Published as: R. Reichert, J. Nievergelt, W. Hartmann: Programming in schools - why, and how? In C. Pellegrini, A. Jacquesson (Hrsg.): Enseigner l'informatique,pp 143-152. Georg Editeur Verlag, 2001.

Programming in schools – why, and how?

Raimond Reichert, Jürg Nievergelt and Werner Hartmann, ETH Zurich

Abstract

Today's society sees Information and Communications Technology (ICT) almost exclusively via useful application software, with little understanding of "what goes on behind the screen". This has resulted in many highschools abandoning introductory programming in favor of teaching application skills. We argue that schools of general education should emphasize fundamental, timeless concepts underlying ICT, and that personal experience with writing small programs is a good way to introduce such concepts. If programming is taught for its intrinsic intellectual value, rather than as a tool for computer users, we seek the simplest settings that suffice to illustrate the concepts to be taught. We present Kara, a toy world where a robot is programmed as a finite state machine.

1. Introduction

Computers permeate the technical infrastructure our society takes for granted: microwave ovens, washing machines, vcrs, ticket machines, electronic payment or online flight reservation systems are just some examples. The internet and world wide web have linked humanity. Soon, ubiquitous computing will likewise link machines: everywhere extremely small, interconnected computers will communicate with each other.

A sound general education is necessary to keep up with the challenge of understanding these fast-changing technologies and to be able to cope with their increasing complexity. Schools face a difficult decision: what aspects of computer science to try to convey to students in the limited time available? Computer science comprises many different levels, but the general pulic only sees its applications. Educators may be tempted to concentrate on short-lived, product specific aspects of applications; to focus on low-level skills, such as “how to do it”, rather than on understanding, on “why and how it works”. Yet mastery of today’s release may not be helpful a few years later when confronted with a new system or a different application. Moreover, teachers often focus on application-specific skills not only because these skills are of immediate use, but also because they themselves are just users of computers and do not know much about what is going on “behind the screen”.

Schools that provide general education must promote an understanding of the fundamental concepts of computer science. Later, on the job, there is rarely time to study fundamentals. In language courses, for example, students learn not only vocabulary and grammar, but also basic concepts of communication, such as: How do I structure and formulate my thoughts for a specific audience? Physics courses teach

the basic laws of nature, such as conservation of energy, and their consequences. Such concepts of general education form the foundation which helps us keep up with life-long learning.

What are some of the fundamental concepts of computer science, and how can they be taught in school? One of the most fundamental aspects is that all computer systems are controlled by programs, i.e. by rigorously defined algorithms expressed in a formal notation. 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 „program“ is surely among the most fundamental concepts required to understand computers.

2. Programming as part of general education

Why should students be taught programming? After all, for just about every conceivable application, ready-made software packages provide tools much more powerful than what almost any user could write – and we are all users of software such as word processors, spreadsheets, search engines etc. If computer users no longer program, does it follow that the art of programming should only be taught to computing professionals? If school was only about the productivity of future office-application users, one might agree with this view. Instead of teaching programming, teachers might concentrate on how to use “Word” more proficiently.

But this point of view, introducing computer science as a set of application specific skills, has a minor and a major flaw. The minor flaw is that most introductions to office-application packages concentrate only on short-lived details of the package being used, making any transfer of knowledge to a different package difficult. Concepts which might be useful across applications, such as how information is encoded and structured, are often neglected, even though they facilitate mastery of new software.

The major flaw of a computer education focused on skills only is harder to explain. Let us 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: they check the preconditions (input) of a theorem or formula and derive the conclusions (output). 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 involves significant time devoted to proofs, even though most stu-

dents 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. Mathematics users do not need to know the proof of every formula they use; they need to understand mathematical thinking, based on the concepts of theorem and proof.

Similarly, applying computers should be a matter of understanding, not just of pushing the right keys. Computer users do not need to know the source code of every application they use, but they should have an intuition of what constitutes a program. This calls for personal experience with writing, testing and debugging a number of small programs. In the days of ready-made application software, we do not need programming as a tool, but rather as background knowledge that helps us understand what computers can and cannot do. A similar statement can be made for any kind of general education. General education is rarely applied for immediate use, but helps us put details of transient importance into a larger perspective. [Nievergelt 99] discusses the value of programming as part of general education in more detail.

3. First steps in programming – mini-languages

How to introduce students to the fundamental concepts of programming? If a teacher wants to teach programming, what language should he use? C++, Java, Delphi? These languages are made for the professional programmer; they are powerful and complex. They are also object-oriented, posing an additional challenge for the teacher: whereas object-orientation is relevant to “programming in the large”, it is a non-trivial additional hurdle for a beginner. And last but not least, programming environments with their project-management and debugging facilities are complicated to handle and not suited for an easy introduction to programming.

