Informatics in Education, 2007, Vol. 6, No. 2, 283–306

© 2007 *Institute of Mathematics and Informatics, Vilnius*

Recursion Versus Iteration with the List as a Data Structure

Izabella FOLTYNOWICZ
Theoretical Chemistry Department, Faculty of Chemistry, Adam Mickiewicz University
ul. Grunwaldzka 6, PL 60-780, Poznań, Poland
e-mail: iza@rovib.amu.edu.pl

Received: November 2006

Abstract. A reversible sequence of steps from the specification of the algorithm and the mathematical definition of the recurrent solution through the recursive procedure, the tail recursive procedure and finally to the iteration procedure, is shown. The notation for analysing recursive function execution as well as modified flow charts of an algorithm to identify the differences between the iteration and the tail recursion are proposed. All the procedures are written in Logo, so the lists are used as the data structure. Transformation from the recursive procedure to the iterative procedure and vice versa can be shown in such a way in every language in which the recursion is allowed. All examples are one-recursion-call examples and all except one are the functions of discrete mathematics.

Key words: recursion, iteration, tail-end recursion, Logo, loop, list (data structure), discrete mathematics.

1. Introduction

This article deals with one of the main concepts of programming: iteration versus recursion. I describe a sequence of steps developed while working in computer labs with my students of the first year of the master's degree programme of "chemistry with informatics". Many of them are not well prepared or even unprepared by the secondary school to study informatics. The name of the course is "Algorithms and data structures" but in view of the students' level it must contain also the basic principles of programming. I decided to use Imagine Logo (Kalaš *et al.*, 2001) as one of the tools. Logo is underestimated and easy to use educational tool, especially for recursion. The main difference between Logo and the other languages from the point of view of this article is that Logo operates on list, similarly to all dialects of Lisp. While working with students I realized the necessity of using one well-defined and convenient way of notation for analysis of the procedure working. I think that the meaning of the use of any notations (and annotations) is out of question not only for me. For example (McCartney *et al.*, 2005) wrote: "any annotation was better than none". I would extend it to say that a good notation is of key importance (I do hope to be an exception from the rule to leave well alone). There are a lot of notations and the problem is which one to suggest to our students, to become the most convenient, clearest, unique and fruitful (I assume that there really exist one notation that

is good for all). Finally I have come to an effective way of notation which, in my opinion, fulfilled these criteria, and employed it successfully at my lab classes. As follows from my experience, it is better to suggest one good way of notation at the beginning of the course than to struggle with a lot of other ideas later. In the tests and exam's questions I used to put blank tables to be filled in to avoid ambiguity. Certainly I must remember that the sequence of steps I propose, as every method, has its own restrictions. In particular, this method works well for one-recursion-call examples. The work of the procedure in two-recursion-call examples will be shown in the next paper (Foltynowicz, 2007).

Every exercise starts from a specification of the algorithm: the definition of the inputs (parameters of a function) and outputs (returns) taking into regard the data structures. Than we formulate the mathematical recursive solution: base case and general, recursion case, and translate it into a function written in Logo. After this we ought to analyse the work of the recursive procedure and try to improve it by removing the collecting output phase at the expense of creating an additional accumulating parameter (in other words transform the recursion to the tail-end recursion). For the purpose of analysing the work of the recursive and tail recursive procedures I suggest the table with tree columns: the first for the logic value of the stop condition, the second for the recursive call phase and the third for the collecting output phase. The arrows shows the order of execution ruled by the stack (based on the principle *Last In First Out* = LIFO). The tail recursion (or tail-end recursion) is a special case of recursion that can be easily transformed into an iteration (for example 4). Replacement of the recursion with iteration, can drastically decrease the amount of stack space used and improve efficiency but is rather less intuitive than recursion. Every recursive version has an equivalent iterative version and vice versa. For the purpose of analysing the work of the procedure we make a table which needs to be well defined. The table notation for demonstrating the working of iterative algorithms is not novel and is shown for completeness and clarity of the whole process. The number of its columns is equal to the number of variable names used in the procedure with one additional column for logic values of the loop condition. The number of rows is equal to the number of values assigned to the variable by the assignment statement during the execution of the procedure for given data i.e. the number of rows is the number of iterations plus one (the first row is for the initial values before the loop). The order of filling the table is from left to right and must be the same as the order of execution of the statements in the algorithm. For explanation of the differences between the iterative and recursive versions of the function the modified flow charts for the functions are proposed. They begin from the name of the procedure (function) and the names of the formal parameters and end with the variable name returned by the function. The arrows show the direction of overwriting the subsequent values of actual procedure parameters.

Testing the execution of the function for given data it is worth discussing the place of putting in some checking code, which usually is the print statement of the values of the function parameters. The working of the method is illustrated in several simple and important examples of discrete mathematics. As follows from my experience, such a training helps to understand the interrelation between the mathematical definition, recursive and iterative procedures and flow chart, which are all different ways of algorithm notation. If

we carry out the analysis of several examples in such an ordered way, we quickly discover the following two rules:

1. The loop condition for the most general (in the sense of being always possible to use) *while* loop is a logical negation of the base case in recursion.
2. In the recursive version the assignment statements of iterative version are replaced by exchanging subsequent values of actual procedure parameters (it should be emphasised that these values are overwritten: the next on the previous one as in the assignment statement).

It sounds like conclusions of the paper, but I decided to place them in the introduction because these are the conclusions for learners, not for teachers, and they ought to be discovered by students. Discovering is more time consuming and more valuable then suggesting the rules at the beginning of the process.

Regularities and invariants which, without any doubts, are essential for considerations of both correctness and efficiency (Ginat, 2003) will be only mentioned at the level of beginners.

I hope that the method of transforming a recursive function into an iterative one, shown below, will turn to be useful for every teacher. All algorithms can be executed for small integer numbers so all examples are good for written exercises and tests done without help of a calculator or a computer. In the lab classes the use of a computer is obligatory for trying and checking.

