The Role of Variable in Programming: Examples and Methodology

TÖRLEY Gábor, ZSAKÓ László

Abstract. One of the hardest notions to define in programming is the variable and the related command of assignment. In our opinion, it is exactly these difficulties that are responsible for the reluctance towards programming. The reason for this, according to us and others [7], is the multifunctional nature of the variable: it can be used for various purposes. Its concept "in our heads" and in the programming languages is markedly different in this respect.

Keywords: programming, variable, programming methodology.

1. Everyday Algorithms

The fact that during our everyday activities we perform several algorithms, from sequences, branches, and repetitions, to non-deterministic and parallel structures, facilitates our conception of the algorithm [2, 3, 6].

There are programming languages designed for beginners (like ELAN), in which the concept of algorithm is included, but not the concept of data [6].

The data world appears in manifold ways: first, the data are some kind of *objects,* which can be grouped into *classes* (and not as variables in the traditional sense). Such a class is the family, where there is a mother, a father, children, etc. The elements (objects) of the class are the specific family. In everyday algorithms, such objects appear in diverse forms; in default cases, their values are constant, only rarely variable.

Sidenote: Typically, beginners are wary of the concepts of class and object-oriented programming. The underlying reason is that in the classical, von Neumann-style execution of programs, the coder sits in the position of the "processor" and performs the program sequentially, contrary to the above mentioned, everyday-world experience of the concept. This parallelism, implicitly or explicitly present in the object-oriented approach, creates confusion. Languages for beginners include so-called seeded objects (like the turtle in Logo or the cat in Scratch) to avoid this problem.

Interestingly, the structures of the classes are complex straight away; the basic types appear only as parts of the class structure. Contrarily, programming education almost always begins with basic types. The primary composition mode is the direct multiplication (the record concept), which can be complemented by special multitudes: a *set* of clothes, *queue* in the supermarket, etc.

Arrays appear in an odd way too; their indexes can be the floors an elevator passes (negative indices are also possible), or the stations of a railway (where the index can be the very name of the station). In this sense, we can call arrays as indexed structures.

2. The Role of Mathematics

Mathematics introduces number types (integer, real; even if mathematics uses natural and rational numbers as well) and sequences (infinite is possible in mathematics) into the data world. The latter can lead to the concept of the array.

Then there are the matrices as well, but we can bump into two-index structures much earlier, for example, during spreadsheet operations; in fact, they can even precede the sequence concept of mathematics.

The peculiarity of spreadsheets is that they are not "embodiments" of what we could traditionally call variables, rather a functional model of solving a problem [4]. There are constants in certain cells, while in others, functions are applied for specific cells or cell groups. Therefore, functions have parameters by definition. In default cases, however, we calculate all these functions together. If the value of some cells changes, the functions are recounted. Instead of the algorithmic activities, which would apply variables, this model features operations (such as sum, maximum, etc.) defined for ranges (table parts) [9].

Note that everyday algorithms contain similar calculations too (for example, if we buy three cappuccinos, for 2 EUR per piece, then we will pay 6 EUR), but not even then will we store the "sum" in a separate "variable."

3. Why is Programming Difficult?

There are many articles about the difficulties of teaching programming. Hofuku et al. [5] illustrate it by a very simple example-pair:

```
int i;
for (i = 0; i < 10; i++) {
  printf("Hello");
}
```

```
int i = 0;
while (i < 10) {
  printf("Hello");
  i++;
}
```

As they write, *"both programs (…) show that the instruction repeats display a string Hello 10 times. To understand these programs, learners should know variables, data type (int), variables initialization, assignment, compare operators, Boolean values, and increment operator. Many concepts are necessary to understand these simple structures."* The problem identified by the authors relates to the concept of the variable.

Nevertheless, a drawing procedure in Logo, drawing a square with :n side length, causes much less trouble. We draw a square with :n side length by repeating making :n steps forward and turning 90 degrees to the right four times.

```
to square :n
  repeat 4 [forward :n right 90]
end
```

Therefore, here there is no need for the classical concept of the variable.

Usually, it is difficult for students to understand how variables work and the role and concept of them [8].

