Educational Multimedia and Hypermedia, 1994

edited by
Thomas Ottmann
Ivan Tomek

Proceedings of *ED-MEDIA 94*—
World Conference on
Educational Multimedia and Hypermedia
Vancouver, BC, CANADA; June 25-30, 1994

AACE Association for the Advancement of Computing in Education

Teaching programming to novices: a review of approaches and tools

P. BRUSILOVSKY
International Centre for Scientific and Technical Information
Kuusinen str. 21b, Moscow 125252, Russia. E-Mail: plb@plb.icsti.su

A. KOUCHNIRENKO
Department of Mechanics and Mathematics,
Moscow State University, Moscow, Russia. E-mail: agk@agk.msk.su

P. MILLER
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA, 15213 USA. E-mail: plm@cs.cmu.edu

I. TOMEK
Jodrey School of Computer Science, Acadia University
Wolfville, Nova Scotia, Canada. E-mail: Ivan.Tomek@AcadiaU.ca

Abstract: We review three different approaches to teaching introductory programming: the incremental approach, the sub-language approach, and the mini-language approach. The paper analyzes all three approaches, providing a brief history of each and describing an example of a programming environment supporting this approach. At the end, we summarize the results obtained up to date.

There have been many efforts to develop special methodologies, languages and tools for supporting the initial steps in programming education. A number of special programming environments for novices were designed. In this review we analyze these attempts and summarize the experiences. The review is intended for both the practitioners who are interested in applying effective ways of teaching introductory programming, and researchers who work in the field of programming instruction and learning.

We outline three different approaches to teaching introductory programming. We call them the incremental approach, the sub-language approach, and the mini-language approach. The paper analyzes all three approaches, providing a brief history of each and describing an example of a programming environment supporting this approach. At the end, we summarize the results obtained up to date and discuss directions for further research and development.

The Incremental Approach

The first attempts to overcome the problems of teaching introductory programming use the approach that we call the incremental approach to teaching a programming language. This approach was first time described for a wide audience by Holt et al (1977) for the case of teaching PL/1 and reapplied and extended independently by several authors including Holt's group and others (Holt & Hume 1980; Atwood & Regener 1981; Ballman 1981; Tomek, Muldner & Khan, 1985).

In the incremental approach, the language being taught is presented as a sequence of language subsets. Typically, each subset introduces new programming language constructs while retaining all the constructs of the preceding subsets. Each subset is complete, i.e., it is precisely defined and can be learned or implemented without the following subsets. The whole programming language is then learned

by examining progressively larger subsets. At each stage the new constructs and elements of a current subset are introduced to the student and then the new knowledge is mastered by solving a number of programming problems. This permits greater concentration on each set of new language operators and control structures.

Since each subset forms a complete sublanguage, a language processor can be specially constructed to support the learning of each subset. The first idea of Holt et al. (1977) was to take advantage of current subset restrictions to improve error handling and error message generation. Later it was suggested (Ballman, 1981; Tomek, Muldner & Khan, 1985) to combine the advantages of the incremental approach and program visualization. Program visualization is important for novices to more easily understand the semantics of a programming language construct. The problem with this principle is that many aspects of program execution can be visualized, and this requires a lot of screen space and overloads the student's working memory. It also creates implementation problems. Implementing visualization using the incremental approach makes it possible to focus the attention of the student on the new features that are being introduced in the current language subset. This helps to avoid screen overload and student working memory overload and simplifies the design of visualization.

As many language subsets as desired can be specified in order to provide advanced error handling and space for detailed visualization. For example, the incremental approach was suggested for teaching Pascal by Holt and Hume (1980) who divided the language into 8 subsets, Tomek, Muldner and Khan (1985) divided the same language into 10 subsets, and Atwood and Regener (1981) divided the language into 15 subsets. One of the most developed applications of the incremental approach is the PMS system described in the next subsection.

