восстановление и завершение)

В этой таблице «время» возрастает по строкам (сверху вниз). Новый вызов располагается на следующей строке со сдвигом вправо. При восстановлении очередного экземпляра также переходим на следующую строку, но сдвигаемся влево относительно положения в строке записи о завершении предыдущего экземпляра. В стек последовательно помещаются экземпляры (или все необходимые для их работы данные) с номерами 1, 2, 3, 4. Экземпляр с номером 5 – *терминальный*. После его завершения последовательно восстанавливаются и завершаются экземпляры с номерами 4, 3, 2, 1. В общем случае порождается последовательность рекурсивных вызовов с аргументами $n(1)$, $n(2)$, ..., $n(t)$ (индекс аргумента = номер экземпляра). Говорят, что *прокладывается трасса* $n(1)$, $n(2)$, ..., $n(t)$. Затем эта трасса *проходится* в обратном порядке, т. е. $n(t)$, $n(t-1)$, ..., $n(2)$, $n(1)$. Вычисления могут производиться как при прокладке трассы, так и при обратном проходе по ней. В данном примере все умножения производятся при обратном проходе по трассе, а если учесть, что сама трасса $n, n-1, n-2, ..., 1, 0$ заранее определяется по заданному n, то рекурсивный алгоритм вычисления fact(n) легко превратить в итеративный, описав вычисления, производимые при обратном проходе по трассе $0, 1, 2, ..., n-1, n$:

```
y:=1; {y=fact(0)}
for i:=1 to n do y:=y*i {y=fact(i)};
{y=fact(n)}
```

Приведём ещё один пример рекурсивной функции на языке Паскаль, а именно вычисление a^n, соответствующее рекурсивному определению (2.7):

```
function power(a: Float; n: Nat0): Float;
        var p: Float;
begin
        if n=0 then power:=1
        else    begin
                        p:= sqr(power(a, n div 2));
                        if Odd(n) then p:=p*a;
                        power:=p
                end
end {power}
```

Процесс вычисления power(a, 11)= a^{11} показан в табл. 2.2.

Таблица 2.2

Вызов функции и результат	Экземпляр функции и фаза исполнения
power(a, 11)= sqr(power(a, 5))$\cdot a$	экз1 (вызов)
power(a, 5)=sqr(power(a, 2))$\cdot a$	экз2 (вызов)
power(a, 2)=sqr(power(a, 1))	экз3 (вызов)
power(a, 1)=sqr(power(a, 0))$\cdot a$	экз4 (вызов)
power(a, 0)=1	экз5 (вызов и завершение)
power(a, 1)=sqr(1)$\cdot a=a$	экз4 (восстановление и завершение)
power(a, 2)=sqr(a)=a^2	экз3 (восстановление и завершение)
power(a, 5)=sqr(a^2)$\cdot a=a^5$	экз2 (восстановление и завершение)
power(a, 11)= sqr(a^5))$\cdot a=a^{11}$	экз1 (восстановление и завершение)

Фактически это рекурсивный вариант «индийского» бинарного алгоритма [2],[13] возведения в степень (или, точнее, из этого рекурсивного алгоритма легко извлекается «индийский» алгоритм).

В качестве примера использования рекурсивной *процедуры* рассмотрим решение *задачи о ханойских башнях*. Задача заключается в следующем. Имеются 3 колышка, на которые можно нанизывать диски (кольца) различного диаметра. В исходном состоянии все n дисков находятся на одном из колышков, так что каждый из дисков лежит на диске большего диаметра, т. е. диски расположены в виде пирамиды. Требуется переместить пирамиду с одного колышка на другой, используя третий как промежуточный. За один шаг можно переместить один диск с колышка на колышек. При этом диск с большим диаметром не может быть помещен на диск с меньшим диаметром.

Можно показать [9], что задачу нельзя решить менее чем за $2^n - 1$ ша-

гов (перемещений). Алгоритм, который решает задачу ровно за 2^n–1 шагов, является рекурсивным и основан на следующем простом соображении. Для удобства дадим колышкам названия: *откуда, куда, рабочий.* Пусть исходно диски лежат на колышке *откуда,* переместить их требуется на колышек *куда*, используя как промежуточный колышек *рабочий.* Решим задачу в 3 этапа. На первом этапе переместим верхнюю часть из (n –1)-го диска исходной пирамиды на колышек *рабочий* (считая, что умеем это делать). Второй этап состоит из одного шага – перемещения самого большого диска с колышка *откуда* на колышек *куда* (после этого он оказывается на своем окончательном месте). На третьем этапе пирамида из (n –1)-го диска перемещается с колышка *рабочий* на колышек *куда* (аналогично первому этапу) поверх находящегося там диска наибольшего диаметра. Задача решена.

