De "faire" à "faire faire": au coeur de la programmation: quelques réflexions didactiques

Duchâteau, Charles

Publication date:
1992

Document Version
Version revue par les pairs

From "DOING IT ..." to "HAVING IT DONE BY ...": The Heart of Programming. Some Didactical Thoughts.

Charles Duchâteau

CeFIS, Facultés N-D. de la Paix, rue de Bruxelles, 61, B-5000 Namur, Belgium

Abstract : The meaning of the word *problem* is very specific in the context of programming. Indeed, the problem does not lie in the *task* which is only the starting point but in the need of *having it done subsequently by a "performer"*. This contribution proposes a didactical approach and an environment for the learning of this "having it done by ..." which is the heart of programming. It insists on the necessity of developping a meaningful description of the constraints brought by the performer in the world of the tasks to program and allows the learning of "true" programming through a metaphorical description of the computer-performer.

Keywords : computer education and literacy, programming environment, didactics of programming, problem-solving, metaphors, mental representations, microworlds.

1. Introduction

One on the main purposes of this contribution can be found in the following remark of Mendelsohn [16] : "*Not many educational objectives can be accomplished if would-be learners are frustrated by unnecessary difficulties with the programming language and environment. It seems to the authors that far too much attention has been given to the debate on transfer of competence, and far too little to attacking the problems of ease of learning and ease of use.*"

The following reflexions and propositions result from a "on the job training" in programming and from researches on didactics. Those researches are based on a ten year long experience of teaching algorithmics and programming to several audiences of adults. These audiences consist mainly of teachers who will be (in the best case) or who are already themselves in charge of a course on computer science at a secondary level. Hence, the goal is not to train programming specialists but to give these teachers, in a short time and in an appropriate way, the most important concepts and methods of programming. You also have to stress methodology and the process of discovering or creating the concepts[1]. That nearly means that the "packing" is as important as the "contents".

---

[1] I completely agree with the remark of Jacques ARSAC in [1] : "*J'ai pratiqué la programmation... J'ai été contraint de réfléchir à ce que je faisais, pour pouvoir en rendre compte à mes étudiants. J'ai été encore plus sévèrement obligé d'y réfléchir face à ce public d'une extrême exigence intellectuelle et qui ne vous fait aucune concession : les professeurs de lycée*."

For many years the learning of programming was confused with the one of a programming language. After that, the knowledge of such a language was considered as unimportant : the main thing, before the expression in a programming language, was the prior *analysis* of the *problem*. The *top-down approach* should have been the guide leading from the *problem* to solve to the *program*. But this point of view on the secondary role of the languages can only be adopted by programming experts : they have so completely mastered the syntax of these esoteric languages that their conventional rules seem natural for them. For a long time these experts have built, step by step, through the practice of some of these programming languages, mental representations of the constraints of the computing device. So they are able to consider details of the languages as unimportant. As mentionned by du Boulay in [5] : "*Experienced programmers forget just how many of the implicit conventions about programming they have absorbed in their exposure to a variety of programming languages.*". But we know, and this is the main theme of this contribution, programming is "having it done by ..."; hence the constraints brought by the abilities of the performer are essential and omnipresent. Is it possible to describe these constraints that are a main and basic part of a "programming problem" without using a particular language to command this performer ? What could be the meaning of a prescribed *top-down* approach if you do not clearly tell the learner where and what is the "*ground floor*" ?

However, we have got to admit it, *the high level programming languages are made for programmers, not for learning programming*. As reported from Bonar by Mendelsohn [16] : "*current programming languages are unsuitable for novices. They point out that their emphasis in economy of expression ... is misplaced in designing languages for novice programmers."*. Without any doubt, there are good reasons for not stressing the syntactical details of these languages. Nevertheless, in spite of these prevalent opinions, teaching programming *inevitably* goes on giving the language more than its due. Very often, the "algorithm" or "the previous analysis" are only a remake, using the mother tongue, of the program that had been written before, as pointed out by Rogalsky [18] : "*L'observation d'élèves débutant en informatique dans un cadre scolaire semble indiquer que les méthodes d'analyse enseignées ne sont pas vues par les élèves comme un outil pour résoudre leurs problèmes de programmation, mais comme un contrat (avec l'enseignant) qu'il faut respecter. L'expression de la phase de travail qui correspond à l'analyse apparaît assez souvent comme une paraphrase du texte du programme écrit, qui peut précéder, accompagner, voire succéder à l'écriture directe dans un langage de programmation.*".