PMS - a system to make learning Pascal easier

PMS (Pascal Made Simple) was developed to help students of computer science understand the semantics of standard Pascal constructs (Tomek, Muldner & Khan, 1985; Tomek & Muldner, 1986). The package was designed in the first half of the 1980's for IBM PC computers and the implementation reflected the limitations of this machine - its small memory and minimal graphics capabilities. The package provided programming environments for ten languages, each of them a subset of Pascal designed to illustrate some aspects of the language in an optimal way. (Hardware limitations made it impossible to make consecutive sublanguages full subsets of preceding subsets.) The individual languages included MINI (primitive data types, I/O, assignment, basic file i/o, control structures), NUMBERS (illustrating real and integer arithmetic and their limitations), TYPES (enumerated types and arrays), RECORDS (records and arrays), VARIANTS (records with variants), SETS, BINARY (binary files), STRINGS, PROCEDURES (procedures and functions), and POINTERS (dynamic data objects and pointers).

Each language has its special display tailored to the aspects of Pascal that it was designed to illustrate, but all shared the same structure to ensure consistency of user interface which was considered very important for novice computer users. In particular, each minilanguage screen consists of a menu line at the top of the screen, a source code window and a data window occupying most of the screen, and an output window on the bottom. Each language features a screen oriented syntax driven editor and an interpreter. The screen editor is capable of intercepting errors as soon as the user types them and starts a new line so as to minimize user frustration with syntax errors and divert their attention from dealing with them and focusing on syntactic details. The editor produces intermediate code executed by the interpreter in either step-by-step or block fashion. The interpreter highlights programming constructs in the source code as they are being executed and the effect is simultaneously shown in the data window. To illustrate the approach taken by PMS to visualize programming concepts, figure 1 shows a program demonstrating the concept of pointers: the data window shows pointers as simulated memory 'addresses' obtained as randomly generated integers.

PMS was used for several years in an introductory Pascal course with about 100 students each year, in weekly 2 hour labs compulsory for all students. The students were asked to execute a series of prewritten programs demonstrating essential concepts, and to write and execute their own programs under supervision. According to both students and instructors, the experience with PMS was very positive and the look 'into the machine' as PMS was executing programs considerably helped students understand the meaning of data types, data structures, pointers, operation of procedures and functions, recursion (PROCEDURES feature a simulated stack), and other concepts. The main reason why PMS

```
PROGRAM Pointers;                                  * p1       <0008>
VAR p1, p2: ^CHAR;                                 * p2       <0009>
          c: CHAR;                                 * c         Q
 BEGIN                                             *
             WRITE('Enter a character');           *
             READLN(c);                            *
             NEW(p1);                              *
             p1^ := c;                             *
             NEW(p2);                              *
             WRITE ('Enter a character');          *
             READLN(p2^);                          *
             WRITELN('Characters ',p1^,p2^);       *
             {Now exchange}                        ********************
             c := p2^; p2^ := p1; p1^:= c;         *<0009>  q
             WRITELN('Exchanged: ',p1^,p2^)        *<0008>  Q
END.                                               *
*******************************************************************
Enter a character: Q
Enter a character: q
Exchanged: Qq
```

Figure 1: Example of a POINTERS program depicting pointers as simulated memory addresses. Note the I/O window at the bottom showing the only text that the user would normally see on the screen when executing the program in a regular programming environment.

was eventually abandoned was that it did not have enough facilities to support new features of the Borland Pascal taught in the program such as Units and graphics, and there were no resources to update it.

The mini-language approach

The idea of the mini-language approach is to design a small and simple language to support the first steps in learning programming. In most of the existing mini-languages a student learns what programming is by studying how to control an actor, which can be a turtle, a robot or any other active entity, acting in a microworld. Although an actor can be a physical device, the student usually deals with a program model of such a device and observes the behavior of the executive on a screen. A special miniature programming language is used to control the actor. The language includes small set of commands that the actor can perform, and a set of value-returning queries and control structures. Most mini-languages include all basic control structures (conditional execution, looping, recursion, etc.) and a mechanism for creating some kind of sub-program.

The development of the mini-language approach was seriously influenced by *turtle graphics* of Logo (Papert, 1980). Logo was not designed especially for the purpose of teaching programming but the "turtle subset" appears to be a good tool for introducing programming to novices and it provides genuine insight into problem solving with a computer. Note that unlike most of the actors of mini-languages described below, the turtle of Logo is "blind", it can't check its microworld. The "turtle subset" also does not support such classic control structures as Pascal-like *if* and *while*.