Описанный способ действий представим в виде рекурсивной процедуры:

```
procedure Hanoi( n: Nat; откуда, куда, рабочий : Col);
begin
    if n=1 then Step(откуда, куда)
    else
    begin
        {1}   Hanoi( n-1, откуда, рабочий, куда);
              Step(откуда, куда);
        {2}   Hanoi( n-1, рабочий, куда, откуда)
    end {if}
end {Hanoi}
```

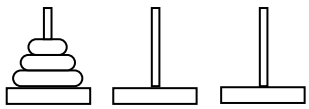
В процедуре Hanoi элементарный перенос одного диска реализован вызовом процедуры Step(*откуда, куда)*.

Пусть **type** Col=(a,b,c). Процесс выполнения Hanoi(3, a, b, c) показан в табл. 2.3. Заметим, что в отличие от предыдущих примеров процедура содержит 2 рекурсивных вызова, поэтому в стеке должно содержаться указание на номер вызова для того, чтобы продолжить работу восстановленного экземпляра с нужного места. По этой же причине преобразование рекурсивного алгоритма в итеративный (см. [9]) значительно сложнее, чем в предыдущих примерах.

Результат работы процедуры Hanoi в приведённом примере – прочитанная по табл. 2.3 сверху вниз последовательность элементарных шагов

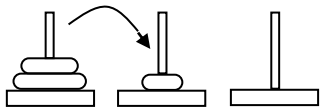
Таблица 2.3

Вызов функции и результат	Экземпляр функции и фаза исполнения
Hanoi(3, a, b, c)	экз1 (вызов)
{1}Hanoi(2,a,c,b)	экз2 (вызов 1 из экз1)
{1}Hanoi(1,a,b,c)≡Step(a,b)=*ab*	экз3 (вызов 1 из экз2 и завершение)
Step(a,c)=*ac*	экз2 (восстановление)
{2}Hanoi(1,b,c,a)≡Step(b,c)=*bc*	экз4 (вызов 2 из экз2 и завершение)
	экз2 (восстановление и завершение)
Step(a,b)=*ab*	экз1 (восстановление)
{2}Hanoi(2,c,b,a)	экз5 (вызов 2 из экз1)
{1}Hanoi(1,c,a,b)≡Step(c,a)=*ca*	экз6 (вызов 1 из экз5 и завершение)
Step(c,b)=*cb*	экз5 (восстановление)
{2}Hanoi(1,a,b,c)≡Step(a,b)=*ab*	экз7 (вызов 2 из экз5 и завершение)
	экз5 (восстановление и завершение)



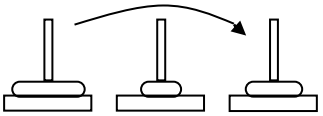
a *b* *c*

Исходное состояние



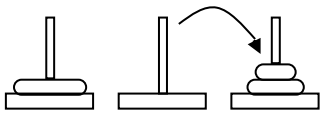
a *b* *c*

1-й перенос: *ab*



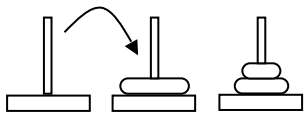
a *b* *c*

2-й перенос: *ac*



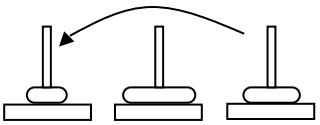
a *b* *c*

3-й перенос: *bc*



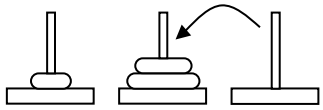
a *b* *c*

4-й перенос: *ab*



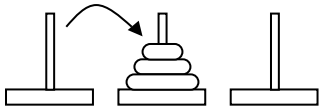
a *b* *c*

5-й перенос: *ca*



a *b* *c*

6-й перенос: *cb*



a *b* *c*

7-й перенос: *ab*

Решение задачи о Ханойских башнях при $n = 3$ (7 перекладываний)

Step(p,q)=pq, т. е. последовательность пар имён колышков: *ab ac bc ab ca cb ab*. Сам процесс переносов (в виде последовательности состояний колышков и дисков после каждого шага) изображен на рисунке.

2.3. Особенности программирования рекурсивных алгоритмов

Пример 2.1 *(Накапливающие параметры).* Вместо рекурсивного определения функции fact(n) рассмотрим вспомогательную функцию двух аргументов fct(n,m), которую определим рекурсивно и через которую выразим функцию fact(n):

fct(n,m) ≡ **if** n = 0 **then** m **else** fct(n – 1,m·n);

fact(n) ≡ fct(n,1).