This contribution will try to show and to prove that if a certain language remains undoubtedly necessary, the prior analysis can make sense. Therefore we have to create intermediate levels between the world of the task and the one of a particular programming language : through images and metaphors, the teacher will be able to share with the learners the mental representations about the constraints of the performer he has gradually discovered through mastering some programming language(s). This relates to Mendelsohn's assertion : "*... the solution is to use an intermediate representation, one which minimizes initial difficulty but which leads into real programming.*" [16].

2. Task or problem

One of the main reasons for promoting the teaching of programming as an important part of a general training is the development, through programming, of *problem solving* skills. This was pointed out by many authors and well expressed by Van der Veer [19] : "*There has been*

quite some discussion recently about the urgency of incorporating computer literacy in our school curricula. An argument frequently adduced in support of such a development is the notion that experiences of computer programming turn people into better problem solvers.". And yet, if you look at some of the elementary books about programming, you can find the words "*problem*" or "*analysis*" linked with the description of simple and tedious tasks like :

- "to tell what is the date tomorrow (knowing what is the date today)";
- "to write a word back to front";
- "to search for a name within a list";
- "to throw a die until you get a series of 3 sixes and to tell the number of casts";
- "to choose the greatest among three numbers";
- "to sort a list"; ...

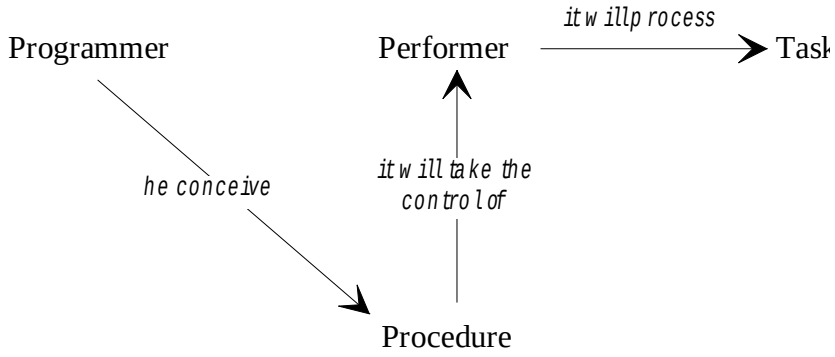
Even if you try to find other examples, all these activities are only simple, dull and tedious tasks and nobody can understand why one speaks about "problems". And even when the teacher insists on "analyzing the problem" of "choosing the greatest among three numbers" or (more difficult !) of "counting the number of words in a sentence", this request is meaningless for learners. What could be the significance of "analyzing" such stupid jobs ! Even the question "how do you manage it" is often insane and useless when facing these dull tasks : you can achieve it nearly without thinking ! In other words, the "Domain Space" as described by Gilmore [12] is not a "Problem Space" for beginners.

But we know that, as we are involved in programming, each of these tasks will give raise to a real and often difficult *problem*. Indeed the challenge does not lie in the task which is only the starting point : it would be so easy to "sort", to "choose the greatest number..." ourselves; the difficulty is not "to do it" but "*to have it done by a performing device*". It is in this gap between a nearly unconscious *acting* (doing it) and the problem of "*having it done by* someone -or something- else" that programming lies.

So, given this view, *programming* is

- faced with a *task* (about handling information, for classic programming) on the one hand,
- faced with the precise and complete indication of the abilities of a *performer*, on the other hand,
- to indicate by writing down a *procedure* (= directions for use), taking into account these abilities, how *to have the task done by* the performer.

