Appeared in the Proceedings of the 2001 IEEE Symposia on Human-Centric Computing Languages and Environments, pages 135–141. Stresa, Italy, September 2001.

Kara, finite state machines, and the case for programming as part of general education

W. Hartmann, J. Nievergelt, R. Reichert
Department of Computer Science, ETH Zürich, Switzerland
{hartmann, nievergelt, reichert}@inf.ethz.ch

Abstract

As a major evolutionary step in computer technology, users have come to rely on ready-made application software, rather than writing their own programs. If computer users no longer program, does it follow that the art of programming should only be taught to computing professionals? We argue the case for programming as a component of general education. Not because of any direct utilitarian benefit, but in order to gain a personal experience as to what it means, and what it takes, to specify processes that evolve over time. An analogy to mathematics education shows that schools teach the concept of „proof", although in daily life people use mathematical formulas without knowledge of their proof.

Programming practiced as an educational exercise, free from utilitarian constraints, is best learned in a toy environment, designed to illustrate selected concepts in the simplest possible setting. As an example, we present the programming system Kara based on the concept of finite state machines.

1. General education in an ever-changing world

The rapid evolution of the information and communication technology that permeates modern society affects our expectations of education at all levels. Top priority is assigned to content that is practically oriented, relevant to today's job world, immediately useful. This emphasis on practicality is not new. Folklore identifies „the 3 R's: reading, 'riting, 'rithmetic" as the main task of elementary school. These skills, limited to the privileged few throughout most of recorded history, became a necessary part of general education as a consequence of the industrial age.

But in addition to tangible, sellable skills, education has always included topics about which most adults later say „I've never used it since I left school". This category includes the majority of the topics any one student endured, namely most of those outside the profession chosen later. Is it a waste of time to study disciplines of no direct use? Perhaps yes, perhaps not, it all depends on the topic and on whom you ask. But generally we accept the necessity of exposure to topics not directly related to any material benefit. Young children, champions at learning, are interested in just about anything new, without any thought of whether they might ever use it. The value of learning cannot be measured in dollars and cents – and if we try, we end up short-changing ourselves.

The fashionable clamor for „relevant education" is all too easily misinterpreted as training focused mainly on skills of immediate use. This emphasis leads to mastery of low-level skills, such as „how to do it", rather than to understanding concepts and principles, of „why and how it works". In a rapidly changing field such as information and communication technology, mastery of today's release may not be of much value when a new system is installed a few years later. The precious school years must be dedicated primarily to knowledge of long-lasting value, to what is known as general education.

2. „The 4 R's: reading, 'riting, 'rithmetic, 'rogramming"

As a major technical achievement of the past two decades, computer users no longer need to write programs to solve their problems. For just about every conceivable application, ready-made software packages provide tools much more powerful than what almost any user could write. If computer users no longer program, does it follow that the art of programming should only be taught to computing professionals? The plausibility of this argument has led many high schools and colleges to replace the introductory programming courses widely taught in the 70s and 80s by skill courses on how to use application packages. Text processing, spreadsheet calculations, web surfing are certainly essential job skills today; they deserve a brief introduction, whereafter any user can reach any desired skill level on his own. But the lion's share of the teacher's and the students' attention should be devoted to fundamental concepts of timeless validity. What are some of these concepts?

Modern society relegates an ever-growing number of every-day tasks to machines. These machines act as controllers that initiate actions based on their current state and on received inputs. The number of possible

behaviors, of sequences of actions triggered by different environmental conditions, is usually so huge as to be impossible to enumerate. Yet, we aim to be sure that each and every possible behavior, of which only a tiny fraction will ever be played out, is „correct“ in some precise sense. The way to do that is to write a specification that captures the practical infinity of processes that may evolve over time, depending on received inputs. A program is such a formal specification, and the concept of „program“ is surely among the most fundamental concepts required to understand computers. We argue the case for programming as a component of general education. Not because of any direct utilitarian benefit, but in order to gain a personal experience as to what it means, and what it takes, to specify processes that evolve over time.

