Education and Information Technologies **2** 65–83 (1997)

Mini-languages: a way to learn programming principles

PETER BRUSILOVSKY
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA.
E-mail: plb@cs.cmu.edu

EDUARDO CALABRESE
Dipartimento di Ingegneria deli' Informazione, University of Parma, 43100 Parma, Italy.
E-mail: educal@ce.unipr.it

JOZEF HVORECKY
University of Economics, 83220 Bratislava, Slovakia.
E-mail: hvorecky@vseba.sk

ANATOLY KOUCHNIRENKO
Department of Mathematics, Penn State University, University Park, PA 16802, USA
E-mail: agk@math.psu.edu

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

Mini-languages are a visually intuitive, simple and powerful way to introduce students to programming. They are a good foundation for general computer science instruction, provide insight into programming for the general population, and teach algorithmic thinking. The goal of the paper is to provide an extensive review of the mini-language approach to teaching programming. For different audiences and in different countries, the authors have extensive experience in design and application of mini-languages. We outline the problems that motivate the application of this approach, present a brief history, review several existing mini-languages, and provide discussion of lessons learned. In particular, we discuss how to choose a mini-language for a particular group of students and list some requirements for a successful application of a mini-language. We conclude with a discussion of possible future directions of the mini-language approach development.

KEYWORDS: secondary education; higher education; informatics; languages; logo; programming.

1360–2357

INTRODUCTION

There have been many efforts to develop special languages for supporting the initial steps in programming education as an 'easy start' for novices (Mendelson *et al.*, 1990). Genuine insight was given by the 'turtle graphics' of Logo (Papert, 1980). The success of Logo in general and turtle graphics in particular stimulated the development of the mini-language approach to teaching the principles of programming.

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 behaviour of the executive on a screen. The actor can perform a small set of commands and answer several value-returning queries. Usually the student controls the actor, first by giving isolated commands and then by writing small programs in a special miniature programming language. This miniature language includes commands and queries of the actor and several control structures. Most mini-languages include all basic control structures (conditional execution, looping, recursion, etc.) and a mechanism for creating new instructions (or other kind of subprograms). In this paper we use the term mini-language to name a combination of an actor with a language to control it.

There are multiple worthwhile objectives for teaching mini-languages. Mini-languages can provide a solid foundation for learning a general purpose language, such as Pascal, C or LISP. Mini-languages also 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. It is the acquisition of algorithmic thinking as an explicit, familiar and powerful tool.

This paper provides a review of the mini-language approach. We outline the problems that motivate the application of this approach, present a brief history of the approach and a review of existing mini-languages and provide an extensive discussion of lessons learned. In particular, we discuss how to choose a mini-language for a particular group of students and list some requirements to a successful mini-language. All authors have a deep, positive experience in the design and application of mini-languages for different audiences in different countries. Our suggestions are based on experiences of teaching hundreds and thousands of real students. We conclude with a discussion of possible future directions of the approach development.

DISCUSSION

Drawbacks of the standard approach

Nowadays, most of the students who start to learn the principles of programming are taught by the old classic approach. This approach is based on using a general-purpose programming language, such as Pascal, Modula-2, LISP or C, a professional programming environment for the chosen language, and a set of problems from the area of number processing and symbol processing. We believe that the effectiveness of the classic approach is quite low in general, and the younger the students the worse the classic approach works. In particular, using general-purpose languages creates the following three kinds of obstacles for novice programmers:

(1) General-purpose languages are too big and too idiosyncratic. The conceptual basis of the language together with the main principles of programming combine to form a large amount of material. This volume alone makes it difficult to understand the material properly, thereby failing to form a strong cognitive infrastructure. Instead of emphasing fundamental principles, the languages evoke secondary notions that reflect the subtleties of the given language and its implementation.

(2) General-purpose languages provide little leverage for understanding their basic actions and control structures. The languages are not visual and their basic functions are carried out behind an opaque barrier. Professional programming environments do not usually provide any visualization capabilities. With the process of program execution hidden, the student develops an input–output orientated understanding. In this way the absence of visual feedback hampers mastery of language semantics.

(3) Since general-purpose programming languages are orientated on number and symbol processing, the first possible problems used in teaching the language are far from the students' everyday experiences and are not attractive for them. Developing applications that are both informative and interesting requires learning a considerable language subset and writing quite big programs, and this introduces another distraction–the need to master the programming environment. As a result, the first difficult steps in learning to program are usually neither well motivated nor supported by work on the computer.

The mini-language alternative

There are two reasons why mini-languages are a good alternative. One is that they provide a good foundation for general computer science instruction. The other is that they open a door onto new educational opportunities. We believe that principles of computer programming is a topic that should be learned starting at the early stages of school. It provides the basis for logical and abstract reasoning that is fundamental to the learning process.

The first advantage of mini-languages is, as the name suggests, that they are small. Mini-languages have a small syntax and simple semantics. A student, even

a very young one, can thus come to grips with the entire mini-language and employ it with interesting results. In the words of Papert (1980), students can 'expropriate' the mini-language, making it their property. The time required to master a mini-language itself is small, so the students can spend most of their efforts on more important issues: mastering algorithm development and program design in a university setting or understanding the principles of programming for a more general audience.

A second important advantage is that mini-languages are built on metaphors that are intrinsically engaging and visually appealing. It is possible to create rich sets of problems that both cover the fundamental ideas and dovetail with the students' life experiences. This causes students to want to expropriate a mini-language.