Hence, the heart of algorithmics (and programming) is to be seen in the following diagram :



This is the fundamental diagram which is the center of all the programming problems. This approach helps to understand why children working with LOGO are really at the heart of programming. The depiction of the performer is in this case very simple : it is a "turtle" which

can move, turn,... when it gets the corresponding command and which can leave a track of its motions. So, the problem is not to draw a circle, a house or a ship (which is without any interest) but to have this pictures drawn by the turtle-performer by giving it the suitable directions. The learner does not draw anything, he *has it drawn* by the turtle, so he programs.

It should be noted that in this context a metaphorical level (the features of the turtle) takes place between the language level (LOGO) and the task level (to draw). This intermediate level allows to bring into the world of the task, not the constraints of the computer (technical device), but rather the requirements and rules imposed by the world of the performer, thus creating the specific programming problem (to have it drawn ...).

One can also find the same diagram as the essential component of the *problems* related to the use of digitally controlled machine tools, to robotics, to the command of automatons... and as a main part of the microworlds for programming learning.

Two important comments must be made here :

a. The "having it done" required by the programming problems is not a "live" one where you can apply straight to the performer; in this case one can give orders to a present performer and react on-the-job to the consequences of the actions that one orders to it. Here, one must "unfold" in advance the whole task that must be done, to obtain a prior procedure describing with all necessary details, without any ambiguity and with a total exhaustiveness both the elementary instructions and the description of the organization of its actions.[2]
b. A well set problem in this field must include the description of the task but also *the complete and precise indication of the abilities of the performer that will carry it on*. Otherwise it is impossible to write the procedure : you cannot have something done by someone (here something) if you do not know what he (it) can do. I shall indicate further on what must be included in the description of any performer to allow the conception of a procedure. We find here a very difficult aspect of the first steps of programming learning : one enjoins the learner to proceed by stepwise refinements without clearly telling him what the basic abilities of the performer are. There are many "problems" that only describe "the task to be done by" without any precision concerning the performer that will process it : these are *not* the terms of programming problems.

The agreement on this conception of procedural programming viewed as "having a task done subsequently by a performer" gives immediately a way for learning it. First, the learner has to discover and understand the various elements and the writing rules of a procedure. At this step he can train himself to conceive and write procedures about straightforward "tasks to be done by" by very simple performers (without any similarity with the computer-performer). The essential aim of this first step is to discover the main components of procedures (that will remain the same for the computer) and particularly to master the use of flows of control.

Next, the teacher has to depict the basic features of the computer-performer. This can be done in the same way as for the previous performers. This description will be a metaphorical

---

2 The difference between the direct command of a "visible" performer and the drawing up of a procedure forcing to anticipate the future behaviour of the performer is essential. Again, one can see this important difference through LOGO activities : at first, the learner can order the turtle-performer, visible on the screen and which moves and draws one by one as it follows these instructions; subsequently, the novice programmer can enter another world with the creation and the writing down of a program through the use of the editor : now the turtle is no more visible, the instructions have no immediate nor visible effects. One of the main difficulties of programming is to conceive the programs when the performer is absent. The same comment can be made concerning the gap between the direct use of some software and the writing down of macro-instructions.

one, not about the technical characteristics of the computer, but presenting the main features of the computer viewed as a performer. Especially, the language used to give orders will be as simple as possible, from a syntactical point of view, but full of imagery and meaningful from a semantic point of view. These images are very important : as Cohors-Fresenborg points it out [3] : "*There is a resonance between external representation and cognitive structures*".

In the end, some simple programming tasks will be taken up : they will allow to acquire the first skills, the "tricks of the trade". So, the learner will gradually build upon the initial image of the computer performer more mental representations of its abilities, according to the kind of tasks he becomes able to have done (thanks to the mastering of these tricks). In other words, supplementary metaphorical levels are superposed to the original depiction of the basic features of the computer performer; these additional strata correspond, not to the primitive actions of the performer, but to more complex possibilities. The corresponding tasks need no analysis : *it is no longer a problem* to have those done by a performer, perceived as more and more powerful. This has been shown by some authors [5,13,15,...] and well depicted by Arzarello [2] : these meaningful mental representations allow fundamental skills of the experts.

