Athens Journal of Technology and Engineering - Volume 9, Issue 2, June 2022 – Pages 77-94

Didactic Connection between Spreadsheet and Teaching Programming

By Gábor Törley[*], László Zsakó[±] & Péter Bernát[°]

When we talk about problem-solving skills, then, generally, programming comes to our minds as an activity that can develop algorithmic thinking and abstraction. Regarding the spreadsheet, the software application area could be our first, and mathematics could be our second thought. When spreadsheets and programming are mentioned together, programming of macros is in focus, which is in fact programming. In this paper, we want to focus on how these two areas impact each other, and we want to emphasize that the spreadsheet is an efficient tool to develop algorithmic thinking. Moreover, there is more "crosstalk" between these two tools. This paper will show through examples that there is a two-way connection between spreadsheet and programming; that is why it can be useful to build the concepts of these two topics mutually on each other.

Keywords: *spreadsheet, programming, problem solving, algorithmic thinking, teaching methodologies*

Introduction

Usually, spreadsheet teaching is not classified as a problem-solving tool. For example, according to the Curriculum Framework of the National Core Curriculum (NAT) 2012, the topic "Problem-solving with Information Technology (IT) tools" deals with only programming and algorithms (NAT 2012). Spreadsheets are part of the topic "Using application systems", and the goal of spreadsheets is retrieving information.

The new Curriculum Framework 2020 (NAT 2020) adopts a different and more suitable approach according to which spreadsheets are a new topic in the field of "Developing problem-solving skills". With this change of approach, teaching methods developed earlier by the Faculty of Informatics of Eötvös Loránd University, are followed (Zsakó 2015a, 2015b). In our work, we will show that the new Curriculum Framework has the right approach.

In our paper we will go through the key concepts of programming and we will demonstrate how these concepts can be taught by spreadsheet. We will focus, in particular, on how to present and teach programming theorems in spreadsheets. For this reason, we will define the concept of programming theorems after the literature review.

[*]Senior Lecturer, Faculty of Informatics, ELTE Eötvös Loránd University, Hungary.
[±]Associate Professor, Faculty of Informatics, ELTE Eötvös Loránd University, Hungary.
[°]Senior Lecturer, Faculty of Informatics, ELTE Eötvös Loránd University, Hungary.

Literature Review

The spreadsheet teaching field includes basic programming concepts (Tort 2010), like data types, operations, variables, and functions. Tort also suggests adding procedures, scopes of variables, data tables, sorting, etc. because a spreadsheet can be considered as a program and building a spreadsheet is partly programming. If we use a model of a spreadsheet explicitly, then we can help learners in the process of abstraction, which is a very important part of programming.

According to Szalayné Tahy (2016), a table spreadsheet can be considered as a program with data and pre-defined algorithms. Although students can see a table/spreadsheet on their screens, they need to understand the "program", which consists of their solutions implemented by functions.

Csernoch and Bíró (2015) claim that spreadsheet software can be used as a problem-solving tool. Their method, called Sprego, "is a deep approach metacognitive problem-solving environment, which has borrowed and combined proven methods from high level programming languages. The three milestones of Sprego are

- using as few and as simple general-purpose functions as possible,
- building multilevel formulas,
- building array formulas." (Csernoch and Bíró 2015 p. 27)

This method can develop students' computational thinking and algorithmic skills. Teaching spreadsheet has an important role in Information and Communications Technology (ICT) education because students learn several aspects of computer science and develop skills connected to this field, for example, handling data structures, database management, programming principles, logical and computational thinking, and algorithmic skills. Sprego also promotes schema construction through authentic problem-solving and algorithm construction (Csapó et al. 2020).

Many fundamental programming concepts have their equivalents in spreadsheet. Kankuzi et al. (2017) propose that before an introductory programming course, students should learn spreadsheet programming, where the fundaments of programming are indirectly introduced to them through problem solving by using spreadsheet.

According to Warren (2004), if we use spreadsheet before teaching a programming language, then it takes less time to get to more complicated algorithms.

Programming Theorems/Patterns of Algorithms

Programming tasks can be categorized into groups according to their types, which is useful because for each group we can create an algorithm rule/schema that solves all the tasks in that specific group. These task types are called *programming theorems* because their solutions are justifiably the correct solutions.

Essentially, programming theorems/patterns of algorithms are abstract specifications and algorithms that we want to use as schemas in order to solve a programming task. The aim of *specification* is to give the task in a formalized way (it can be an "interface" between the programmer and the customer). Specification has four components: input, output, precondition and postcondition. *Input* is the input data of the task; *precondition* gives information on the input (i.e., which statements should be fulfilled by the input data); *output* is the result of the task; and *postcondition* is statements used to get the result (how we reach the result-state from the first input-state) (Harangozó et al. 1998).

We can recognize the suitable programming theorem from the task description. Once we have done this, we can use the specific data of the general task type, and in the general algorithm substitute them with the task-specific data. Applying this method will lead us to the correct solution.

In these tasks we usually have to assign a certain result to one (or more) data collection(s), which, for simplicity's sake, we will handle as some sort of sequences. In simple cases sequences can be illustrated as arrays (Szlávi et al. 2019).

Programming theorems/patterns of algorithms are proven templates as a basis on which we can build our solutions later. (This way development will be quicker and safer.) We note here that our term "Patterns of algorithms" differs from the usual definition (LMU n.d.). According to our wording, pattern refers to a task-schema and not to a problem-solving strategy.

We can categorize programming theorems in three groups. We would like to summarize the essence of these algorithms (Szlávi et al. 2019).

Basic Programming Theorems

- **Sequential computing (sequence calculations):** We have an input sequence, and we have to calculate a single value from that. We will use the same operation on every element of the sequence.
- **Counting:** We have an input sequence, and we have to count how many of them have a given attribute.
- **Decision:** Let us determine if there is an item with a given attribute among the elements of an input sequence.
- **Selection (linear selection):** We have an input sequence, and we have to select an element which has a given attribute, assuming that at least one such element exists in the input sequence.
- **Search (linear search):** We have an input sequence, and we have to search for an element that has a given attribute, and we do not know whether such an element exists in the sequence. (Search is the construction of *decision* and *selection*.) Dijkstra calls this algorithm "Linear search theorem" (Dijkstra 1976, p. 105).
- **Maximum selection:** We have to pick/find the greatest (or smallest) value from the input sequence.

