DE GRUYTER OPEN

ICTE Journal

2015, **4**(3): 32–39
DOI: 10.1515/ijicte-2015-0012

DIDACTICS OF PROGRAMMING

Ľubomír Salanci
Department of Informatics Education, Faculty of Mathematics, Physics and Informatics
Comenius University, 842 48 Bratislava, Slovak Republic
salanci@fmph.uniba.sk

Abstract

Programming is as an important part of informatics at Slovak schools, and therefore we put focus on didactics of programming. We have observed various issues that are related to teaching and didactics of programming. These issues should be mastered by future teachers of informatics that we prepare at our faculty. In order to prepare future teachers we have designed a course of didactics of programming. For example, we have observed that our students – future teachers do not differentiate between levels of complexity when trying to teach various programming topics, or they skip important steps when explaining solution of a problem. We came to conclusion that it is necessary to design various activities related to teaching of programming and problem solving that allow students to collect their own practical experiences by resolving various didactical problems and to develop their critical thinking about teaching.

Keywords

programming and problem solving, didactics of programming, future teachers of informatics, didactical problems

Introduction

Informatics education has various regional specifics in different countries. Therefore, we briefly explain the role of programming at primary and secondary schools in Slovakia. Then we take a closer look at didactic of programming which we consider as an important part in education of students – future teachers of informatics.

The role of programming in Slovak schools

Programming has been for decades a very important part of school informatics. In the 70s there were only a few high schools that had classes with a focus on programming (e.g. Gymnázium Jura Hronca in Bratislava). Informatics has become a compulsory subject at high schools in the 80s. During this period, the teaching of informatics was reduced to teaching of programming

(programming = the second literacy). By contrast, in the 90s with the arrival of personal computers, internet and applications there was trend to teach controlling computer, select applications or type-writing.

Currently, we pay attention to programming again. It's because programming allows us to teach pupils to solve problems: to explore a given task, to choose a suitable representation of handled information, to invent and to write a solution of the problem, to evaluate and to correct its own or other's solutions. The problem solving is currently considered as one of the most important competence. Then, such understanding of programming gives us a different sight on what is the goal of school programming.

This also means that we do not plan to train professional programmers, developers – perhaps the 99% of pupils will not be programmers. Our goal is not perfectly learn a programming language, libraries, and selected set of algorithms or specialized technologies. Of course, if pupils are asking for more challenging topics, we will be happy if a teacher is competent to teach them. But then, especially in relation to an evaluation of students, the teacher must realize that they are not compulsory components of school's informatics.

Problems solving from the perspective of didactic of programming

Let's imagine for example, that we are at the high school. Pupils have learned how to use variables and cycle. Pupils were training and using them to solve several tasks during preceding lessons.

Now, we want to teach to solve problems by accumulating result. We give pupils the following assignment: *"Cunning trader sells goods so that subtly increases its value. We pay 1 euro on the first purchase. On each latter purchase, he asks 1 euro more. How much would we pay altogether, if we bought goods 10 times?"* How should we proceed in class if pupils are not able to solve the problem by themselves?

It turns out that it does not enough to show pupils the solution, only:

```
s := 0;
for i := 1 to 10 do s := s + i;
```

It is because pupils do not understand to the given solution. For example, they do not understand how the assignment `s : = s + i` works. Even, additional explanations like *"it adds numbers from 1 to 10 to the variable"* will not help.

Moreover, pupils did not experience the process of solution inventing and they did not see how the program arose. Therefore they could not be able to solve similar assignments. If we continue to teach in this way and pupils stay in role of spectators, programming will happen to them a "magic", which only masters the teacher. They get bored from the programming; they get frustrated because they actually do not know programming.

How can we proceed better?