4. Types of Variables

Sajaniemi et al. [7] summarize the roles and types of variables in the following table (table 1):

Role	Informal definition
Fixed value	A data item that does not get a new proper value after its initialization
Stepper	A data item stepping through a systematic, predictable succession of values.
Walker	A data item traversing in a data structure.
Most-recent holder	A data item holding the latest value encountered in going through a succession of unpredictable values, or simply the latest value obtained as input.
Most-wanted holder	A data item holding the best or otherwise most appropriate value encountered so far.
Gatherer	A data item accumulating the effect of individual values.
Follower	A data item that gets its new value always from the old value of some other data item.
One-way flag	A two-valued data item that cannot get its initial value once the value has been changed.
Temporary	A data item holding some value for a very short time only.
Organizer	A data structure storing elements that can be rearranged.
Container	A data structure storing elements that can be added and removed.
Other	Any other variable.

Table 1: Roles of variables [7]

Our categorization shares some of the elements of the above table. Our methodology, however, fits better with constructivist pedagogy, reflecting the path of how learners get in contact with variables during their studies and how they get to understand their concept. It will also become clear that the two main groups of our categories are linked to whether the notion of assignment has already been introduced or not. Accordingly, the first group contains variable types where the assignment is not yet present [10].

The most classical form of dealing with a "variable" is when it comes to the constants of other subjects (e, pi, g, etc.), which are essentially *numerical constants with names*. They are not variables in the sense that practically they replace a character sequence. (As a consequence, programming languages typically assign the given value to such constants while translating the code; that is, they are not truly treated as variables.) These constants generally do not even appear in the description of tasks (or their names may come up, but it is their value that needs to be used).

Constants in task descriptions play a similar role. For example, from the description "let us read the first 100 numbers", it comes that just like pi, we could name 100 as maxn, for instance. Such constants (with or without names) can appear in task descriptions or input conditions. Nevertheless, they will also be added to the code during translation.

There are constants (so-called *quasi constants*) which appear as constants for people (for example, we know the height of a given "n" person), but in the program, they will turn up as special variables that will get value at some point (as the data are retrieved), and then we will use them only for calculating. In several classical programming tasks, input variables will not change after being retrieved.

```
for i := 1 to n do
  read(x[i])
end for
calculate(n, x)
```

In many cases, it might occur that the retrieved values can be considered constants, but we use a value sequence for the same variable, and we even process it right away. Such constants can be called *quasi constants for multiple purposes.*

```
for i := 1 to n do
  read(y)
  calculate(y)
end for
```

Perhaps the first notion that can be truly interpreted as a variable is the *state component,* such as the position, the direction, etc. of the turtle in the Logo language. We can get the values of the state component, and we can also change them with the help of the dedicated functions or operations. Nevertheless, assignments in the classical sense should be avoided in the case of state components. Most often, state components have their name, so there is no need for a new definition.

The notion of *parameter* is brought in by the procedure (function). In the functional approach (and, for example, in the turtle graphics of Logo, too) parameter is essentially the name of the value, which will be copied in the place of the parameter during function call (with lazy evaluation a bit later). In this case, every parameter is strictly an input parameter only; thus, value is assigned only once, which does not change. The result of the function is a value (perhaps complex), which might be part of an arbitrary expression.

The first data appearing really as variables come up when we need to calculate formulas. These are called *variables calculated from others* (x := f(y)). Their role lies in their name, so essentially these variables are the names of function values stored in variables in classical programming languages.

```
Read(y)
x := f(y)
Write(x)
```

We can build the above algorithm easily on a functional approach: let us calculate something with the input data and write out its result:

```
Write(f(Read))
```

This functional thinking can facilitate and prepare the introduction of variables (which is why it is useful to apply programming languages containing functional programming elements, like Logo or spreadsheets built on functional bases).

Variables, calculated from others, *containing only value* (y := f(x); z := y * (y + 1)) are used solely for the purpose of efficiency, shortening, etc. In the following algorithm excerpt, aimed to calculate the second-based time difference between two dates, TA and TB are such variables.

```
TA := A.hour * 3600 + A.min * 60 + A.sec
TB := B.hour * 3600 + B.min * 60 + B.sec
difference := TB - TA
```

When processing data multitudes (like sequences, sets, etc.), we might need to handle each of their elements, for which *variables applied for a given range* can be used. Such are direct or indirect loop variables.

```
for i := 1 to n do …

for x in H do …
```

```
x.first; while not x.end do … x.next
```

Certain loop types may allow for the partial traversal of a structure as well.

