Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 65–74
doi: 10.14794/ICAI.9.2014.2.65

Evaluation of programming languages from educational perspective

Péter Szlávi, Viktória Heizlerné Bakonyi

ELTE IK
szlavip@elte.hu
hbv@inf.elte.hu

Abstract

In this paper we examine the suitable programming languages (PL) in primary and secondary education. The aim of this examination is to sketch a viewpoint system which helps informatics teachers to judge PLs from the aspect of their educational usefulness, the opportunities they offer or rather the difficulties they pose, in order that they can compare them. This evaluation is far from being self-serving; it can lay the base for harmonizing various ideas in informatics teaching, especially the ones related to programming and the tools (PLs) that we can reach our goals with. Naturally, we must not forget about the importance of development environments as tools in teaching, but it will be the topic of another paper.

Keywords: Programming languages, Programming methodology, Elementary and Secondary School

MSC: 68U99

1. Introduction

1.1. Why is this viewpoint interesting?

When shaping their educational program, schools in Hungary have to act upon the learning content in the NAT (NAT is the abbreviation of National Education Plan) [1].

Informatics knowledge changes rapidly, thus – compared to other subjects – the contents and the technology used in informatics have to be reconsidered more often. Informatics teachers have a serious role in adapting curriculum frameworks to a particular school.

It is obvious that it is a must to choose at least one programming language (PL) and one development environment (DE) to teach students the skill of writing

algorithms and other programming related knowledge set in the NAT. In addition, in order to attract students, more and more schools offer deeper knowledge than the NAT requires.

If you change to a new PL and/or a new DE in your school – either because of a change in the NAT and in the curriculum framework, or for any other reason – you have to evaluate which features of this "novelty" i.e. the newly introduced PL+DE need more time and which ones can be handled in fewer lessons.

We hope that our viewpoint system will help readers to evaluate the strengths and weaknesses of the novelty before actually introducing it and to scale the curriculum more punctual.

1.2. The educational aims of programming languages

On the one hand, you should make students become familiar with factual information about the chosen PL+DE until they can routinely perform tasks. Through this, you have to show them the most important features, the syntax and the semantic criteria of the PL, and how to develop with the given PL in a specific DE.

On the other hand, you should point to the general features of the PL-class (e.g. procedural, logical, functional; object-oriented etc.) which the specific PL belongs to. You should make it clear what the "philosophy" of this PL-class is, what the key ideas are; what elements (and not what specific syntax!) constitute the program code in this PL; how the program is executed etc. (In addition, you should sketch a general picture about the DE, as well.)

It is worth noticing that using a PL similar to a macro language can be profitable when teaching application systems, as well (e.g. VBA and MsOffice, or the PL based on C–Pascal–Lisp–BASIC running on Mapple [2]).

PLs and DEs serve to assist both programming and developing algorithmic thinking, but they can also operate as a tool for understanding models. What does it really mean?

The educational benefit of experiments is widely known. If you use a simulation program during lessons, when the computer serves as an experimental tool, you will describe the real world with some idealistic parameters. Although some people do not consider it applicable in the teaching–learning process just because of idealization, we think that this is not a disadvantage, but rather an advantage as compared to real experiments.

Simulation modelling i.e. taking part in designing and implementing the model offers many more opportunities than simply using a ready-made application. The model has to become an active participant of the experiment; it has to become a computerized tool for us. Understanding the logic of the computer-simulation and programming it helps us understand the real world and its behavior.

We must notice that this attitude – algorithmic problem solving – is somehow universal i.e. it is independent of a specific theme or of the level of idealization. (As opposed to classical mathematical methods, which become more and more complicated when reducing idealization. At first a linear equation or a system of linear equations is enough to solve a problem, whereas later you need some higher

degree equations the solution of which requires some other – completely different – methods. In the end, you might even have to use differential equations.) [3] Regarding programming, you must be aware that there are PL+DEs which are created specifically to implement simulation programs [4], but it is also possible to use a general, well-known PL+DE in order to enable students to achieve success. [5] We will write about the required conditions for that (PL+DE) later.