The operations performed by the actor are always visible, revealing the semantics of language construction. Visual queues enable the novice to understand semantics of introduced constructs, elucidate principles of program structure and execution, protecting them from developing misconceptions. Visibility provides a feedback for exploratory learning and problem solving. Visual metaphors make it easier to develop interesting problems for important concepts. Problems that achieve visible and meaningful results aid concept mastery by reinforcing them with problem solving activities.

Another important advantage is that the designer of a mini-language is not restricted to the syntax and semantics of any 'big' language, which may not be suitable for novices. As a language designed especially for an educational purpose and for a well-defined audience, a mini-language can take advantage of the narrow definition of the class of its users by using students' native languages for keywords and providing only data types and control structures essential for these students. In the same way, the designer of a programming environment for a mini-language can take advantage of the small sizes and known audiences of the languages and design a highly attractive and effective environment.

A brief history of mini-languages

The development of the mini-language approach was seriously influenced by the turtle graphics of Logo (Papert, 1980). In some sense, the turtle-graphics set can be considered the first example of a mini-language. 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 strongly contributes to the overall incredible success of Logo. Logo provides genuine insight for further development of mini-languages; however, Logo turtle graphics is not the best example of a mini-language. Note that unlike most of the actors of mini-languages described below, the turtle of Logo is 'blind', it cannot check its microworld. The 'turtle subset' also does not support such classic control structures as Pascal-like *if* and *while*. To introduce these constructs the language needs to contain some predicates that provide feedback when controlling the actor.

The first and still the most popular real mini-language 'Karel the Robot' was designed by Pattis as a 'gentle introduction' to Pascal for university students taking their introductory programming course. Karel the Robot was completely described in a book with the same name (Pattis, 1981) written by Pattis, who also designed the first programming environment for Karel. 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 intersecting streets and avenues, walls and beepers (Fig. 1). Karel can also carry some beepers in his 'bag'. The main actions of Karel are move, turn left, pick beeper and put beeper. 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.

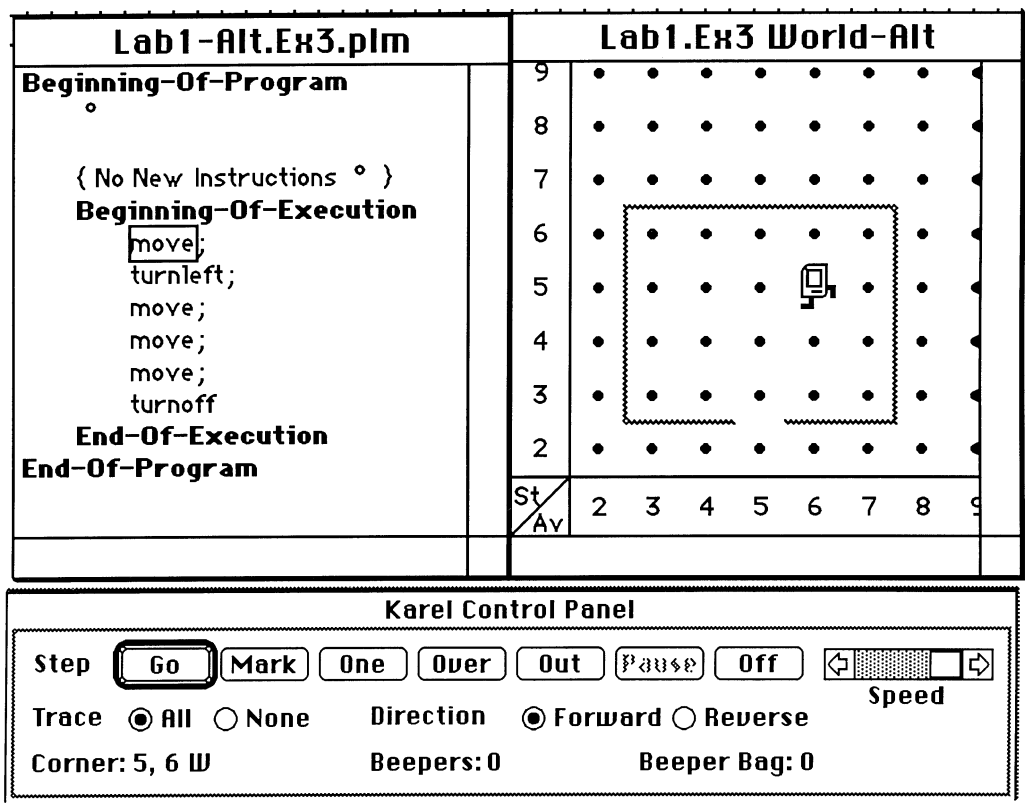

Figure 1. Karel the Robot

Karel the Robot has been very important both as a mini-language and in stimulating other work. The following are several mini-languages that were directly inspired by Karel and use some of its features: Martino (Olimpo *et al.*, 1985) and Marta (Calabrese, 1989) in Italy, Darel (Kay and Tyler, 1993) in Australia and Karel-3D (Hvorecky, 1992) in Slovakia. The original Karel the Robot is now in use in many universities and colleges in the USA. A new edition of the Karel the Robot book has been issued recently (Pattis *et al.*, 1995).

Karel-3D

Karel-3D (Hvorecky, 1992) is an extension of the idea of Karel the Robot. Karel-3D (Fig. 2) expands the original Karel into several directions to make it more suitable for special audiences, such as pre-school students or secondary school pupils.

- The robot can move in the three-dimensional space displayed in a separate window. Other windows are used for typing programs, for menus and for context-sensitive help.
- The robot can lay a brick, stand on it and (as a result of repeated actions) move up and down. Thus, the movement of the robot is also 'three-dimensional'.