Complex Programming Theorems

- **Copy (calculation with a function):** We have an input sequence with N elements, and we have to assign N other elements to these. The type of assigned values can differ from the type of original values, but the count (N) remains the same, as well as the order. In other words, we will use the same operation on each of the elements of the sequence, but the output will be a sequence.
- **Multiple item selection:** We have to list all elements from the input sequence which have a common attribute A.
- **Partitioning:** We have to list all elements from an input sequence which have a common attribute A, and then also list those ones not having attribute A. So, we "assign" all the elements of the input to one of the output sequences. (Of course, there can be more than two attributes.)
- **Intersection:** We have two sets as input (with elements of the same type), and we have to list all elements that are part of both sets. (This is the construction of *multiple item selection* and *decision*.)
- **Union:** We have two sets as input (with elements of the same type), and we have to list all elements that are included at least in one of the sets. (This is the construction of *copy, multiple item selection* and *decision*.)

Constructed Programming Theorems

- **Conditional copy:** We will calculate the same operation on each element of the sequence which have the given attribute and another operation on each element which does not have the given attribute. (This is the construction of *multiple item selection* and *copy*.)
- **Conditional summation:** Sum of elements with a certain attribute. (This is the construction of *multiple item selection* and *summation*.)
- **Conditional maximum search:** find the maximum of the elements that have a certain attribute. (This is the construction of *decision* and *maximum selection*.)
- There are at least K elements with the given attribute (This is the construction of *search* and *counting*.)

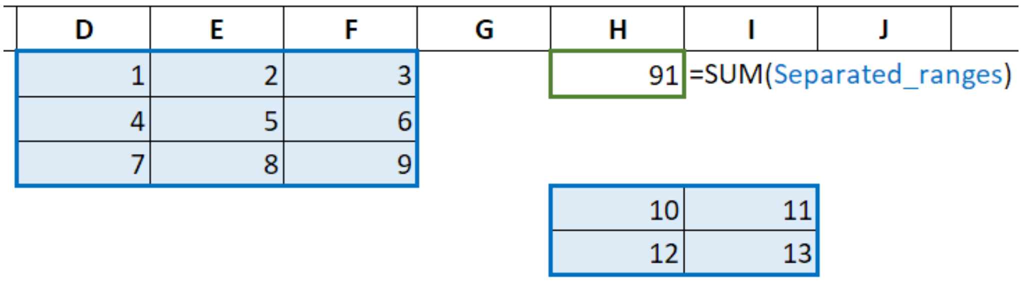
Basic Programming Concepts in Spreadsheet

Variable and Data Type

Cells, one of the most important basic concepts of spreadsheets, are comparable with variables, one of the most important basic concepts of programming. Like variables, cells are named containers (they have a default name, but can also be renamed) in which data can be written and from which the same data can be retrieved. If the user enters the data, it corresponds to reading a value from the user into a variable. If a formula enters the data, it corresponds to storing the result of a calculation in a variable. However, unlike variables, cell content is constantly visible, so no instruction for displaying output is required. It is important to note that in spreadsheet it is not the cell that has a type, but the value stored in it, as even values from different types can be written in the same cell. However, data validation can be set to a cell to limit the type of data that can be entered in it, which is like declaring the type of a variable.

An example of the specialty of spreadsheet's variable concept is that we can assign a name to separated ranges as well and we can use it as a parameter (see Figure 1); we cannot do this in programming.

Figure 1. *Separated Ranges as a Single Variable and as a Parameter in Spreadsheet*



D	E	F	G	H	I	J
1	2	3		91	=SUM(Separated_ranges)	
4	5	6				
7	8	9				
				10	11	
				12	13	

The first column of Table 1 contains the "data types" of spreadsheet, while the second one shows the construction of these types in programming from primitive data types. "Data types" are enclosed in quotation marks in the first case because the spreadsheet does not implement them as true data types. We add that professional programming languages often include the appropriate composite data types so that the programmer does not have to construct them.

Table 1. *"Data Types" in Spreadsheet and Their Construction in Programming*

"Data Type" in Spreadsheet	**Construction in Programming**
Number	integer, real
Currency	integer, real + output formatting
Accounting	integer, real + output formatting
Date	integer, real + output formatting or record type (struct)
Time	integer, real + output formatting or record type (struct)
Percentage	integer, real + output formatting
Fraction	record type (struct) + output formatting
Scientific	integer, real + output formatting
Text	string
Logical	Boolean

Behind the scenes, in fact, spreadsheet deals with 4 data types: logical, number, text and error (error data type is not the subject of our paper). All the "data types" of spreadsheet are real numbers except text and logical. This means that all numeric "data types" of spreadsheet are representations; more precisely, they are output formatting (see Table 2).

Table 2. *Representations of Spreadsheet's Number Type*

"Data Type"	**Displayed in Cell**	**Stored Number**
Number	123456.00	123456
Currency	$123 456.00	123456
Accounting	$123 456.00	123456
Date	01.03.2238	123456
Time	12:00:00	123456.5
Date and Time	01.03.2238 12:00 PM	123456.5
Percentage	75.00%	0.75
Fraction	2/3	0.66666667
Scientific	1.23E+05	123456

There is a great similarity between the spreadsheet's cell format and programming languages' formatted output (i.e., decimal places, format numbers in thousands, etc.).

Function and Data Type

Understanding spreadsheets requires a function-like way of thinking (introduction to functional programming). Using parametrizing functions and nested functions in spreadsheet can support the understanding of parametrizing and parameter passing in conventional programming languages.

In order to form the correct type-concept, spreadsheet has an important role because there are specific functions that can be interpreted only on specific types. For example, SUM and AVERAGE functions can be interpreted only on numeric data, and each arguments of the logical functions, such as AND or OR, must be logical values. Students can understand that the type is not only a set but the applicable operations as well. There is a great difference between digits as string and numbers (difference between "23" and 23). Constant data show this difference as well.

Array and Matrix

Although spreadsheet has a special variable concept (Szlávi et al. 2018), a deeper understanding of functions can support students' understanding of the difference between scalar and sequence and what it means to travers a sequence. In spreadsheet, a sequence can be stored in an array or in a matrix. The best tool for comprehending the concept of indexing can be the INDEX function that executes the indexing operation on a selected range.

A single cell is not suitable for storing complex types, such as a record, but using several adjacent cells can be a solution.

Record

If we view a table in spreadsheet as a table in a database, then its rows can be considered records, the fields of which are defined by the columns. That is, the table can be considered an array of records, or even an array of objects, which can lead to the concept of object-oriented programming.

Conditional and Loop

The IF function can help to understand the conditional control structure as well as the logical (Boolean) type and operations (AND, OR functions).