3. The depiction of a performer and the writing of a procedure

It is usual to distinguish two essential components in a procedure : the basic instructions corresponding to the *primitive actions* of the performer on the one hand, the *organizing instructions* (flows of control) which allow the ordering of these actions on the other hand. One of the main parts of the flows of control (alternative or repetition) are the *conditions* which can be assessed by the performer.

Hence, to describe a performer we have to specify :

- the list of the basic actions it can perform and the corresponding instructions;
- the control flow patterns allowed (sequence, alternative, repetition, jump, subroutine call, recursive call, ...), and
- the list of conditions the performer can evaluate.

It is also essential to depict the "working environment" of the performer : this gives a meaning to the previous lists and links the task that has to be done with the abilities of the performing device.

3.1. An example : the robot with a dropper

Its working environment is as follows :

- a alignment of glasses; these glasses are already more or less full of water;
- a dropper the robot can use;
- a big container full of an inexhaustible amount of water; the robot can fill the dropper from this bowl.

The initial state of the dropper is unknown; there is at least one glass but, all the glasses can already be full.

The procedure to be written down can mention the following commands :

- *choose the first glass* : whatever it was doing at the moment this instruction is encountered, the robot comes (or comes again) to deal with the first glass;

- *choose the following glass*[3] : the robot leaves the glass it was handling to deal with the following one;
- *press a drop into the chosen glass*[3] : the robot lets a drop fall into the glass it is dealing with at that moment;
- *fill the dropper* : the robot fills the dropper out of the container without forgetting the glass in front of it, if need be.

The following conditions may appear in the appropriate flows of control :

- *all the glasses are full; the chosen glass is full; the dropper is empty*.

Moreover, the opposite conditions are also allowed just as the assembling resulting from some of these conditions linked together with the words "AND" or "OR".

We still have to specify the flows of control which are available. At that moment, to learn an accurate style of (procedural) programming, it is better that the flows of control stay the same whatever the performer. These can belong to structured programming (sequence, alternative, repetition, subroutine call), to "GO TO" programming (sequence, jump, conditional jump), or even to "recursive" programming.

The tedious *task* would consist in filling this set of glasses; the *problem* lies in the conception and the writing out of the procedure, taking into account the robot's abilities -and nothing more-, to tell it how to fill all the glasses.

3.2. Some comments on that approach

First, a few remarks :

- The only "experiments" that have been made used the flows of control of structured programming.
- In those problems, there is no gap between the world of the task and that of the "language" of the performing-robot; naturally this one has the abilities to execute the task. The only problem lies in the correct organisation of the basic actions and of the needed conditions. So, the right use of the appropriate flows of control is the main (or even the only) difficulty.
- There are no syntactical constraints in the world of the performer : this world is semantically very close to that of the task and brings very few constraints from the language point of view. These constraints lie only in the use of flows of control.
- These problems seem far away from the real "programming problems" and they are not yet linked with computer. Nevertheless, we can often find here, among the many wrong answers of the learners, some general behaviors, reported by other studies :
  - The uses of "WHILE" statement are more difficult and more rare than those of the "REPEAT" statement.
  - The learners sometimes believe that the conditional IF ... THEN ... statement is sufficient to get a repetition and use that in place of the WHILE statement.
  - The procedure (subroutine) call is introduced at this moment as single instruction for a complex action (in opposition to basic instruction for simple action); so, the performer cannot execute this instruction properly and an explanatory annex is necessary; these explanations for performing the complex action take the form of a subsidiary procedure

---

[3] It is better not to insist on circumstances which would make the ordered actions impossible. So, the statement "choose the following glass" would lead to difficulties when the robot dealt with the last one; "put a drop in the chosen glass" is impossible when the dropper is empty or if none of the glasses had yet been chosen ... To insist too much on these restrictions is giving clues about solution to the learners.

next to the main procedure. In view of this non technical presentation, novice programmers use this flow of control pretty well.