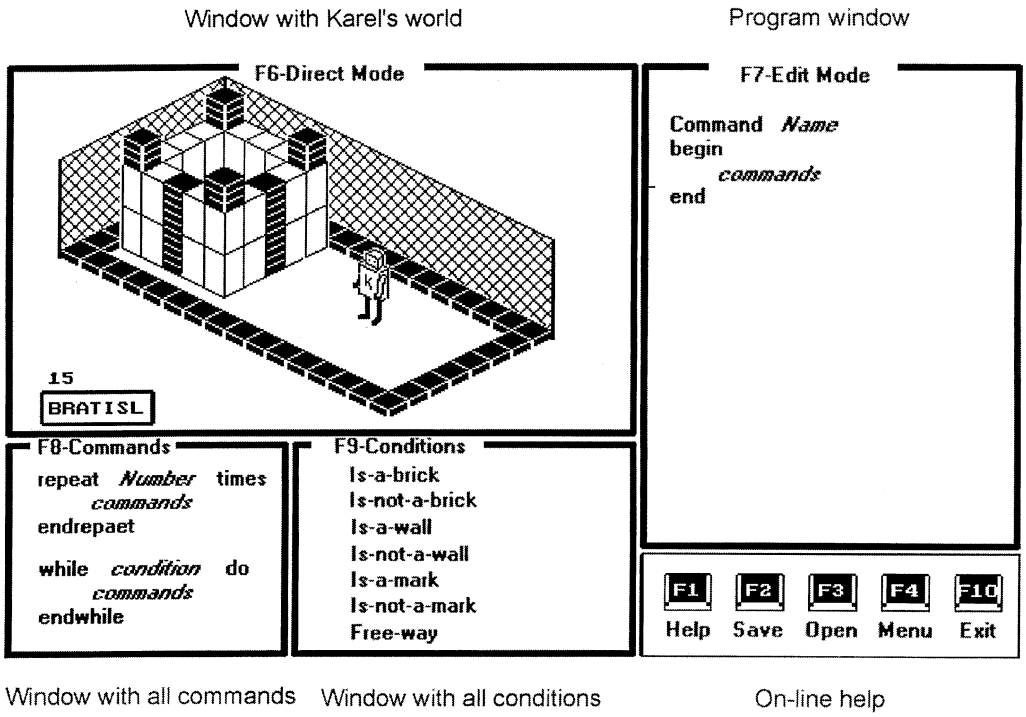

Figure 2. Karel-3D

- The robot obeys either pressing keys (in a steering or navigation mode) or commands (in a programming mode). A family of languages is created in this way. Small children use steering, older ones start with programming.

Many Czech and Slovak schools use Karel-3D as their first (and sometimes the only) programming language. Karel-3D is supported by a textbook (Gasparovicova and Hvorecky, 1991).

Marta

Marta (Calabrese, 1989) is a screen robot based on Logo and is similar to the robot Karel. It can be used as a soft introduction to programming from pre-school age to adults. Being written in Logo, Marta offers the following advantages over Karel:

- Marta can be driven in navigation mode with single keystrokes, which makes it suitable even for pre-school children.
- Marta can build her world, as she can put or remove walls.
- Marta can be driven in the dark where the obstacles and beepers are not visible, although they are still there.
- New commands and operations can easily be defined by writing Logo procedures.
- Several levels of programming are available: one-line programs, multiple-line programs and Logo procedures.

The only limitations with respect to Karel and that Marta's world is bounded to a 7×15 grid and that she cannot put down more than four beepers at an intersection.

Josef the robot

Some mini-languages were designed independently of Karel, but with some influence of Logo. One example is Josef the Robot (Tomek, 1982; 1983). Josef was designed at the same time as Karel and its philosophy is a combination of Logo and the ideas used in Karel, namely the idea of a rich microworld of Logo and the idea of preparing the user for programming constructs of conventional languages as in Karel. Josef provides a relatively rich microworld–a robot living on the screen and capable of graphics and other operations, particularly in a simulated real world represented by a map. It uses constructs similar to conventional languages and advanced concepts such as interrupts. One of the features intended to avoid syntactic difficulties that obscure the essential concepts of problem solving by programming is that variables and procedure arguments are untyped.

Other languages

Several mini-languages were designed and used widely in the USSR where the book by Pattis was not available, e.g. Wayfarer, Turingal and Tortoise. For more information about Russian mini-languages see Brusilovsky (1990).

Wayfarer. Wayfarer was suggested in 1980 to support a new course on programming for the students of the Mechanics and Mathematics Department of the Moscow

State University (Kouchnirenko and Kouchnirenko, 1988). The goal was to challenge the students, from the first lesson, with a set of interesting problems. This set of problems concentrated on the task of controlling a simple actor 'Wayfarer' moving on a checked field with walls. The solutions were written in a simple mini-language called MINI containing *if–then–else* and *while–do* statements, procedures without parameters, commands and queries to an actor. Several years of experience proved that four–six weeks of work with Wayfarer gave the student a solid ground for the study of general notions of programming.

Turingal. The mini-language Turingal (Brusilovsky, 1991) was designed in 1983 for computer science students of the Moscow State University. This language provides control of the well-known basis of algorithmic theory–the Turing machine, which works with a tape of symbols. The elementary operations of the mini-language are simple and visual; the movement left and right along the tape and typing the symbols on the tape. To control the machine, Turingal offers a set of control stuctures (conditional statement, loops, case) and subroutines, with syntax and semantics similar to the structures of PASCAL (Turingal =TURING machine + pascAL). One of the main reasons for presenting the mini-language is to bring the student to the mastery of the semantics of these well-known stuctures before starting to learn Pascal.