Loop, as a language element, is not part of spreadsheets, but its concept can be discovered on different levels. For example, if we perform the same operation on all elements of a column in an adjacent column using a copied formula (for example, calculating prices increased by some percentage), we are processing the elements of the column just as a loop traverses an array. In addition, elements of columns (or ranges) can be traversed using array formulas.

Deeper comprehension of lookup functions can lead to the concept of conditional loops because if we look for something, then we can pose a question whether we need to examine all of the elements of the sequence in order to give a definite answer.

Programming Theorems in Spreadsheet

Programming theorems can be demonstrated in three different levels in spreadsheet:

1. with the appropriate built-in functions, students can become familiar with the concept of programming theorems;
2. using spreadsheet as an algorithm visualization tool, students can understand how programming theorems work;
3. most programming theorems can be implemented using array formulas based on the postconditions of their specifications.

In the following, we would like to present these three levels.

Understanding Programming Theorems Using Built-in Functions

Problems to solve with spreadsheet and with programming are often similar, so it is no surprise that the spreadsheet has the functions that implement most of the programming theorems. Table 3 summarizes the connections between spreadsheets and patterns of algorithms.

Table 3. *Connection between Spreadsheet and Patterns of Algorithms*

Patterns of Algorithms	**Built-in Functions in Spreadsheet**
sequential computing (conditional as well)	SUM, SUMIF, SUMIFS, AVERAGE, AVERAGEIF, AVERAGEIFS, DSUM, DAVERAGE, CONCAT
counting	COUNTIF, COUNTIFS, DCOUNT, DCOUNTA
decision	IF(COUNTIF), IF(COUNTIFS)
selection	VLOOKUP, HLOOKUP, XLOOKUP, INDEX(MATCH), DGET
search	*decision + selection*
maximum selection	MAX, MIN
copy (map)	there is not any special function, it can be implemented by copying the reference/formula (Figure 6.) or by creating an array formula
multiple item selection	filter and advanced filter
conditional maximum	MAXIFS, MINIFS, DMAX
Kth maximum	LARGE, SMALL
Sort	SORT (sorting criteria exists but we do not know anything about the method)

It should be noted that the *selection* programming theorem can only be implemented in spreadsheet with crucial limitations. In the case of this programming theorem, the attribute to be examined can be any logical condition. On the contrary, lookup functions (except DGET) can only find an item equal to a specified value in an arbitrary (unordered) range. Although the DGET function can search using any logical condition, it only provides a solution if exactly one element meets the condition.

We would like to highlight *decision, selection* and *search* algorithms, showing how they can be implemented in spreadsheet. It can be presented that VLOOKUP and MATCH functions implement only the *selection* algorithm because they do not give any meaningful answer if the element which we looked for does not exist. *Decision* algorithm should be rephrased: Does the specific element or the element with the specific attribute exist? This way of thinking is connected to the postcondition of *decision* algorithm. As we deduct *linear search* algorithm from the construction of *decision* and *selection* algorithms, we will use the same construction in spreadsheet. For example:

IF(COUNTIF()>0;VLOOKUP();“None”)

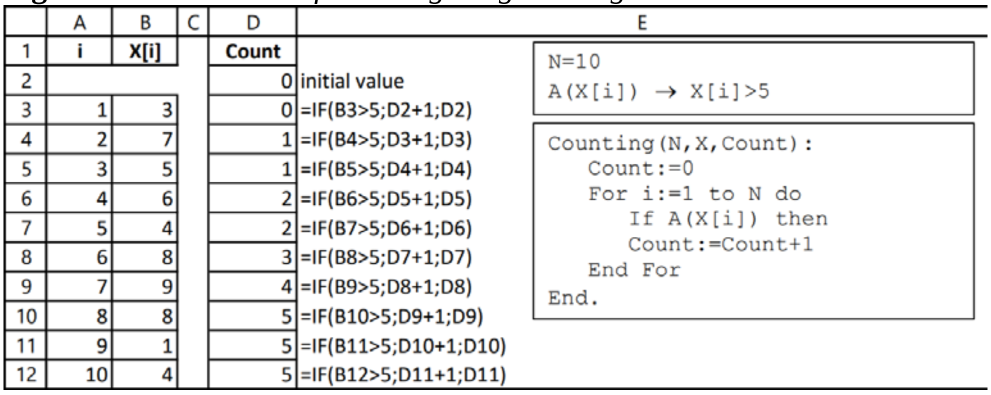
Algorithm Visualization of Programming Theorems

As mentioned above, spreadsheet has the functions with which most of the programming theorems can be implemented. However, these functions hide the actual calculations from the user. Spreadsheet can also support understanding an algorithm step by step and in this way it can support understanding how an algorithm, such as a programming theorem, works. In other words, spreadsheet can visualize the input, the output and the state of the output variable at each step of the algorithm. This means that spreadsheet can show us the whole state space (i.e., input, output, local variables). As examples, we would like to present a possible visualization of the following programming theorems: *counting*, *maximum selection*, *decision*, *conditional maximum search*, and *copy*.

The *counting* programming theorem stores the current number of elements having a given attribute A in an auxiliary variable. It first sets the auxiliary variable to 0, then uses a For loop to traverse the sequence, and if the current element has attribute A, it increments the value of the auxiliary variable by 1.

In our example (see Figure 2), the sequence has 10 elements in an array, and the attribute A is whether the element is greater than 5. Our visualization shows the current value of the auxiliary variable in column Count using the formula shown in Figure 2.

Figure 2. *Visualization of Counting Programming Theorem*



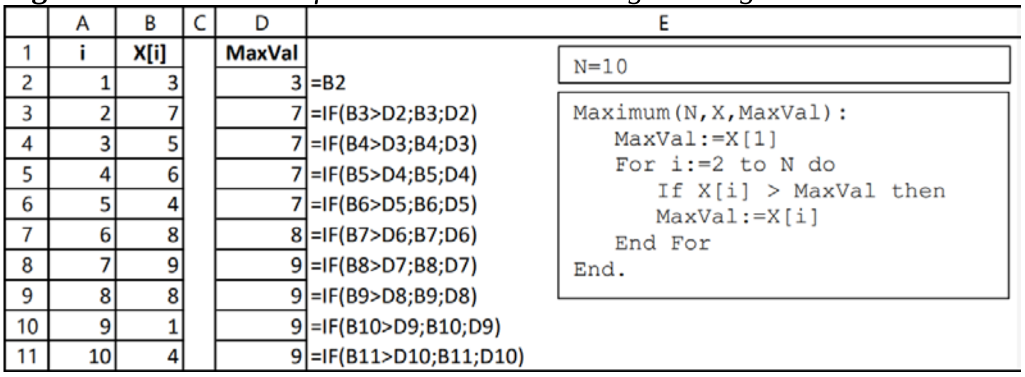
	A	B	C	D	E
1	i	X[i]		Count	
2				0	initial value
3	1	3		0	=IF(B3>5;D2+1;D2)
4	2	7		1	=IF(B4>5;D3+1;D3)
5	3	5		1	=IF(B5>5;D4+1;D4)
6	4	6		2	=IF(B6>5;D5+1;D5)
7	5	4		2	=IF(B7>5;D6+1;D6)
8	6	8		3	=IF(B8>5;D7+1;D7)
9	7	9		4	=IF(B9>5;D8+1;D8)
10	8	8		5	=IF(B10>5;D9+1;D9)
11	9	1		5	=IF(B11>5;D10+1;D10)
12	10	4		5	=IF(B12>5;D11+1;D11)

```
N=10
A(X[i]) → X[i]>5
```

```
Counting(N,X,Count):
   Count:=0
   For i:=1 to N do
      If A(X[i]) then
      Count:=Count+1
   End For
End.
```