There is no need to introduce beginners to the complexities inherent in professional programming languages and environments, where you need a manual just to display “hello world”. 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. The main ideas can be conveyed with a few selected, simple examples.

Artificial toy-worlds have a long tradition as programming environments suitable for novice programmers. The value of such an environment is not to be measured by what you can program with it, but rather by the cost-effectiveness balance, by comparing how easy or hard it is to gain a specific insight. One of the first mini-environments was LOGO [Papert 80]; probably the most popular environment to date is Karel, the robot [Pattis, 81]. These inspired many other mini-language environ-

ments; some in the tradition of Karel the robot, like [Bergin 97] or [Boles 99] or those presented in [Brusilovsky 97], an overview of mini-languages. Other environments can be found, for example, in [Dann 00], [Fenton 89], [Kahn 95], [Smith 94].

Karel is a robot living in a virtual world on the screen and is programmed in an imperative programming language. For a beginner, any full-scale language is a challenge. How could this first hurdle be reduced? Instead of using a conventional programming language, use a simpler model of computation such as finite state machines. This idea was proposed in [Nievergelt 99] and has its advantages: finite state machines are part of every-day devices and can be illustrated by examples such as vcr's or vending machines. It is easy to represent them in a graphical manner. Paths of execution are defined statically as paths in a directed graph; no other control structures are needed.

We present a learning environment, designed to be as simple as possible, for the first steps in programming. Kara, the programmable ladybug [Reichert 01], [Reichert, Nievergelt, Hartmann 00], may seem to be just another mini-language environment designed in the tradition of Karel the robot, but it differs from its predecessors in its drastic quest for conceptual simplicity: the choice of finite state machines as model of computation.

The first steps of programming Kara are easy and playful. There is no need to learn the syntax of a programming language, since Kara is programmed in a purely graphical manner. At first glance, Kara's abilities are highly limited. However, there is a wide range of tasks that Kara can be programmed to do in his world, including some tough problems. And when students reach the limits of Kara and want to move on, they are well equipped to learn a professional programing language – Java – with the learning environment JavaKara. This environment is designed to make the first steps in Java easy and visual, without any unneccessary, Java specific baggage. The environments Kara and JavaKara, along with a wide selection of programming problems and their solutions, can be downloaded from the world wide web: http://www.educeth.ch/karatojava.

4. Kara – a programming environment based on finite state machines

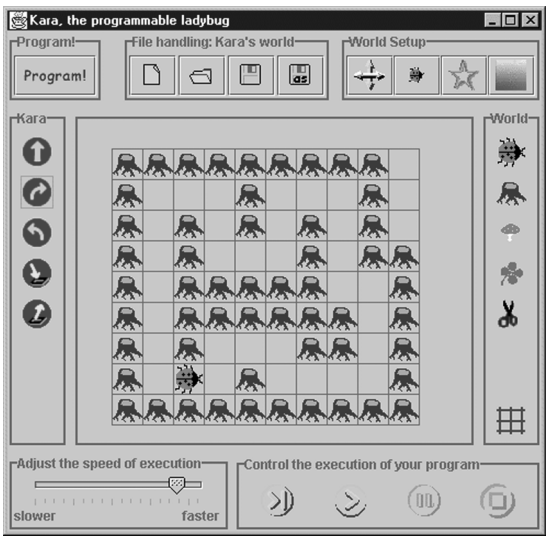

Figure 1: Kara and his world

Kara is a programmable ladybug living in a graphical world on the screen (Figure 1), a rectangular grid of squares. He can only move from one square to an adjacent square. On any square you may find an unmoveable tree trunk; a moveable mushroom; a cloverleaf; and of course, Kara himself. He can be programmed to do specific tasks, for example, to pick up or put down an arbitrary number of cloverleaves. Five sensors help Kara recognise his surroundings: Is there a tree ahead? Is there a tree to my left? Is there a tree to my right? Is there a mushroom ahead? Am I standing on top of a cloverleaf? Kara knows only a few commands: one step ahead; turn left by 90°; turn right by 90°; put down a cloverleaf; pick up a cloverleaf. Kara may execute any number of commands depending on his sensors' values .