Tortoise. The Tortoise mini-language was designed to support a part of the computer literacy course for 14–15 year-old students in Russian schools (Brusilovsky, 1994). It is similar to the Turingal, but adds some features that make it attractive for younger students. For example, the tape of symbols was substituted by a two-dimensional field of symbols.

Lessons learned

After more than ten years from the appearance of mini-languages the time is now suitable to summarize some lessons learned. The authors of this paper have had positive experiences using different mini-languages for different audiences in different countries. Comparing and analysing our experiences provides some generalizations and the following sections discuss several aspects of mini-language development and application.

Intended audience

The mini-languages used by the authors of this paper are designed for different age groups, ranging from elementary school students (Marta and Karel-3D), through secondary school ages (Tortoise and advanced version of Karel-3D) to college freshmen (Wayfarer, Turingal and Karel). All authors reported success in engaging their audiences and in teaching problem solving and fundamental principles. In this sense the mini-language approach is a kind of 'silver bullet' that gives good results in teaching the principles of programming for different audiences.

At the same time there is no 'right' or perfect mini-language. There are different mini-languages and they suit different needs, because the type of actor and the

set of mini-language control structures depend on the age, background, interests and learning goals of the student. For example, when one of the authors tried to redesign the Turingal mini-language environment used previously in the university for school students who were three years younger, he had to redesign the executive and the microworld as well. The Turing machine executive with a one-dimensional tape microworld that was used in the Turingal mini-language does not seem to be interesting for the majority of 14 year-old students who never had a course on the theory of algorithms. The new mini-language called Tortoise adds two-dimensional world, colour and sound to the classic Turing machine to make the microworld attractive for these students.

Another example shows that an actor can be extended to accommodate the extended learning goals. Several years ago the paradigm of actor–executive, supported originally by the Wayfarer, was applied in a secondary school course 'Foundations of Computer Science and Technology' (Kouchnirenko *et al.*, 1993). The main idea was to use executives to motivate the introduction of variables, arrays and simple calculation procedures. To support it the Wayfarer was replaced by a more powerful executive Robot in a richer microworld. For example, Robot can measure the 'temperature' and 'radiation level' at the current position of the field.

The audience can differ in the overall goal of using the mini-language. There are two main reasons to apply the mini-language approach. The first reason is to support an introductory computer science course in a university setting. Here a mini-language can be used to provide a 'gentle introduction' into one of the general purpose programming languages (Pattis, 1981) and to support mastering such general skills as algorithm development, program design and program debugging (Comar and Pintelas, 1989; Ferguson, 1978; Kay and Tyler, 1993). The second reason is to support the language-independent learning of the principles of programming and algorithmic reasoning (Brusilovsky, 1991; Gasparovicova and Hvorecky, 1991; Olimpo *et al.*, 1985). In the early stages of mini-language application, the first goal dominated. At present, the second goal has become more important as the principles of computer programming is a topic that should be learned widely (and probably at the early stages of school), because it provides the basis for logical and abstract reasoning that is important in any learning process and everyday life.

Note that the application of a mini-language is never the goal itself, but a method of mastering a set of notions and skills. If this set contains not only programming concepts, but also some concepts from another domain, a mini-language might be useful to learn this domain (as Logo is used to learn geometry). There is for instance the mini-language SOLO aimed at the student of psychology (Eisenstadt, 1983).

Important features of a mini-language

Some features of mini-languages are important for success. The mini-language should be *simple* in both its syntax and semantics. Simplicity is essential–see the

classic paper by du Boulay *et al.* (1981) for a discussion of simplicity. The mini-language should be *naturally visible* most operations performed by the actor should make visible changes in the microworld represented on the screen. The mini-language should be *attractive* and meaningful for the intended category of students. A good example for it is the Japanese Algo-Arena (Kato and Ide, 1993) mini-language system applying the metaphor of Sumo wrestling, which is meaningful and appealing for young Japanese students, or another system (Comar and Pintelas, 1989) applying the metaphor of shopping in a supermarket. The above requirements stem from the very idea of a mini-language. Other desired features are not so obvious and stem from experience. The mini-language should be *conversational* like Basic or Logo. It means that any mini-language command can be executed in both navigation mode (single command execution) and programming mode (complete program execution). Our experience shows that immediate execution and navigation modes are important for a young audience. The original Karel did not support a navigation mode, but both extensions of Karel for primary and secondary school students, Karel-3D and Marta, add this feature.

A mini-language should be a *modular* language. It should contain a mechanism for creating abstract instructions (procedures). All the procedures should be independent units. Such a procedure can be considered as a new command of the actor, which can be used in both navigation and programming mode.

The procedures designed for a particular problem can be later re-used for solving subsequent problems. Logo, Karel, Josef and Wayfarer provide good examples of modular languages.

The role of programming environment

An important lesson is that selecting a mini-language approach is not enough for successful teaching of the principles of programming. The learning should be supported by a good programming environment. Such an environment should keep both the microworld 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. This makes the connection between a mini-language command or construct and its effect on the microworld obvious. The interpreter should also provide visualization 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.

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 strucutre 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 to give them meaningful names. A related useful component of the environment is a graphical

program design tool. Such a tool makes the program stucture visible and understandable for the student.

A good example of a mini-language programming environment with all the desired features is provided by Karel Genie (Miller *et al.*, 1994) a novice programming environment for the Karel mini-language.

