EDUKACJA – TECHNIKA – INFORMATYKA W BUDOWANIU LEPSZEJ PRZYSZŁOŚCI

Elżbieta Sałata, Agata Buda

EDUCATION – TECHNOLOGY – COMPUTER SCIENCE IN BUILDING BETTER FUTURE

Redakcja naukowa

Elżbieta Sałata & Agata Buda

Wydawnictwo Uniwersytetu Technologiczno-Humanistycznego w Radomiu

Radom 2018

Monografia 222

Recenzenci:
Doc. Ing. Ladislav Rudolf, Ph. D., University of Ostrava
Dr Krzysztof Symela, ITE-PIB w Radomiu

Redakcja naukowa
Elżbieta Sałata & Agata Buda

Opracowanie wydawnicze
Monika Fetraś

Projekt strony tytułowej
Robert Bondarowicz

26-600 Radom, ul. Malczewskiego 29
www.uniwersytetradom.pl, e-mail:wydawnictwo@uthrad.pl

ISSN 1642-5278
ISBN 978-83-7351-860-5

Wyd. I

Péter Szlávi, Gábor Törley, László Zsakó

THE MOST DIFFICULT NOTION OF PROGRAMMING: THE VARIABLE

Everyday algorithms

The fact that during our everyday activities we perform a number of algorithms, from sequences, branches, and repetitions, to non-deterministic and parallel structures, facilitates our conception of the algorithm. [1, 2, 5]

Interestingly enough, there are programming languages designed for beginners (like ELAN), in which the concept of algorithm is included, but not the concept of data. [5]

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.

Side-note: Typically, beginners are wary of the concepts of class and object-oriented programming. The reason for this is that, contrary to the above mentioned, everyday-world experience of the concept, in the classical, von Neumann-style execution of the program the coder is sitting in the position of the "processor" and is performing the program sequentially. This parallelism, implicitly or explicitly present in the object-oriented approach, creates confusions. To avoid this, so-called seeded objects (like the turtle in Logo or the cat in Scratch) are built into languages for beginners.

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 indices can be the floors an elevator passes (negative indices are possible too), 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.

The role of mathematics

It is mathematics that 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 [3]. In certain cells there are constants, while in others functions applied for specific cells or cell groups. Therefore, functions by definition have parameters. In default cases, however, we calculate all these functions together. If the value of given cells change, 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). [7]

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".

Why is programming difficult?

Many articles have already been written about the difficulties of teaching programming. Hofuku et al. [4] illustrate it by a very simple example-pair:

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

Types of variables

Kuittinen and Sajaniemi [6] summarize the roles and types of variables in the following table (Table 1).

Table 1.

Role	Informal definition
Fixed value	A variable which is initialized without any calculation and whose value does not change thereafter.
Stepper	A variable stepping through a succession of values that can be predicted as soon as the succession starts.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values.
Most-wanted holder	A variable holding the "best" value encountered so far in going through a succession of values. There are no restrictions on how to measure the goodness of a value.
Gatherer	A variable accumulating the effect of individual values in going through a succession of values.
Transformation	A variable that always gets its new value from the same calculation from value(s) of other variable(s).
Follower	A variable that gets its values by following another variable.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Organizer	An array which is only used for rearranging its elements after initialization.
Temporary	A variable holding some value for a very short time only.
Other	Any other variable.

Our categorization shares some of the elements of the above table. Our methodology, however, fits better with constructivist pedagogy, reflecting the path 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 concept of assignment has already been introduced or not. Accordingly, the first group contains variable types where assignment is not yet present.

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 in fact not 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 we could name 100, just like pi, as maxn for instance. Such constants (with or without name) 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, the assignment 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 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 the 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 names, so essentially these variables are the names of function values, which are stored in variables in classical programming languages.

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

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

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

With 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 only 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 with 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.

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 equation 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:=i
while i≤N do
  …
  i:=i+1
end while
```

where the changes of the i variable is 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]
```

Frequently, we need to perform operations on data or data multitudes that are trivial to do parallelly, but for a 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.

Table 2.

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 assignment 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.

cd. **Tabele 2.**

Group name	Informal definition
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 which 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.

Comparative analysis

Comparing Kuittinen and Sajaniemi's [6] categories with ours, this what 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.

- Their “Transformation” category matches our “Quasi constants used for multiple purposes.”
- Their “One-way flag” category matches our “State variables.”
- Their “Temporary” group corresponds to our “Temporary storage space.”
- The rest of the groups cannot be matched directly. It is worth noting that Kuittinen and Sajaniemi do not take into account whether 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 (Fig. 1).\

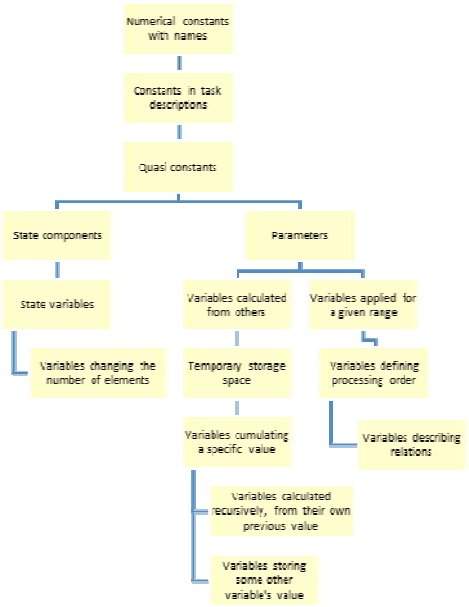

Figure 1

Conclusion

When introducing the concept of the variable, it is best to apply approaches which are 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 the use of 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) are also important when introducing the concept of the variable.

References

1. Bell T., Witten I. H., Fellows, M., Computer Science Unplugged, *An Enrichment And Extension Programme For Primary-Aged Children*, http://csunplugged.org
2. Bernát P., Zsakó L., *Methods of Teaching Programming – Strategy*, XXX. Didmattech 2017, Trnava University in Trnava, 2018, pp. 40-51.
3. Cunha J., Fernandes J.P., Mendes J., Saraiva J., *Spreadsheet Engineering*, "Lecture Notes in Computer Science", 2015, Vol. 8606, pp. 246-299.
4. Hofuku Y., Cho S., Nishida T., Kanemune S., *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, Springer, Potsdam 2013.
5. Koster C.H.A., *Systematisch leren programmeren*, Educaboek, 1984.
6. Kuittinen M., Sajaniemi J., *Teaching Roles of Variables in Elementary Programming Courses*, ITiCSE'04, 28-20.06.2004, Leeds, United Kingdom, https://pdfs.semanticscholar.org/11ce/7795412e240c3af8b88d97087994f9d290bc.pdf
7. Szalayné Tahy Z., *How to teach programming indirectly – using spreadsheet application*, "Acta Didactica Napocensia", 2016, Vol. 9, No. 1, pp. 15-22.

Summary

One of the hardest notions to define in programming is the variable and the related command of assignment. In our opinion, these are exactly these difficulties that are responsible for the reluctance towards programming. The reason for this, according to us and others [6], 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.