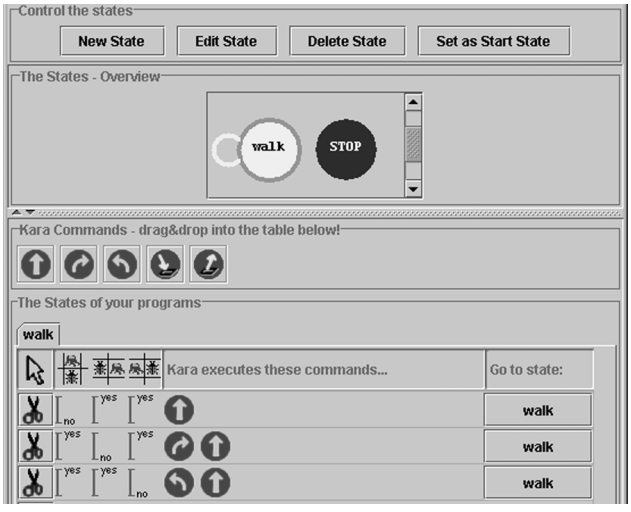

Figure 2: Programming Kara

Let's consider an example. Kara is to walk endlessly along a circuit as shown in Figure 1. A circuit has the following property: each square in the circuit has exactly two

free adjacent squares. This property makes writing the program shown in Figure 2 straight-forward. If we assume that Kara starts program execution with one free square behind him, then exactly one of the square to his left, in front, or to his right must be free. This is the square he visits next. Now the square behind him is free again, re-establishing the invariant of the state “walk”.

Beyond simple examples such as the round trip above, or placing clover leaves in a chess board pattern, there are many challenging tasks Kara can solve, such as playing simplified versions of PacMan or Sokoban, computing a Pascal triangle modulo 2, or move the towers of Hanoi.

A physical Kara robot is harder to realize, since such a robot highlights the problems of leaving the precisely defined digital world. Legokara [Meier, Reichert, Zürcher 00; http://www.educeth.ch/informatik/karatojava/legokara] is a realization of Kara as a Lego Mindstorms robot (Figure 3), and comes with a compiler to generate the byte code for the RCX robot controller from Kara’s finite state machines. LegoKara is programmed in the same graphical environment as Kara; there is no need to know the Lego programming environment. With a detailed construction manual, one can build a LegoKara in about two hours. One has to confront problems which do not exist in the idealized, discrete world of the virtual Kara. For example, Kara may think there is no wall in front of him – because his wall-sensing sensor infrared beam illuminates only a small portion of the scene ahead of him. Or how the parameters for a 90° turn have to be adapted to the physical environment, to the texture of the ground.

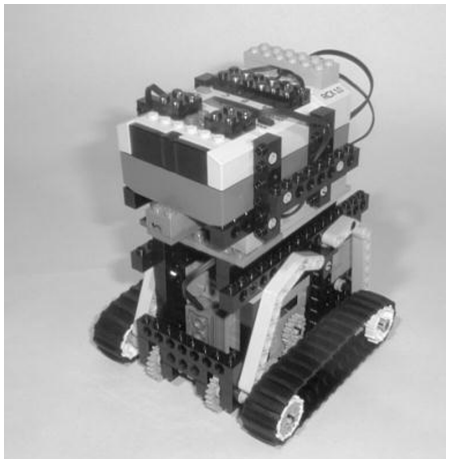

Abbildung 3: LegoKara-Roboter

5. The transition from Kara to Java

How to progress from an artifical mini-environment such as Karel the robot to a real-world programming language like Java? With most mini-environments, the students have to abandon both the virtual world and its programming language. JavaKara, i.e. Kara programmed in Java, aims to offer a smoother transition. Students know the virtual world already, so they can concentrate on learning the Java language and the handling of a Java compiler. They can learn the basics of Java with a series of exercises of increasing complexity designed to motivate and to introduce the programming constructs of Java.

```
import roboapp.javakara.JavaKaraProgram;

public class Spiral extends JavaKaraProgram {

  void walk (int distance) {
    for (int i = 0; i < distance; i++) {
      kara.putLeaf();
      kara.move();
    }
  }

  protected void myProgram() {
    final int MAX_LENGTH = 20;
    int d = 2;
    while (d < MAX_LENGTH) {
      walk (d);
      kara.turnRight();
      walk (d);
      kara.turnRight();
      d = d + 2;
    }
  }
}
```

Figure 4: A simple JavaKara program

At first, students only work with a subset of Java. Predefined templates hide the object oriented constructs of Java. This makes it easier for students to focus on those language constructs that can be found in any imperative programming language, such as decisions, loops, methods, boolean expressions, base types like integers or booleans, arrays etc. Figure 4 shows a program which draws a cloverleaf spiral. Some things are defined by the JavaKara environment: A program for JavaKara has to be derived from the class `JavaKaraProgram`; the main program has to be in the method `myProgram`. `JavaKaraProgram` mainly offers access to the object `kara`, and is the link between new programs and the programming environment.

As Figure 4 shows, students have contact with objects from the beginning: Kara is an object on which they can call methods to execute commands or to query his sensors. However, students do not need to know exactly what an “object” is. They just have to know how to call the relevant methods such as `kara.move()`.

JavaKara offers a step-by-step introduction to the fundamental concepts of an imperative programming language. Our website http://www.educeth.ch/

karatojava/ contains over 40 examples. Once students have grasped the concepts behind these examples, they know about loops, branches, methods, variables, paramters, arrays, expressions. After working through these examples, they should be well equipped to read a textbook on Java and to learn about object orientation and class libraries – moving into the realm of “real-world” programming.