The first one-recursion-call example, factorial of a number, the most frequently used in literature, is treated as the model and is described in detail.

2. Factorial of a Number

1. Specification of the algorithm:
   Input: integer number $n \geqslant 0$.
   Output: $n!$
2. Mathematical definition of recursive solution:
$$\mathsf{Factor}(n) = \begin{cases} 1 & n = 0, \\ n \cdot \mathsf{Factor}(n-1) & n \geqslant 1. \end{cases} \tag{1}$$
3. Logo recursive procedure and its work:
```
to FACTOR :n
  if :n = 0 [op 1]
  op :n * FACTOR :n – 1
end
```
(2)

   where op means output, procedure is the function which gives the value as the output (the same as *return* in pseudocode and in most of programming languages). The calling of the procedure and the subsequent result are shown below:
```
? show FACTOR 5
120
```

In Table 1 trace of the procedure execution for the actual parameter $n = 5$ is placed. Stop condition comes from the base case. The arrows shows the order of execution. The rule is simple: what cannot be executed must be written symbolically onto the stack (recursive call column) and after reaching the stop condition it must be executed (collecting output phase column). The recursive call phase column shows symbolically that the recursive call requires the compiler to allocate storage on the stack at run-time for every call that has not yet returned.

Table 1

Stop condition: :n = 0	Recursive call phase	Collecting output phase
	? FACTOR 5	
0	5 * FACTOR 4	120
0	4 * FACTOR 3	24
0	3 * FACTOR 2	6
0	2 * FACTOR 1	2
0	1 * FACTOR 0	1
1	1	1

The procedures equivalent to the procedure (2), but written in a slightly different way, are shown only in Table 2. Such differences in notation do not introduce any new information to the subject of this work, and will therefore not be considered further.

Table 2

to factorial1 :n	to factorial2 :n	to factorial3 :n
if :n < 2 [op 1]	op ifElse :n < 2 [1]	test :n < 2
op :n * factorial1 :n – 1	[:n * factorial2 :n – 1]	ifTrue [op 1]
end	end	ifFalse [op :n * factorial3 :n – 1]
		end

4. Since the transformation of the recursion into tail recursion implies an introduction of an additional (accumulating) parameter into the function, the question arises on the initial value of this parameter. In other words: we must answer the question what is the value of the parameter this function must be called with:

   Mathematical definition:

$$\text{Factorp}(n, f) = \begin{cases} f & n = 0, \\ \text{Factorp}(n-1, n \cdot f) & n \geqslant 1. \end{cases} \tag{1p}$$

5. Logo tail recursive procedure and its work:

```
to FACTORp :n :f
  if :n = 0 [op :f]
  op FACTORp :n – 1 :n * :f                    (2p)
end
```

```
?show FACTORp 5 1
120
```

Nothing has to be done after the recursive call, the solution is collected in the additional parameter. The tail recursive function is less memory consuming then the recursive one, there is no collecting output phase during the execution and usually the compiler optimizes the tail recursive version to include the simple loop.

Table 3

Stop condition: :n = 0	Recursive call phase	Collecting output phase
	? FACTORp 5 1	
0	FACTORp 4 5	
0	FACTORp 3 20	
0	FACTORp 2 60	
0	FACTOR 1 120	
0	FACTOR 0 120	
1	120	

6. Recursion versus tail recursion:

   In Table 4 two procedures (2) and (2p) are placed once more for exact comparison and some symbols are bold. It is worth watching carefully what happens: the operator **(n *)** acts on the additional parameter **:f** instead of the recursive call FACTOR :n - 1, parameter **:f** is outputted instead of **1** in the base case:

Table 4

to FACTOR :n	to FACTORp :n **:f**
if :n = 0 [op **1**]	if :n = 0 [op **:f**]
op :**n** * FACTOR :n – 1	op FACTORp :n – 1 **:n * :f**
end	end

We used to say that tail recursion (or tail-end recursion) is a special case of recursion in which the last operation of the function is a recursive call. What does it mean? It means that if the recursive call is an element of the arithmetic formula (the case of FACTOR) or is the parameter of other procedure (the case of decbin for example; see below) the recursion is not tail recursion. Fig. 1 shows how the difference between recursion and tail recursion could be illustrated by the flow charts modified for functions.

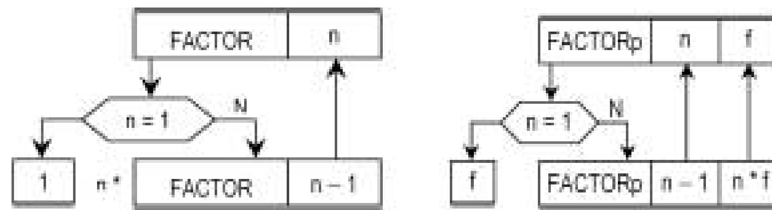

Fig. 1. Flow charts of recursive and tail recursive versions of the factorial function.

7. Iterative version written in Logo.

```
to Factorit :n
 let "f 1
 repeat :n [let "f :f * repc]        (2it)
 op :f
end
```

In Imagine Logo in loops: repeat, for, while and forEach can be used repcount (short: repc). It outputs a positive integer, which is the number of repetitions. It begins from 1 Repcount behaves like a variable which is always increased by 1 in the body of the loop. Repeat loop is the most typical loop of Logo.

repeat :n [*body of the loop which is the list of statements*]

We must remember that the typical data structure in Logo is the list. It is worth emphasizing that there is no loop (which will terminate) without the loop condition, even if it is not explicitly written in the language. The loop condition for repeat loop is 1 <= repc <= :n of course. The procedure performance for $n = 5$ is shown in Table 5.