To further support our argument for programming as part of general education, consider an analogy with mathematics teaching. Many professions in science and engineering require the use of mathematical results; scientists and engineers are users of mathematics in the same sense that many people are users of computers. Thus, one might argue that scientists and engineers only need to learn how mathematical theorems are applied; that the concept of „proof“ is only relevant to professional mathematicians. But centuries of experience show that any math instruction at the college level and beyond involves significant time devoted to proofs, even though most students are unlikely to ever prove a theorem outside their math courses. The reason is plain: we do not trust a „mathematics user“ to apply formulas or mathematical software in a reliable and responsible manner if he has never understood the concept of „proof“. Applying mathematical results is a matter of understanding, not just of pattern matching. Similarly, applying computers should be a matter of understanding, not just of pushing the right keys.

The analogy between the role of programming as an introduction to computer science on the one hand, and the concept of proof in an introduction to mathematics on the other, is highlighted by analogous trends and debates in recent times. At the same time as programming was deleted from many curricula in grades K9-12, the concept of proof came under attack by some curriculum reform movements. But recently the importance of proofs in mathematics teaching is being emphasized again. In April 2000, the US National Council of Teachers of Mathematics (NCTM) published new Principles and Standards for School Mathematics (PSSM). [Ferrini 00] is a good overview of PSSM. In the chapter “Reasoning and Proof” of PSSM we find:

“Instructional programs from prekindergarten through grade 12 should enable students to

- recognize reasoning and proof as fundamental aspects of mathematics;
- make and investigate mathematical conjectures; develop and evaluate mathematical arguments and proofs;
- select and use various types of reasoning and methods of proof.

By the end of secondary school, students should be able to understand and produce mathematical proofs, arguments consisting of logically rigorous deductions of conclusions from hypotheses and should appreciate the value of such arguments.”

In analogy with proofs leading to insight into mathematical reasoning, programming develops insight into the way computers work.

But how to introduce students to the fundamental concepts of programming? There is no need to introduce beginners to the complexities inherent in professional programming languages and environments. Programming practiced as an educational exercise, free from utilitarian constraints, is best learned in a toy environment, designed to illustrate selected concepts in the simplest possible setting. The fundamental concepts of programming may be intellectually demanding, but they are not complex in the sense of requiring mastery of lots of details.

We present a learning environment, designed to be *as simple as possible*, that highlights a few important programming concepts. The programming system Kara is based on the concept of finite state machines and serves to drive toy „robots“ around the floor or the screen. Whereas the task of programming robot motion has been used by many systems since LOGO, the combination with finite state machines as a model of computation is new. Finite state machines are the key to the simplicity of the resulting system. They are well known from every-day devices such as traffic lights or vending machines, which makes it possible to explain to students the concept of state machines in a concrete, “hands-on” way.

3. Kara, the programmable ladybug

Kara [Reichert, Nievergelt, Hartmann 00], [Reichert 01] is a programming environment that realizes the goal of introducing programming by means of finite state machines. The use of finite state machines also has a practical, user-interface related advantage: a finite state machine can be constructed easily in a purely graphical manner. In a simplified version of „turtle geometry“ [Papert 80], and in the tradition of Karel, the robot [Pattis 81], Kara is a programmable ladybug living alone in a flat world. Fig. 1 shows the main window with the grid world and icons for commands to control Kara and to edit the world.

Appeared in the Proceedings of the 2001 IEEE Symposia on Human-Centric Computing Languages and Environments, pages 135–141. Stresa, Italy, September 2001.

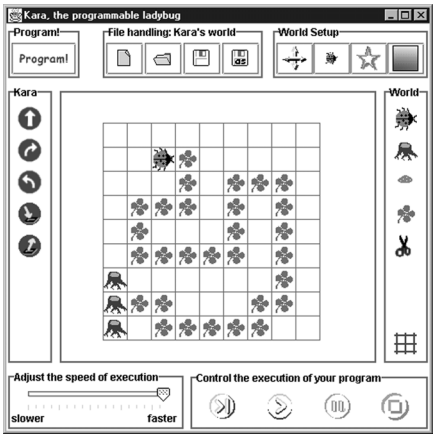
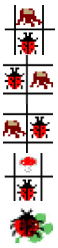


Fig 1: The world of Kara

In Kara's world, there are only a few types of objects: unmovable tree trunks, which completely occupy a square; mushrooms which Kara can push around; and cloverleaves, of which Kara can lay down or pick up any number; and of course, Kara himself. Kara is placed on either a free square or a square occupied by only a cloverleaf, facing one of four possible directions. Sensors inform Kara about its immediate surroundings:



Is there a tree on the square in front of Kara?