6. A simple system illustrates deep issues

What fundamental concepts of computer science lay hidden behind the playful user interface of Kara? To an experienced computer science teacher, many – you just need to extract them and to present them in a manner accessible to novices. Here we merely mention some examples, as the presentation of each single concept offers enough material to cover several lessons. Let’s start with some aspects concerning systematic programming.

Invariant, proof of correctness. A finite state machine is a potentially complex control structure. Transitions are jumps that could go “anywhere” within the machine. It is therefore imperative that each state be associated with an invariant which describes the relationship between Kara and his environment. An invariant must hold everytime the automaton is in a particular state. The verification that each transition transforms the invariant of the originating state (pre-condition) into the invariant of the target state (post-condition) leads to a proof of correctness.

Macros, Procedures. Kara programmers notice that certain sequences of commands are often reused. If Kara wants to know whether there is a cloverleaf on the square behind him, he has to turn around, step ahead, check the cloverleaf sensor, store the result, turn around and step ahead again – a complicated dance to procure a simple information. This illustrates a fundamental principle of programming: how can you avoid writing repeatedly the same sequence of commands? The solution to this problem are subprograms; for finite state machines, one would use macros, to be inserted where needed.

Let’s now turn to one of the core questions of computer science: what can be computed with different kinds of computing models? This question leads to the theory of automata and formal languages. The “finite state machine” Kara serves to illustrate different computing models and shows how important it is to precisely define all the details. Subtle differences with respect to what Kara can do to his world change his power of computation. In this theoretical discussion we consider an idealized version of Kara living in a world of infinite size; we will consider different assumptions of what we allow Kara to do. In the least powerful version (a), the world is empty except for Kara, and remains that way. In the most powerful version (b), we allow Kara to read and write characters of a finite alphabet – the empty square and the cloverleaf suffice to represent the {0, 1} alphabet.

(a) The actual finite state machine. In this model we assume that Kara is not allowed to change the world; he may only move himself. To illustrate this restriction, consider the possible walks Kara can do in an empty world of infinite size. If there is nothing in the world to "read", the automaton will soon either stop or fall into a periodic pattern which it repeats endlessly. If there are features for Kara to read, but he is not allowed to write, he can do more interesting tasks. Finite state machines recognize regular languages, i.e., strings of characters which follow simple rules. What is Kara's two-dimensional analogon to character string recognition? We consider a language L, a set of cloverleaf drawings of finite size, enclosed by trees. The question is whether Kara can be programmed to recognize exactly those drawings in L and reject any other drawing. Kara may not change the world to answer this question – if he wants to memorize anything, he has to do it using states.

(b) Turing machine. If Kara may lay down and pick up cloverleaves at arbitrary places, he can use the world as a "read/write" memory of unbounded size. Under this assumption, Kara becomes the most powerful computing model in the hierarchy of automata – a Turing machine which can compute anything algorithmically computable (given suitable encoding of input and output).

*(c) Concurrent processes***.** If multiple robots cooperate with each other to solve a task, then one is confronted immediately with difficult problems of concurreny [Nievergelt 99].

7. Conclusion

The role of computer science in schools changed several times during the past couple of decades. The most recent change replaced programming in favor of a more application-oriented curriculum. We regard this as a mistake, because the justification for programming in schools is not its immediate applicability in real life, but in its general educational value: a sound, intuitive understanding of what it means to delegate to a machine control of ever more complex every-day processes.

If one agrees with this analysis, the question becomes how one can teach, in a short span of time, some fundamental concepts of programming without being distracted by irrelevant system-specific details. This is a challenge to computer science educators which can be tackled in more than one way. We hope that Kara will be used as one way to introduce students to the timeless fundamentals of programming.

Literatur

[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.

[Dann 2000] 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.

[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.

[Meier, Reichert, Zürcher 00] Meier, R.; Reichert, R. ; Zürcher, S.: LegoKara, die Lego-Mindstorms-Version von Kara. ETH Zürich, KS Baden, 2000.
http://educeth.ethz.ch/informatik/karatojava/legokara

[Nievergelt 99] Nievergelt, J.: „Roboter programmieren“ – ein Kinderspiel. Bewegt sich auch etwas in der Allgemeinbildung? Informatik Spektrum 22, Nr 5, 364-375, Oktober 1999.

[Papert 80] S. Papert: 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. John Wiley & Sons, 1981.

[Reichert, Nievergelt, Hartmann 00] Reichert R., Nievergelt J., Hartmann W.: Ein spielerischer Einstieg ins Programmieren mit Kara und Java. Informatik Spektrum Vol 23 Nr 5, Oktober 2000.

[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.