First, we discuss with pupils about the assignment. We verify for example, if they understand it. Therefore we ask pupils: *"How much do we pay for the first purchase?"*, *"How much do we pay for the second one?"*, *"Third?"*, *"And the last?"*

We ask them further: *"How much money will we pay for all purchases together?"* In this case, the assignment has intentionally a trivial solution. So our pupils quickly reply that we actually need to sum the numbers:

```
1 + 2 + 3 + 4 + ... + 10
```

We will guide pupils to the informatics' solution: *"We could count the numbers by ourselves. But we have a computer. So, let computer add up numbers for us."* Our pupils do not know the mathematical formula for the sum of numbers, yet and we do not want to reveal it. We go further: *"We can't write the program as it is on the blackboard, because the computer can't guess what three dots ... means. We must list all the numbers."* Here we could end up, because the problem is solved. But this is this just the beginning.

Thus we continue: *"But let's imagine how the program would look like if we decided to sum up 1000 numbers. If we have to list all 1000 numbers, we would be tired. Can this be programmed somehow smarter?"* Now, we pushed pupils to their limits. Our motivation is based on the principle: we can solve the problem somehow, but the solution is awkward, so it does worth to learn something new.

Next we begin to examine the problem a little by better. The following analogy might help pupils: *"How would we sum the numbers without a computer?"* As teachers, we know that the final program will work in a similar way. Therefore, we want from pupils to discover that the solution is composed of a series of small steps.

We realize some steps together: *"First, we add numbers 1 and 2; then we add number 3; then 4, and so on."* In parallel, we illustrate each step, number and totals on the blackboard Fig.1.

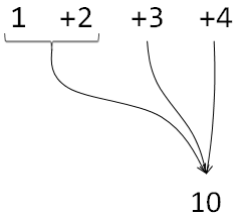

Fig. 1: Illustrations on the blackboard.

It is important for pupils to figure out that the result is produced gradually, not at one time. Moreover, the following scheme is discovering: *"We see that the sum is produced gradually. We add a number to the last result. It creates a new sum."* And further: *"We must remember the last result. Otherwise we would not know to continue counting. We wrote the result on the blackboard. Our program stores it using a variable. Let's call it for example* `s`*."* For pupils, it is important to experience the moment when the need to use a variable arises, and to understand the purpose for what the variables are used in final program.

Our illustrations on blackboard were yet informal. Now we bring them closer to the final program. *"In the beginning we have nothing. Therefore, after the program has started we set the variable* `s` *to zero"*. We are explaining and we are writing on the blackboard:

```
s := 0
```

We use familiar commands to write steps and we are verbally commenting them: *"Gradually, we add the individual numbers:"*

`s := s + 1` ... *"When this command executes the variable* `s` *will have a value 1."*

`s := s + 2` ... *„The variable* `s` *will contain value 3."*

`s := s + 3` ... *„s = 6"*

`s := s + 4` ... *„s = 10"*

...

`s := s + 10`

Such tracing, writings or drawings on the board are very important in order to give pupils the opportunity to discover the following repeating pattern: *"The first command* `s := 0` *is a special. But others look like this:* `s: = s + something`*. And the* `something` *changes from 1 to 10."*

Our pupils already know the programming construction of `for` loop and they have perfectly trained it. We ask: *"How to make a program that changes something from 1 to 10?"*

So we get to the notation:

```
s := 0;
for i := 1 to 10 do s := s + i
```

Next, pupils should train the new principle by solving similar graduated problems. For examples: Change the program to add up `100` numbers; to add up to *n*; Sum squares of numbers (`1 + 4 + 9 + 16 +` ...); Draw rectangles with sides gradually increased by 10; and other.

Note: sometimes it is not necessary to perform such detailed procedure. Some pupils need just a small hint. By contrast, with others we must go through all the steps with all 10 numbers.

Abstract reasoning in programming

We can see that by problem solving, we guide pupils to discover connections and relationships, to generalize solutions and to write their solutions using an abstract language.

It is interesting that by higher the level of abstraction we use notations which are shorter, but more difficult to understand.

For example, the sum of 10 numbers in Python programming language can be written as follows:

```
s = 0
```

```
for i in range(11):
    s = s + i
```

But also in this way:

```
s = sum(range(11))
```

It is possible that the notation `sum (range (11))` is still clear to us. But the principle that is behind it is far from simple.

For example, we can easily modify the solution with `for` loop to sum squares of numbers. Simply, we change the formula in the body of the cycle:

```
s = s + i * i
```

Can we write it using `sum (...)`? What should be in the brackets?