The above data (may they be constants or variables) do not logically require the related but more complicated notion of *assignment*. The classical notion of assignment is the generalization of the memory cell of von Neumann-type computers. We can load data into this named storage; we can get and change them, among others. Consequently, such a notion of the variable does not need to be present in all computing models.

State variables can have a special role. In simple cases, they are logical values (like processed, unprocessed) or in classical graph traversal, they are the colors denoting the state of the points (white, grey, or black). State variables are similar to the state components of the turtle or robots, but here assignment commands make sense, traditional variables are involved.

We often use state variables for ending loops.

```
needed := true
while needed do
  …
  if … then needed := false
  …
end while
```

Programming based on the theory of finite automata frequently applies such state variables.

There are several operations with which we can add or distract elements in data multitudes. These are the *variables changing the number of elements* (such as stack, queue, etc.). They lack classical assignment, but they have *push, pop,* etc. operations. The elements we add or distract, however, are classical variables, which can undergo any kind of operation.

The assignment is probably the hardest to understand with *variables cumulating a specific value*. The main difficulty with them is that in many languages, the description of assignment highly resembles the description of equality check in mathematics (for example, C++: x = f(x, y)), which is mathematical nonsense. Such variables collect the values of a data structure with some cumulative operation.

```
x := …
for y in H do x := f(x, y)
```

A classical version is summation:

```
x := 0
for y in H do x := x + y
```

or maximum selection:

```
x := minval
for y in H do x := max(x, y)
```

The variable of classical counting loops can be such cumulating-type variables:

```
i := 1
while i ≤ N do
  …
```

```
  i := i + 1
end while
```

where the changes of the i variable are often marked with special commands (for example, i++, inc(i)).

Variables calculated recursively from their own previous value are similar to cumulative variables. They frequently appear in mathematical tasks. While the factorial can be described with a simple cumulative variable:

$$n! = \begin{cases} 1 & if \quad n = 0 \\ n * (n-1)! & if \quad n = 1 \end{cases}$$

```
fact := 1
for i :=1 to n do fact := fact * i
```

with Fibonacci numbers, it is easier to calculate the next element of the sequence from the previous values (which not surprisingly is simpler to understand than the cumulative variable of the factorial).

$$Fib(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ Fib(n-1) + Fib(n-2) & if \quad n > 1 \end{cases}$$

```
fib[0] := 0
fib[1] := 1
for i := 2 to n do fib[i] := fib[i - 1] + fib[i - 2]
```

Easier example is the arithmetic sequence for this role:

```
seq[1] := 2
d := 3
for i := 2 to n do seq[i] := seq[i - 1] + d
```

Frequently, we need to perform operations on data or data multitudes that are trivial to do parallelly, but for sequential execution, we need *temporary storage space,* for example, swapping two variables:

Parallelly	**Sequentially**

`swap(A, B):`
`  A, B := B, A`
`end swap`
	`swap(A, B):`
	`  s := A; A := B; B := s`
	`end swap`

or the cyclic rotating of a sequence:

```
rotate(A):
  c := A[1]
  for i := 2 to n do A[i - 1] := A[i]
  A[n] := c
end rotate
```

Similar to state variables are *variables storing some other variable's value.* Typically, we need them when we want to process a multitude by completely traversing it, but as a result, we expect just one element. For example, such a variable is the one containing the index of the maximum value element in maximum selection:

```
maximum(A):
  max := A[1]
  for i := 2 to n do
    if A[i] > max then max := A[i]; index := i
  end for
  maximum := index
end maximum
```

If the formation time and the processing time of the data are different (in time or order), we need variables to temporarily store the data in a given order (stack, queue, dequeue, priority queue, etc.). These are the *variables defining processing order*.

Another group of the variables (and with this one, we are getting far from the first types) constitutes *variables describing relations*. They are typically used in hierarchical and network data structures (trees, graphs, etc.), but structures linked from a certain aspect are similar too.