1.3. Minimal expectations towards PLs

Human expectations: a PL should not require deep, specific knowledge from students (depending on the profile of the school). We usually prefer free software giving the opportunity to use them at home easily.

Computer expectations: a PL should not to require expensive hardware (e.g. extra quick processor or big RAM etc. The software is required from to run stably. The compiler had better be used on different operating systems e.g. C++ or FreePascal (on Windows and on LINUX). It had better not require other installations, which would mean extra costs and would needlessly load resources when the computer is used for other purposes.

Further expectations: to give a standard for improvement. A good example could be, for instance, the BASIC language, which can be used for teaching application systems, too (Office). In addition, we can mention a "drawing" PL (e.g. Logo/Imagine or Scratch) created in accordance with the needs of the young children. Using it, one can give a slight sketch of programming structures, improving towards procedural languages. Moreover, using Imagine can lead children from drawing to the world of objects. Our opinion is that in secondary schools it is important to understand the base of object oriented programming. In this age-group it is worthy paying attention to how the knowledge gained can be used at the advanced school-leaving exam [6] or later in vocational training.

2. The evaluation of programming languages

2.1. The simplicity of the language

Easy learning is ensured by the simplicity of a language, which depends on several factors: base-words (instructions); how they fit each other and the source-code; the program structure and consistency.

2.1.1. Base-words, identifiers

The quality of base-words influences the quickness of first steps, as they are the ones starters meet first. The "ancient" BASIC contained only some meaningful keywords: `IF`, `THEN`, `ELSE`, `GOTO`, `FOR`, `NEXT`, `INPUT`, `GOSUB`, `RETURN`, `PRINT`, and `DIM`. It is hard for a PL to be simpler than this. There is no doubt that one can

easily remember the keywords if a base word is short, which makes typing it in simple.

Now let us step over the base-words: in the early versions of BASIC there was a special symbol (called sigil) at the end of variable names, the role of which was to simplify typing, but it can be severely criticised for its lack of expressiveness. The logic of Perl is similar to this, where variable-names get a "pre"-tag, which shows the structure of data ($ – scalar, @ – array, % – hash table). Though these sigil-s are short and meaningful, they do not help beginners to learn programming.

By demanding formatting characters in reading and writing functions to specify the type of the variables, C does not make the first steps easy for inexperienced programmers, either.

The over-emphasis of readability e.g. in COBOL is also thought-provoking since in this language even arithmetical operational signs have a name: `ADD, SUBSTRACT, MULTIPLY, DIVIDE` ...

It can be stated that only symbols which are well-known and used widely in other fields of science (e.g. mathematics) can be accepted as abbreviations – and of course they are to be used with the same meaning. Thus they are short and their meaning is clear for everybody. In most languages the four basic arithmetical signs are just such ones.

The mode of fitting base-words into the program can also be interesting: is it necessary to highlight them – with uppercase (like in ELAN, COMAL or MODULA), with lowercase (as in C-like PLs) or some other way (e.g. with an apostrophe as in ALGOL 60)? Generally, the necessity to distinguish uppercase and lowercase characters is also an important methodical question, which somehow determines the programming style, too. (Unlike Pascal and BASIC, C-like languages are case-sensitive. It is true that it makes typing more complicated, since you should pay attention to these niceties, too, but it might make the structure of the program clearer, enhance its readability and, finally, train users to be punctual.

2.1.2. The program structure and the simplicity of program structures

The transparency of the program is a cardinal question both in writing and reading the code. Regarding transparency, a simple structure is an obvious advantage. (General advice: teachers must do their best to raise the need in students to write adequately structured codes.) In several PLs, including C-like languages, the place of data declarations is not fixed; what is more, they can be missed (e.g. in JavaScript and PHP). Therefore, a programmer has little to remember about them. There is something here, however, that is worth considering. In this case a miss-typing can create a new variable with the miss-typed identifier, which you do not want. To find such errors is not an easy task at all.