In teaching of programming, we must distinguish between different levels of demanding of concepts, problems, examples and solutions. Only then we can offer to our pupils an affordable way to new knowledge.

The ability of kids to think abstractly is determining factor in teaching of programming. For example, kids up to 12 years are in stage of concrete operations. The stage of abstract thinking starts later (Rybár, 1997).

This means, that under a certain age it does not make sense to expect from pupils a general solution (to use variables, or formulas with unknown values). Therefore, assignments for pupils at primary schools are formulated in such way that they work with a small number of specific elements, with pictures or objects that can be touched.

Cognitive process

We used the example with the sum of numbers in order to realise that a new programming knowledge is formed in certain stages:

Motivation

↓

Collecting own experience (elementary, then a little bit complex)

↓

Clarification of the rules and relationships (cleaning)

↓

Knowledge

↓

Crystallization (training of new knowledge)

These stages come out from constructivism, constructionism (Ackermann, 2010), theory of mathematics education (Hejný, Kuřina, 2009) and our observations.

Moreover, these stages are present when we teaching more challenging programming topics, even when we are teaching non-programming topics (for example, working with text or working with graphics). Also, our experience with professional teachers shows us, that they consider these stages as completely natural.

Pupils have different problems while learning a programming. For example, one research (Paz, 2006) discovered that some pupils have a misconception of variables. Consider the following example:

```
a := 1;
x := 2 + a;
a := 10;
```

Now, we can ask pupils what is the value of variable `x`? The correct answer is 3. But, some pupils may answer that the value of variable `x` is 12. From the perspective of didactics of programming we ask: *Why do they think so?*

Didactics of programming in teacher education

We have experience that general public usually underestimate didactics and teaching. We often hear opinions that *"didactics is useless"*, *"just use common sense"* and similar.

Even students – future teachers have a distorted idea about primary and secondary school, about pupils and teaching itself. This happens probably because these students experienced the teaching from one side only – as pupils.

Students – future teachers often do not understand that a good lesson is driven by certain rules. They underestimate preparation to teaching and to lesson. They significantly overestimate abilities of pupils. Students – future teachers do not believe that the children will not understand them, or children will not to perceive new concepts because of rapid pace.

Teaching of informatics and programming especially, consists a number of traps. A simply said: We know, how we shouldn't teach. But it is difficult to find the right approach - the right way.

We may conclude from the previous chapters that to teach a programming is as demanding as to teach math.

Our faculty has been offering some didactical courses for 25 years. But these courses were focused on general didactics, general pedagogy, advanced programming, or problems solving at level of various competitions. Therefore, it was decided more than 10 years ago that our students need a course of didactics of programming for ordinary pupils, in an ordinary classroom.

No such similar course of didactic we had found, therefore we had to design it from the scratch. We wanted to meet pragmatic expectations of our students: they would like to learn how to teach programming. Gradually, we have created a course which consists of minimum lectures about various theories. Instead of that we have invented a series of activities for our students

that allow them to get experience with teaching of programming and to clarify very basic principles of teaching.

It was shown that activities should be arranged in such order that our students could gradually understand various didactical problems. For example, we observed that nearly all students initially do not understand to didactical problems related to teaching younger pupils. Probably, the age gap is too big and it is difficult to imagine for our university students a thinking of younger pupils. Therefore, we have tried to overcome the gap by familiarizing our students with didactical problems in reverse order: from higher secondary down to primary school.

So, the first activity of our course is focused on school-leaving test (maturation test) in informatics (*Monitor, 2004*). First, each student solves it. We want put our students in the role of their future pupils. Also we want to familiarize students with seriousness of the exam. We have observed that clever students consider that the test as very simple, some others as too difficult. A discussion about assignments, tasks follows then. We also focus on distinguishing and naming informatics' concepts that are tested in different questions, but also on a way that they are tested.

Previous activity gradually passes into debate about the national curriculum and the role and objectives of informatics and programming in education.