Figure 3 shows how to visualize the *maximum selection* programming theorem. This algorithm uses a For loop and checks whether the current value of the sequence (in our example: the array) is higher than the local maximum. If yes, then we change the value of variable MaxVal to the current value of the array. In the first step, the local maximum is the first element of the array and that is why we start the loop counter from 2.

Figure 3. *Visualization of Maximum Selection Programming Theorem*



	A	B	C	D	E
1	i	X[i]		MaxVal	
2	1	3		3	=B2
3	2	7		7	=IF(B3>D2;B3;D2)
4	3	5		7	=IF(B4>D3;B4;D3)
5	4	6		7	=IF(B5>D4;B5;D4)
6	5	4		7	=IF(B6>D5;B6;D5)
7	6	8		8	=IF(B7>D6;B7;D6)
8	7	9		9	=IF(B8>D7;B8;D7)
9	8	8		9	=IF(B9>D8;B9;D8)
10	9	1		9	=IF(B10>D9;B10;D9)
11	10	4		9	=IF(B11>D10;B11;D10)

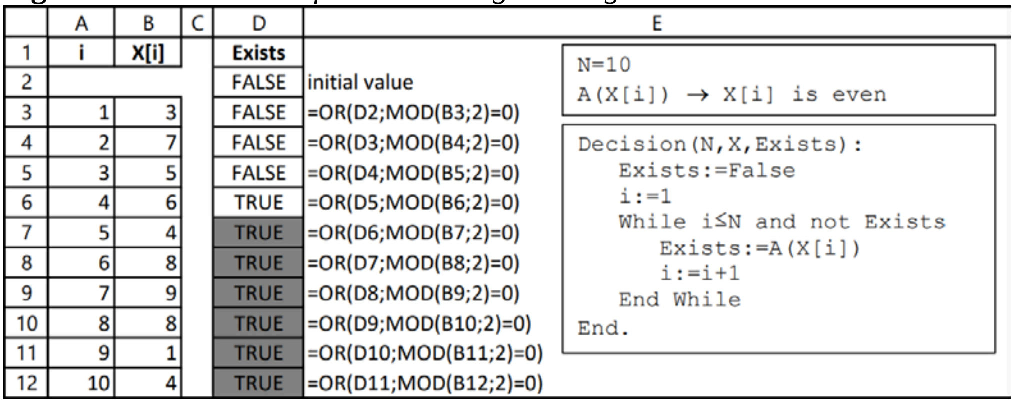
```
N=10
```

```
Maximum(N,X,MaxVal):
   MaxVal:=X[1]
   For i:=2 to N do
      If X[i] > MaxVal then
      MaxVal:=X[i]
   End For
End.
```

The *decision* algorithm checks the elements of the array until attribute A becomes true for the current element. In our example attribute A is that the value is even. Since we do not need to always check all the elements of the array, there is a while loop in the algorithm. In the while loop, we check whether the current element has attribute A and then we increment variable i, which means we go to the next element. If there are elements to be checked and the current element did not have attribute A, we go into the loop, otherwise we exit from the loop.

The visualization in spreadsheet in Figure 4 shows well that as soon as the current element has Attribute A, the variable Exists changes from false to true and after that it will not change back to false (if there is not any element with attribute A then Exists will remain false). According to the algorithm, however, if Exists is true then there is no need to check further. To emphasize this, we can easily create a conditional formatting that darkens (or even hides) the cells belonging to the skipped steps. In our example, conditional formatting was applied to range $D$3:$D$12 with rule "=D2".

Figure 4. *Visualization of Decision Programming Theorem*



	A	B	C	D	E
1	i	X[i]		Exists	
2				FALSE	initial value
3	1	3		FALSE	=OR(D2;MOD(B3;2)=0)
4	2	7		FALSE	=OR(D3;MOD(B4;2)=0)
5	3	5		FALSE	=OR(D4;MOD(B5;2)=0)
6	4	6		TRUE	=OR(D5;MOD(B6;2)=0)
7	5	4		TRUE	=OR(D6;MOD(B7;2)=0)
8	6	8		TRUE	=OR(D7;MOD(B8;2)=0)
9	7	9		TRUE	=OR(D8;MOD(B9;2)=0)
10	8	8		TRUE	=OR(D9;MOD(B10;2)=0)
11	9	1		TRUE	=OR(D10;MOD(B11;2)=0)
12	10	4		TRUE	=OR(D11;MOD(B12;2)=0)