Can we state then that the uncontrolled one is really simpler? That is to say if there are fewer rules to be learnt, the programmer's work will become easier? The answer is obvious: fewer rules are only profitable when the code remains transparent and the simplification does not cause mysterious errors. So our example about "easy" declaration is far from being good. Moreover, it suggests negligence, which

is definitely harmful.

If rules can be used on various levels or in various situations and their usage is consistent – i.e. consistence greatly contributes to the memorability of rules.

Let us quote some good examples! Think of the hierarchical program-structure of Pascal: `program`-structure → `procedure / function`-structure.

There are, however, some negative examples in Pascal, too. For instance, the incomprehensible dependencies of place in syntax: the different usage of key-words `var` and `const` in different situations, namely when declaring local variables or using them as formal parameters.

Here we should mention the uncertainty caused by the different condition-representation of while and repeat cycles in the Pascal language. The high variety of cycles (the fullness of instructions) can also be disturbing: `Do {While|Until} Loop` and `Do Loop {While|Until}`. As opposed to it, in C-like languages the condition-controlled loops – both pre-condition and post-condition ones – use a "staying inside condition", so they are easier to remember.

The issue of starting and ending a complex structure: e.g. instruction blocks in the Pascal language are signed by Begin-End key-word pair, whereas in C-like languages (C, C++, C#, Java, JavaScript, PHP etc.) they are replaced by { and },which is short and clear at first glance. In Pascal, however, Begin-End can often be omitted, and even Case-End, Record-End, Repeat-Until can sometimes be omitted, too, whereas in C-like languages you can leave out {} around a single instruction at cycles or at if structures. There is no doubt that the shortening in a single instruction (in any language) is very dangerous and spoils the memorability of rules! Similarly, in some versions of the BASIC language the if instruction has an end tag if there is a line break in the code; but if you write it in one line, there is no end tag. It can be extremely disturbing for beginners! In Python or F# it is not necessary to use key-words to mark the end of structures, because the block structure is obvious due to the indentions. Killing two birds with one stone: it spares some typing and the structure remains recognizable and readable. Nevertheless, do not fail to recognize its seamy side, either! It might damage the need for conscious structure-shaping, which will make further steps more difficult.

We have reserved some good examples for the end! Let us mention some consistent usage in ELAN: `IF-ENDIF, REP-ENDREP`... in Maple: `if-fi, do-od`... or in UNIX shell: `if-fi, for do-done` and `case-esac` structure.

2.2. Typicality

2.2.1. Simple code-writing, easy learning

A good PL (in primary and secondary education) semantically does not differ dramatically from an algorithmic language; it is only a "precise version" of it. It is important that you need not make significant transformations on the program for code-writing. Let us show the difficulties through some negative examples.

First of all, let us take the standard Pascal and the need of changing a function to a procedure due to the restrictions of function return type. (Good news: in

FreePascal the return type of a function can be a record or an array as well!) We can mention another typical problem in procedural PLs which usually demands "transformation": the usage of general multiway branches. As there are only a few possible acceptable types in multiway branches, they often have to be transformed into several, nested if statements. Another problem occurs using a Repeat-Until structure in Pascal, where you should negate the condition.

The operator syntax in C-like languages '=' instead of ':='; '==' instead of '=', which differs from algorithmic languages is a real source of hazard. In BASIC you will meet only "half" of the above problems, namely the signs for equality and basic assignment operators are the same!

2.2.2. It is a perfect model of the language-class

A good PL consistently and sensibly contains the features that are important and relevant to its language-class. For instance, Java and C# are object-oriented languages, and they clearly show all important OOP features. Prolog is an excellent example of the logical language-class. On the other hand, the "eclectic" Logo is not an ideal language, at all, since it does not show clearly either the typicality of an automatic language, or the features of a functional-language.