The first and still the most popular mini-language *Karel the Robot* was designed by Richard Pattis for university students taking their introductory programming course (Pattis, 1982). Karel contains all important Pascal-like control structures and teaches the basic concepts of the notions of sequential execution, procedural abstraction, conditional execution, and repetition. The overhead of full high level programming languages is reduced as there are no variables, types or expressions in Karel. The actor, robot Karel, performs tasks in a world that consists of intersection streets and avenues, walls, and beepers. Karel can also carry some beepers in his "bag". The main actions of Karel are move, turnleft, pickbeeper, and putbeeper. A set of 18 predicates allows Karel to check the state of his microworld. For example,

Karel can determine the presence of nearby walls, if there are any beepers in his bag or at his location, and the direction he is facing. By writing programs that cause Karel to perform carefully selected tasks, students gain experience with the fundamentals while using a pleasant and persuasive metaphor.

Among other mini-languages designed in the beginning of eighties independently of Karel we should mention *Josef the Robot* (Tomek 1982, 1983), *Wayfarer* (Kouchnirenko & Lebedev, 1988), and *Turingal* (Brusilovsky, 1991).

Karel Genie, the mini-language programming environment

Karel Genie is a novice programming environment specifically designed for the Karel mini-language. Karel Genie is a member of the Macintosh based Genie family of structure editor based programming environments developed at Carnegie Mellon University. To support the novice programmer, the Karel Genie provides a set of specially designed tools, which include a structure editor, a program decomposition view for both looking at and editing programs, and a runtime system with advanced visualization tools. The Karel Genie is an integrated programming environment. Editing the program, executing the program and taking a high level view of the call structure are all presented within a single user-interface, allowing students to move from one activity to another with little cognitive overhead.

The details of precise syntax are not a burden for the user of Karel Genie because it is a structure editor. Maximum support is provided to the novice programmer since program construction can be conducted entirely through menu interaction. Every syntactically legal program transformation and only syntactically legal program transformations appear as alternatives in the popular Macintosh pull-down and pop-up menus. As novices develop expertise they begin to exploit Karel Genie's text edit features. The granularity of text edits is kept small and errors tend to be rare and trivial to repair.

Procedural abstraction is especially supported by the Karel Genie graphical program design tool. Students decompose a problem into simpler sub-problems by dragging the mouse from the program root. The environment invites the student to name a new-instruction. (Karel's sub-programs are called new-instructions.) If the new-instruction is not already defined, the Karel Genie builds a new-instruction shell with its name inserted at the correct point. Whether the instruction is new or not, a call to the sub-program is inserted appropriately.

Karel Genie is highly visual and successfully uses multiple representations. This is clearly illustrated in Karel Genie's runtime system. From the point of view of the novice this is not a separate system, but simply an extension of the same environment in which the program was written. Code is highlighted as it is executed. At the moment a "move" is executed in the code window, a Karel icon moves in the world. Selection, repetition, and recursion are similarly visually reinforced. This is extremely valuable in teaching basic concepts as well as in introducing powerful and "advanced" concepts such as loop invariance. The runtime environment also has a visual call stack. This makes procedure invocation and return clear. Coupling this with the fact that Karel programs can be single stepped and run either forward or backward, students who are taught properly in the Karel Genie are using procedures in appropriate ways from as early as the second day of class!

The Karel Genie has been in use in secondary schools and universities throughout the US. for nearly ten years. In addition to CMU it has been used in computer science classes at Harvard University, New York University, Stanford University, Swarthmore College, Ohio State and a number of other institutions.

The sub-language approach

The idea of the sublanguage approach is to design a special *starting subset* of the full language containing several easily visualizable operations. Such a subset can support the first steps of learning programming and helps later in introducing more complex programming concepts. The sub-language approach is quite similar to the mini-language approach, but differs from it in one important feature. While the mini-language approach uses a special miniature language with its own commands and control structures, the sub-language approach provides only a set of commands and queries as an extension of some "big" programming language. These commands and queries are used in combination with standard control structures of the "big" language. The sub-language approach was also influenced by the turtle graphics of Logo. A set of four "turtle" commands of Logo provides the first example of a sub-language. The success of Logo in general and turtle graphics in particular stimulated further research and resulted

in several successful attempts to install turtle graphics commands into general-purpose programming languages as PL/1, Pascal, Basic, and Smalltalk.