- The conception of a procedure is often based on a mental, essentialy sequential execution; this makes the use of flows of control difficult; the beginners start from the mental representation of the succession of actions and thus from the basic abilities of the performer rather than from the global task; they write down the procedure by imagining the work of the performer. This difficulty is well known and good sumarized in Hoc's assertion [14] : "*Another characteristic of beginner's strategies could be seen as generating the statements by mental execution of the program.*".

As far as difficulties relating to this special approach are concerned, some comments have to be made :

- This bypass by programming that kind of robot-performer delays the programming of the computer (as a performer) and would set back the contact with the computer (as a machine). Nevertheless, the learners want to work with the computer as early as possible; this wish can be fulfilled by using some software, as for example a text editor the mastering of which will be essential when the time comes.
- Adults understand quickly why and how this approach prepares computer programming which is the main goal; hence they accept the artificial problems about these made up performers. As they quickly perceive the difficulties and the real challenge of these problems of "having it done by...", they also agree both with the useless tasks and the artificial performers.
- It is pretty easy to have the mistakes found by the learners from some answers : you find a counter-example (= a situation where the proposed procedure will lead to errors) and you act the role of the performer, following step by step the incorrect procedure. However, this type of correction is far away from the debugging allowed by real programming or by the LOGO microworld. In these later cases you can really try your procedure by submitting them to the performer. In the present situation, one could realize a multimedia environment : a set of video sequences would be organized to present the different steps of the execution according to any procedure; so the learner would see the result of his production, as in a LOGO environment. But, the actual difficulty is to set a system that could automatically find the initial conditions out of an incorrect procedure (state of the glasses, of the dropper, ... in the preceeding example) showing the errors of this procedure. The game is not worth the candle !

4. Depiction of the computer-performer

The following stage is to allow the discovery of the basic abilities of the computer, as a performer, according to the describing rules similar to those of the previous robot-performers. Too often, this discovery of the main features about the computing device takes place only through the learning of a specific programming language; at best, the mental representations about the computer (as a performer) are only by-products of the language : the syntactical constraints make it more uneasy to bring to light the semantical aspects of the device. So, enlightening metaphors and, most of all, a meaningful and syntactically unconstraining language are essential. Therefore, the teacher has to give attention to details (unimportant from the expert programmers point of view) which can facilitate the building of adequate mental representations about the computer-performer. See Hoc [15] : " ...*whatever the*

programming language, beginners have to learn the operating rules of what is called the "device" underlying the programming language" and Green [13] : "*it seems that existing notations too often act to raise barriers against programming...*". The present purpose is to propose a discovery of the operating rules through appropriate images and words.

4.1. The computer as a performer or the computer as a machine

Many researches insist on the need of describing the device the constraints of which originate the "programming problem" in order to make the training easier. The description of the computer as a machine (RAM, CPU, DOS, ...) is no use; we have rather to picture it as a performer (when it seems to "understand" the programs through the compiler), to draw its profile through a metaphoric depiction of its main features. These are, at first, the same whatever the procedural programming language.

Some of the components of the environment of the computer-performer are of course linked with the computer structure : keyboard, screen... But it is better to break here with a too "hardware" description of the computer : it is sufficient that the metaphorical features be relevant to the pursued goal, programming learning.

4.2. The main features of the computer-performer portrait

It is of course out of the question to detail this portrait : anyway, it is not very new nor original, save for being true to the framework of the depictions of the previous performers : environment, basic actions and the corresponding commands, conditions and flows of control. So, in short :

4.2.1. The environment

4.2.1.1. The computer-performer is a "drawers' manager"

It has at its disposal a set of labelled "drawers". The characteristics of such a drawer are :

- Its label, which cannot be removed or handled (by the performer during the execution). This label will be used, in the procedure, to indicate either the drawer itself, or its contents.
- The contents of the drawer : what will pass through it are "data" (integer, real number, string)[4].
- The type of the drawer, which determines the kind of data that can take place inside of it.

It also has at its disposal a (very) big data trunk : it can take any data from it, at the appropriate moments (when the available procedure will command to do so).