Typicality harmonizes with one of the expectations mentioned in 1.3: a PL should serve as a basis for stepping forward. For example after using standard Pascal, you have good opportunities to continue programming in OOP or in 4GL. In addition, C-like languages also have several proper "follow-ups".

2.3. Usability

2.3.1. Developing – advanced programming style

PLs used in education highly affect the users' programming style! Therefore, a good choice here is of utmost importance. The most important features to be taken into consideration:

a) The PL must not contradict the top to down program-planning strategy – it should support the use of parameterized procedures or functions. One of the few undisputedly positive examples is ELAN. Writing the code from "down to top" (first declare and then use!) principle is damaged in most procedural languages. In Pascal, C or C++, you can bridge the "problem" of writing a code "down to top" by using the keyword `forward` (in Pascal) and using head-lines (so called prototypes) before main splitting the declaration and definition. (From a methodological aspect, however, you have to ask whether it is a real problem or not. Actually it can be stated that the algorithm is ready by the time you start code-writing.)

   Here let us mention "drawing" Logo which you can use to show the "top to down" and "down to top" planning method, as well. Making the details richer and richer step by step, we use "top to down" method but at the same time

you can also draw by using ready-made elements, which is a clear example for "down to top" planning.

b) It should support algorithmic abstraction – whether it has the traditional (procedural) structure-organizing instructions: sequence, branches, cycles, procedures/functions/operators. From this point of view, in most C-like PL-s (C# is an exception) the semantic meaning of algorithmic multi-way branches differs from the code – we think of `break` in `switch`, which can be left out. An additional question: are there overloadable operators at all? N.B. What we mean here is overloadability by the developer. In Pascal there are no such possibilities, but in C++ and C# there are. These features become important at a later phase of the teaching process, but they can be used well sooner, as well.

c) It should support data abstraction – the adequate variety of elementary types. The existence of enumerated type and type-construction tools are important from the point of view of abstraction. (The non-existence of automatic I/O-operations is an interesting problem!) For example, such a type-construction tool is record in Pascal and `struct` in C, C# and Java. There is quite a great difference between the array conception of Pascal and C. The correct manageability of types – the unity of representation + implementation, propose serious methodological questions. Like in standard Pascal, there is no possibility for overloading operators: the usage of a newly implemented or a built-in type will be quite different in syntax. The same task can be resolved properly in C++ or C#. If you make use of the opportunities of an object-oriented language, you can step to a new level of type-creation.

   In most modern languages (C#, C++, Java, BASIC, Object Pascal) the familiar data structures (like queue, stack, list) are built in. Now the question is whether they show an idealized structure or not – usually they do not. In C++, for instance, `List` type contains a `swap` operation (which turns the order of the elements), thus disturbing the clear notion of list type. Regarding methodology, there is an important question: is it useful for students to create their "own" data structures at least once or had they better use the ready-made ones?

d) It is free of "illegal" and dangerous opportunities. For example in Logo, which is a semi-functional and semi-automata-like PL, there is no room for variables and assignment statement, but the language still offers it with make. In a procedural PL class `Goto, Stop, Halt` and `Exit` are extraneous, which were used for error handling a long time ago. Many of them are still present in Pascal or in C-like languages. [7].

e) The possibility of modularity has a methodological result, too: the teacher can offer some ready-made parts and the students have to concentrate only on the main problem. Most 4GL environments support this module based "frame-program using" educational strategy.

2.3.2. Usage – topic-universality

It should possess a real "topic-universality"! It means that the chosen language should support program writing in any topic occurring at school: students should be able to use it when implementing a simulation-application which requires high calculation speed, large amounts of memory and even good graphical skills for displaying. These features are determined mainly by hardware but slowness can also be caused by the execution strategy of the language (if it is an interpreted one). You should take it into account, too!