To support the novice programmer, the Karel Genie provides a set of specially designed tools (Fig. 3), which include a structure editor, a program decomposition view for both looking at and editing programs, and a run-time 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 Karel Genie stucture editor provides maximum support to the novice programmer, because program construction can be conducted entirely through menu interaction. As novices develop expertise they begin to exploit Karel Genie's text edit features. The Karel Genie graphical program design tool lets students decompose a problem into simpler sub-problems by dragging the mouse from the program root.

Karel Genie is highly visual and successfully uses multiple representations. This is clearly illustrated in Karel Genie's run-time system. The 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. Karel programs can be single stepped and run either forward or backward. The run-time environment also has a visual call stack. This makes procedure invocation and return clear.

The Karel Genie has been in use in secondary schools and universities throughout the USA for nearly ten years. In addition to Carnegie Mellon University 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.

Thus a good mini-language is important, but not enough for success. A programming environment is very important too. In particular, just applying the Logo language with its famous Turtle graphics for teaching programming is often not enough. In recent papers, Logo authorities argue the importance of extended programming environments for Logo with variable visualization, stepwise execution and even a program design tool (da Rocha, 1993; De Corte, 1993). We should note that most of the recent successful mini-language environments (Karel-3D, Tortoise, KuMir) include both a structure editor and a run-time system with extended visualization. For example, in Karel-3D the text of any program can be created as a combination of commands and tests taken from

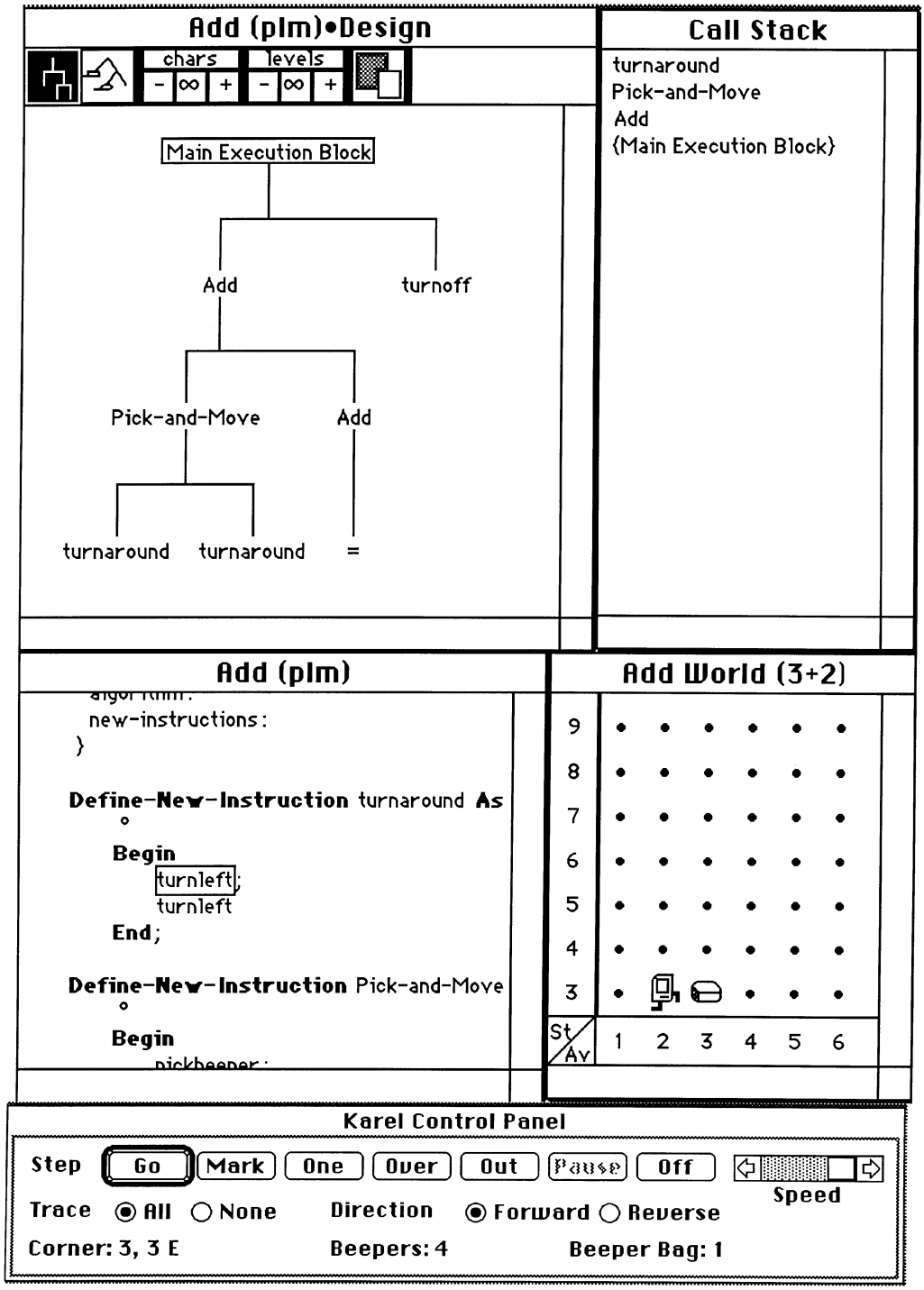

Figure 3. Karel Genie: editor and design view used and run time with tracing call stack

menus. Problems that children have with typing are avoided this way. The programming mode contains various debugging and visualization tools. The execution of a program can be slowed down or speeded up, if necessary. The executed commands can be highlighted.

