Teaching Programming Using the Karel the Robot Paradigm Realized with a Conventional Language

by Roland H. Untch
Department of Computer Science
Clemson University *

Abstract

An excellent method for introducing students to computer programming was described by Richard E. Pattis in his book *Karel the Robot.* By initially limiting the student's language repertoire to easily grasped imperative commands whose actions are visually displayed, the Karel approach quickly and effortlessly introduces the student to such concepts as procedures and the major control structures. However, some who have used the technique as a "quick-start" introduction to programming have noted some problems in the transition from using the Karel language to the conventional language used for the rest of the course (e.g. Pascal). By embedding the Karel programming paradigm in a conventional language, we have been able to eliminate these transition problems while retaining the pedagogical merits and spirit of Karel. This paper provides a brief review of the *Karel the Robot* programming paradigm, considers the transition problems, describes our novel use of Karel in a conventional language, and informally presents the results of using our version of Karel.

1 Introduction

Those of us charged with introducing programming to students do not have an easy task. The ability to analyze and solve problems is a skill not easily learned. To improve our student's problem solving and programming skills, we teach them a number of tools and techniques that enable them to "simplify" problems, that is, manage the complexity and size of the problems. We teach them about functional decomposition and stepwise refinement. We explain the concept of structured programming and discuss the merits of top-down development. In the process we introduce a number of tools, such as hierarchy charts and pseudo-code.

Unfortunately too much data and too many data types confuse the beginning student. Usually the scope of early programs is restricted in an effort to lessen the data complexity. The result, unfortunately, is programs that are often so simple as to be trivial. The early programs that we have traditionally been forced to assign (for instance programs to calculate mortgage payments, determine prime numbers, or compute Fibonacci series) do not require the use of "modern programming techniques". It is consequently difficult to establish the need and practice the use of the work management and design tools that the students will ultimately need.

Many have wrestled with this difficulty and a few satisfactory solutions have emerged [1, 2, 3, 7]. A rather novel solution was developed at Stanford University by Richard E. Pattis and is described in his book *Karel the Robot: A Gentle Introduction to the Art of Programming* [6, 5]. What Pattis did was develop a robot programming paradigm, called Karel (pronounced "Carl") that was entirely imperative. To quote from the preface of Pattis' book: "The careful omission of variables and data structures from Karel's language ... allows the immediate exploration of the rich domain of abstraction and control structures." Having used this approach since early 1985, we have found that he was correct.

Others have reported on their use of Karel. Lt. Colonel Kenneth L. Krause states that at the United States Air Force Academy, "Karel proved to be enormously successful, the value of which far exceeded that of a mere motivator. Students easily grasped the subtleties of Karel and his language. They displayed impressive capabilities to employ top-down design/stepwise refinement techniques in solving relatively complex problems. They gained a solid appreciation of language structure, programming errors, and program behavior. In short, Karel lived up to all the claims of the author and represents a powerful pedagogical tool." [4]

*This draft: March 1990

Weaning students from Karel, however, causes some problems. It is a solution to those problems that we intend to present. In the following sections we will briefly review the *Karel the Robot* programming paradigm, describe the transition ("weaning") problems, present our method of using Karel in a conventional language, and indicate how this method addresses these transition problems.

2 What is Karel the Robot?

Karel is essentially a programmable cursor that can move across the flat world of a CRT screen. Shown on the screen is a gridwork of vertical and horizontal lines (avenues and streets) that form intersections or street corners. Karel is restricted to moving from street corner to street corner, one such move at a time. Additionally, Karel can pivot 90 degrees to the left when requested. Karel can only face North, South, East, or West and can always determine which direction he is facing.

To add variety to Karel's environment, Pattis added beepers, flashing symbols that Karel can detect ("hear"), pick up, carry, and put down. It is possible to program Karel to locate beepers, transport them, or place them in some graphic pattern. To bound Karel's environment there are wall sections that can be placed between streets and form impenetrable barriers. Karel can detect ("see") wall sections that are immediately to his front or sides. These wall sections can be used, to give two examples, to form obstacle courses that Karel must navigate or to represent hurdles that Karel must "jump" in a hurdle race.

Karel initially understands only five imperative commands: `move`, `turnleft`, `pickbeeper`, `putbeeper`, and `turnoff`. These are Karel's so-called primitive instructions. When these commands are executed in a Karel program, the results are depicted on the student's screen. A `move` instruction, for instance, will graphically show Karel moving from one street corner to the next. Should a wall section be in the way, however, Karel will signal the message "`Error Shutoff`" in protest and terminate execution of the program.