4.2.1.2. The computer-performer can communicate with the outside

Indeed, its working room includes a keyboard-door and a screen-window. Through the door, it can receive data and it can show data through the window too.

4.2.1.3. The computer-performer has at its disposal a workbench with some tools

These tools will give new value on the basis of some data that the performer will put into it. So there are tools named SUM OF..., DIFFERENCE OF..., RANDOM NUMBER, ...

Among these tools, some have a special status : they will allow the programmer (= the procedure writer) to write down suitable conditions. These tools give as result a comparison between data (of the same type) : IS LESS THAN..., IS GREATER THAN..., IS EQUAL TO..., ...

---

[4] The booleans are too early at this stage. The only use of the boolean type is made through the conditions, that will later be seen as booleans expressions and, as we know, this will lead to many difficulties.

4.2.2. The basic commands (instructions to act)

There are only three elementary instructions :

- the "fill in the drawer instruction" :
  Put any value *into* any drawer
- the instruction to fetch data from the outside
  Go to the (keyboard-)door, wait for the value someone will give to you and put it into any drawer
- the display instruction
  Display any data *on the (screen-)window*.

The meaning of "any data" in these commands must still be explained. Such data can be written down as :

- constant, as
  Put 1 *into* (the) Total (labelled drawer) or *Display* 'Hello'
- drawer label (in this case, what is meant is a copy of the data lying into the drawer) :
  Put (a copy of the contents of the drawer labelled) Name *into* (the drawer with the label) Client
  Display (a copy of the contents of the drawer labelled) Sum
- the result given by the use of one or many tools (we shall call this an expression) :
  Put (the SUM OF (a copy of the contents of the drawer labelled) Counter WITH 1 *into* (the drawer with the label) Counter
  Display (the) PRODUCT OF (the copy of the contents of the drawer labelled) Counter WITH (the copy of the contents of the drawer labelled) Age

4.2.3. The conditions

These are comparisons of data by using the tools previously mentionned, as
Counter IS LESS THAN 10 ; Name IS EQUAL TO 'Smith'; ...

As for previous performers, these comparisons can be used to build more complex conditions with the words AND, OR and NOT.

4.3. Some comments

This description has been reduced to a skeleton and has erased many details. (See [7-9] for more information). I must insist on the importance of some of them.

a. As much as possible, the previous metaphors have to be translated into pictures. So I have realized a set of videos to depict these images. So, when you can *see* that the consultation of a drawer do not clear it (it is only a copy of the value inside of it that the performer take off) but that the filling of a drawer erases the value previously inside of it, many difficulties related to the concept of a "variable" disappear.
b. As programming experts, we often have a guilty conscience to choose unusual terms and words instead of the customary technical jargon. Nevertheless, it may be better to wait for the creation of appropriate mental representations in the learner's mind, before introducing the usual but often less meaningful words. So, I prefer to talk about "drawer" than about "variable", about "filling" rather than about "assignment", about "shelves" before about "array"...[5]

---

[5] Rather than going on for many years with the study of the learning problems and barriers partly caused by inadequate linguistic constraints, one could devote a part of this time and of these efforts to design, test and study alternative presentations and languages. Not programming languages but languages to learn programming.

c. Some details are probably important and would be evaluated : so, is it significant to depict "closed drawers" (you -and the performer- cannot see what value is in it) rather than "open compartments" ? (See [15] and Cohors-Fresenborg conception of variable [3].). Is it relevant to mark out the "tools" by names that stress the result (the SUM OF ...) rather than the action (TO ADD ...) ?

d. One of the most important features of the computer-performer is its *amnesia*. You can give orders to it only in the present imperative form[6] without any reference to the past : its only memory is the *current* contents of the "drawers". Terms as "some time ago" and tenses like the past tense are meaningless.

e. The main criticism one could make of such a depiction of the computer-performer is the underlying anthropomorphism. The excessive character of metaphors and images is itself the very remedy to this criticism : no learner, even the very beginners, can believe that "it really happens like this inside the computer"; they obviously understand that, to conceive and write program they can do "as if...".