Очевидно, что все умножения производятся при вычислении fct(n,m) по ходу прокладки трассы. Отсюда может быть получен другой вариант итеративного алгоритма вычисления факториала:

```
y:=1;
for i:=n downto 1 do y:=y*i      {y*fact(i-1)=fact(n)};
{y=fact(n)}
```

В некоторых случаях использование накапливающих параметров либо позволяет упростить переход к итеративному алгоритму, либо дает более эффективную версию рекурсивных вычислений (см., например, [14], с. 24, пример 1.5).

Приведём ещё один элементарный, но поучительный пример использования накапливающих параметров. Пусть требуется дать рекурсивное определение операции (функции) Минус1(n) вычитания единицы из натурального числа n, использующее только операцию прибавления единицы. Такая функция (операция) обладает свойствами Минус1(0) = 0 и Минус1(n + 1) = n. Введём функцию двух аргументов

$f(n,m)$ ≡ **if** n = 0 **then** 0 **else if** m + 1= n **then** m **else** $f(n, m + 1)$,

тогда Минус1(n)≡ $f(n,0)$.

Пример 2.2 *(Числа Фибоначчи)*. Для неотрицательных целых чисел (номеров) n последовательность Фибоначчи $\{F(n)\}$ определяется рекуррентным соотношением $F(n) = F(n-1) + F(n-2)$ при $n > 1$ с начальными условиями $F(0) = 0$, $F(1) = 1$. Легко написать рекурсивную функцию, вычисляющую $F(n)$, следуя непосредственно этому рекуррентному соотношению:

```
function Fib(n: Nat0): Nat0;
begin
    if n=0 then Fib:=0 else
        if n=1 then Fib:=1
            else  Fib:= Fib(n – 1)+Fib(n – 2)
end {Fib}
```

Процесс вычисления Fib(6) представлен в табл. 2.4. Очевидно, что при этом одни и те же значения функции вычисляются многократно. Например, количество вызовов Fib(0) равно 5, Fib(1) вызывается 8 раз, Fib(2) – 5 раз, Fib(3) – 3 раза, Fib(4) – 4 раза и Fib(5) – 1 раз. Можно показать [9], что число выполнений операции «+» при работе этой рекурсивной функции есть $F(n) - 1$. При больших n имеется оценка $F(n) \cong \Phi^n/\sqrt{5}$, где $\Phi = (1+\sqrt{5})/2 \cong 1.618$. Ясно, что такой способ вычислений очень неэффективен.

Причина этого не в недостатке самой рекурсии (и при программировании циклов возможны неэффективные решения), а в неудачном (неадекватном задаче) её применении. Действительно, так как используется рекуррентное соотношение второго порядка, то естественно вычислять сразу пару соседних чисел Фибоначчи. Для этого определим рекурсивную процедуру, вырабатывающую пару значений $F(n)$ и $F(n-1)$:

```
procedure Fib2(n: Nat; var u, v: Nat0);
{результат: u=F(n), v=F(n–1) при n>0}
begin
    if n=1 then begin u:=1; v:=0 end
        else
        begin
            Fib2(n-1, v, u);      {v=F(n-1), u=F(n-2)}
            u:=v+u                {u=F(n)}
        end {if}
end {Fib2}
```

Процесс вычислений при $n = 6$ представлен в табл. 2.5. Здесь при вызове процедуры значения выходных параметров (неопределенные) обозначены значком «*» (например, Fib2(4,*,*)), а при завершении процедуры значения выходных параметров u и v записаны жирным курсивом (например, Fib2(4,***3,2***)). В таком варианте рекурсии операция «+» выполняется лишь $n - 1$ раз. Очевидно, что этот способ вычислений аналогичен обычному итератив-

См. отдельную страницу

```
Fib(6)                                  !!!!!!!!!!!!!!!!!!!!!!!!!!8
    Fib(5)                              !!!!!!!!!!!!!!!5
        Fib(4)                          !!!!!!!!3
            Fib(3)                      !!!!!2
                Fib(2)                  !!!1
                    Fib(1)              !1
                    Fib(0)              !!0
                Fib(1)                  !!!!1
            Fib(2)                      !!!!!!!1
                Fib(1)                  !!!!!!1
                Fib(0)                  !!!!!!!0
        Fib(3)                          !!!!!!!!!!!!!2
            Fib(2)                      !!!!!!!!!!!1
                Fib(1)                  !!!!!!!!!!1
                Fib(0)                  !!!!!!!!!!!0
            Fib(1)                      !!!!!!!!!!!!!1
    Fib(4)                              !!!!!!!!!!!!!!!!!!!!!!!!!3
        Fib(3)                          !!!!!!!!!!!!!!!!!!2
            Fib(2)                      !!!!!!!!!!!!!!!!!1
                Fib(1)                  !!!!!!!!!!!!!!!!!1
                Fib(0)                  !!!!!!!!!!!!!!!!!!0
            Fib(1)                      !!!!!!!!!!!!!!!!!!!1
        Fib(2)                          !!!!!!!!!!!!!!!!!!!!!!1
            Fib(1)                      !!!!!!!!!!!!!!!!!!!!!1
            Fib(0)                      !!!!!!!!!!!!!!!!!!!!!!0
```

