Another Experiment with Teaching of Programming Languages

by

Farhad Mavaddat

Research Report CS-80-40
Department of Computer Science
University of Waterloo

September 1980

Faculty of Mathematics

University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Another Experiment with Teaching of Programming Languages

by

Farhad Mavaddat

Research Report CS-80-40
Department of Computer Science
University of Waterloo

September 1980

ANOTHER EXPERIMENT WITH TEACHING OF PROGRAMMING LANGUAGES

by

FARHAD MAVADDAT

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada

ABSTRACT

The key issue in successful teaching is that of keeping the interest of students alive. In a first course on computer programming, this can be achieved by presenting interesting problems of reasonable complexity early in the course. This is often a difficult task, as most of the common programming languages used for instruction are intangible and therefore cumbersome to master before any serious programming may start.

Here a simple and tangible programmable machine is introduced. It is shown that important programming concepts can be defined and exercised in terms of possible operations on this machine. It is also shown that a seemingly difficult problem can be solved by novice programmers within the first few sessions of an introductory course.

INTRODUCTION

Any reasonable introduction to computer programming should focus on at least two objectives. The first is destructive in nature and consists mainly of undoing the popular image of computers created by the vast and often misleading influence of the media and of people uninformed about computing. It should be achieved by showing that a digital computer is nothing but a fast and obedient servant, capable of following exactly the instructions presented to it by its human master. The instructor should emphasize that a computer will obey all such instructions and nothing but those instructions. It is only based on this shattered image that the instructor is able to get into constructive programming and, to the student's surprise, show that many of his original beliefs have in fact been true and possible, though in a different way.

It is unfortunate that many first courses on computer programming deal only with the destructive aspects and leave the constructive side to subsequent courses. This may not be so serious for those who are required by the nature of their studies to take additional programming courses. But it does have serious consequences for those who don't get the chance to go through any such additional courses. They are left with a faulty image of what computation is and are ignorant of its inherent power beyond that of a number cruncher.

Based on these observations I have been examining ways of bringing some of the constructive aspects of computing into a first course on computer programming. The favourable results of one such experiment were reported in [1]. This paper deals with a second technique which can be applied to less sophisticated students or can precede the method of [1] for a more mature audience.

OBSERVATIONS

Both experiments are based upon the observation that in most first courses on computer programming the instructor sets himself the task of simultaneously teaching two rather distinct concepts, namely "algorithmic design" and "language machines".

By the knowledge of "algorithmic design" I mean the body of programming know-how shared by all programmers regardless of the language in which they are programming. It is what aids (does not have to be re-learned) any programmer fluent in one programming language in learning another distant language, without going through all that is required for a novice to programming.

By any "language machine" I mean those concepts peculiar to any given language, usually defined through its syntax and semantics. It is important to note that "language machines" are the vehicles by which "algorithmic design" concepts manifest themselves and as such they are inevitable to any discussion of algorithmic concepts.

The point that I would now like to stress is that the choice of real "language machines" such as Fortran, Algol, or Pascal is often inappropriate as the vehicle for introducing algorithmic concepts. The student must absorb too many concepts and details before being able to start any constructive thinking, something which is beyond the time tolerance of most fashionable, term-oriented courses.

One solution to this problem is that of looking for new programmable machines which are more tangible, less complicated, and therefore more suitable as a vehicle for discussing the "algorithmic design" concepts. Such a "language machine" would be considerably simpler to master, and if it has certain necessary pre-requisites then the instructor would be better able to show some of the constructive power of programming early in the course. One such machine was reported in [1]. In the rest of this paper, we shall study another such machine that I have used with reasonable success.

The Maze Machine

The aim in programming a maze-machine is to supply a sequence of instructions (a "program") which will take a person though an arbitrary maze from a "start" point to an "end" point. To do this we must assume a maze exists and must list the instructions which a person walking the maze can understand and obey.

Towards the first point, students are given a number of maze configurations. A typical maze is shown in Figure (1). The "start" position and orientation of the person are also shown. Students are told that the person walking through the maze is capable of understanding and executing only two instructions: STEP and RIGHT. A person obeying the STEP instruction advances in his present direction by one unit of the maze without changing his direction. He will turn to the right by 90 degrees, without changing his position in the maze, upon execution of the RIGHT instruction. We will refer to STEP and RIGHT as "basic capabilities". Formulations of the maze problem using other basic capabilities are discussed in Appendix I.

1. Sequencing

At this point students are asked to write a sequence of instructions (a program), using only basic capabilities which, when obeyed by the person, will guide him from the "start" to the "end" point of a specific maze (see Figure (2-a)). By varying the maze and/or the set of basic capabilities (see Appendix I), the importance of following a particular sequence of instructions can be emphasized.