The above categorization is summarized below:

Group name	Informal definition
Numerical constants with names	Names to substitute character sequences, behind which there are clearly defined values, like e, pi, g, etc.
Constants in task descriptions	Constants which can appear for example in input conditions.
Quasi constants	They appear as constants for people, but in the program, they will turn up as special variables that will get value only when retrieved.
State components	We can get or change their values, but assignments in the classical sense should be avoided.
Parameters	Name of the value which will be copied in the place of the parameter upon function call (with lazy evaluation a bit later).
Variables calculated from others	Names of function values which are stored in variables in classical programming languages.
Variables applied for a given range	Such are direct or indirect loop variables.
State variables	Compared to state components, the notion of assignment commands is included.
Variables changing the number of elements	The operations for adding or distracting elements in data multitudes, they do not involve classical assignment.
Variables cumulating a specific value	They collect the values of a data structure with some cumulative operation.
Variables calculated recursively, from their own previous value	Variables which are calculated from their own previous value
Temporary storage space	Variables that store data for only a very short time.
Variables storing some other variable's value	When we want to process a multitude by completely traversing it, but as a result, we expect just one element.
Variables defining processing order	Variables which temporarily store the data in a given order.
Variables describing relations	Variables that describe the relations of hierarchical and network data structures.
All other variables	Every variable that does not belong in any of the above is categorized into this group.

Table 2: Categorization of variables

5. Comparative Analysis

Comparing Sajaniemi's [7] categories with ours, thus we can conclude:

- Both categorizations follow constructivist principles.
- Their definition of "fixed value" matches our categories "numerical constants with names" and "constants in task descriptions."
- Their "stepper" group fits our "variables applied for a given range" category.
- The variables in their "most-recent holder," "most-wanted holder," and "gatherer" groups correspond to our "variables cumulating a specific value." In our view, it is always

a cumulative function that gives value to a variable, and it depends on the programming theorem behind the function whether the variable belongs to the "most-recent holder," the "most-wanted holder," or the "gatherer" group.

- A part of the variables grouped as "gatherer" falls into our "variables calculated recursively, from their own previous value" category. In our opinion, it is important to make a distinction here because wherever the concept of recursion comes up, it is a different level of cognition compared to cumulative variables.
- Our "quasi constants used for multiple purposes" is the part of their "most-recent holder" category.
- Their "one-way flag" category matches our "state variables."
- Their "temporary" group corresponds to our "temporary storage space."
- Their "container" group is similar to our "variables describing relations" and "Variables defining processing order". We have separated these two groups because data structures in our former group are linear, and the latter group contains non-linear data structures usually.
  - The rest of the groups cannot be matched directly. It is worth noting that Sajaniemi et al. do not consider whether the assignment has already been introduced into the learning process or not, whereas in our categorization, it was a key factor. On the other hand, our grouping was based on the everyday concept of algorithm.
  - The different categories and concepts of the variables are related to each other in the following way (figure 1):