f. The learning environment "Images pour programmer" ("Images for programming")[8], besides the videos, the accompanying documents and the book, also includes the software START that allows the novice programmer to follow -in a schematic environment similar to those previously depicted- the execution of the first programs he has written down, using a language which is very close to the one we have described above, according to du Boulay's remark [5] : "*... the implementation of the programming language and the preparation of teaching materials must be developed as a whole, so that each refers to the machine in the same terms*.". Nevertheless, two comments must be made as to the limits of this "language" :
   - Even if this language is very meaningful according to the previous metaphors, syntactical constraints have now to be respected. It is a pity, but it will still be like this for many years and for any "programming" language !
   - This software does not allow the discovery of the computer-performer in an interactive way : you cannot give a command and see the corresponding action of the performer; you can only follow the execution of short programs, written down using a text editor. Other environments allowing for *exploration* of the basic abilities of the computer-performer would be necessary before environment allowing *programming* of this latter one. (See du Boulay [6] for an example of such a complete environment).

g. This first depiction of the main features concerning the computer-performer is quite simple and straightforward. Some additional abilities will be added to the three basic actions but they will not change very much the above description. Of course, the abilities of the performer depend on the set of "tools" laying on its "working bench" to handle numerical and string data. But, it is useless to present to beginners complete lists of these "handling data tools" too early : the learners would not know what to do with them. These additional possibilities will be presented later, when they will become necessary for programming some tasks. Anyway, the tools for handling numbers are often expected and easily accepted by learners. The situation will be absolutely different with the tools for handling strings : the operations that they allow are not similar to those that are familiar to a human being.[7]

---

6 The most appropriate tense would be the future imperative to express better this "having it done in the future" ! But this tense does not exist nor in English, nor in French...

7 We face here a common feature to the computer as a performer and to the computer as a machine : data processing is always formalist. This is right for number processing : it is naturally formalist. But it is very

5. Some "tricks of the trade" in programming

The main concepts are now available; the most important thing remains to be done : to have these concepts used by learners. So, the latter will, step by step, create more complex and more complete mental representations about what they can have done by the computer-performer. This is what I call "to master the first tricks of the trade", according to Hoc's remark [14]: "*I suggest that a too early training of top-down programming methods should be avoided, before the subject has correctly learned the constraints of the computer operation on the overall structure of the procedures, and knows a wide range of concrete plans to be transferred*.".

Now, the beginner has to build gradually, on a base which is made up of the basic features about the computer-performer, new images about it, its constraints and its possibilities. He has to go from what is basic to what is possible. Step by step, he must discover new tasks that do not match the basic abilities of the performer but which are no longer programming "*problems*" : the corresponding skills are mastered. (See [2, 14, 15]).

The teacher has to play an important part in the attainment of those mental representations and programming skills.

a. First he has to list the desirable "tricks of the trade" that have to be mastered. It is very difficult to find them by using introspection : these basic skills are deeply integrated into the expert know-how; they are nearly unconsciously activated when necessary. Hence, it is not easy to remember those "details" that formerly were a problem but that are now quite instinctive reactions.
b. On the basis of this set of skills, we then have to find out some programming tasks that will make them essential. The best is to propose a problem that as often as possible isolates "in the pure state" such a trick of the trade.This skill has of course to be fulfilled through many different problems to take place in the programmer's mental abilities : it is a characteristic of the attainment of such "tricks of the trade" to become possible only by exercizing it in many different contexts.

6. By way of conclusions

During the last ten years, computer science and especially algorithmics and programming have been taught in schools at a secondary level. Whatever the didactics approach, the main challenge to take up concerns the unsignificant, unrewarding or mindless nature of the first tasks to program. There is an increasing gap between the high qualities and the performances of recent software (that learners also use) and the simple and useless tasks proposed to beginners.

On the other hand, many concepts or knowledge in procedural programming become more and more helpful for the correct use of these new softwares : macro-commands, programming features,... These are wonderful opportunities to teach general algorithmics concepts, no more by using classical programming languages, but through these new "programming" environments (see [4]). What would be the depiction of the "performer" when you program while using a word processor ? What are the basic skills in those contexts ?