The role of the problem set

A good mini-language should be complemented with a good set of attractive and meaningful problems for students to solve. Problem solving is the most effective way of mastering the mini-language and, consequently, supporting important concepts. The set of problems must contain problems of various complexity (be ready also to challenge the top 10% of your students with attractive but difficult problems) and cover all important concepts. These problems must be interesting for the student both from the point of the goal to be achieved, and the process of solution development. Sometimes these problems look like a puzzle rather than a 'serious' programming task and the problem-solving activity becomes a kind of a game. We recommend keeping a set of attractive problems in mind when designing a mini-language. A mini-language that looks very attractive but for which an insufficient number of problems are available loses its attractiveness very quickly. Often the choice of an actor is driven by the goal of making the set of problems the student is expected to solve attractive and meaningful. Compare, for instance, two problems:

(1) For a given matrix, A, find some pair, i, j, such that $Ai, j = 0$;
(2) Move Robot to some position where the radiation level is equal to zero.

A mini-language augmented with a good set of problems can support problem-driven learning: when the student is presented with a new meaningful problem, a new language construct is then introduced that helps to solve the problem. This approach has been used fruitfully in textbooks (Kouchnirenko and Kouchnirenko, 1988; Kouchnirenko *et al*., 1993).

The above considerations concern traditional mini-languages that can be described as 'one microworld–many problems'. There is, however, an opposite approach 'one microworld–one problem', where the microworld is orientated on solving one, though difficult, problem, like finding an exit from a labyrinth. Two good examples of this are the TRAPS system (Witschital *et al*., 1989) where the task is to move an actor through a playing board filled with various obstacles, and Algo-Arena (Kato and Ide, 1993) where the task is to control a Sumo wrestler. Here the student learns to program by solving more and more complex subproblems of the problem, or solving the problem for more and more complex microworld configurations (as in TRAPS), or designing more and more complex and better program solutions (as in Algo-Arena). A limited set of problems in these systems is compensated by an attractive and meaningful microworld. We think that systems of this kind can be suitable to support a small course on the principles of programming for a young audience.

Using several microworlds

Several mini-languages with different actors can be used sequentially or simultaneously in an introductory programming/design course for computer science uni-

versity students. The course can comprise several regular mini-languages (Kouchnirenko and Kouchnirenko, 1988) or several one-problem mini-languages as suggested in (Comar and Pintelas, 1989; Ferguson, 1987). Different microworlds can be used to stress different aspects of the subject (for example floating point numbers) or to teach different parts. Here the problem is that understanding a new mini-language requires a considerable amount of time and mental effort. The application of any additional mini-language should be justified. Two lessons learned from the experience are:

(1) having started to use a mini-language the student expected to use it for a considerable amount of time, and
(2) the miniature languages used to control different actors should be similar (at least they should have the same control stuctures).

Mini-languages and sub-languages

An approach very similar to the mini-language approach is the sub-language approach. The idea of the sub-language approach is to design a special starting subset of the full language containing several 'easy to visualize' operations. Such a subset can support the first steps of the learning programming and helps later in introducing more complex programming concepts. 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. Usually the sub-language approach applies the same idea of an actor in a microworld, and the starting subset is really a set of commands and queries performed by the actor. Moreover, the same kind of actor can be controlled by a mini-language, or by a sub-language.

The sub-language approach differs from the mini-language approach 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. Good experience in designing sub-languages has been accumulated in Russia. The programming environment KuMir, which supports the Russian secondary school course on 'Foundations of Computer Science and Technology' (Kouchnirenko *et al.*, 1993) applies several interesting actors (each one can be considered as a sub-language) within the same educational programming language (E-language).

KuMir is an integrated educational environment, combining a text editor with a zero-response-time incremental compiler and a simple debugger. The basic philosophy that governs KuMir is that the language has a compact core that can be dynamically extended by loading one or more separately prepared modules, i.e. executives providing new microworlds and actors, or even new numerical packages and abstract types of data. This approach allows easy customization of the system. The school teacher can start with one of the traditional actors, like Robot, then add more sophisticated options (Kouchnirenko *et al.*, 1993). At the university level, special packages like complex and rational numbers or plane

geometry can be added. During the last three years KuMir has been used at the Department of Mathematics of Moscow State University in support of undergraduate courses of mathematics.

We think that the sub-language approach is often 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 is 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. Using sub-languages can be recommended for university introductory programming courses. Here the sub-language approach can be successfully combined with several actors for different purposes (as in KuMir).

FURTHER SCOPE

Supporting other paradigms

This paper discusses only the approaches related to teaching procedural paradigm languages. The mini-language approach, however, can be applied successfully with other paradigms. An example of applying the mini-language approach for teaching parallel programming paradigms is the Robot Brothers project (Olimpo, 1988). Three good examples of mini-languages designed to support object orientated paradigm are Playground (Fenton and Beck, 1989), Gravitas, (Sellman, 1992), and KidSim (Smith *et al.*, 1994). The latter mini-language can also serve as an example of learning some physics concepts (laws of gravitation) along with the principles of programming, and as an example of how to design an attractive mini-language for a particular category of students. Several authors consider object-orientated mini-languages as the most natural and suitable way to teach introductory programming and 'control technology' to younger students (Kato and Ide, 1993; Resnick, 1990; Whalley, 1992).