Таблица 2.5

Вызов функции и результат	Экземпляр функции и фаза исполнения
Fib2(6,*,*)	экз1 (вызов)
Fib2(5,*,*)	экз2 (вызов)
Fib2(4,*,*)	экз3 (вызов)
Fib2(3,*,*)	экз4 (вызов)
Fib2(2,*,*)	экз5 (вызов)
Fib2(1,*,*)	экз6 (вызов)
Fib2(1,***1,0***)	экз6 (завершение)
Fib2(2,***1,1***)	экз5 (восстановление и завершение)
Fib2(3,***2,1***)	экз4 (восстановление и завершение)
Fib2(4,***3,2***)	экз3 (восстановление и завершение)
Fib2(5,***5,3***)	экз2 (восстановление и завершение)
Fib2(6,***8,5***)	экз1 (восстановление и завершение)

ному варианту вычисления чисел Фибоначчи, где на каждом шаге цикла также вычисляется пара соседних чисел Фибоначчи.

2.4. Рекурсивные функции на последовательностях

Пусть в файле f_in типа Seq=**File of** Ordinal находится последовательность ω элементов типа Ordinal, для которого определено отношение «>». Требуется вычислить значение функции Max(ω) ='*максимальный элемент последовательности ω*'. Как известно [15], это индуктивная функция, которая может быть вычислена итеративно:

```
function Max_iter( var f_in: Seq): Ordinal;
    var x, y: Ordinal;
begin{прочитанная часть файла L(f_in) – пуста, а непрочитанная R(f_in)= ω}
    y:=MinOrdinal;
    {inv: y=Max(L(f_in));  bound: Length(R(f_in)) }
    while not Eof(f_in) do
    begin       Read(f_in, x);
                y:=max(x,y)
    end{while};
    Max_iter:=y
end{Max_iter}
```

Здесь использована функция max(*x*,*y*) нахождения максимального из

двух аргументов

```
function max( a, b: Ordinal): Ordinal;
begin      if a>b then max:= a else max:= b      end{max}
```

Решение задачи с помощью функции Max_iter даётся следующим фрагментом: Reset(f_in); y:=Max_iter(f_in); Close(f_in).

Рассмотрим *рекурсивную* функцию Max_rec для этой же задачи:

```
function Max_rec( var f_in: Seq): Ordinal;
    var x: Ordinal;
begin
    if Eof(f_in) then Max_rec:=MinOrdinal
    else  begin
                Read(f_in, x);
                Max_rec:=max(x, Max_rec(f_in))
          end{if}
end {Max_rec}
```

Решение даётся фрагментом: Reset(f_in); y:=Max_rec(f_in); Close(f_in).

На более абстрактном уровне можно определить функции-селекторы First и Rest над непустой последовательностью $\omega = x_1x_2...x_n$ (см. [15]):

First(ω)= x_1, Rest(ω)= $x_2x_3...x_n$.

Тогда рекурсивное определение Max(ω) примет вид

Max(ω)≡ **if not** Null(ω) **then** max(First(ω), Max(Rest(ω))) **else** MinOrdinal.

В общем случае для индуктивной функции $f(\omega)$ (f: $\Omega(X) \to Y$, где $\Omega(X)$ обозначает пространство последовательностей над алфавитом X) имеем

$f(\omega)$≡ **if not** Null(ω) **then** G(First(ω), f(Rest(ω))) **else** $f0$,

где $f0 = f$(NullSeq) и G – соответствующая функция $G: X\times Y \to Y$.