2. Procedures or Subprograms

After seeing only a few programs students are already aware of the need for more powerful instructions. Repetitive use of three RIGHTs to perform a "left" and of consecutive STEPs for multi-step forward movement is a nuisance which gives the instructor the occasion of permitting them to use such more powerful instructions. Nonetheless, by way of pointing out that the person's basic capabilities are not changed, students are required to describe these more powerful instructions by separate smaller programs (subprograms). Such subprograms are referred to as "extended capabilities". It is explained that the person obeying these instructions is required to search a list of such extended descriptions on encountering an instruction that he does not recognize as one of his basic capabilities. If such an extended definition is found he is required to obey it and, upon completion, returning to the instruction following the "extended capability". Return to the calling point can be accomplished by use of the END instruction in the subprogram.

The reader should now appreciate the ease with which the concept of a subprogram can be introduced this early in the course. Those who prefer top-down approach can also benefit from this by proper reversal of the presentation sequence. Figure (2) shows the basic program and its more interesting form, in which extended capabilities are used.

3. Looping

So far things have been rather dull. The first step towards eliminating some of this dullness can be accomplished by writing programs for mazes with repetitive structure.

Figure (3) shows one such maze, consisting of three repetitions of the maze in Figure (1). The use of a do n times ... construct in this paper is quite arbitrary, and in fact the instructor will be better off using a construct more similar to the one that he intends to use for the main language of the course.

4. Conditional Statements

If the programmer, when writing his program, is not aware of the number of repetitions required, or if he wants to write it for a class of mazes with a varying number of repetitions of the same format, then he must expect from the person executing his instructions some co-operation in inspecting the maze and in returning information about it to his program.

This is a very important concept which will be used later when generalizing the maze algorithm. But for the time being it is sufficient to expect that the person be able to realize, at least, if he is out of the maze or not.

Representing this by the boolean basic capability OUT?, the program can be generalized into any number of repetitions. Figure (4) shows one such program for any number of repetitions of the maze shown in Figure (1).

5. Extension of Boolean Capabilities

For the sake of unifomity we also allow the students to write extended capabilities of boolean type. Under this scheme an extended capability will return to the calling program with a TEND statement if the condition under investigation is found to be true. Similarly it will return with a FEND statement for false cases. Figure (5) introduces a new extended capability, namely IN?, which is the complement of the basic capability OUT?. It is further used for writing the maze program of Figure (4) in a new way, as shown in Figure (5).

6. Generalized Maze Algorithm

As the last step in the process of building more general algorithms, students are asked to write a program which will guide the person through any maze subject to the existence of at least one path between the start and end points. Compared to the steps taken so far this is a giant step and will probably shock some of the students (especially the more intelligent ones). The fact is that it is possible and can be managed rather easily (though perhaps not very efficiently) with the capabilities which have been discussed so far, supplemented by one other basic capability which will be introduced now.

In Section (4) we discussed the need for testing a property of the maze and feeding the results back into the execution sequence of the given algorithm. Now we require that a person also be able to inspect the possibility of further progress in the maze by one unit in the direction he is currently heading. This new basic capability will be represented by the boolean basic capability FRWD?. With this added capability students should be able (in principle) to write a program which will guide a person through any unknown maze.

For those who can not write such a program on their own, the instructor may describe the "right hand" algorithm which enables any person to cross an unknown maze by constantly trying to keep in touch with the wall on his right (or left) side while walking forward. An implementation of this rule using the capabilities discussed so far is shown in Figure (6).

Subroutines <u>RSTEP</u>? and <u>LSTEP</u>? respectively test whether stepping to the right or left is possible. They do not affect the position or orientation of the person. Subroutine <u>RET</u> performs a 180 degrees rotation without affecting the person's position.

The main algorithm always tests the possibility of stepping to the right, forward, or to the left, in this order and steps in the first possible direction (the right hand rule). After each step a test is made for the possibility of having exited the maze.

7. Summary

The material in this paper has been presented in the order in which I usually present it in class. The number of lectures varies according to the level of the students and is usually between two and six lecture hours. Some obvious details and personal touches have been left out. They are the sort of thing which should be worked out according to the taste and style of the instructor.

Undoubtedly the most important step in a student's progress is that of writing the general program in traversing a maze. It is precisely here that the constructive side materializes and it is very important that the instructor emphasize the fact of constructing apparently very intelligent machines out of very unintelligent and sometimes dull instructions.

A simulation program, displaying the maze and the movements of the person in it on a video display, has proved to be a useful tool in teaching of the course and also debugging of the maze programs.