Real world mini-languages

When teaching the principles of programming for a very young audience of seven–nine years old, special attention should be paid to the attractiveness of an actor. A good way to make an actor attractive for young students is to use a real world actor–some real computer-controlled device as the first 'real turtle' of Logo. A good step towards real world actors is done by robotic toys such as Lego–Logo. Robotic toys can provide good motivation for learning for a very young audience. The authors think this is an area that should receive more attention. Special work has to be done to make real world actors not only attractive, but also really useful in learning general concepts. In particular, robotic toys can be naturally applied to support learning the object-orientated paradigm. Some encouraging results in this area were reported recently (Whalley, 1992). Another example is the use of robotic toys by a concurrent extension of Logo called MultiLogo to teach the principles of concurrent programming (Resnick, 1990). Interesting results in making 'real world programs',

i.e. constructing a program as a real world artefact assembled from modules, are reported in (Kato and Suzuki, 1993). Real world actors and real world programs are not only more attractive, but also support collaborative problem solving activity.

Advanced mini-language environments

A promising way of further development of mini-language programming environments is to extend them with an intelligent tutoring component and a hypermedia component. Intelligent tutors have shown impressive results in mathematics and programming. By connecting intelligent tutoring to mini-languages it is conceivable that learning can be improved by an order of magnitude. Intelligent computer tutors can decrease the amount of non-creative work of the human tutor and reserve the teachers time to work with students who have special needs. Intelligent tutoring components provide the required amount of guidance: suggest the next concept to learn or the problem to solve according to the student's current knowledge, assist the student in the problem solving process, diagnose the student's solution. First examples of creating intelligent tutors for mini-languages were reported in (Brusilovsky, 1992; Dion and Lelouche, 1988; Witschital *et al.*, 1989). The hypermedia component extends the space of exploratory learning, providing student-driven access to conceptual knowledge and examples. Recently, a hypertext component was designed for the KuMir programming environment and a hypertext electronic version of the textbook (Kouchnirenko *et al.*, 1993) is now under development. Some ideas about integration of a programming environment, a hypermedia component and intelligent tutoring systems into a single system can be found in Brusilovsky (1993).

CONCLUSIONS

All the authors of the paper, i.e. five university teachers, representing four different nations, using similar, but distinct mini-languages, report positive results from using mini-languages. However, all our findings and recommendations are results of experience rather than rigorous empirical evaluation. It is time to investigate such claims empirically and report the findings. There are very few papers at present that report any results of classroom studies or special experiments with mini-languages. For example, there were special empirical evaluations to study (a) how skills learned in Karel were transferred to other problem solving settings (Scheftic and Goldenson, 1986), and (b) how the visual nature of the Tortoise mini-language helped students in debugging their programs (Brusilovsky, 1994). We believe it would be of great value to the larger educational community to study the mini-languages in a rigorous and repeatable way.

Promoting the mini-language experience

Much research and development work has been done in the field of teaching introductory programming and a number of powerful environments have been created. Teaching programming to beginners was often used as a testbed to apply new creative ideas about how programming should be taught. It is time now to transfer these

ideas and experiences to 'real life' in the following two senses. First, these excellent novice programming environments should leave laboratories and universities where they have been developed and find their way to schools, universities and homes. Here the role of software companies is important. Good examples are the Karel Genie and KuMir, which are both used widely outside the place where they were developed. But even here additional efforts are needed to turn these environments into programming products that can be bought in a store. Second, current 'big' programming environments, such as Borland Pascal, should learn some lessons from the best novice environment experience. Although current programming environments provide debuggers operating at the source code level and capable of step-by-step execution with display of current data values, they still lack many useful features that exist in the environments we have described above. Good examples of programming environments for professional languages that have most of the required features discussed above are Pascal Genie (Miller *et al*., 1994) from Carnegie Mellon University and PascalMir and FortranMir from Moscow State University.

REFERENCES

Brusilovsky, P. (1990) Languages for teaching the principles of programming (in Russian). *Informatika i Obrasovanije (Informatics and Education)* **5** (2), 3–9.

Brusilovsky, P. L. (1991) Turingal – the language for teaching the principles of programming. In *Proceedings of the Third European Logo Conference*, E. Calabrese (ed.), Parma, 27–30 August 1991, pp. 423–32.

Brusilovsky, P. L. (1992) Intelligent tutor, environment and manual for introductory programming. *Educational and Training Technology International* **29** (1) 26–34.

Brusilovsky, P. (1994) Program visualisation as a debugging tool for novices. In *Proceedings of INTERCHI'93 (Adjunct proceedings)* Amsterdam, 24–9 April 1993, pp. 29–30.

Brusilovsky, P. (1993) Explanatory visualization in an educational programming environment: connecting examples with general knowledge. In *Human-Computer Interaction, Proceedings of Fourth International Conference on Human-Computer Interaction, EWHCI'94, Lecture Notes in Computer Science*. B. Blumenthal, J. Garnastaev and C. Unger (eds), Berlin, Springer-Verlag, **876**, 202–12.

Calabrese, E. (1989) Marta – the 'intelligent turtle'. In *Proceedings of the Second European Logo Conference*, G. Schuyten and M. Valcke (eds), Ghent, Belgium, 30 August–1 September 1989, pp. 111–27.

Comar, C. and Pintelas, P. (1989) An environment for teaching 'program design' by exercises. *Microprocessing and Microprogramming* **28**, 259–64.

da Rocha, H. V. (1993) The use of computational representation in the teaching and learning of programming concepts. In *Proceedings of the Fourth European Logo Conference*, Athens, Greece, 28–31 August 1993, P. Georgiadis, G. Gyftodimos, Y. Kotsanis and C. Kynigos (eds), pp. 153–9.

De Corte, E. (1993) Towards embedding enriched Logo-based learning environments in the school curriculum: retrospect and prospect. In *Proceedings of the Fourth Logo Conference*, Athens, Greece, 28–31 August 1993, P. Georgiadis, G. Gyftodimos, Y. Kotsanis and C. Kynigos (eds), pp. 335–49.