Рассмотрим вариант функции Max2rec с накапливающим параметром:

```
function Max2rec( var f_in: Seq; y: Ordinal): Ordinal;
    var x: Ordinal;
begin
    if Eof(f_in) then Max2rec:=y
    else  begin
                Read(f_in, x);
                Max2rec:=Max2rec(f_in, max(x,y))
          end{if}
end {Max2rec}
```

Решение даётся следующим фрагментом:

```
Reset(f_in); y:=Max2rec(f_in, MinOrdinal); Close(f_in).
```

Пусть функция неиндуктивна. Например, *f*(ω) = *'последовательность ω не убывает'*. Здесь *f*: Ω(Ordinal)→Boolean. Рассмотрим индуктивное расширение *F*(ω)=<*f*(ω), last(ω)> [15]. Используем накапливающий параметр B=*'прочитанная часть последовательности не убывает'*. Кроме того, учтём, что функция *f*(ω) имеет *стационарное* значение false [15]. Тогда рекурсивное вычисление функции *f*(ω) реализуется на языке Паскаль функцией IsSorted:

```
function IsSorted( var f_in: Seq; last: Ordinal; B: Boolean): Boolean;
    var x: Ordinal;
begin
    if Eof(f_in) or not B then IsSorted:=B
    else  begin
              Read(f_in, x);
              B:={B&} (x=>last);
              IsSorted:=IsSorted(f_in, x, B)
          end{if}
end { IsSorted }
```

Законченное решение даётся следующим фрагментом:

```
Reset(f_in); y:=IsSorted(f_in, MinOrdinal, true); Close(f_in).
```

В тех случаях, когда последовательность представлена в программе одномерным массивом (например, типа Vector=**array** [index] **of** T_Elem), индуктивная функция может быть вычислена следующим образом:

```
function fun (const a: Vector; n: index): T_fun;
{последовательность представлена массивом a[1..n]}
begin
    if n=1 then fun:=a[1]
    else  fun:=G(a[n], fun(a, n-1))
end {fun}
```

или в варианте с накапливающим параметром

```
function fun2 (const a: Vector; n: index; y: T_fun): T_fun;
begin
    if n=1 then  fun2:=G(a[1], y)
           else  fun2:=fun2(a, n-1, G(a[n],y))
end {fun2}
```

со стартовым вызовом fun:=fun2(a, n, f0), где f0=f(NullSeq).

Рекомендуется рассмотреть примеры с массивами при вычислении таких функций, как, например, $f(\omega)=\mathrm{Max}(\omega)$, $f(\omega)=\Sigma_{1..n}\, x_i$, $f(\omega)=\Pi_{1..n}\, x_i$ и т. п.

2.5. Взаимно-рекурсивные функции и процедуры

Пусть требуется построить *синтаксический анализатор* понятия *скобки:*

скобки::=квадратные | *круглые*

квадратные::=[*круглые круглые*] | +

круглые::=(*квадратные квадратные*) | –

В этом рекурсивном определении последовательности символов, называемой *скобки*, присутствуют две взаимно-рекурсивные части: *квадратные* определяются через *круглые*, и наоборот, *круглые* – через *квадратные.* В простейшем случае *квадратные* есть символ «+», а *круглые* есть символ «–». Другие примеры последовательностей, порождаемых этим рекурсивным определением:

‘[– –]’, ‘(++)’, ‘[(++)([–(++)][– –])]’, ‘(+[(++)([–(++)][(+[– –])–])])’.

Синтаксическим анализатором назовём программу, которая определяет, является ли заданная (входная) последовательность символов *скобками* или нет. В случае ответа «нет» сообщается место и причина ошибки.

Реализуем основную часть этой программы как булевскую функцию Bracket, которая вызывает две другие (парные) булевские функции Round и Square, определяющие, является ли текущая подпоследовательность частью *круглые* или *квадратные* соответственно. Каждая из функций Round и Square в свою очередь вызывает парную к себе (Square и Round соответственно). По правилам языка Паскаль описания этих функций в программе должны следовать в разделе описаний, например в такой последовательности:

```
function Round : Boolean; Forward;          {опережающее описание }
function Square : Boolean;
        begin
        … {тело функции}
        end{ Square };
function Round : Boolean;
        begin
        … {тело функции}
        end{ Round };
function Bracket: Boolean;
        begin
        … {тело функции}
        end{ Bracket }
```

Пусть входная последовательность читается из файла F, а результат и вспомогательные сообщения выводятся в файл G. Оба эти файла будут глобальными для функций Bracket, Round и Square. Вспомогательные сообщения квалифицируют ошибки в записи последовательности *скобки* в том случае, когда результат функции Bracket есть False. Для формирования этих сообщений будет использована процедура Error.