Is there a tree on the square to Kara's right?

Is there a tree on the square to Kara's left?

Is there a mushroom in front of Kara?

Is there a cloverleaf underneath Kara?

Kara can execute a couple of primitive actions:

Advance one square in the current direction.

Turn right by 90°, on the current square.

Turn left by 90°, on the current square.

Put down a cloverleaf.

Pick up a cloverleaf.

The world of Kara, and Kara himself, are intentionally as simple as described above. We kept the number of types of objects, and of sensors and commands, to a minimum with which we could still pose challenging tasks.

Using these sensors and commands, finite state machines for Kara are specified in the visual program editor shown in Fig. 2. The program shown makes Kara follow a trail of cloverleaves such as the one in Fig. 1, and eat them up.

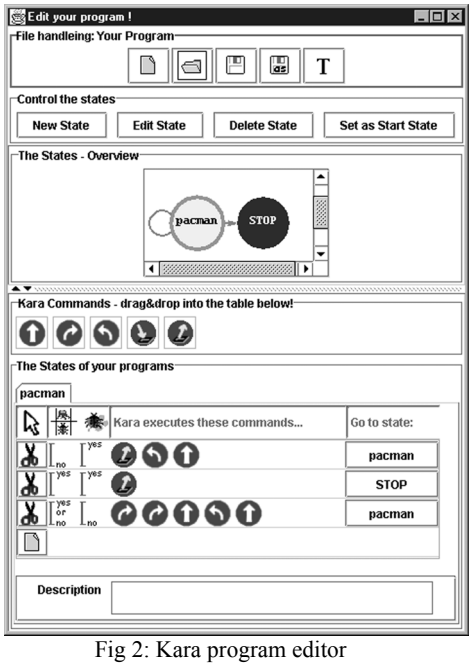


Fig 2: Kara program editor

4. Instructive finite state programs

Finite state machines are suitable for controlling processes that react to local conditions only. Indeed, given an environment like that of Karel the robot or Kara, it turns out that a program specified as a finite state machine is often more concise than a program for the same task written in a conventional programming language.

Consider a wall of trees of arbitrary shape, with arbitrary appendages, as shown in Fig. 3a. Kara must be programmed to endlessly cycle around the perimeter of the wall, hugging it with his right side. The following finite state program, presented in the notation of the Kara environment, solves the problem (Fig 4a, b).

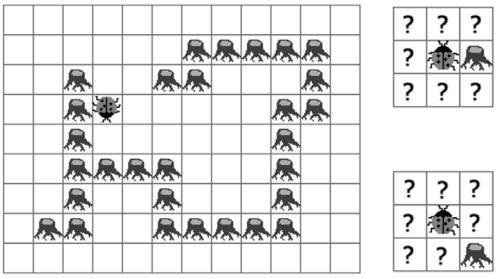


Fig 3: a) A wall of trees and
b) the predicate „wall to the rear-right"

Kara starts in state Seek, not knowing anything about its position, as expressed by the vacuous assertion „precondition = true". The function of this state is to bring about the postcondition „wall to the rear-right",

illustrated in Fig.3b: There is a piece of wall to the right of Kara, or diagonally to the rear-right, or both. As long as the bug has not sensed any piece of wall (neither ahead nor to the right), it moves forward. If it senses a wall, either ahead or to the right, it positions itself so as to fulfill the postcondition „wall to the rear-right“. The „don't care" notation „yes or no" serves as an abbreviation.