New instructions can be defined to extend Karel's vocabulary. For example, to define a `turnright` instruction one would write:

```
DEFINE-NEW-INSTRUCTION turnright AS
BEGIN
     turnleft;
     turnleft;
     turnleft
END
```

This definition must be repeated in each program that wishes to use a `turnright` instruction.

Karel is able to respond to the elements in his environment by testing a fixed set of predicates. The predicates Karel can evaluate or test are:

```
front_is_clear          front_is_blocked
left_is_clear           left_is_blocked
right_is_clear          right_is_blocked

next_to_a_beeper        not_next_to_a_beeper

facing_north            not_facing_north
facing_south            not_facing_south
facing_east             not_facing_east
facing_west             not_facing_west

any_beepers_in_beeper_bag
                        no_beepers_in_beeper_bag
```

(Note: Pattis used hyphens as separators within identifiers; we have changed these to the more commonly used underscore.)

These predicates are used in Pascal-like control statements. For example, when moving from corner to corner in a hurdle race, Karel could alter its actions based on whether a hurdle (wall section) is immediately in front of him or not. The code for this would resemble the following:

```
IF front_is_clear
     THEN move
     ELSE jump_hurdle
```

where `jump_hurdle` is presumably some instruction that the programmer has defined using the mechanism previously described.

Karel programs are either manually executed or run under a Karel simulator. The simulator is generally a simple but complete programming environment containing both an editor and an interpreter for Pattis' Karel language. Thus students using the simulator must first learn both this special pedagogical language and the commands of the simulator environment before they can test their logic.

3 The problems of transition

Once the basic features of Karel are mastered, and the student is thoroughly acquainted with the programming techniques mentioned in the introduction above, the

student is next taught to program in a conventional programming language, such as Pascal or Modula-2. Unfortunately this transition can be difficult. Some students become frustrated. Comments such as "bring back Karel" [4] are occasionally expressed. The transition is difficult because the student is simultaneously asked to learn a new language *and* a new operating environment *and* is presented with a new domain of problems.

Although Pascal-like, the Karel language is not Pascal. As such it is fraught with minor syntactic differences that must be identified and assimilated by those learning Pascal. This process can be maddening, especially to students who are still uncertain of their programming skills. And, if the language to be learned is not Pascal, the frustration level mounts. Students subsequently learning C or Ada, for example, are often confused by the use of the semicolon as a statement terminator instead of a separator. Finally, what skill the student acquired in deciphering error messages produced by the Karel simulator is of little avail with the new compiler.

This frustration with mid-course "retooling" is made worse by the need to learn a new editor and operating environment. We are all personally familiar with feelings of impatience and frustration when working with an alien editor or operating system. (Confess—how many of us run an old-fashioned but familiar editor, like `vi` or `spf` or `emacs`, on our personal computers?) Such feelings can be especially disheartening to a beginning programmer. Moreover neither the student nor the instructor can afford to spend much time on delving into the details of this new environment. At this point in the course the instructor must use the established momentum to discuss other topics. Similarly, students who may have felt free to experiment with an editor at the beginning of the term now have other demands on their time.

The greatest problem in the transition, however, comes from the underlying reason for the transition. Except for the implicit data object that is Karel's world, the two-dimensional screen, the student has not been taught how to declare and manipulate any data. Consequently the conventional language is introduced as a vehicle for teaching about variables, expression evaluation, assignment statements and the like. Alas, the initial programs assigned in the conventional language lack the intuitive feel of the Karel programs. For example, a student could easily ascertain that a Karel program was incorrect when Karel, say, tried running into a wall. A program to calculate mortgage payments, on the other hand, is less easily verified. In fact, all too often students resort to "democracy" to validate their results; that is, they compare answers and the majority output wins.

The solution to these transition problems is, of course, DON'T demand the students learn a different language, DON'T demand the students learn to use a different environment, and GRADUALLY switch to other problem domains. All this can be accomplished by embedding Karel in an existing conventional language. This is what we have done.

4 A method of using Karel in a conventional language

Instead of viewing the five Karel primitives as statements in a language, we elect to view the Karel primitives as invocable procedures that manipulate the screen data object. When thought of this way, it is a relatively straightforward matter to implement them as such. (If you wish, you may consider the screen as an abstract data type. The Karel primitives are operations on this data type.) These procedures are stored in a library where they can be referenced by (linked with) the student's program code.