Функции Round и Square реализованы так, что они читают очередной символ входной последовательности и далее действуют в прямом соответствии с рекурсивными определениями частей *круглые* и *квадратные* соответственно. При этом в функции Bracket приходится читать первый символ входной последовательности дважды. Можно было бы избежать этого, используя «заглядывание вперёд», однако такая реализация менее прозрачна.

```
Program SyntaxAnalysisOfBracket;
{Bracket = скобки, Round = кругл, Square = квадр        }
{ скобки ::= квадр | кругл                              }
{ квадр ::= + | [кругл кругл]                           }
{ кругл ::= – | (квадр квадр)                           }

var F,G : Text;
        b: Boolean;

procedure Error (k: Word);
begin
 WriteLn(G);
 case k of
  1:{Bracket} WriteLn(G,'! - Лишние символы во входной строке.');
  2:{Bracket} WriteLn(G,'! - Недопустимый начальный символ.');
  3:{Square} WriteLn(G,'! - Отсутствует "]".');
  4:{Square} WriteLn(G,'! - Отсутствует "+" или "[".');
  5:{Square} WriteLn(G,'! - Очередной квадр - пуст.');
  6:{Round} WriteLn(G,'! - Отсутствует ")".');
  7:{Round} WriteLn(G,'! - Отсутствует "–" или "(".');
  8:{Round} WriteLn(G,'! - Очередной кругл - пуст.');
  else {?} WriteLn(G,'! - ...');
 end{case};
end{Error};

function Round : Boolean; Forward;
```

```
function Square : Boolean;
{ квадр ::= + | [кругл кругл] }
 var   s: Char;
       k: Boolean;
begin  Square := False;
       if not Eof(F) then
       begin Read(F,s); Write(G,s);
             if s='+' then Square := True
             else if s='[' then
             begin
                   { квадр ::= [кругл кругл] }
                   k := Round;
                   if k then k:={k and } Round
                   else {первый кругл ошибочен};
                   if k then  {оба кругл правильны}
                         if not Eof(F) then
                         begin
                               Read(F,s); Write(G,s);
                               if (s=']') then Square:=True else Error(3)
                         end
                         else {нет ]} Error(3)
                   else {хотя бы один кругл ошибочен};
             end {конец анализа [кругл кругл]}
             else { не + и не [ } Error(4)
       end
       else {квадр - пуст} Error(5)
end {Square};

function Round : Boolean;
{ кругл ::= – | (квадр квадр)  }
 var   s: Char;
 k: Boolean;
begin  Round := False;
       if not Eof(F) then
       begin Read(F,s); Write(G,s);
             if s='–' then Round := True
             else if s='(' then
             begin { кругл ::= (квадр квадр) }
                   k := Square;
                   if k then k:={k and }Square
                   else {первый квадр ошибочен};
                   if k then {оба квадр правильны}
```

```
                    if not Eof(F) then
                    begin
                        Read(F,s); Write(G,s);
                        if (s=')') then Round:=True else Error(6)
                    end
                    else {нет )} Error(6)
                else {хотя бы один квадр ошибочен};
            end {конец анализа (квадр квадр)}
            else { не – и не ( } Error(7)
        end
        else {кругл – пуст} Error(8)
end {Round};
function Bracket : Boolean;
{ not Eof(F) }
 var b: Boolean;  c: Char;
begin
        b:=False;
        Read(F,c); Reset(F);
        if  (c='+') or (c='[') then b:=Square
        else    if (c='–') or (c='(') then b:=Round
                else {недопустимый начальный символ} Error(2);
        Bracket := b and Eof(F);
        if b and not Eof(F) then {лишние символы} Error(1);
end {Bracket};
begin { SyntaxAnalysisOfBracket }
        Assign(F, 'Bracket.DAT'); Assign(G, 'BracketRes.DAT');
        Reset (F); {Rewrite} Append(G);
        WriteLn(G,'Анализатор скобок:');
        if not Eof(F) then
        begin
            b := Bracket;
            WriteLn(G);
            if b then  WriteLn(G,'ЭТО СКОБКИ!')
            else  WriteLn(G,'НЕТ, ЭТО НЕ СКОБКИ!');
        end
        else WriteLn(G,'Пусто!');
        Close(G);
end.
```

Отметим взаимную симметричность текстов процедур Round и Square, что соответствует симметричности определения понятия *скобки* относительно частей *круглые* и *квадратные* соответственно.

Результаты выполнения программы на некоторых тестовых данных приведены в табл. 2.6.

Таблица 2.6

Номер теста	Исходные данные (файл Bracket.dat)	Результат (файл BracketRes.dat)
1	(++)	Анализатор скобок: (++) ЭТО СКОБКИ!
2	[(++)([–(++)][– –])]	Анализатор скобок: [(++)([–(++)][– –])] ЭТО СКОБКИ!
3	[(++)([–(+–)][– –])]	Анализатор скобок: [(++)([–(+– ! - Отсутствует "+" или "[". НЕТ, ЭТО НЕ СКОБКИ!
4	[(++)([–(++)][– –])](	Анализатор скобок: [(++)([–(++)][– –])] ! – Лишние символы во входной строке. НЕТ, ЭТО НЕ СКОБКИ!