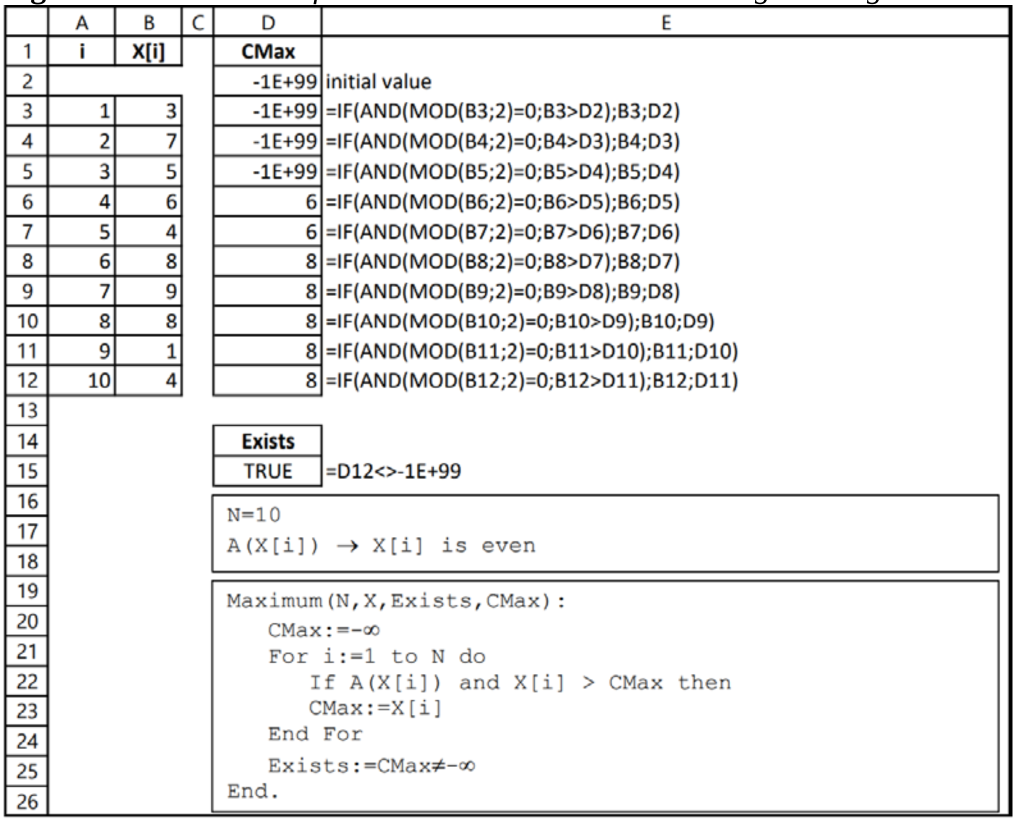
```
N=10
A(X[i]) → X[i] is even
```

```
Decision(N,X,Exists):
   Exists:=False
   i:=1
   While i≤N and not Exists
      Exists:=A(X[i])
      i:=i+1
   End While
End.
```

The *conditional maximum search* programming theorem searches for the largest item in the series that satisfies the specified condition. Of course, it is not certain that there is an element in the series that satisfies this condition, which is why the output will also contain a logical value (variable Exists) that will be true if and only if the condition was true for at least one element.

This programming theorem is based on the *maximum selection* programming theorem described above. Now, however, it is not certain that the first element can be considered the maximum so far; instead, minus infinity will be the initial value of the conditional maximum (variable CMax). Furthermore, the current maximum value is substituted with a larger element only if that larger element satisfies the condition. At the end, the output logical value is set to true if and only if the value of the conditional maximum differs from minus infinity (see Figure 5).

Figure 5. *Visualization of Conditional Maximum Search Programming Theorem*



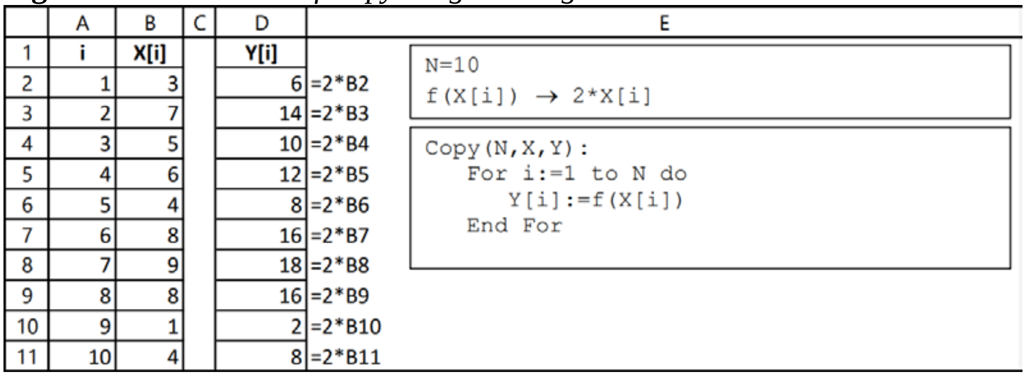
	A	B	C	D	E
1	i	X[i]		CMax	
2				-1E+99	initial value
3	1	3		-1E+99	=IF(AND(MOD(B3;2)=0;B3>D2);B3;D2)
4	2	7		-1E+99	=IF(AND(MOD(B4;2)=0;B4>D3);B4;D3)
5	3	5		-1E+99	=IF(AND(MOD(B5;2)=0;B5>D4);B5;D4)
6	4	6		6	=IF(AND(MOD(B6;2)=0;B6>D5);B6;D5)
7	5	4		6	=IF(AND(MOD(B7;2)=0;B7>D6);B7;D6)
8	6	8		8	=IF(AND(MOD(B8;2)=0;B8>D7);B8;D7)
9	7	9		8	=IF(AND(MOD(B9;2)=0;B9>D8);B9;D8)
10	8	8		8	=IF(AND(MOD(B10;2)=0;B10>D9);B10;D9)
11	9	1		8	=IF(AND(MOD(B11;2)=0;B11>D10);B11;D10)
12	10	4		8	=IF(AND(MOD(B12;2)=0;B12>D11);B12;D11)
13					
14				Exists	
15				TRUE	=D12<>-1E+99

```
N=10
A(X[i]) → X[i] is even
```

```
Maximum(N,X,Exists,CMax):
   CMax:=-∞
   For i:=1 to N do
      If A(X[i]) and X[i] > CMax then
      CMax:=X[i]
   End For
   Exists:=CMax≠-∞
End.
```

There is not any function that can directly implement the *copy (map)* algorithm. If we execute the same operation on the elements of the input sequence, the output will be a sequence. The "copying formula" (actually copying reference) feature of the spreadsheet shows that during the *copy* algorithm we "copy" the formula so we "copy" the operation as well. This way we can visualize the *copy* programming theorem (see Figure 6).

Figure 6. *Visualization of Copy Programming Theorem*



	A	B	C	D	E
1	i	X[i]		Y[i]	
2	1	3		6	=2*B2
3	2	7		14	=2*B3
4	3	5		10	=2*B4
5	4	6		12	=2*B5
6	5	4		8	=2*B6
7	6	8		16	=2*B7
8	7	9		18	=2*B8
9	8	8		16	=2*B9
10	9	1		2	=2*B10
11	10	4		8	=2*B11

```
N=10
f(X[i]) → 2*X[i]
```

```
Copy(N,X,Y):
   For i:=1 to N do
      Y[i]:=f(X[i])
   End For
```

Implementation of Programming Theorems based on their Postconditions

In the case of advanced spreadsheets, array formulas can map all the patterns of algorithms (programming theorems), and there can be a connection among array formulas and postconditions of programming theorems.