A well less known experience in designing sub-languages has been accumulated in Russia. Russian researchers Zvenigorodsky and Kouchnirenko extended the ideas of Logo turtle and developed a more formal and broader concept of an *executive* which can be applied in both the mini-language approach and the sub-language approach. The executive consists of an actor (the executive itself) working in a microworld and a small set of commands and value returning queries that the actor can perform. The concept of an executive makes possible the use of several executives within the same programming language. One of the first Russian languages for teaching programming - Robik (Zvenigorodsky, 1982) - contains a mechanism for an easy change of executives providing a way to work with different sublanguages within a language. A number of executives can be developed within Robik with their own microworlds, commands and queries. Another approach was developed in the Moscow State University by Kouchnirenko (Kouchnirenko & Lebedev, 1988). This approach and the supporting programming environment is described in the following section.

KuMir - a programming environment with zero response time

Since 1985, learning programming became a part of the high school program in USSR. In the first school textbook, academician Andrew Ershow introduced a simple ALGOL-like language (later nicknamed as E-language, in honor of A. Ershow). The first programming environment for the E-language called E-practicum and it's last version called KuMir are distributed widely in Soviet schools and universities. Several different executives were designed and used in several implementations of E-practicum and KuMir (bileg, landrower, builder, etc). The current school textbook (Kouchnirenko, 1990, 1991, 1993) which has been distributed in some 5.6 millions of copies is based on an improved version of E-language. This textbook introduces the basic concepts of programming by a problem approach. Posing problems to control two executives Robot and Drawer, the textbook naturally introduces such concepts as subprogram, parameter, *loop n times*, *while* and *if* constructs, integer and real variables, arrays, etc.

KuMir is an example of so-called zero response time programming environments. It is an integrated system combining a text editor with a zero-response-time incremental compiler and an unsophisticated source-level debugger. The proper name for such a system is "editor-compiler": while you are typing your program the compiler is working on it, so the program is ready for execution anytime and without any delay. For the first time such kind a system, namely the well-known Cornell program synthesizer, was introduced in (Teitelbaum, 1981). KuMir was developed to reduce the time wasted by teachers and students because of trivial technical problems. The main features of KuMir are as follows.

- All statically detectable errors are found while editing.
- User's and compiler's "domains of influence" are strictly separated. The user may modify source text in the text's window without any restrictions. The only thing the compiler may do is to write messages in the message window. The user may react immediately or postpone the reaction. Thus, the user may transform a program from one correct state to another correct state passing through intermediate incorrect states ignoring all messages.
- Diagnostic messages appear as soon as possible without any special user intervention. Messages appear in all places associated with the error and vanish as soon as the source of the error is eliminated.
- KuMir uses the metaphor of "marginal notes" to attach system messages to the text of the program. The screen is divided into two windows: the left window contains program text, the right window (the "margin") is used for diagnostic messages by the compiler and the debugger. The diagnostic message appears near the line it refers to. While tracing the program, the values of the evaluated expressions are shown in the margins. This visualization feature can be used by beginners to write and debug small programs without input/output commands.

A special feature of KuMir is the possibility of using several executives simultaneously. From a software engineering point of view, the executive in KuMir is a package containing several procedures and functions. External executives can be loaded dynamically. After loading a package, the executive

self-integrates into the language: the names of its procedures become visible and KuMir statically checks type correspondence of their parameters. New executives can be designed directly in E-language itself, thus making it possible for the users of the environment to design executives according their own needs. The mechanism of executives substantially extends the range of interesting but solvable problems. Many interesting programs can be divided into two parts: a small header, providing user interface, and a huge package containing tens of procedures doing all the work. If such a package has implemented as an executive, then to design such a program and get some interesting result, the student has to implement only the header itself. For example, to implement a graphical editor over an appropriate executive, the student will usually have to write only 50-80 lines.

Wide experience proved that KuMir, equipped with Robot, Drawer, and other executives is a good tool for the programming courses for 12-16 years old school students. During the last three years KuMir was also used at the Department of Mathematics at Moscow State University to allow students to solve a lot of simple problems quickly: during the first term, the students spend about 20 hours to solve 90 problems on programming, calculus and algebra.