Table 5

variables		loop condition:
:f	repc	repc <= :n
1	1	1
1*1=1	2	1
1*2 = 2	3	1
2*3 = 6	4	1
6*4 = 24	5	1
24*5 = 120	6	0

In Logo, in contrast to other popular programming languages, the repeat loop is more convenient than the for loop, because in the last one the end value of repeats cannot be the name of the variable, it must be a number. The way to overcome this difficulty is to make a list by the sentence (se) or list procedure. All alternative notations of the iterative procedure (2it) are collected in Table 6.

Table 6

to Factorw1 :n let "f 1 while [repc <= :n] [let "f :f * repc] op :f end	to Factorw :n let "f 1 while [:n > 0] [let "f :f * :n let "n :n – 1] op :f end
to Factor1 :n let "f 1 for "i list 1 :n [let "f :f * :i] op :f end	to Factor_1 :n let "f 1 for "i (list :n 1 –1) [let "f :f * :i] op :f end

8. Iteration versus tail recursion

   Iterative version was chosen in its most general version: with *while* loop (this kind of loop is possible to use in every case) and with loop condition being the logic negation of the stop condition from the recurrent version[1] and placed below in the Table 8 with the tail recursive version for comparison. The performance of such procedure (FACTORit) is the same as the tail recursive one (FACTORp). Let us for example see (Table 7) how the procedure FACTORit is executed for the actual parameters: n = 5 and :f = 1:

Table 7

variables		loop condition:
:f	:n	:n <> 1
1	5	1
1*5=5	4	1
5*4 = 20	3	1
20*3 = 60	2	1
60*2 = 120	1	0

Fig. 2 shows how the difference between iteration and tail recursion could be illustrated by the flow charts modified for functions. Roughly speaking we can say that every loop (while, for, repeat) works "horizontally" and every recursion works "vertically". As a consequence, the statements placed between the beginning of the procedure and the loop condition are performed only once (they are placed outside the body of the loop) in contrast to the instruction placed between the beginning of the recursive procedure and the stop condition, which are performed as many times as the procedure is called by itself (they are placed inside the body of the loop). So we should not place the instruction that must not be repeated between the beginning of the recurrent procedure and the place of

[1] The stop condition could be :n = 1 if we assume :n >= 1

Table 8

iteration	tail recursion
to FACTORit :n :f while [:n <> 1] [let "f :f * :n let "n :n – 1] op :f end	to FACTORp :n :f if :n = 1 [op :f] op FACTORp :n – 1 :n * :f end
? show FACTORit 5 1 120	? show FACTORp 5 1 120

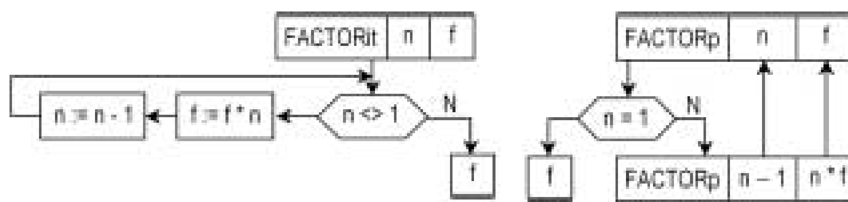

Fig. 2. Flow charts of iterative and tail recursive versions of the factorial function.

its recursive call. Generally, we can also say that in the recurrent way of the realization of repeated processes we use the procedure parameters: some of them are the data which do not change, some of them change during the recurrent calls. At first the formal parameters are replaced by the actual ones. Then the current actual parameters are exchanged by overwriting of the succeeding values of the parameters. The process of exchanging of the procedure parameters in the procedure which contains calling itself is replaced by the assignment statements which are an indispensable part of the body of the loop in the iterative version.

Tail recursion is the special case of recursion that is semantically equivalent to the iteration constructs normally used to represent repetition in programs. Because tail recursion is equivalent to iteration, tail-recursive programs can be compiled as efficiently as iterative programs.

In Appendix 1 four easy examples without structural data and two with the list as a data structure are presented in the form of a laboratory exercise. Euclidean algorithm for finding the greatest common divisor is very important but not typical: it is defined by tail-end recursion.

3. Arithmetic of Integers: Two Integer Functions div and mod

Although these functions are usually implemented in programming languages it is worth performing this task to understand better what they really are.

1. Specification of the algorithm:
   Inputs: two natural numbers a, and b

Outputs: the quotient of an integer division a by b: $a\ div\ b$, and the remainder of division a by b: $a\ mod\ b$

2. Mathematical definition of recursive solution
   These two function are interrelated:

$$div\,(a,b) = \begin{cases} 0 & a < b, \\ 1 + div(a-b,b) & a >= b; \end{cases}$$
$$\operatorname{mod}(a,b) = \begin{cases} a & a < b, \\ \operatorname{mod}(a-b,b) & a >= b. \end{cases}$$

3. Recursive version and its execution (Table 9):

```
to divmod :a :b
 if :a < :b [show :a op 0]
 op 1 + divmod :a - :b :b
end
```

```
? show divmod 15 7
1
2
```

Table 9

Stop condition: :a < :b	Recursive call phase	Collecting output phase
	? divmod 15 7	
0	1 + divmod 8 7 ↓	↑ 2
0	1 + divmod 1 7	1
1	0	0

4. The tail-end recursive version:

$$div\ \text{mod}\ (a,b,q) = \begin{cases} q, a & a < b, \\ div\ \text{mod}\ (a-b,b,q+1) & a >= b. \end{cases}$$

```
to divmod :a :b :q
 if :a < :b [op se :q :a]
 op divmod :a – :b :b :q + 1
end
```

```
? show divmod 6 7 0
[0 6]
```

```
? show divmod 15 7 0
[2 1]
```

5. Iterative version and its execution (Table 11)

Table 10

Stop condition: :a < :b	Recursive call phase	Collecting output phase
	? divmod 15 7 0	
0	divmod 8 7 1	
0	divmod 1 7 2	
1	[1 2]	

For clarity data a and b do not change, variable r is used for changing a.

```
to divmod :a :b
 let ”q 0 let “r :a
 while [:r >= :b][let ”q :q + 1 let ”r :r – :b]
 let ”L se :q :r
 op :L
end
```

```
? show divmod 15 7
[2 1]
```

In the Table 11 additionally the invariant (Kenneth and Wright, 1992) is shown.