To understand the postcondition of some programming theorems, spreadsheet can be a good support. We need to use array formulas. In many cases, the implemented solution by spreadsheet is obvious: for example, *summation, counting, conditional summation, copy,* and *conditional copy.* For instance, the postcondition of *counting* looks like this:

$$Count := \sum_{\substack{i=1 \\ A(Array_i)}}^{N} 1$$

Where A is the attribute function, N is the size of the sequence (in this case: array). This means if the given array-element has attribute A then we add 1 to Count. In spreadsheet, this formula can be implemented literally with the array formula. The Greek letter great sigma means that we add more elements to each other and the condition below that decides at which elements we should add 1 to Count. The operation of great sigma will implement the SUM function, and the operation of the conditional will implement the IF function. That is why the following spreadsheet formula will implement the postcondition of count algorithm correctly:

{=SUM(IF(A(array);1;0}

The SUM function will sum an array with elements 0 and 1 (the output of IF function) and those elements will be 1 that has A attribute (in other words: where the value of A function is true).

In our previous work (Szlávi, Törley & Zsakó, 2019), we have proven that all the programming theorems can be deduced to the *sequential computing* theorem. We have claimed that the *decision* algorithm deduced to *sequential computing* gives the correct solution based upon a Boolean array where the ith element of the array is true if the ith element of the input array has A attribute. *Decision* algorithms have two variants: the first one checks if there is an element in the input array that has attribute A, while the second one checks if every element in the input array has attribute A. It can be proven easily that the following array formulas implement the *decision* algorithm:

- existing element with A attribute: {=OR(A(array_element))}
- every element with A attribute: {=AND(A(array_element))}

We note here that we could implement this theorem with "normal" (i.e. not array) formulas (for example COUNTIF, COUNTIFS functions) but this way of thinking would not lead us to an efficient algorithm and we could not connect it to the postcondition.

Array formulas could help to understand the combination/construction of programming theorems. A good example of this is the *conditional maximum search* algorithm that is the construction of *decision* and *maximum search*. We will combine the postcondition of these algorithms, which means if an element exists that has attribute A in the array then we calculate the maximum of these elements:

{=IF(OR(A(array_element));
MAX(IF(A(array_element);array_element; “”));“NONE”)}

The connection between algorithm patterns’ postconditions and array formulas can be seen in Table 4.

Table 4. *The Connection of Algorithm Patterns’ Postcondition and Array Formulas*

Pattern of Algorithm and Postcondition	Array Formula Implementation
Summation (sequential computing)$\sum_{i=1}^{N} Array_i$	{=SUM(array)}
Counting $\sum_{\substack{i=1 \\ A(Array_i)}}^{N} 1$	{=SUM(IF(A(array);1;0))}
Decision (exists) exist := ∃i∈[1..N]: A($Array_i$)	{=OR(A(array)}
Decision (all) all := ∀i∈[1..N]: A($Array_i$)	{=AND(A(array))}
Conditional sum $\sum_{\substack{i=1 \\ A(Array_i)}}^{N} Array_i$	{=SUM(IF(A(array); array;0))}
Conditional maximum exist := ∃i∈ [1..N]: A($Array_i$) and exist à MaxVal = $MAX^{N}_{\substack{i=1 \\ A(Array_i)}} \; Array_i$	{=IF(OR(A(array)); MAX(IF(A(array); array; “”));“NONE”)}
Copy ∀i∈[1..N]: F($Array_i$)	{=F(array)}
Conditional copy ∀i∈[1..N]: A($Array_i$) is true: F($Array_i$) else $Array_i$	{=IF(A(array);F(array); array)}
Multiple item selection $Count := \sum^{N}_{\substack{i=1 \\ A(Array_i)}} 1$ and $\forall \mathrm{i} \in [1..\mathrm{Count}]: \mathrm{A}(Y_i)$	{=IF(A(array); array; “”}

We can see a connection between the formulas of postconditions and the formulas of spreadsheet. This can be seen on Table 5.

Table 5. *The Connection between Formulas in Postcondition and Formulas in Spreadsheet*

Formula in Postcondition	Formula in Spreadsheet
$\sum_{i=1}^{N} Array_i$	SUM(array)
$MAX_{i=1}^{N} Array_i$	MAX(array)
F(Array_i)	F(array)
A(Array_i)	IF(A(array);array; “”)
$\exists$i$\in$[1..N]: A(Array_i)	OR(A(array)
$\forall$i$\in$[1..N]: A(Array_i)	AND(A(array)

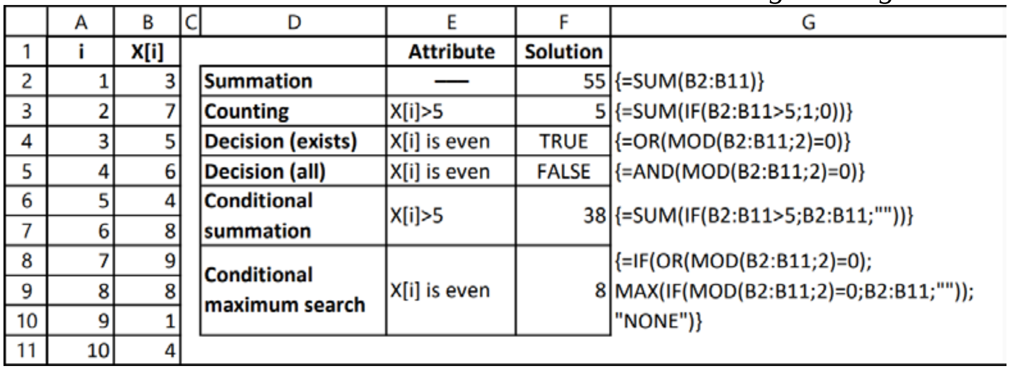
Table 5 shows that we have “building blocks” and by combining these “blocks” a more complex postcondition can be built. This combination shows how programming theorems can be constructed.

We should take a note on *maximum selection* and *multiple item selection* programming theorems. *Maximum selection* cannot be implemented with an array formula because we cannot compare and refer to the elements of the array in the memory (like we showed in Figure 3). CMax can be implemented with an array formula because IF function can select those array elements for MAX function which have A attribute.

The result of *multiple item selection* is an array (which is Y in the postcondition). Count will be the number of those elements which have A attribute (like at count programming theorem) and it will be the number of elements of the output array in Figure 8. In spreadsheet, we do not need to output the number of the output array.

The implementation of some programming theorems with scalar output using array formulas, based on the postconditions of their specifications can be seen in Figure 7.