We then simply use the constructs and syntax of the underlying conventional language to build our Karel programs. To extend Karel's vocabulary, we use procedures. For example, to define `turnright` in Pascal, we would write:

```
(* Pivot KAREL 90 degrees to right *)
 procedure turnright;
     begin
         turnleft;
         turnleft;
         turnleft
     end; (* turnright *)
```

Similarly, the Karel predicates can be viewed as parameterless Boolean functions that return screen state information. Rather than implement them as such, it is advantageous to implement them as global Boolean variables whose values are set by the primitive procedures as they are executed. Not only is this somewhat more efficient, but some compilers demand that functions with no parameters nonetheless be invoked with an empty argument list. Thus instead of simply writing `front_is_clear` we would have to write `front_is_clear()` which adds a useless, and potentially confusing, set of parentheses. Using these Boolean predicates in conventional control statements, we are able to write code like the following Pascal example:

```
    procedure sparse_harvest_to_wall;
        begin
            if next_to_a_beeper then
                pickbeeper
            while front_is_clear do
                begin
                    move;
                    if next_to_a_beeper then
                        pickbeeper
                end;
        end; (* sparse_harvest_to_wall *)
```

It is convenient to add a sixth Karel primitive, `turnon`, to initialize the screen. The `turnon` primitive reads from an external file the information necessary to initialize the encapsulated screen data structure, initializes the Karel predicates, causes the screen to be displayed on the terminal, and returns. After that, execution proceeds much as it would under a Karel simulator.

Any necessary declarations of external variables and procedures can be hidden from the student by placing them in a text file that is included by some standard compiler directive. This "include statement" is accepted by the students as a given. This inserted code is not visible on the student's source listing. An example of a complete Karel program, as prepared by a student, follows.

```
(* An expanded version of the Stair Cleaning *)
(* Task program from Chapter 3 of Pattis.    *)

PROGRAM stairs (INPUT,OUTPUT,SITUATION,REPORT);
%INCLUDE 'KAREL:KAREL(PASCAL)'

(* Pivot KAREL 90 degrees to right *)
procedure turnright;
    begin
        turnleft;
        turnleft;
        turnleft
    end; (* turnright *)

(* Climb on to next step *)
procedure climb_stair;
    begin
        turnleft;
        move;
        turnright;
        move
    end; (* climb_stair *)

(* Attempt to remove a beeper *)
procedure pickbeeper_if_present;
    begin
        if next_to_a_beeper then
            pickbeeper
    end; (* pickbeeper_if_present *)

begin (* main *)
    turnon;
    while front_is_blocked do
        begin
            climb_stair;
            pickbeeper_if_present
        end;
    turnoff
end. (* main *)
```

For the student's earliest assignments, the two lines

```
PROGRAM progid (INPUT,OUTPUT,SITUATION,REPORT);
%INCLUDE 'KAREL:KAREL(PASCAL)'
```

are given and used without explanation. The student is simply asked to change the program identifier name from assignment to assignment.

The following is a snapshot of the student's screen midway through the execution of the above program. (It has been edited slightly to fit on the page.)

```
 MOVE
 CORNER  FACING  BEEP-BAG  BEEP-CORNER
  (4, 5)   EAST       2         0
ST.+-----------------------------------
 9 | .   .   .   .   .   .   .   .   .
   |
 8 | .   .   .   .   .   .   .   .   .
   |
 7 | .   .   .   .   .   .   .   .   .
   |
 6 | .   .   .   .   .   .   .   .   .
   |
 5 | .   .   .   .   .   1   .   .   .
   |                   +---+
 4 | .   .   .   .   > | . | .   .   .
   |               +---+   |
 3 | .   .   .   . | .   . | .   .   .
   |           +---+       |
 2 | .   .   . | .   .   . | .   .   .
   |       +---+           |
 1 | .   . | .   .   .   . | .   .   .
   +-------+---------------+-----------
     1   2   3   4   5   6   7   8   9  AVE.
```

5 Teaching with this version of Karel

Let us now examine how to introduce programming to students using this version of Karel. In addition to discussing course objectives and administration, the first lecture gives an overview of the computer system the students will be using. The students are assigned accounts, taught how to log on and off the system, and given a command that lets them execute a sample Karel program. Asking them to use the system prior to next lecture ensures some familiarity of the operating environment (lab locations, usage procedures, terminals, etc.) prior to the detailed presentation of that environment. Also, seeing Karel skitter across the screen rouses their curiosity. If, as occasionally happens, the accounts are not ready for the first lecture, a preliminary look at Karel is substituted instead. As the lectures continue with discussions of Karel programming, the labs can concentrate on teaching the students how to use the operating system, the hardware, and, especially, the editor. Exercises where the students enter and execute an existing Karel program are particularly helpful in building confidence and providing practice. Some of these programs are subsequently modified to illustrate new concepts. For instance, the use of procedures is introduced very early—at the end of the second lecture or the beginning of the third. To nail down this concept, the students are asked to define `turnright` and `turnaround` procedures. They then take a practice program and replace sequences of `turnleft` instructions with `turnright` and `turnaround` procedure calls as appropriate.