What we can learn from the presented experience

In the above sections, we have described three approaches to teaching programming for novices and presented some experience with these approaches. The three approaches were developed independently and constitute different strategies to make learning programming easier. However, the environments of all three share several similar features stemming from the same goal: to overcome the beginners' problems of learning programming. The analysis of these similarities can provide some good ideas about how to teach programming to novices and how a novice programming environment should be constructed. We distill three lessons from experience:

- Teaching programming should start from a small, simple language subset. Small size and simplicity protect novices from cognitive overload. New features are sensibly added to a well understood base, rather than overwhelming the novice.

- Execution should visually reveal the semantics of language constructs and, where possible, elucidate principles of program structure and execution. Visual queues enable the novice to understand semantics of introduced constructs, protecting them from developing misconceptions. Visibility provides a feedback for exploratory learning and problem solving.

- Concepts need to be embedded in rich contexts. Visual metaphors make it easier to develop interesting problems for important concepts. Problems that achieve visible and meaningful results aid concept mastery by reinforcing with problem solving activities. Newly learned concepts should be immediately applied to solving meaningful attractive problems.

Another important lesson is that selecting the best approach is not enough for successful teaching of programming. Even with the right approach, learning programming should be supported by a good programming environment. Such an environment should keep both the microworlds and the student program visible on the screen. The program should typically be executed one instruction at a time, while the interpreter highlights programming constructs in the source code as they are being executed and the effect is simultaneously shown in the microworld. For the mini-language and visible subset approaches, the interpreter should also provide visualisation for those concepts of the language that are not visualized by the microworld. Important features include visualization of variables and the stack of subroutine calls. Such an interpreter supports both understanding of programming constructs and program debugging. The environment should also provide a structure editor, to increase student productivity, and enable the student to concentrate on the more important parts of problem-solving. The structure editor protects the student from making most syntax errors and provides immediate diagnostics for the remaining ones. Providing menus or hot keys, the structure editor also solves the contradiction between the requirements to enter the constructs easily and giving them meaningful names.

Two other important points to discuss are the areas of applicability of the three approaches and main differences between them. The main distinction between the three approaches is whether the approach uses a special executive or special visualization as in the mini-language and sub-language approaches,

or regular language operations visualized by the environment as in later implementations of incremental approach. The advantages of the first two approaches are in attractiveness and additional motivation which are provided by a well-designed visible executive. Programming problems applied to the actor in a microworld are usually more attractive and meaningful for the students. Sometimes these problems look as a puzzle rather than a 'serious' programming task and the problem-solving activity becomes a kind of a game. We think that if the novices being taught are well-motivated computer science students then an executive or a special visible subset are not vitally required, but even here they can help students in their first programming lessons. For younger students and for those who learn programming as a part of some computer literacy course or on their own, the application of a motivating executive is very important for the success of learning. The younger the students are the more attractive the executive should be.

The second distinguishing point is whether an executive or a visualization subset is used as a part of a bigger language such as Logo or Pascal as in the sub-language approach, or whether a special miniature language is designed to control the executive. We think that the sublanguage approach is better when the student's direct goal is to learn a particular big language. The student can learn most of the control structures and operators of the language more easily with the help of the visible subset. However, if the goal in not to learn a particular language but to learn the principles of problem solving by programming which can later be followed by learning a "real" language, the mini-language approach is better. Mini-languages can provide a sound basis for systematic problem solving for people who will program only to customize their spreadsheet, database, or CAD package, or another application program. Mini-languages open a door to new educational opportunities. Regardless of the student's eventual penetration into programming and regardless of the age of the student there is a positive residue of the study of a mini-language.