Table 11

Variables:		Loop codition:	Invariant:
q	r	r >= n	q · n + r = m and r >= 0
0	15	true (15 > 7)	true (0·7+15=15 and 15>0)
1	8	true (8 > 7)	true (1·7+8=15 and 8>0)
2	1	false (1 < 7)	true (2·$7 + 1 = 15$ and 1>0)

The idea of checking the correctness by inverting this algorithm is not good, because although the correspondences $(a, b) \to a\ div\ b$ and $(a, b) \to a\ mod\ b$ are unique (are two two-dimensional discrete functions, but not one-to-one functions), the invert correspondences are not functions at all (the inverse function of the two-dimensional not one-to-one function does not exist). It would be instructive to show to the students the plots of these two discrete functions but it is not easy to find a proper tool. The Excel *3-D Surface* plot type of the functions $f(x, y) = z = x\ div\ y$ and $f(x, y) = z = x\ mod\ y$ could be well understood if we remembered that the sets of x, y and z contained discrete values. Compare the plots made in Excel with the ones made in Sigma-Plot, which are really discrete. I hope it helps understand what these function really are.

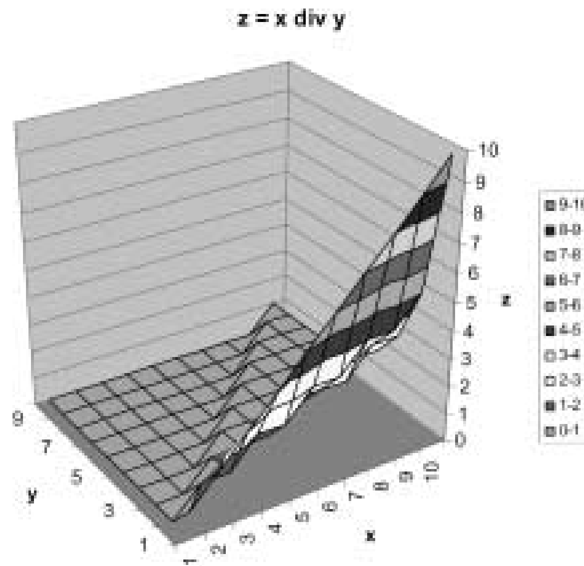

Fig. 3. Plot of the function $z = x\ div\ y$ made in Excel.

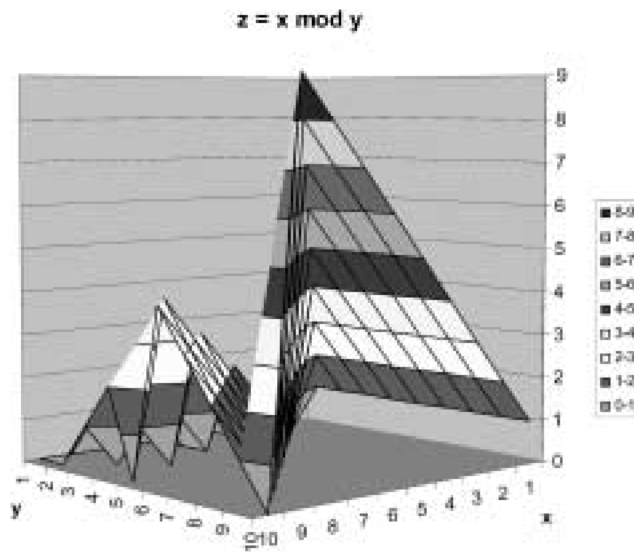

Fig. 4. Plot of the function $z = x\ mod\ y$ made in Excel.

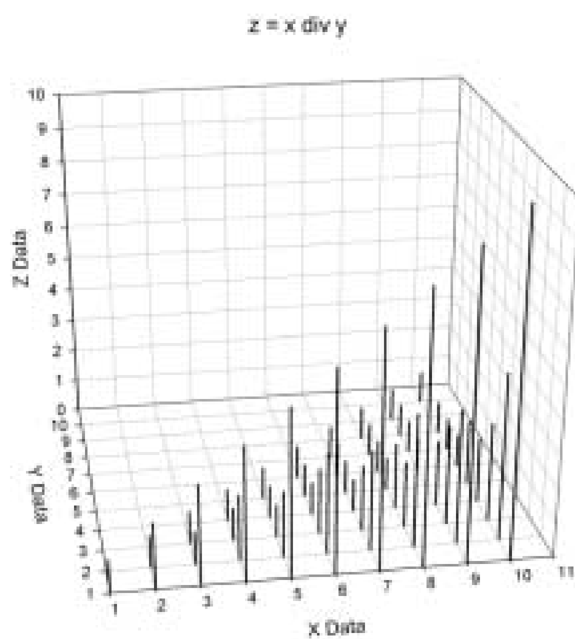

Fig. 5. Plot of the function $z = x\ div\ y$ made in Sigma-Plot.