---

disturbing for text processing : we do not process strings formally, we use words... We are here in the heart of computer science (or, better, of data processing) : it is an endless quest for trapping and confining meaning into form.

What are the mental representations about the spreadsheet-performer or the word-processor-performer ? What about the flows of control in these specific programming environments ?

There are still many unanswered questions ! In the coming years, as also pointed out in [2], the psychology of programming has to pay attention not only to classical programming and programming languages but more and more to the new programming environments provided by recent softwares.

7. References

1. Arsac, J. : Préceptes pour programmer. Dunod, Paris, 1991.
2. Arzarello, F., Chiappini, G.P., Lemut, E., Malara, N., Pellerey, M. : Learning to program : a cognitive apprenticeship. These proceedings.
3. Cohors-Fresenborg, E. : Registermachine as a Mental Model for Understanding Computer Programming. These proceedings.
4. Dagdilelis, V., Balacheff, N., Capponi, B. : L'apprentissage de l'itération dans deux environnements informatiques. ASTER 11, 45-66, Paris, (1990).
5. du Boulay, B., O'Shea, T., Monk J. : The black box inside the glass box : presenting computing concepts to novices. Man-machine studies (International journal of) 14, 237-249 (1981).
6. du Boulay, B., Patel, M., Taylor, C. : Programming Environment for Novices. These proceedings.
7. Duchâteau, C. : Quand le savoir faire ne suffit plus. CeFIS, Facultés. N-D. de la Paix, Namur, 1989.
8. Duchâteau, C. : Images pour programmer. Un environnement pour l'apprentissage de la programmation. CeFIS, Facultés. N-D. de la Paix, Namur, 1989.
9. Duchâteau, C. : Images pour programmer. Apprendre les concepts de base. De Boeck-Wesmael, Bruxelles, 1990.
10. Duchâteau, C. : Incursion au pays du faire faire. CeFIS, Facultés N-D de la Paix, Namur, 1983.
11. Duchâteau, C. : Programmer ! Pour une découverte des méthodes de la programmation. Wesmael-Charlier, Leuze-Longchamps, 1983.
12. Gilmore, D. : Visibility and Novice Programming Environments. These proceedings.
13. Green, T.R.G. : Programming Languages as Information Structures. In : Psychology of Programming (Hoc, J-M., Green, T., Samurçay, R., Gilmore, D. , ed), 117-138. Academic press, 1990.
14. Hoc, J.-M. : Analysis of Beginners'Problem-Solving Strategies in Programming. In : The Psychology of Computer Use (Green, T.R.G., Payne, S.J., Van der Veer, G.C., ed), 143-158. Academic Press, London, 1983.
15. Hoc, J.-M., Nguyen-Xuan, A. : Language Semantics, Mental Models and Analogy. In : Psychology of Programming (Hoc, J-M., Green, T., Samurçay, R., Gilmore, D., ed), 139-156. Academic press, 1990.
16. Mendelsohn, P., Green, T.R.G., Brna, P. : Programming Languages in Education : The Search for an Easy Start. In : Psychology of Programming (Hoc, J-M., Green, T., Samurçay, R., Gilmore, D., ed), 175-204. Academic press, 1990.
17. Rogalski, J. : Enseignement de méthodes de programmation dans l'initiation à l'informatique. In : Actes du colloque francophone sur la didactique de l'informatique, 61-74. Editions de l'EPI, Paris, 1989.
18. Rogalsky, J., Samurçay, R., Hoc, J-M. : L'apprentissage des méthodes de programmation comme méthodes de résolution de problème. Le travail humain 51, 309-320 (1988).
19. Van der Veer, G.C., Van de Wolde, G.J.E. : Individual Differences and Aspects of Control Flow Notations. In : Psychology of Programming (Hoc, J-M., Green, T., Samurçay, R., Gilmore, D. , ed), 107-120. Academic press, 1990.