2.4. Widespread documentations

The existence of standards and/or documentations is a necessary – but not definitely sufficient – condition of introducing a PL into the curriculum. Without them you cannot even think about it. In the modern "shareware-world" it is a crucial issue whether there are easily available sources of information to learn the language or you are left to your own resources to get some uncertain information from others. In addition, it is not worthy teaching a language which is likely to disappear soon; you had better choose one that has been present stably. Incidence can guarantee this. (Facts that can help incidence: its freeness and the support of a developing community on the web.)

3. Summary

Finally, we would like to collect our thoughts related to the aim of education!

1. In the field of informatics, the aim of teaching programming languages – beyond the basic aim, i.e. the development of algorithmic thinking – is to establish and efficiently assist the teaching of code-writing. Some PLs which are close to macro languages help a deeper understanding of application systems and customizing them by programming. We can state that only a unique informatics subject is capable for these tasks.

2. On the other hand, teaching programming and programming languages – as we mentioned before – can be connected to other subjects, as well, through simulations, for instance. Principally, but not exclusively, simulation models can be implemented in common educational projects where other science subjects are also involved. Such interdisciplinary tasks help students to develop their programming skills and – at the same time – to better understand the other subject by making experiments and creating models.

When choosing a PL, which of course depends on the target audience, the learning and professional viewpoints may differ. Learning the first PL, the main criterion must be learnability, simplicity and its ability to shape proper notions. It is worth considering learnability features in later phases, too, but professional viewpoints

should become more and more important, and learnability as a viewpoint can even be neglected.

By learnability we mean the following:

- Use only some keywords, possibly keywords taken from the learners' native language!
- The syntax should be simple, and given by some easily memorable formulae!
- The result of the program should be spectacular i.e. particularly rich in visualization and/or offer the possibility of animation, which adds extra motivation!
- It should work with ordinary notions!
- It should be stable and well-documented!

We believe that the most important professional viewpoints are as follows:

- The PL should have standard structures (both data and instruction)!
- It is required to support top to down planning; in order to do so, it should possess refinements (procedures, functions, operators)!
- It should have compound instructions due to principles of structured programming!
- There should be a method to define and construct the most important compound data types!
- It should support modular programming, which is important both from the points of view of teachability, project approach teaching and advanced programming!
- It must enable students to go further both into the OOP and into 4GL!
- It should be accessible in several platforms!

References

[1] Nemzeti Alaptanterv (2012), `www.kormany.hu/download/c/c3/90000/MK_12_066_NAT.pdf` (Accessed 15 April 2014)

[2] Monagan, M., Programming in Maple: The Basics, `amath.colorado.edu/computing/mmm/MapleProgr.pdf` (Accessed 15 April 2014)

[3] Szlávi, P., Szimuláció vs. matematikai modellezés (e-manuscript) (2000), `people.inf.elte.hu/szlavi/InfoOkt/SzoftErt/SzimVsMat.pdf` (Accessed 15 April 2014)

[4] Bernát, P., Szimuláció az oktatásban a NetLogo szimulációs környezettel, in InfoDidact'12 Konferencia (2012)

[5] Szlávi, P., „Darázs-modell" (e-manuscript) (2000-2012), `people.inf.elte.hu/szlavi/InfoRendsz/Darazs/Darazs.pdf` (Accessed 15 April 2014)

[6] Az informatikai alapismeretek vizsgatárgy írásbeli és szóbeli érettségi vizsgáihoz – A 2012. év vizsgaidőszakaira érvényes szoftverek listája (2012), `www.oktatas.hu/pub_bin/dload/kozoktatas/erettsegi/info2012osz/infalapism_irasbeli_szoftverlista_2012okt.pdf` (Accessed 15 April 2014)

[7] Szlávi, P., A Programkészítés didaktikája (PhD dissertation) (2004), `www.inf.elte.hu/karunkrol/szolgaltatasok/konyvtar/Lists/Doktori%20disszertcik%20adatbzisa/Attachments/32/Szlavi_Peter_Ertekezes.pdf` (Accessed 15 April 2014)