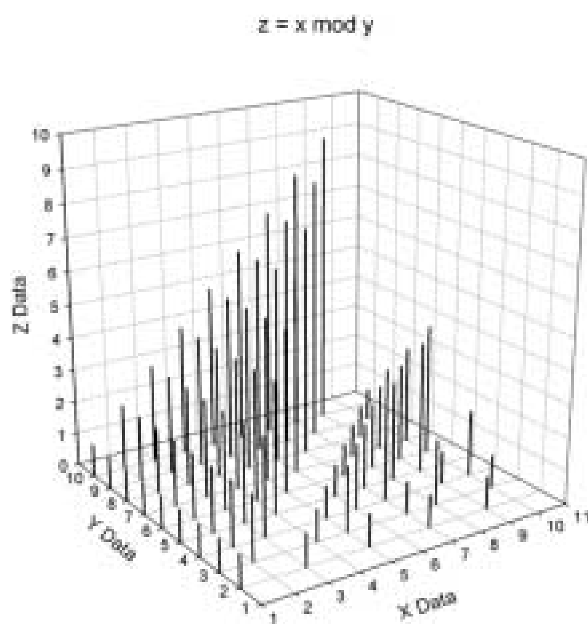

Fig. 6. Plot of the function $z = x\ mod\ y$ made in Sigma-Plot.

4. Transformation from Decimal to Binary Representation of a Given Number

1. Specification of the algorithm:
   Input: integer number $n \geqslant 1$ (in decimal representation).
   Output: string or list of bits being the binary representation of n (output is not a simple variable – it is structured).
2. Mathematical definition of recursive solution:

$$decbin(n) = \begin{cases} 1 & n = 1, \\ n \bmod 2 \text{ put as last } decbin(ndiv2) & n > 1. \end{cases}$$

3. Logo recursive procedure and testing its execution (Table 12).

```
to decbin :n
 if :n = 1 [op 1]
 op lput mod :n 2 decbin div :n 2
end
```

```
? show decbin 12
   1100
```

Table 12

Stop condition: :n =1	Recursive call phase	Collecting output phase
	? decbin 12	
0	lput 0 decbin 6 ↓	↑ 1100
0	lput 0 decbin 3	110
0	lput 1 decbin 1	11
1	1	1

4. The tail-end recursion:

$$decbin(n,p) = \begin{cases} p \text{ with 1 putted as first} & n = 1, \\ decbin(n\ div\ 2, n \bmod\ 2 \text{ put as first of } p) & n > 1; \end{cases}$$

or

$$decbin(n,p) = \begin{cases} p & n = 0, \\ decbin(n\ div\ 2, n\ \bmod\ 2 \text{ put as first of } p) & n > 1. \end{cases}$$

```
to decbinp :n :p
 if :n = 1 [op fput 1 :p]
 op decbinp ( div :n 2 ) ( fput mod :n 2 :p )
end
```

Using a string (Table 13):

```
? show decbinp 12 ”
1100
```

Table 13

Stop condition: :n=1	Recursive call phase	Collecting output phase
	? decbinp 12 “	
0	decbinp 6 0	
0	decbinp 3 00	
0	decbinp 1 100	
1	1100	

or using a list (Table 14):

```
? show decbinp 12 []
[1 1 0 0]
```

Table 14

Stop condition: :n=1	Recursive call phase	Collecting output phase
	? decbinp 12 [ ]	
0	decbinp 6 [0]	
0	decbinp 3 [0 0]	
0	decbinp 1 [1 0 0]	
1	[1 1 0 0]	

5. Recursion versus tail recursion:
It is worth watching carefully what happens (Table 15).

Table 15

to decbin :n	to decbinp :n **:p**
if :n = 1 [op **1**]	if :n = 1 [op **fput 1 :p**]
op **lput mod :n 2** decbin div :n 2	op decbinp (div :n 2) (**fput mod :n 2 :p**)
end	end

It can be done in a slightly different way (Table 16).

Table 16

to decbin1 :n	to decbin1p :n **:p**
if :n = 0 [op **[ ]**]	if :n = 0 [op **:p**]
op **lput mod :n 2** decbin1 div :n 2	op decbin1p div :n 2 **fput mod :n 2 :p**
end	end

6. Iterative version and its execution (Table 17).
It is more convenient to use the iterative versions which responds to the procedures decbin1 and decbin1p.

```
to decbinit :n
 let "L [ ]
 while [:n <> 0] [let "L fput (mod :n 2) :L let "n div :n 2]
 op :L
end
```

Table 17

variables		loop condition:
:L	:n	:n <> 0
[ ]	12	1
[0]	6	1
[00]	3	1
[100]	1	1
[1100]	0	0

How to check whether the algorithms have been correctly implemented? The next algorithm provides a possible answer.

5. Horner Algorithm

1. Specification of the algorithm:
   Inputs: integer number n, list $[a_0 a_1 \ldots a_n]$, x (simple and structuralised variables); $a_0 a_1 \ldots a_n$ and x are real numbers or integer numbers as a special case. $a_0 a_1 \ldots a_n$ are the coefficients of the polynomial of degree n

$$\begin{aligned} w_n(x) &= a_0 x^n + a_1 x^{n-1} + \ldots + a_n x^0 \\ &= x(a_0 x^{n-1} + a_1 x^{n-2} + \ldots + a_{n-1}) + a_n. \end{aligned}$$

   Output: real number $w_n(x)$, evaluation of the polynomial at x.
2. Mathematical definition of recursive solution:

$$w_n(x) = \begin{cases} a_0 & n = 0, \\ x \cdot w_{n-1}(x) + a_n & n \geqslant 1; \end{cases}$$

   or with all data as parameters; in that case a is a set (list) of coefficients defining the polynomial and we should answer the question if it changes:

$$w(n, x, a) = \begin{cases} a_0 & n = 0 \\ x \cdot w(n-1, x, a \text{ without } a_n) + a_n & n \geqslant 1. \end{cases}$$

3. Logo recursive procedure and its execution (Table 18).

```
to Horner :n :x :a
 if :n = 0 [op first :a]
 op :x * (Horner :n – 1 :x bl :a) + last :a
end
```

Let ours example be:

$$w_4(x) = 3x^4 + 0x^3 - 4x^2 + 4x - 5.$$

```
? show Horner 4 –2 [3 0 –4 4 –5]
19
```