Another activity is focused on teaching a problem solving. We start with the cunning trader problem, as we have already described in this paper. At the beginning of this activity, we change roles. Students become teachers and they try to navigate us to the solution. We play a role of pupils who do not understand anything. Therefore we ask students: *"Where did this formula come from? For what is this variable? Why is this cycle there? How did it arise?"* During this "game", students realize that teaching is not a trivial task.

Subsequently, each student chooses some new assignment. He or she solves the assignment. Then he or she thinks about how to explain a solution to pupils. Finally, he or she demonstrates teaching in front of other classmates. Classmates are playing a role of pupils. This activity is funny and very edifying.

Our objective is to teach students to see the steps which lead to solution of a problem. Many students of our faculty are far excellent in programming. Schools programming problems are trivial for them and they solve such problems automatically, by heart. It is similar as to us to answer: *"How much is 100 – 50?"* Probably, we immediately respond that the result is 50. But pupils are taught the subtraction during several lessons of math in school. Therefore, we need these students to eject from such automatic mode.

During other activities: our students analyze and evaluate textbooks or books for professional programmers; we discuss about programming languages, about their advantages and disadvantages from a perspective of teaching; we discuss about different programming topics which are suitable for primary and high schools, students are taught to distinguish the stages of cognitive process (it is not easy for students).

At the end, students get recommendation how to teach individual programming topics; how to avoid didactical problems; or what examples and assignments are appropriate for pupils of

various ages. Finally students present a lesson on a chosen topic. We and other students evaluate performance according to negotiated rules.

Conclusion

The goal of our school programming is to learn algorithmically solve problems. We have observed complex relationships in teaching of programming an age of pupils, their motivation; a choice of programming language and environment; order of topics etc. There are several publications that are focused on didactical aspects of programming (Armoni et al, 2010). We try to answer not only the question *"How to teach?"*, but also *"Why so teach?"*

We have argued in this paper that it is not easy to transfer didactical theories to future teachers. We must carefully choose those parts which our students are able to understand. Therefore we familiarize students with basic principles of teaching programming by performing many didactical activities. So, students by themselves gradually get to know the important moments of the teaching process. Not only positive, but also negative situations are valuable, such as: didactical problems caused by poorly specified assignment; lack of motivation; overcomplicated examples, usage of undefined terms during explanation of new concepts; or excessive ambition of teacher to solve complex tasks too early.

We have created a series of educational materials (Salanci et al., 2010) in which we have summarized or experiences from leading courses of didactics of programming that we developed within several years.

References

RYBÁR, Ján: *Úvod do epistemológie Jeana Piageta. (Introduction into epistemology of Jean Piaget).* Bratislava : IRIS, 1997. ISBN 80-88778-43-3.

ACKERMANN, E.: Constructivism(s): Shared roots, crossed paths, multiple legacies. In: *Proceedings of Constructionism 2010*. Bratislava : Comenius University, in association with The American University of Paris, 2010. p. 13. ISBN 978-80-89186-65-5

HEJNÝ, M. and F. KUŘINA. Dítě, škola a matematika. (*Children, school and mathematics*) Praha : Portál, 2009. ISBN 978-80-7367-397-0

PAZ, T.: What Can We Learn from Educationally Disadvantaged High School Students about Variable? In: *Proceedings of ISSEP 2006*. Vilnius. p. 29-39. ISBN 9955-680-47-4.

Monitor 2004 – pilotné testovanie maturantov – Informatika – Test I-2. Bratislava : Štátny pedagogický ústav, EXAM, 2004.

ARMONI, Michal, and Tamar BENAYA, David GINAT, Ela ZUR. Didactics of Introduction to Computer Science in High School. In: *Lecture Notes in Computer Science Volume 5941*. Berlin: Springer, 2010, p. 36-48. ISBN 978-3-642-11375-8.

SALANCI, Ľubomír and Andrej BLAHO, Monika TOMCSÁNYIOVÁ. *Didaktika programovania 1, 2, 3*. Bratislava: Štátny pedagogický ústav, 2010. ISBN 978-80-8118-065-1, ISBN 978-80-8118-090-3, ISBN 978-80-8118-079-8.