Seek: precondition = true, postcond. = „wall to the rear-right“

Fig 4: a) State Seek of the wall following program

Track: precond. = postcondition = „wall to the rear-right“

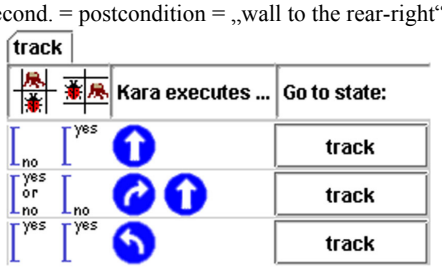


Fig 4: b) State Track of the wall following program

The mathematically inclined reader may find it instructive to prove the program correct by checking that the state Track maintains the invariant „wall to the rear-right“, and that, in each transition, Kara progresses in his march along the wall. Here we merely point out how several fundamental ideas of the theory of computation, of algorithms and programs can be illustrated in a simple, playful setting whose rules are quickly understood.

Fig.5 shows a cloverleaf pattern which represents the integers mod 2 of a Pascal triangle, with its apex at the top left corner of the grid.

It is remarkable that the four-state finite state machine shown in Fig. 6 suffices to draw this intricate pattern. The states 'carry0' and 'carry1' implement modulo 2 addition. The state 'next row' drives the bug across the grid in raster scan fashion.

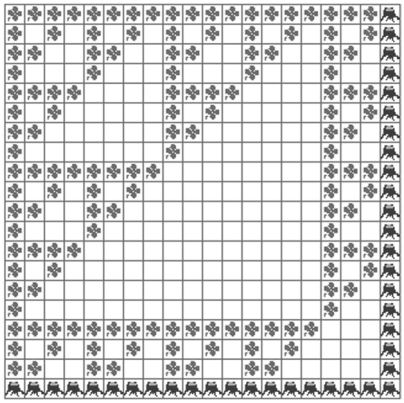


Fig. 5 Clover leaves and empty squares represent a Pascal triangle modulo 2.

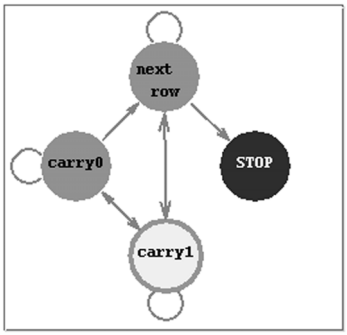


Fig.6 State diagram of the Pascal triangle program.

Fig.7 shows the detailed specification of the state 'carry1'. In this state, Kara already knows that the field to his rear-right is occupied by a cloverleaf. And to that field, it has to add the value of the current field. The bug consults its two sensors „am I facing a tree trunk“ and „am I sitting on a clover leaf“. The four possible answers to these two binary questions trigger different sequences of actions, followed by transitions to different next states. Consider the first transition as an example. Kara is not facing a tree trunk, meaning it has not yet reached the end of the current row. And it is not sitting on a cloverleaf, meaning it is reading a 0.

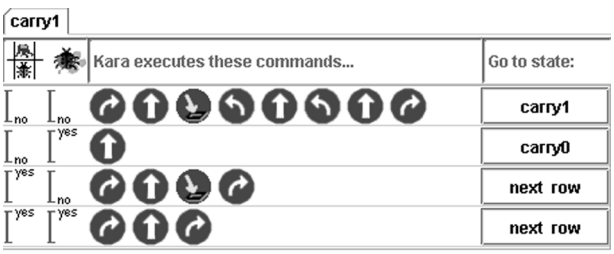


Fig.7 Specification of the state carry1.

Since 1 + 0 = 1, Kara deposits a cloverleaf, thus writing the result 1 of this mod 2 addition. Kara must write on the correct square and must position itself properly for the next bit addition. This requires the complicated dance expressed by 7 motion commands surrounding the "write" command.

5. Experience

Kara has been used for the past years in several courses to teach the fundamentals of programming to beginners, typically high school students who never had any exposure to programming, and teachers with little background in computing. The feedback we have received from all these users is highly positive. Beginners appreciate the fact that they succeed in solving non-trivial problems after being coached for just a few hours. More expert users are surprised to learn that the conceptually simple structure of programming using finite state machines raises fundamental questions of computer science.

Our programming courses start with a one day introduction to Kara, followed by a "real-world" programming language (Java, Pascal). The feedback from a survey of over a hundred high school students and over a hundred teachers can be summarized as following:

- Not surprisingly, Kara was most motivating for those students with the least prior experience in programming.
- All students found Kara either "very motivating" or "motivating" – no student was "indifferent" or found Kara "demotivating" or "very demotivating".
- Also, all students found the handling of Kara's user-interface either "very easy" or "easy" – no student was "indifferent" or found the user-interface "hard" or "very hard" to use.