Table 18

Stop condition: :n = 0	Recursive call phase	Collecting output phase
	? Horner 4 –2 [3 0 –4 4 –5]	
0	–2 * Horner 3 –2 [3 0 –4 4] + (–5)	–2 * (–12) – 5 = 19
0	–2 * Horner 2 –2 [3 0 –4] + 4	–2 * 8 + 4 = –12
0	–2 * Horner 1 –2 [3 0] + (–4)	–2 * (–6) – 4 = 8
0	–2 * Horner 0 –2 [3] + 0	–2 * 3 + 0 = –6
1	3	3

4. The tail-end recursion. Let N be the degree of the polynomial (N is a constant) and n the parameter whose value changes from N to 0 (n is a variable):

$$w(n, x, a, p) = \begin{cases} p & n = 0, \\ w(n-1, x, a \text{ without } a_{N-n}, p \cdot x + a_{N-n}) & n \geqslant 1. \end{cases}$$

The initial value of parameter p can be 0.

```
to Hornerp :n :x :a :p
 if :n = 0 [op :p]
 op Hornerp :n – 1 :x bf :a :x * :p + first :a
end
```

```
? show Hornerp 5 –2 [3 0 –4 4 –5] 0
19
```

Table 19

Stop condition: :n =0	Recursive call phase	Collecting output phase
	? Hornerp 5 –2 [3 0 –4 4 –5]	
0	Hornerp 4 –2 [0 –4 4 –5] 3	
0	Hornerp 3 –2 [–4 4 –5] –6	
0	Hornerp 2 –2 [4 –5] 8	
0	Hornerp 1 –2 [–5] –12	
0	Hornerp 0 –2 [ ] 19	
1	19	

It is also possible to take the initial value of a as a list of coefficients without the first one and p as the a_0 (Table 20):

```
? show Hornerp 4 –2 [0 –4 4 –5] 3
19
```

Table 20

Stop condition: :n =0	Recursive call phase	Collecting output phase
	? Hornerp 4 –2 [0 –4 4 –5] 3	↓
0	Hornerp 3 –2 [–4 4 –5] –6	
0	Hornerp 2 –2 [4 –5] 8	
0	Hornerp 1 –2 [–5] –12	
0	Hornerp 0 –2 [ ] 19	
1	19	

We can also conclude that for iterative solution it would be more convenient to use the opposite convention of writing coefficients to that used for recursive solution, namely:

$$w_n(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_0=\big(\ldots(a_nx+a_{n-1})x\ldots+a_1\big)x+a_0.$$

Using coefficients in array we do not remove one coefficient in every step, as is convenient to do using list.

5. Recursion versus tail recursion
   It is worth watching carefully what happens (Table 21):

Table 21

to Horner :n :x :a if :n = 0 [op **first :a**] op :**x** * (Horner :n – 1 :x **bl :a**) **+ last :a** end	to Hornerp :n :x :a **:p** if :n = 0 [op **:p**] op Hornerp :n – 1 :x **bf :a** :**x** * **:p + first :a** end
? show Horner 4 –2 [3 0 –4 4 –5] 19	? show Hornerp 4 –2 [0 –4 4 –5] 3 19 or ? show Hornerp 5 –2 [3 0 –4 4 –5] 0 19

We can remove one parameter in such a way (Table 22):

Table 22

to Horner1 :x :a if count :a = 1 [op first :a] op :x * (Horner1 :x bl :a) + last :a end	to Horner1p :x :a :p if empty? :a [op :p] op Horner1p :x bf :a :p * :x + first :a end
? show Horner1 –2 [3 0 –4 4 –5] 19	? show Horner1p –2 [0 –4 4 –5] 3 19 or ? show Horner1p –2 [3 0 –4 4 –5] 0 19

The base case could be the same in Horner1 and Horner1p, of course. I leave it up to the reader to check how they work.

6. Iterative version and testing its execution (Table 23).
The first proposition:

```
to Hornerit :n :x :a :w
  repeat :n [let ”w :w * :x + first :a let ”a bf :a]
  op :w
end
? show Hornerit 4 -2 [0 -4 4 -5] 3
19
```

Table 23

variables		loop condition:
:w	:a	repc <= :n
3	[0 –4 4 –5]	1
3*(–2)+0=–6	[–4 4 –5]	1
–6*(–2)–4=8	[4 –5]	1
8*(–2)+4=–12	[–5]	1
–12*(–2)–5=19	[ ]	0

We can conclude that this procedure is executed exactly like the tail recursive one.
The second proposition (Table 24):

```
to Hornerit :x :a
  let ”w first :a
  let ”a bf :a
  repeat count :a [let ”w :w * :x + item repc :a]
  op :w
end
? show Hornerit –2 [3 0 –4 4 –5]
19
```

Table 24

variables		loop condition:
:w	:a	:a <= 4
3	[0 –4 4 –5]	1
3*(–2)+0=–6	[0 –4 4 –5]	1
–6*(–2)–4=8	[0 –4 4 –5]	1
8*(–2)+4=–12	[0 –4 4 –5]	1
–12*(–2)–5=19	[0 –4 4 –5]	0

The third proposition (Table 25):

```
to Hornerit :x :a
 let "w first :a
 let "a bf :a
 while [count :a <> 0] [let "w :w * :x + first :a let "a bf :a]
 op :w
end
```

Table 25

variables		loop condition:
:w	:a	count :a <> 0
3	[0 –4 4 –5]	1
3*(–2)+0=–6	[–4 4 –5]	1
–6*(–2)–4=8	[4 –5]	1
8*(–2)+4=–12	[–5]	1
–12*(–2)–5=19	[ ]	0

Now I come back to the question asked at the end of the previous section. Verification whether the algorithm works correctly could require conversion of the binomial representation of a given number into the decimal one. This is equivalent to the evaluation the polynomial defined by the coefficients being the binary representation at x equal to 2. For our example we have:

```
? show decbinp 12 [ ]
[1 1 0 0]
```

Now we can check:

```
? show Horner1p 2 [1 1 0 0] 0
12
```

or more compact, with the procedural parameter decbinp 12 [ ] (a parameter of a procedure that is itself a procedure):

```
?show Horner1p 2 decbinp 12 [ ] 0
12
```

It is not a proper mathematical proof of the algorithm correctness, but it is sufficient for the beginners.