Since only a small subset of the language is being used at this point, the error messages produced are similarly constrained. Students rapidly learn to associate certain types of messages with certain mistakes. (On an indulgent day, we might say they "learn how to read the error messages".) As their language repertoire increases, they become increasingly adept at identifying the source of any lexical or syntactic errors.

By the fourth week of instruction the students have authored and run at least four or five complete programs (not to mention the exercise programs given to them). Not only does practice indeed make perfect, but this early amount of activity sets a pattern of work that continues throughout the rest of the term. In fact, our students typically complete 12 to 14 programs in a semester, with the next to last program being a file handling program of approximately 1100 lines.

Even more importantly, the Karel programs by their very nature are well structured. Thus, when it comes time to formally discuss such issues as, say, stepwise refinement, the students already have an intuitive grasp of these issues.

Once procedures and flow control structures are well understood, it is time to discuss variables, expression evaluation, parameter passing, and the like. Since we are using a conventional language, we can immediately give illustrations of these concepts to our students. Moreover, graphic, Karel-style programming assignments using variables and numerical expressions are easily developed. For example, the following Pascal subroutine is part of an assignment where Karel needs to count the number of beepers on the current corner. The routine is invoked by `count_beepers_on_corner( number );`

```
procedure count_beepers_on_corner
            (var number : integer);
    var   i : integer;
    begin
        number := 0;
        while next_to_a_beeper do
            begin
                pickbeeper;
                number := number + 1
            end;
        for i := 1 to number do
            putbeeper
    end; (* count_beepers_on_corner *)
```

Not only are such assignments still easily understood by the student and the results readily apparent, but these problems retain the important element of being "**big**". By that, we mean they need a lot of subroutines. This is important! How else can we motivate the need for parameters and functions and top-down development?

Even long after the Karel section of the course has been completed, Karel examples come in handy. When discussing hierarchy charts, for example, it is easier to demonstrate with some relatively compact yet meaningful Karel-style modules rather than some contrived conventional ones. Certain types of exam questions are easier to write given the student's Karel background and the ability to thus make certain implicit assumptions about a problem.

All of the above illustrate the primary pedagogical advantage of using Karel in this way. **The students always add to their stock of knowledge and never need to relearn (or even unlearn) something.** Nothing is wasted (almost).

6 Summary

Karel is extremely useful in introducing students to computer programming. When implemented[1] and taught as described above, certain pedagogical problems inherent in the simulator approach do not arise. Students do not experience as much frustration. Moreover, by eliminating certain extraneous issues, this approach is more efficient.

References

[1] Adams, J. Mack, Philippe J. Gabrini, and Barry L. Kurtz, *An Introduction to Computer Science with Modula-2*, D. C. Heath and Company: Lexington, Massachusetts, 1988.

[2] Drew, Mark S. and Shane D. Caplin, "Batch Logo – A Strategy for Introducing PL/I and Structured Programming to Gifted High School Students", *SIGCSE Bulletin*, vol. 16, no. 2, June 1984, pp. 13-16.

[3] Harvey, Brian, *Computer Science Logo Style*, The MIT Press: Cambridge, Massachusetts, 1985.

[4] Krause, Kenneth L., Robert E. Sampsell, and Samuel L. Grier, "Computer Science in the Air Force Academy Core Curriculum", *SIGCSE Bulletin*, vol. 14, no. 1, February 1982, pp. 144-146.

[5] Miller, Phillip L. and Lee W. Miller, *Programming by Design: A First Course in Structured Programming*, Wadsworth Publishing Company: Belmont, California, 1987.

[6] Pattis, Richard E., *Karel the Robot: A Gentle Introduction to the Art of Programming*, John Wiley and Sons: New York, 1981.

[7] Tomek, Ivan, *The first book of Josef: An Introduction to Computer Programming*, Prentice-Hall, Inc.: Englewood Cliffs, New Jersey, 1983.

[1] This paper describes our older VAX-based, Pascal version of Karel. Currently we are using a PC-based, Logitech Modula-2 version of Karel.