The written comments of the students show that they thought Kara had allowed them to focus on problem solving, on the logic and the correctness of their programs, without being distracted by the environment or by the textual syntax of a "real-world" programming language. They also liked the simplicity of the user-interface, which reminded some of "early computer games".

Kara is used widely in Germany, Austria, and Switzerland (originally, the material was only available in German; now it is also available in English). To our surprise, Kara is not only used at the high school level, but also in K6-8 grade schools, and even in introductory courses at universities.

The programming environment Kara is published on the Web [Reichert 01], along with teaching materials: a wide selection of exercises and their solutions, and slides for introductory presentations. Our web server registers, on average, over 11'000 requests per month, and the programming environment is downloaded over 500 times a month.

The major question Kara users raise is „how do we get to real programming?" In order to facilitate this transition, the extended system JavaKara allows more advanced users to program the ladybug in Java. JavaKara contains templates of procedures that call the built-in Kara operations, letting the user learn how to „program by example".

6. From Kara the bug to robots and Java

The architecture of the Kara software package is designed to be flexible, with the goal that different programming models might later be added, beyond the finite state machines currently implemented. Therefore, the „integrated development environment" of Kara was split into two parts, which are interconnected by a small, clearly defined interface (Fig. 8):

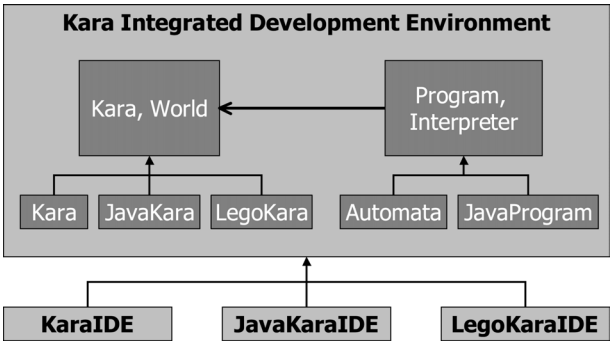


Fig. 8 Software structure of Kara

- Kara and his world. This part defines the properties of the world and the objects that can inhabit it, it defines the commands which Kara understands, and his sensors. From the outside, Kara can only be accessed through a „robot" interface which allows you to query him about his commands and about his sensors. These commands can of course be executed, and his sensors can be queried for their current states. However, Kara neither knows nor cares how he is being controlled and programmed; no assumptions are made about what and how the „outside" might want to do with Kara.
- Program and interpreter. The program package defines how Kara is programmed – or more precisely, how any entity implementing the „robot" interface can be programmed. The interpreter is responsible for executing the programs, controlling the entity through the „robot" interface. Neither the program model nor the interpreter need to know anything about the entity they are controlling except for the information obtained through the interface.

This separation of concerns makes it possible to construct a whole family of Kara applications. The world model and the programming model could be replaced by other implementations without having to reprogram the other part. The „standard“ implementation of the world model is Kara itself, and the standard implementation of the programming model are finite automata. We exploited the separation of the two parts when we created LegoKara and JavaKara.

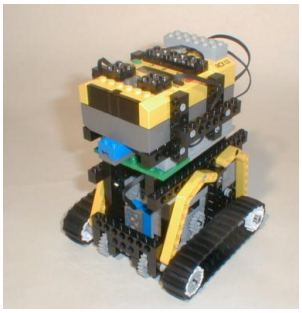


Fig. 8 Kara as a Lego Mindstorms robot

LegoKara (Fig. 8) is an implementation of Kara as a Lego Mindstorms robot. A detailed manual explains how to build a LegoKara, which differs slightly from the virtual Kara. For simplicity's sake, the robot lacks the ability to lay down or pick up cloverleaves, and it does not have the ability to distinguish walls from objects that could be pushed around (mushrooms in Kara's world). LegoKara is programmed using finite automata, in the same programming environment in which Kara is programmed. One can test programs in the simulated world of Kara and compile the automata to RCX (the Lego micro controller) byte code to be downloaded to the robot.