In Appendix 2 two other examples with the list as a data structure and the more interesting student's results (Tables 28, 29) are presented.

6. Conclusions

A method has been proposed to compare and explain the relations between recursive and iterative versions of the same algorithm and illustrated by a few basic examples. Such a training convinces that we really understand what does it mean that every algorithm has

its recursive and iterative version and we can do the transformation from one version to the other. In this paper I decided to present the sequence of steps in the direction from the recursion to the iteration, although sometimes when preparing it I started with the iteration version, which is another argument for the correct performance of the method. The method shown in the paper has his own restriction: it is working well for the functions with the one recursion call in the body of the procedure. All examples discussed in the paper are the functions of discrete mathematics or can be used for integers. These examples I have used to compose different questions to test the knowledge and understanding of the students. The reader can use my examples to construct a lot of *fixed-code* and *skeleton-code* questions (McCartney *et al.*, 2005) for students. I was working with my students with this method only for one year. In one year time I will probably have more interesting results on its performance and may inform you about it. I will be grateful for any comments.

Appendix 1

In Table 26 you will find the recurrence mathematical definitions and (or) recurrent procedures whose common feature is that they are not the tail-end recurrent procedures (excepting the Euclidean algorithm which is defined by tail-end recursion).

- Paste them into the program Imagine and check how they work. Illustrate their work in the table on exemplary data (low whole numbers).
- Give a specification of the algorithm realised by a given procedure (input data, results).
- Modify the procedures to make the tail recurrence (recurrence with an additional parameter). Paste the procedure tested and illustrate in the table its work for the same data.
- Replace the recurrence procedure with tail-end recursion by the iterative procedure. Paste the procedure tested and illustrate in the table how it works for the same data.

Table 26

$H1(a) = \begin{cases} 1 & a = 0, \\ 2 * H1(a-1) & a > 0; \end{cases}$ to H1 :a if :a = 0 [op 1] op 2 * H1 :a - 1 end	$H2(a) = \begin{cases} 1 & a = 1, \\ a + H2(a-1) & a > 1; \end{cases}$ to H2 :a if :a = 0 [op 0] op :a + H2 :a - 1 end
$GCD(a,b) = \begin{cases} b & a = 0, \\ GCD(b \bmod a, a) & a > 0; \end{cases}$ to GCD :a :b if :a = 0 [op :b] op GCD mod :b :a :a end	$GCD1(a,b) = \begin{cases} a & b = 0, \\ GCD1(b, a \bmod b) & b > 0; \end{cases}$ to GCD1 :a :b if :b = 0 [op :a] op GCD1 :b mod :a :b end

To be comtinued

Continuation of Table 26

$$GCDs(a,b) = \begin{cases} a & a = b, \\ GCDs(a-b,b) & a > b, \\ GCDs(a,b-a) & a < b; \end{cases}$$

```
to GCDs :a :b
 if :a = :b [op :a]
 if :a > :b [op GCDs :a – :b :b]
 op GCDs :a :b – :a
end
```

```
to GCDs :a :b
 if :a = :b [op :a]
 ifelse :a > :b [op GCDs :a – :b :b]
 [op GCDs :a :b – :a]
end
```

$$A2(a) = \begin{cases} 1 & a = 1, \\ 2 * a - 1 + A2(a-1) & a > 1; \end{cases}$$

```
to A2 :a
 if :a = 1 [op 1]
 op 2 * :a - 1 + A2 :a - 1
end
```

```
to listrand :n :v
 ; e.g. listrand 10 100
 if :n = 1 [op lput random :v []]
 op fput random :v listrand :n - 1 :v
end
```

The example A2(a), finding 2 power of a for a given natural a by the method used in Charles Babbage Difference Engine, is very easy but worth mention because it attracts our attention to the fact that usually there are exist two good working versions of the iterative algorithm: with the step -1 (typical for recursion) and $+1$, and shows what comes from this fact for the definition of the algorithm (Table 27).

Table 27

to A2w :a :p while [:a > 1] [let "p **:p + 2 * :a - 1** let "a :a - 1] op :p end	to A2w1 :a :p while [:a > 1] [let "a :a - 1 let "p **:p + 2 * :a + 1**] op :p end

At least the last example. The parameters L1 and L2 are lists containing ordered numbers (the order is from the lowest to the highest):

```
to mergel :L1 :L2
 if empty? :L1 [op :L2]
 if empty? :L2 [op :L1]
 ifelse first :L1 < first :L2 [op fput first :L1 mergel bf :L1 :L2]
                              [op fput first :L2 mergel :L1 bf :L2]
end
```

Could you propose the merge-sort algorithm with the list as a data structure? (This question, given after analyzing the usual version with the table as a data structure, turned to be too difficult or not enough interesting for the students.)

Appendix 2

Task 1

Write iterative and recurrent procedures that outputs the list made of the same whole numbers as the original list (parameter of this function) but in the reverse order (the first element is the last, etc.). Could you find an *in situ* algorithm? Analyse the work of your procedures. Paste below the codes of tested programs.