Программа в процессе анализа входной последовательности выводит её либо целиком, если она правильная (см., например, тесты 1 и 2), либо до того символа, который является ошибочным (см., например, тесты 3 и 4). Сообщение об ошибке выводится с новой строки после знака «!».

2.6. Цель, требования и рекомендации к выполнению задания

Цель выполнения задания: ознакомиться с основными понятиями и приёмами рекурсивного программирования, получить навыки программирования рекурсивных процедур и функций на языке программирования Паскаль.

Требования и рекомендации к выполнению задания:

1) проанализировать полученное задание, выделив рекурсивно определяемые информационные объекты и (или) действия;

2) разработать программу, использующую рекурсию;

3) сопоставить рекурсивное решение с итеративным решением задачи;

4) сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи.

2.7. Задания

1. Для заданных неотрицательных целых n и m вычислить (рекурсивно) биномиальные коэффициенты, пользуясь их определением:

$$C_n^m = \begin{cases} 1, \text{если } m = 0, n > 0 \text{ или } m = n \geq 0, \\ 0, \text{если } m > n \geq 0, \\ C_{n-1}^{m-1} + C_{n-1}^m \text{ в остальных случаях.} \end{cases}$$

2. Задано конечное множество имен жителей некоторого города, причем для каждого из жителей перечислены имена его детей. Жители X и Y называются *родственниками*, если (а) либо X – ребенок Y, (б) либо Y – ребенок X, (в) либо существует некоторый Z, такой, что X является родственником Z, а Z является родственником Y. Перечислить все пары жителей города, которые являются родственниками.

3. Имеется n городов, пронумерованных от 1 до n. Некоторые пары городов соединены дорогами. Определить, можно ли попасть по этим дорогам из одного заданного города в другой заданный город. Входная информация о дорогах задаётся в виде последовательности пар чисел i и j ($i < j$ и $i, j \in 1..n$), указывающих, что i-й и j-й города соединены дорогами.

4. Напечатать все перестановки заданных n различных натуральных чисел (или символов).

5. Функция $f(n)$ определена для целых положительных чисел:

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ \sum_{i=2}^{n} f(n \text{ div } i), & \text{если } n \geq 2. \end{cases}$$

Вычислить $f(k)$ для k= 15, 16,…, 30.

6. Построить синтаксический анализатор для понятия *простое выражение.*

простое_выражение::=простой_идентификатор |

(*простое_выражение знак_операции простое_выражение*)

простой_идентификатор::= буква

знак_операции:: = – | + | *

7. Построить синтаксический анализатор для понятия *вещественное число*.

вещественное_число::= *целое_число* . *целое_без_знака* |
целое_число . *целое_без_знака*E*целое число* |
*целое_число*E*целое_число*

целое_без_знака::=цифра | *цифра целое_без_знака*

целое_число::=целое_без_знака | + *целое_без_знака* | –*целое_без_знака*

8. Построить синтаксический анализатор для понятия *простое_логическое.*

простое_логическое::= TRUE | FALSE | *простой_идентификатор* |
NOT *простое_логическое* |
(*простое_логическое знак_операции простое_логическое*)

простой-идентификатор::=буква

знак-операции::= AND | OR

9. Разработать программу, которая по заданному *простому_логическому* выражению (определение понятия см. в предыдущей задаче), не содержащему вхождений простых идентификаторов, вычисляет значение этого выражения.

10. Построить синтаксический анализатор для определяемого далее понятия *константное_выражение.*

константное_выражение::=ряд_цифр|
константное_выражение знак_операции константное_выражение

знак_операции::=+ | – | *

ряд_цифр::=цифра | *цифра ряд_цифр*

11. Написать программу, которая по заданному (см. предыдущее задание) *константному_выражению* вычисляет его значение либо сообщает о переполнении (превышении заданного значения) в процессе вычислений.

12. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые* | *фигурные*

квадратные::=[*круглые фигурные*] | +

круглые::=(*фигурные квадратные*) | –

фигурные::={*квадратные круглые*} | 0

13. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | *скобка скобки*

скобка::= (B *скобки*)

14. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | (B *скобки скобки*)

15. Построить синтаксический анализатор для понятия *скобки*.

скобки::=A | A (*ряд_скобок*)

ряд_скобок::= *скобки* | *скобки* ; *ряд_скобок*

16. Построить синтаксический анализатор для понятия *скобки*.