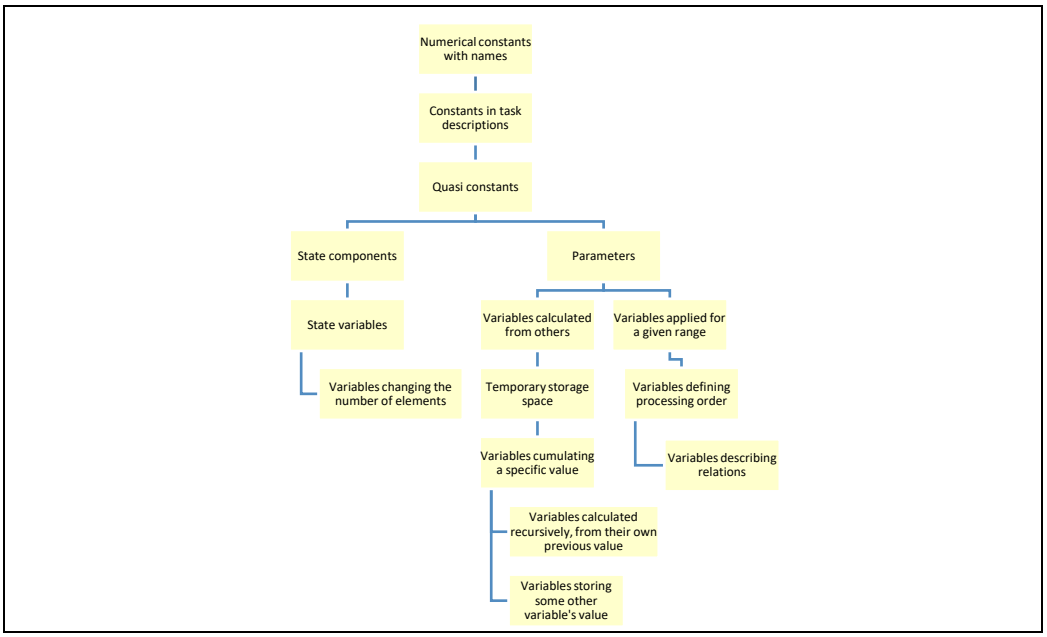

Figure 1: Structure of our categorization

6. Which Type of Variable can be Understood Easily and which with Difficulty?

Usually, the role of scalar variables can be understood easily than the role of more complex data structures. Good examples of that are *quasi constants* and *variables cumulating a specific value*. Variables of these categories can be scalars and arrays as well. For a beginner, it cannot be easy to understand that an array is a variable, which has one identifier, but it contains more elements.

The next difficult role is *parameter* because it can be challenging to understand the difference between formal and actual parameters because they have different identifiers. At the introduction of this category, it can be a good solution to use the same identifiers for formal and actual parameters. Programming languages use different implementations of parameter handling, so learners need to distinguish parameter handling methods. We did not mention the parameter handling method earlier because it depends on the programming language, and we focused on the algorithm, not on coding.

Those variables' role, which are *calculated from their own previous value,* is not difficult to understand but what is difficult is the recursion and the way of thinking behind that. This kind of variables will be used, for example, when we want to solve a recursive problem without recursion e. g. we want to memorize all the elements of a recursive sequence instead of calculating the last element recursively.

7. Object-Oriented Programming

The beginning of learning programming is easy if the learner can imagine herself as the executor of the program. Classical von Neumann programming builds upon this principle with sequential executing. Variables will be needed in order to store, preserve, calculate information. In this paradigm, we split the task in top-down direction to simple procedures, and we pass the necessary data to the procedures.

OOP programs want to represent the phenomena of the real world, which consists of independent and co-ordinate objects. So, when we are programming, the most natural way of thinking is to choose those objects from the real world that are important to us and give them functions.

An object can be described by two parts (like the objects in the real world):

- Attributes: inner and outer attributes, this is a data field.
- Activities: These are the operations that can be executed by or on the object. These are the methods of the object.

Usually, we need more similar objects in a program, so it seems to be expedient to summarize the common attributes. This way, we get a schema of these similar objects, which is called *class,* and each object will be a concrete entity of the class.

Understanding the concept of object and class is difficult, like understanding the concept of variable and type. Beginner programmers can avoid type definitions until they use variables with a given structure (like int, real, etc.). The concept of *datatype* is abstract; it is not easy to understand. If the programmer should define a datatype (or data structure), then she should create all operators, procedures, functions, and data to that. If we skip the concept of inheritance, then the concept of class is very similar to the concept of datatype. But the concept of class is needed for understanding the concept of inheritance.

It is getting more problematic when a solution of a task needs more parallel working objects in time. It is understandable when we want to examine the function of objects individually. But it is challenging to understand the parallel working of the whole system.

In general, most introductory programming textbooks and courses have a syntax-driven organization of the material. The programming language has the "main role", and it is described in a bottom-up way, starting with the simpler constructs of the language and then progressing to more advanced constructs [1].

According to Sorva et al. [11], the defined roles by Sajaniemi et al. [7] can be related to data structures:

- An example of an Organizer is a variable that contains an array of numbers during sorting,
- a variable that references a stack could be a Container,
- a variable that contains a reference to a node in a tree traversal algorithm and a variable that keeps track of the search index in a binary search algorithm can be considered to be Walkers.

As we mentioned in the introduction, beginners are wary of the concepts of class and object-oriented programming.

According to our grouping, *"variables defining processing order"* and *"variables describing relations"* can be understood easier if we refer the former group as classes and refer the latter group as one of the data of a class.

An interesting task can be when we want to represent rational numbers. This can be necessary if we want to calculate very big (i.e., 100 digits) numbers. In this case, every number should be represented in an array. Should we use independent methods, operators, or class methods here?

```
a := add(b, c) (independent method)

a := b + c     (operator)

a := b.add(c)  (class method)
```

In all cases, variable `a` will have the role *variables cumulating a specific value*. We want to answer the question above with another example. We can represent a stack data structure with independent methods or with a class.