Table 28

to revlist :L if count :L = 1 [op :L] ;if empty? :L [op :L] op fput last :L revlist bl :L end	to revlist1 :L if count :L = 1 [op :L] ;if empty? :L [op :L] op lput first :L revlist1 bf :L end
to revlistp :L :p if empty? :L [op :p] op revlistp bf :L fput first :L :p end	to revlistp1 :L :p if empty? :L [op :p] op revlistp1 bl :L lput last :L :p end
to revlistit :L let "L1 [] while [count :L <> 0][let "L1 fput first :L :L1 let "L bf :L] op :L1 end	to revlistit1 :L let "L1 [] repeat count :L [let "L1 fput first :L :L1 let "L bf :L] op :L1 end
to revlistit2 :L let "L1 [] repeat count :L [let "L1 lput last :L :L1 let "L bl :L] op :L1 end	Only one out of twenty students: ? show reverse [1 2 3] [3 2 1]

Task 2

Propose recurrent and iterative realisations of the algorithm of finding the greatest element of the list. Analyse the work of your procedures. Can it be realised in such a way that the original list of elements will not be changed? Paste below the codes of tested procedures.

Table 29

to maxrec :L :max if empty? :L [op :max] if (first :L) > :max [op maxrec bf :L first :L] op maxrec bf :L :max end	to maxrek :L :max if count :L = 0 [op :max] op maxrek bf :L ifelse first :L > :max [first :L][:max] end
to maxrek1 :L :max if count :L = 0 [op :max] if first :L > :max [let ”max first :L] op maxrek1 bf :L :max end	to maxrek2 :L if count :L = 1 [op first :L] op maxrek2 ifelse first :L < last :L [bf :L][bl :L] end
to maxrek3 :L if count :L = 1 [op first :L] op maxrek3 ifelse item 1 :L < item 2 :L [bf :L][butItem 2 :L] end	to maxrek4 :L op ifelse (count :L) = 1 [first :L] [ifelse (item 1 :L) < (item 2 :L) [maxrek4 bf :L] [maxrek4 butItem 2 :L]] end
to maxit :L :max repeat count :L [if first :L > :max [let ”max first :L] let ”L bf :L] op :max end	to maxit1 :L let ”max first :L repeat count :L [if item repc :L > :max [let ”max item repc :L]] op :max end
to maxit2 :L while [(count :L) <> 1] [ifelse (first :L) < (last :L) [make ”L (bf :L)] [make ”L bl :L]] op first :L end	to maxit21 :L repeat count :L [if (count :L) = 1 [op first :L] ifelse (first :L) < (last :L) [make ”L (bf :L)] [make ”L bl :L]] end
to maxit22 :L repeat (count :L) - 1 [ifelse (first :L) < (last :L) [let ”L (bf :L)] [let ”L bl :L]] op first :L end	to maxit3 :L while [(count :L) <> 1] [ifelse (item 1 :L) < (item 2 :L) [let ”L (bf :L)] [let ”L butItem 2 :L]] op first :L end
to maxit31 :L repeat count :L [if (count :L) = 1 [op first :L] ifelse (item 1 :L) < (item 2 :L) [let ”L (bf :L)] [let ”L butItem 2 :L]] end	to maxit32 :L repeat (count :L) - 1 [ifelse (item 1 :L) < (item 2 :L) [let ”L (bf :L)] [let ”L butItem 2 :L]] op first :L end

References

Foltynowicz, I. (2007). Dynamic programming with the list as a data structure. *Informatics in Education* (in submission).
Ginat, D. (2003). Seeking or skipping regularities? Novice tendencies and the role of invariants. *Informatics in Education*, **2**(2), 211–222.
Graham, R.L., D.E. Knuth and O. Patashnik (1994). *Concrete Mathematics. A Foundation for Computer Science.* Addison-Wesley Publishing Company, Inc.
`http://en.wikipedia.org/wiki/Tail_recursion`
`http://www.logo.com/cat/view/imagine-secondary.html`
`http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/k/Kalas:Ivan.html`
McCartney, R., J.E. Moström, K. Sanders and O. Seppälä (2005). Take note: the effectiveness of novice programmers' annotations on examinations. *Informatics in Education*, **4**(1), 69–86.
Ross, K.A., and C.R.B. Wright (1992). *Discrete Mathematics*. Prentice Hall Inc.

I. Foltynowicz is a senior lecturer at the Theoretical Chemistry Department, Faculty of Chemistry, Adam Mickiewicz University, Poznań, Poland. She has many years of experience in teaching quantum chemistry, numerical methods, statistics and theory of probability as well as algorithms and data structures and basic concepts of programming (not so many years in the latter). She has also several years of experience in teaching informatics at a secondary school (it is a rather closed chapter in her live, but who knows). She received her PhD in theoretical chemistry (more exactly theoretical spectroscopy) from the A. Mickiewicz University in Poznań. She is the author of one academic script, one book and a few scientific papers. She is a fan of Logo (first AC-Logo, then Comenius Logo and now Imagine Logo) as an educational tool mainly for recursion and fractal geometry.

Rekursijos palyginimas su iteracija, kai naudojama sąrašinė duomenų struktūra

Izabella FOLTYNOWICZ

Straipsnyje nagrinėjamas uždavinio sprendimas pradedant jo algoritmo specifikacija bei matematiniu apibrėžimu ir baigiant rekursine procedūra. Taip pat analizuojamas rekursinės funkcijos atlikimas ir algoritmų schemos, kurios padeda suprasti iteracijos ir rekursijos, kurios kreipinys būna pabaigoje, skirtumus. Visos procedūros parašytos Logo kalba, taigi, panaudojamos sąrašinės duomenų struktūros. Rekursinės procedūros keitimas iteracine ir atvirkščiai gali būti parodomas tokiu būdu bet kurioje programavimo kalboje, kurioje tik galima naudoti rekursiją. Visuose pavyzdžiuose tėra tik vienas rekursinis kreipinys ir visi, išskyrus vieną, yra diskrečiosios matematikos funkcijos.