Dion, P. and Lelouche, R. (1988) Architecture of an intelligent system to teach introductory programming. In *Proceedings of ITS'88, International Conference on Intelligent Tutoring Systems*, C. Frasson (ed.), Montreal, 1–3 June 1988, pp. 387–94.

du Boulay, J. B. H., O'Shea, T. and Monk, J. (1981) The black box inside the glass box.

Presenting computing concepts to novices. *International Journal on the Man–Machine Studies* **14** (3), 237–49.

Eisenstadt, M. (1983) A user-friendly software environment for the novice programmer. *Communications of the Association for Computing Machinery* **20** (12), 1058–64.

Fenton, J. and Beck, K. (1989) Playground: An object-oriented simulation system with agent rules for children of all ages. In *Proceedings of OOPSLA'89 Object-oriented Programming: Systems, Languages, Applications*, New Orleans, LA, 2–6 October 1989, N. Meyrowitz (ed), pp. 123–37.

Ferguson, D. L. (1987) The design of algorithms. *Machine Mediated Learning* **2** (1, 2), 67–82.

Gasparovicova, L. and Hvorecky, J. (1991) *Kamarati Robota Karla (Friends of Karel the Robot*, in Slovak, Bratislava, Mlade leta.

Hvorecky, J. (1992) Karel the Robot for PC. In *Proceedings of the East–West Conference on Emerging Computer Technologies in Education*, P. Brusilovsky and V. Stefanuk (eds), Moscow, 6–9 April 1992, pp. 157–60.

Kato, H. and Ide, A. (1993) Algo Arena: A system for learning programming through social interactions. In *Proceedings of the Fourth European Logo Conference*, Athens, Greece, 28–31 August 1993, P. Georgiadis, G. Gyftodimos, Y. Kotsanis and C. Kynigos (eds), pp. 111–22.

Kato, H. and Suzuki, H. (1993) AlgoBlock: A tangible programming language. In *Proceedings of the Fourth European Logo Conference*, Athens, Greece, 28–31 August 1993, P. Georgiadis, G. Gyftodimos, Y. Kotsanis and C. Kynigos (eds), pp. 123–33.

Kay, J. and Tyler, P. (1993) A microworld for developing learning design strategies. *Computer Science Education* **3** (1), 111–22.

Kouchnirenko, A. G., Lebedev, G. V. and Svoren, R. A. (1993) *Foundation of Computer Science and Technology* (in Russian). Moscow, Prosvecshenie.

Kouchnirenko, A. and Kouchnirenko, G. L. (1988) *Programming for Mathematicians* (in Russian). Moscow, Nauka.

Mendelson, P., Green, T. R. G. and Brna, P. (1990) Programming languages in education: the search for an easy start. In J.-M. Hoc, T. R. G. Green, D. Gilmore and R. Samway (eds) *Psychology of Programming*, pp. 175–200, London; Academic Press.

Miller, P., Pane, J., Meter, G. and Vorthmann, S. R. (1994) Evolution of novice programming environments: The structure editors of Carnegie Mellon University. *Interactive Learning Environments* **4** (2), 140–58.

Olimpo, G. (1988) The Robot Brothers: An environment for learning parallel programming oriented to computer education. *Computers and Education* **12** (1), 113–18.

Olimpo, G., Persico, D., Sarti, L. and Tavella, M. (1985) An experiment in introducing the basic concepts of informatics. In *Proceedings of the Fourth World Conference on Computers in Education, WCCE'85*. K. Dunkan and D. Harris (eds), Amsterdam, pp. 31–8.

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

Pattis, R. E. (1981) *Karel – The Robot, A Gentle Introduction to the Art of Programming*. London, Wiley.

Pattis, R. E., Roberts, J. and Stehlik, M. (1995) *Karel – The Robot, A Gentle Introduction to the Art of Programming*. 2nd edn. New York, Wiley.

Resnick, M. (1990) MultiLogo: a study of children and concurrent programming. *Interactive Learning Environments* **1** (3), 153–70.

Scheftic, C. and Goldenson, D. (1986) Teaching programming methods and problem solving: the role of programming environments based on structure editors. In *Proceedings of the National Educational Computing Conference*, pp. 231–6.

Sellman, R. (1992) Gravitas: An object-oriented discovery learning environment for Newtonian gravitation. In *Proceedings of the East–West International Conference on*

Human-Computer Interaction. J. Gornostaev (ed.), St Petersburg, 4–8 August 1992, pp. 31–41.

Smith, D. C., Cypher, A. and Spohrer, J. (1994) KidSim: programming agents without a programming language. *Communications of the Association for Computing Machinery* **37**(7), 54–67.

Tomek, I. (1982) Josef, the robot. *Computers and Education* **6** (3), 287–93.

Tomek, I. (1983) *The First Book of Josef*. Englewood Cliffs, Prentice-Hall.

Whalley, P. (1992) An alternative metaphor for teaching control technology. In *Proceedings of the East–West Conference on Emerging Computer Technologies in Education*. P. Brusilovsky and V. Stefanuk (eds), Moscow, 6–9 April 1992, pp. 328–31.

Witschital, P., Stiepe, G. and Kuehme, T. (1989) Experiencing programming language constructs with TRAPS. In *Computer Assisted Learning, proceedings of the Second International Conference, ICCAL'89*, Lecture notes in Computer Science, Vol. 360, H. Maurer (ed), pp. 591–602. Berlin: Springer-Verlag.