JavaKara lets the user program in Java. This makes it possible for Kara to solve problems beyond the capability of finite state machines. The interpreter had to be replaced entirely, and the program editor disappeared. Though running a Java program instead of interpreting a finite automaton was a major change, it only required changes to one part of the application. JavaKara makes it possible to learn Java step by step, from the ground up, in a graphical environment where the actions of programs are automatically visualized. This makes it easier for beginners to grasp the semantics of the different programming constructs, not only the syntax. The wider range of tasks that Kara can solve when programmed in Java highlights the limitations of finite state machines, and leads to an understanding of the power of different models of computation.

7. Related work and conclusions

The design of toy worlds for programming instruction is a popular endeavor. [Bergin 97] is a newer, objectoriented version of Karel, the robot, as are, for example [Buck 01] and [Boles 99], a hamster programmed in Java. [Brusilovsky 97] offers a review of some more toy worlds, most of them closely related to the idea of Karel, the robot. Some visual programming environments, not all directly related to teaching, can be found in [Dann 00], [Fenton 89], [Kahn 95], [Smith 94].

Most of these environments offer many more options and possibilities than Kara does, and some of them are not expressly designed to be used for an introduction to programming. Kara, on the other hand, was designed to be as simple as possible, to offer no more possibilities and options than absolutely necessary, yet still be useable for teaching the first steps in programming in a playful, graphical manner, within a timeframe of 6-12 hours. We contend that one reason why Kara is successful in achieving this goal is its unique choice of model of computation, finite state machines.

In summary, we hold that the software package Kara has proven that some basic ideas of programming can be learned by beginners in a couple of hours, in a playful, enjoyable environment. Even a physical realisation of Kara as a Lego Mindstorms robot is possible. And the same programming environment can be used to dive into the real-world programming language Java, offering a smooth transition from toy-world programming to real-world programming.

References

[Bergin 97] Bergin, J. Karel++: a gentle introduction to the art of object-oriented programming. Wiley, 1997.

[Boles 99] Boles, D. Programmieren spielend gelernt. Mit dem Java-Hamster-Modell. Teubner, 1999.

[Brusilovsky 97] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. (1997) Mini-languages: A Way to Learn Programming Principles. *Education and Information Technologies* **2** (1), pp. 65-83.

[Buck 01] Buck, D., Stucki, D.J. (2001) JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum in *Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education,* p. 16-20.

[Dann 00] Dann, W., Cooper, S., Pausch, R. (2000) Making the connection: programming with animated small world in *Proceedings of the Conference Integrating Technology into Computer Science Education*, pp. 41-44.

[Fenton 89] Fenton, J. and Beck, K. (1989) Playground: An object-oriented simulation system with agent rules for children

of all ages. *Proceedings of Fourth Annual Conference on Object Oriented Programming Systems, Languages, and Applications, OOPSLA'89*. New Orleans, LA, 2-6 October, 1989, pp. 123-137.

[Ferrini 00] Ferrini-Mundy, J.(2000): Principles and Standards for School Mathematics: A Guide for Mathematicians. Notices Of The AMS, September 2000, p. 868-876.

[Kahn 95] Kahn, K. (1995) ToonTalk – An Animated Programming Environment for Children, in *Proceedings of the National Educational Computing Conference.*

[LEGO 98] The LEGO Group: Mindstorms – Robotics Invention System, LEGO 1998. a) User Manual, b) Technical Reference, 110p.

[Papert 80] Papert, S.: Mindstorms. Children, Computers, and Powerful Ideas, Basic Books, NY, 1980.

[Pattis 81] Pattis, R. E.: Karel the Robot – A gentle introduction to the art of programming, Wiley, New York 1981.

[Reichert, Nievergelt, Hartmann 00] Reichert, R., Nievergelt, J., Hartmann, W.: Ein spielerischer Einstieg in die Programmierung mit Java, *Informatik-Spektrum 23 (5)*, Oktober 2000. Springer Verlag.

[Reichert 01] Kara, the programmable ladybug. http://www.educeth.ch/karatojava/

[Smith 94] 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.