Representation with independent methods:

```
stack_elements: array
stack_top: integer
…
procedure push(stack_elements, stack_top, element)
  stack_top := stack_top + 1
  stack_elements[stack_top] := element
end procedure
…
procedure pop(stack_elements, stack_top, element)
  element := stack_elements[stack_top]
  stack_top := stack_top - 1
end procedure
…
```

Representation with a class:

```
class Stack
  Set of values
    elements: array
    top: integer
  …
  Set of operations
    procedure push(element)
```

```
        top := top + 1
        elements[top] := element
      end procedure
      …
      procedure pop(element)
        element := elements[top]
        top := top - 1
      end procedure
      …
End class
```

According to our experiences, if we start teaching programming with the procedural paradigm, then the procedural representation could be a good transition towards object-oriented programming. With this transition, students can understand that the roles of variables do not depend on the programming paradigm, and they can understand the concept of encapsulation if they compare the procedural and the object-oriented representation.

8. Conclusion

When introducing the concept of the variable, it is best to apply approaches based on the data concept of everyday thinking.

This is close to the core concept of one-object languages (Logo, Scratch), except that even in these languages, we should avoid using variables (Logo is much more suitable for this because apart from the state components of the turtle, everything else can be a parameter). Functionalization (such as spreadsheets or Logo's functional elements) is also important when introducing the concept of the variable.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Bibliography

1. J. Bennedsen, M. E. Caspersen: *Teaching Object-Oriented Programming — Towards Teaching a Systematic Programming Process,* Proceedings of the Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts, 18th European Conference on Object-Oriented Programming (ECOOP 2004), 14–18 June, (2004), Oslo, Norway.
2. T. Bell, I. H. Witten, M. Fellows: *Computer Science Unplugged,* An Enrichment and Extension Programme for Primary-Aged Children; http://csunplugged.org (Retrieved: 26.01.2021.)
3. P. Bernát, L. Zsakó: *Methods of Teaching Programming – Strategy.* XXX. DIDMATTECH 2017, Trnava University in Trnava, pp. 40–51, (2017)
4. J. Cunha, J. P. Fernandes, J. Mendes, J. Saraiva: *Spreadsheet Engineering.* Lecture Notes in Computer Science, Vol. 8606, pp. 246–299, (2015)

5. Y. Hofuku, S. Cho, T. Nishida, S. Kanemune: *Why is programming difficult? Proposal for learning programming in "small steps" and a prototype tool for detecting "gaps. in Informatics in Schools.* Sustainable Informatics Education for Pupils of all Ages, Potsdam, Springer, (2013)

6. C. H. A. Koster: *Systematisch leren programmeren,* Educaboek, 1984.

7. J. Sajaniemi, M. Ben-Ari, P. Byckling, P. Gerdt, Y. Kulikova: *Roles of Variables in Three Programming Paradigms,* Computer Science Education, Vol. 16, No. 4, 261–279 Dec (2006)

8. T. S. Sklirou, A. Andreopoulou, A. Georgaki, and N. D. Tselikas: *Introducing Secondary Education Students to Programming through Sound Alerts,* EJERS, Vol. 5, No. 12 (2020): 130–139

9. Zs. Szalayné Tahy: *How to teach programming indirectly – using spreadsheet application.* Acta Didactica Napocensia; Cluj-Napoca Vol. 9, No. 1, (2016): 15–22.

10. P. Szlávi, G. Törley, L. Zsakó: *The most difficult notion of programming: The variable,* In: E. Salata, A. Buda: Education – Technology – Computer Science in Building better future Radom, Poland: Wydawnictwo Uniwersytetu Technologiczno-Humanistycznego w Radomiu, (2018) pp. 108–118., 11 p.

11. J. Sorva, V. Karavirta, A. Korhonen, A.: *Roles of Variables in Teaching.* Journal of Information Technology Education: Research, 6(1), 407–423. Informing Science Institute (2007)

Authors

TÖRLEY Gábor

ELTE Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary,
ORCID: 0000-0002-0496-936
e-mail: gabor.torley@inf.elte.hu

ZSAKÓ László

ELTE Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary,
ORCID: 0000-0002-4614-1509
e-mail: zsako@caesar.elte.hu

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 3, Number 2. 2021

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.3.2.1436

License