Other paradigms

In conclusion, we should mention that this paper discusses only the approaches related with teaching procedural paradigm languages. The lessons learned are, however, important for teaching other programming paradigms as well. Let us see, for example, what has been done in supporting object oriented programming which is becoming more and more popular. The most famous example of an object-oriented programming language is probably Smalltalk (Goldberg & Kay, 1977), for which one of the main motivations was to transport programming closer to the human perspective of the world. However, even though Smalltalk was often advertised as a programming language very suitable for teaching programming, it appears quite difficult for novices (Singley & Carol, 1990). Special object-oriented mini-languages (Fenton & Beck, 1989; Sellman, 1992) as well as an analog of a "visible extension" of Smalltalk (Borne, 1991) were suggested to provide an "easy introduction" to object-oriented programming. The concept of a special environment with extended visualization is also important for the object-oriented approach. A good example of how such an environment can be designed to support the first steps of learning Smalltalk is presented in (Böcker & Herczeg, 1990). Another language and environment that should be mentioned in this context is Prograph (Cox, 1989). Prograph - standing for programming in graphics - is a general purpose object oriented programming language currently running on Macintosh computers. Prograph's interface is purely graphical and supports visual execution. The language has been used to teach high school students.

References

Atwood,J.W. & Regener,E. (1981). Teaching subsets of Pascal. *SIGCSE bulletin*, 13(1), 96-103.

Ballman,T. (1981). Computer assisted teaching of FORTRAN. *Computers and Education*, 5(2), 111-123.

Böcker,H.-D. & Herczeg,J. (1990). Browsing through program execution. In D.Diaper et al (eds.) *Human-Computer Interaction - INTERACT'90.* Amsterdam: North-Holland, 991-996.

Borne,I. (1991). Object-oriented programming in the primary classroom, *Computers and Education*, 16(1), 93-98.

Brusilovsky,P. (1991). Turingal - the language for teaching the principles of programming. In *Proceedings of the Third European Logo Conference*, Parma: A.S.I., 423-432.

Cox,P.T., Giles,F.R. & Pietrzykowski,T. (1989), Prograph: a step towards liberating programming from textual conditioning. In *Proceedings of 1989 IEEE workshop on visual languages*, 150-156.

Fenton,J. & Beck,K. (1989). Playground: An object-oriented simulation system with agent rules for children of all ages. In Object-oriented programming: systems, languages, applications. In *Proceedings of OOPSLA'89,* 123-137.

Goldberg,A. & Kay,A.C. (1977). Teaching Smalltalk: Methods for teaching the programming language Smalltalk; Smalltalk in the classroom, Xerox PARC, June 1977.

Holt,R.C. at al. (1977). SP/k: A system for teaching computer programming. *Communications of the ACM,* 20(5), 301-309.

Holt,R.C. & Hume,J.N.P. (1980). *Programming standard Pascal.* Reston.

Kouchnirenko,A., & Lebedev,G. (1988). *Programming for mathematicians.* Moscow: Nauka, (In Russian).

Kouchnirenko,A., Lebedev,G. & Sworen' R. (1990 ,1991, 1993) *Foundations of Computer Science and Technology.* Moscow: Prosveshchenie. (In Russian)

Miller,P. & Chandhok,R. (1989). The Design and Implementation of the Pascal Genie, In *Proceedings of the ACM Computer Science Conference,* Louisville, KY.

Papert,S. (1980). *Mindstorm, Children, Computers and Powerful Ideas.* New York: Basic Books.

Pattis,R.E. (1981). *Karel - the robot, a gentle introduction to the art of programming with Pascal.* New York: Wiley.

Sellman,R. (1993). Gravitas: An object-oriented discovery learning environment for Newtonian gravitation. In *Proceedings of East-West conference on human-computer interaction,* Moscow, 31-34.

Singley,M.K. & Carrol,J.M. (1990). Minimalist planning tools in an instructional system for Smalltalk programming. In D.Diaper et al (eds.) *Human-Computer Interaction - INTERACT'90.* Amsterdam: North-Holland, 937-944.

Teitelbaum,T. & Reps,T. (1981). The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM,* 24(9).

Tomek,I., Muldner,T. & Khan,S. (1985). PMS - a program to make learning Pascal easier. *Computers and Education,* 9(4), 205-211.

Tomek,I. & Muldner,T. (1986). *A Pascal primer with PMS.* McGraw-Hill.

Tomek,I. (1982). Josef, the robot. *Computers and Education,* 6, 287-293.

Tomek,I. (1983). *The first book of Josef.* Englewood Cliffs: Prentice-Hall.

Zvenigorodsky,G. (1982). Introductory language Robic in an educational programming environment. In *Software for informatics.* Novosibirsk, 72-85. (In Russian)