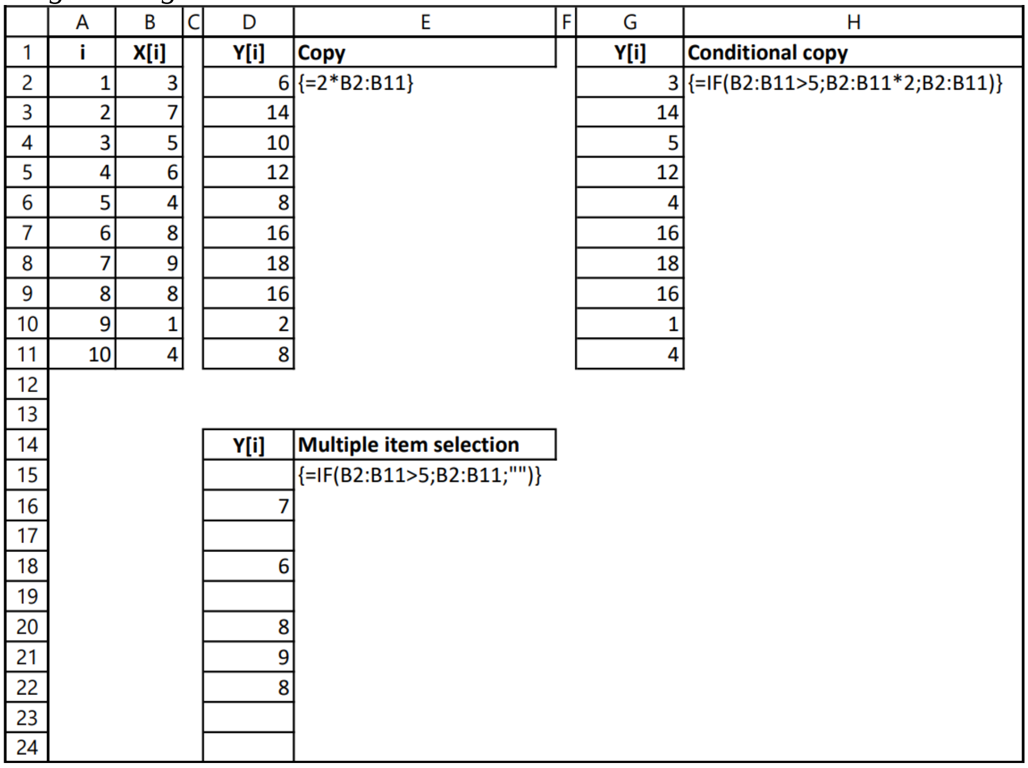
Figure 7. *Implementation of Summation, Counting, Decision (in Two Variants), Conditional Summation and Conditional Maximum Search Programming Theorems*



	A	B	C	D	E	F	G
1	i	X[i]			Attribute	Solution	
2	1	3		Summation	—	55	{=SUM(B2:B11)}
3	2	7		Counting	X[i]>5	5	{=SUM(IF(B2:B11>5;1;0))}
4	3	5		Decision (exists)	X[i] is even	TRUE	{=OR(MOD(B2:B11;2)=0)}
5	4	6		Decision (all)	X[i] is even	FALSE	{=AND(MOD(B2:B11;2)=0)}
6	5	4		Conditional summation	X[i]>5	38	{=SUM(IF(B2:B11>5;B2:B11;""))}
7	6	8					
8	7	9		Conditional maximum search	X[i] is even	8	{=IF(OR(MOD(B2:B11;2)=0); MAX(IF(MOD(B2:B11;2)=0;B2:B11;"")); "NONE")}
9	8	8					
10	9	1					
11	10	4					

Similarly, Figure 8 shows the implementation of some programming theorems with array output. Due to the particularity of spreadsheet, the continuance of array cannot be kept at *multiple item selection* algorithm.

Figure 8. *Implementation of Copy, Conditional Copy, and Multiple Item Selection Programming Theorems*



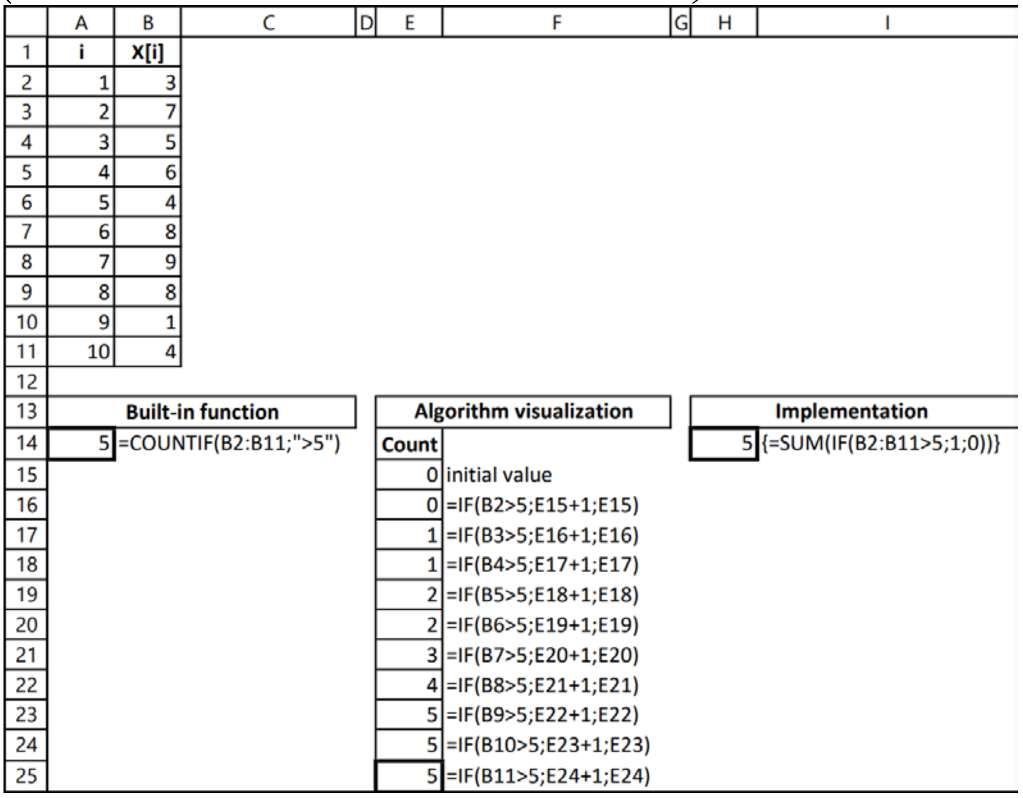
	A	B	C	D	E	F	G	H
1	**i**	**X[i]**		**Y[i]**	**Copy**		**Y[i]**	**Conditional copy**
2	1	3		6	{=2*B2:B11}		3	{=IF(B2:B11>5;B2:B11*2;B2:B11)}
3	2	7		14			14	
4	3	5		10			5	
5	4	6		12			12	
6	5	4		8			4	
7	6	8		16			16	
8	7	9		18			18	
9	8	8		16			16	
10	9	1		2			1	
11	10	4		8			4	
12								
13								
14				**Y[i]**	**Multiple item selection**			
15					{=IF(B2:B11>5;B2:B11;"")}			
16				7				
17								
18				6				
19								
20				8				
21				9				
22				8				
23								
24								