скобки::=A | B | (*скобки скобки*)

17. Функция Ф преобразования текста определяется следующим образом (аргумент функции – это текст, т. е. последовательность символов):

$$\Phi(\alpha)=\begin{cases}\Phi(\gamma)\beta, \text{ если } \alpha=\beta/\gamma \text{ и текст } \beta \text{ не содержит вхождений символа «/»},\\ \alpha, \text{ если в } \alpha \text{ нет вхождений символа «/».}\end{cases}$$

Например: Ф(«ла/ска»)=«скала», Ф(«б/ру/с»)=«сруб», Ф(«ца/ри/ца»)= «царица», Ф(«ум/ри/ва/к/а»)= «аквариум». Реализовать функцию Ф рекурсивно.

18. Пусть определена функция Ф преобразования целочисленного вектора α:

$$\Phi(\alpha)=\begin{cases}\alpha, \text{ если } \|\alpha\|=1,\\ ab, \text{ если } \|\alpha\|=2, \alpha=ab \text{ и } a\le b,\\ ba, \text{ если } \|\alpha\|=2, \alpha=ab \text{ и } b<a,\\ \Phi(\beta)\Phi(\gamma) \text{ , если } \|\alpha\|>2, \alpha=\beta\gamma, \text{где } \|\beta\|=\|\gamma\| \text{ или } \|\beta\|=\|\gamma\|+1.\end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,3,4,5; Ф(4,3,2,1) = 3,4,1,2; Ф(4,3,2) = 3,4,2. Отметим, что функция Ф преобразует вектор, не меняя его длину. Реализовать функцию Ф рекурсивно.

19. Функция Ф преобразования целочисленного вектора α определена следующим образом:

$$\Phi(\alpha)=\begin{cases}\alpha, \text{если } \|\alpha\|\le 2,\\ \Phi(\beta)\Phi(\gamma), \text{если } \alpha=\beta\gamma, \|\beta\|=\|\gamma\|, \|\alpha\|>2,\\ \Phi(\beta\, a)\,\Phi\,(a\,\gamma), \text{ если } \alpha=\beta a\gamma, \|\beta\|=\|\gamma\|, \|\alpha\|>2, \|a\|=1.\end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,2,3,3,4,4,5; Ф(1,2,3,4,5,6) = 1,2,2,3,4,5,5,6; Ф(1,2,3,4,5,6,7) = 1,2,3,4,4,5,6,7; Ф(1,2,3,4,5,6,7,8) = 1,2,3,4,5,6,7,8. Отметим, что функция Ф может удлинять вектор. Реализовать функцию Ф рекурсивно.

20. Построить синтаксический анализатор понятия *список_параметров*.

список_параметров::= *параметр* | *параметр , список_параметров*

параметр::= *имя*=*цифра цифра* | *имя*=(*список_параметров*)

имя::= *буква буква буква*

21. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые*

квадратные:: = [[*квадратные*] (*круглые*)] | B

круглые::=((*круглые*) [*квадратные*]) | A

22. Построить синтаксический анализатор для определённого далее понятия *логическое_выражение*.

логическое_выражение ::= TRUE | FALSE |*идентификатор* |
NOT (*операнд*) |*операция* (*операнды*)

идентификатор::=буква

операция::= AND | OR

операнды::= операнд |*операнд, операнды*

операнд::= логическое_выражение

23. Разработать программу, которая, имея на входе заданное (см. предыдущее задание) *логическое_выражение*, не содержащее вхождений идентификаторов, вычисляет значение этого выражения и печатает само выражение и его значение.

24. Построить синтаксический анализатор для понятия *текст_со_скобками.*

текст_со_скобками::=элемент | *элемент текст_со_скобками*

*элемент::=*A |B | (*текст_со_скобками*) | [*текст_со_скобками*]|
{ *текст_со_скобками* }

25. Построить синтаксический анализатор для параметризованного понятия *скобки(T)*, где *T* – заданное конечное множество, а круглые скобки «(» и «)» не являются терминальными символами, а отражают зависимость определяемого понятия от параметра *T*.

скобки(T)::=элемент(T) | *список(скобки(T))*

*список(E)::=*N | [*ряд(E)*]

ряд(E)::=элемент(E) | *элемент(E) ряд(E)*

3. ОБРАБОТКА СТРОК

3. 1. Представление строк с помощью записи

При реализации алгоритмов будем использовать 2 варианта внутреннего представления строк – с известной длиной и с известным маркером конца строки. В обоих вариантах в программе используется тип *запись*.