Summary of the Three Levels through an Example

As stated earlier, programming theorems can be demonstrated at three different levels in spreadsheet. The first one is about the comprehension and usage of programming theorems using the proper built-in functions. The second one visualizes the algorithms of the programming theorems using only basic operators and functions. In the third level we can implement most of the programming theorems using array formulas, according to their specifications, or more precisely, postconditions. Consequently, all levels can help learning programming theorems from a different aspect.

For comparison, Figure 9 shows the appearance of the *counting* programming theorem at the mentioned three levels.

Figure 9. *Appearance of the Counting Programming Theorem at the Three Levels (The Solution Has a Thick outside Border in Each Level)*



	A	B	C	D	E	F	G	H	I
1	i	X[i]							
2	1	3							
3	2	7							
4	3	5							
5	4	6							
6	5	4							
7	6	8							
8	7	9							
9	8	8							
10	9	1							
11	10	4							
12									
13	Built-in function				Algorithm visualization			Implementation	
14	5	=COUNTIF(B2:B11;">5")			Count			5	{=SUM(IF(B2:B11>5;1;0))}
15					0	initial value			
16					0	=IF(B2>5;E15+1;E15)			
17					1	=IF(B3>5;E16+1;E16)			
18					1	=IF(B4>5;E17+1;E17)			
19					2	=IF(B5>5;E18+1;E18)			
20					2	=IF(B6>5;E19+1;E19)			
21					3	=IF(B7>5;E20+1;E20)			
22					4	=IF(B8>5;E21+1;E21)			
23					5	=IF(B9>5;E22+1;E22)			
24					5	=IF(B10>5;E23+1;E23)			
25					5	=IF(B11>5;E24+1;E24)			

Conclusions

Our paper showed why the spreadsheet (except table formatting and graphs) is part of computational thinking (together with algorithm and programming) rather than digital literacy. Spreadsheets and algorithms both involve problem-solving (skills).

We can find a great similarity between the topics (data, patterns and algorithms) of the two areas and that is why they can support each other when teaching students to learn and understand key concepts.

In the classical order of IT education, students learn spreadsheet before programming. That is why programming knowledge could be built upon spreadsheet (NAT 2012, NAT 2020, Szalayné Tahy 2016). In Hungary, array formulas are taught only in talent development in secondary schools (Molnár 2014), that is why they will not be the part of the regular teaching order); nevertheless our article intended to show that they could be essential tools in programming education.

References

Csapó G, Csernoch M, Abari K (2020) Sprego: case study on the effectiveness of teaching spreadsheet management with schema construction. *Education and Information Technologies* 25(Nov): 1585–1605.

Csernoch M, Biró P (2015) Sprego programming. *Sprego Programming, Spreadsheets in Education (eJSiE)* 8(1): Article 4.

Dijkstra EW (1976) *A discipline of programming*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Harangozó É, Szlávi P, Zsakó L (1998) Joining programming theorems a practical approach to program building. In *Annales Universitatis Scientiarum Budapestinensis. Sectio Computatorica.* Budapest, Hungary.

Kankuzi B, Isong B, Letlonkane L (2017) Using the spreadsheet paradigm to introduce fundamental concepts of programming to novices. In *Proceedings of SACLA'17.* July 3–5, 2017, Magaliesburg, South Africa.

Loyola Marymount University – LMU (n.d.) *Definition of "algorithmic patterns"*. Available at: https://cs.lmu.edu/~ray/notes/algpatterns/.

Molnár K (2014) *Tehetésggondozás az informatikában – Táblázatkezelés*. (Talent development in informatics – Spreadsheet). ELTE Faculty of Informatics. Available at: http://tehetseg.inf.elte.hu/tananyagok/tablazatkez/index.html.

NAT (2012) *National core curriculum framework for informatics in Hungary 2012*. Available at: https://kerettanterv.oh.gov.hu/05_melleklet_5-12/5.2.21_informat_5-10.doc.

NAT (2020) *National core curriculum framework in Hungary 2020*. Available at: https:// www.okta tas.hu/kozneveles/kerettantervek/2020_nat.

Szalayné Tahy Z (2016) How to teach programming indirectly – Using spreadsheet application. *Acta Didactica Napocensia* 9(1): 15–22.

Szlávi P, Törley G, Zsakó L (2018) The most difficult notion of programming: the variable. In E Sałata, A Buda (eds.), *Education - Technology - Computer Science in Building Better Future*, 108–118. Radom, Poland: Wydawnictwo Uniwersytetu Technologiczno-Humanistycznego w Radomiu.

Szlávi P, Zsakó L, Törley G (2019). Programming theorems have the same origin. *Central-European Journal of New Technologies in Research, Education and Practice* 1(1): 1–12.

Tort F (2010) *Teaching spreadsheets: curriculum design principles*. ArXiv, abs/1009. 2787.

Warren P (2004) Learning to program: spreadsheets, scripting and HCI. In *Proceedings of the Sixth Australasian Conference on Computing Education – volume 30*, 327–333. Darlinghurst, Australia.

Zsakó L (2015a) Informatika Nemzeti Alaptanterv 2020. (National core curriculum in informatics 2020). In P Szlávi, L Zsakó (eds.), *INFODIDACT 2015.* (Zamárdi, Magyarország, 11.26.2015.-11.27.2015.) Budapest: Webdidaktika Alapítvány, Paper 1.

Zsakó L (2015b) Informatikai tantervelmélet? Diszciplínák tanítása – a tanítás diszciplínái 1. Tanulmányok a tudós tanár-képzés műhelyeiből. (Curriculum theory in informatics? Teaching of disciplines – Disciplines of teaching volume 1. Essays from the workshop of scientific teacher training). In *ELTE Eötvös Kiadó*, 92–111. Budapest, Hungary.