I believe that both this maze-machine and the one reported in a previous paper [1] are only two examples of a probable wealth of useful machines that exist and could be exploited profitably.

References

Mavaddat, F., "An Experiment with Teaching of Programming Languages", SIGCSE Bulletin, ACM, Vol. 8, No. 2, 1976.

Appendix I

Other possible basic capabilities are { STEP , LEFT}, { 2STEP, BSTEP , RIGHT } and { LEFT , BSTEP }. Here LEFT has its obvious meaning and 2STEP and BSTEP are used to step forward by two maze units or backward by one maze unit, respectively. The 2STEP, BSTEP, combination is particularly useful in cases where the person has to be advanced by an odd number of steps.

Useful exercises can be designed for writing extended capabilities based on these new basic capabilities. Finally, it is worthwhile to ask the students to rewrite the main program for the generalized maze, using the left hand rule.

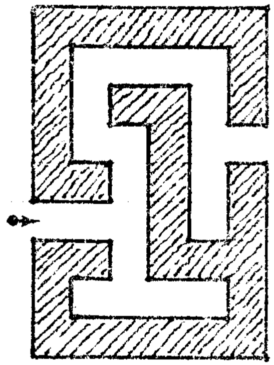

FIGURE (1) - A typical maze

```
STEP;
STEP;
STEP;
RIGHT;
RIGHT
RIGHT
STEP
STEP
RIGHT
RIGHT
RIGHT
STEP
RIGHT
STEP
STEP
RIGHT
STEP
STEP
STEP
RIGHT
STEP
STEP
RIGHT
RIGHT
RIGHT
STEP
END
```

(a)

<u>LEFT</u>

```
RIGHT
RIGHT
RIGHT
END
```

<u>STEP2</u>

```
STEP
STEP
END
```

<u>STEP3</u>

```
STEP
STEP
STEP
END
```

<u>MAIN</u>

```
STEP3
LEFT
STEP2
LEFT
STEP
RIGHT
STEP2
RIGHT
STEP3
RIGHT
STEP2
LEFT
STEP
END
```

(b)

FIGURE (2) - A sequence of instructions to guide a person through the maze. (a) using basic capabilities only, and (b) using an extended instruction set.

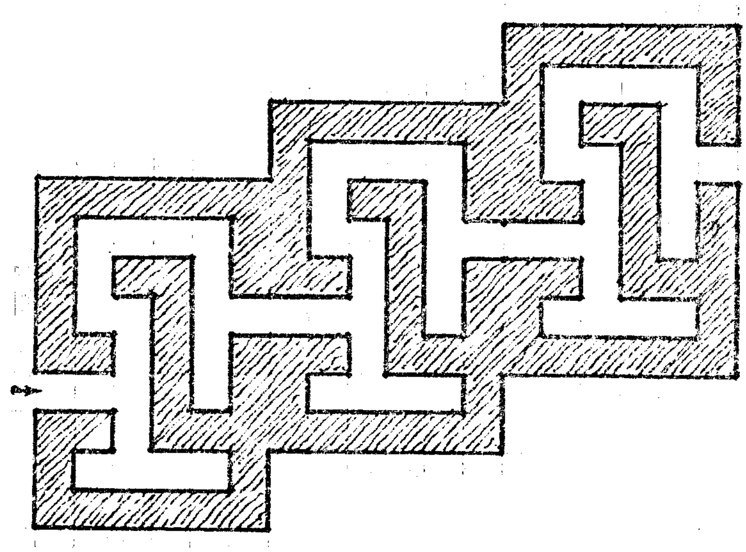

FIGURE (3) - Application of the looping concept to three repetitions of the maze in Figure (1).

```
MAIN

DO 3 TIMES

  STEP3
  LEFT
  STEP2
  LEFT
  STEP
  RIGHT
  STEP2
  RIGHT
  STEP3
  RIGHT
  STEP2
  LEFT
  STEP

END
```

MAIN

```
STEP3
LEFT
STEP2
LEFT
STEP
RIGHT
STEP2
RIGHT
STEP3
RIGHT
STEP2
LEFT
STEP
OUT?  no
END
```

FIGURE (4) - Applying conditional capability to repetitive mazes of any length.

IN?

```
OUT?  no
FEND
TEND
```

MAIN

```
STEP3
LEFT
STEP2
LEFT
STEP
RIGHT
STEP2
RIGHT
STEP3
RIGHT
STEP2
LEFT
STEP
IN?  Yes
END
```

FIGURE (5) - Introducing extended conditional capabilities through the use of FEND and TEND instructions.

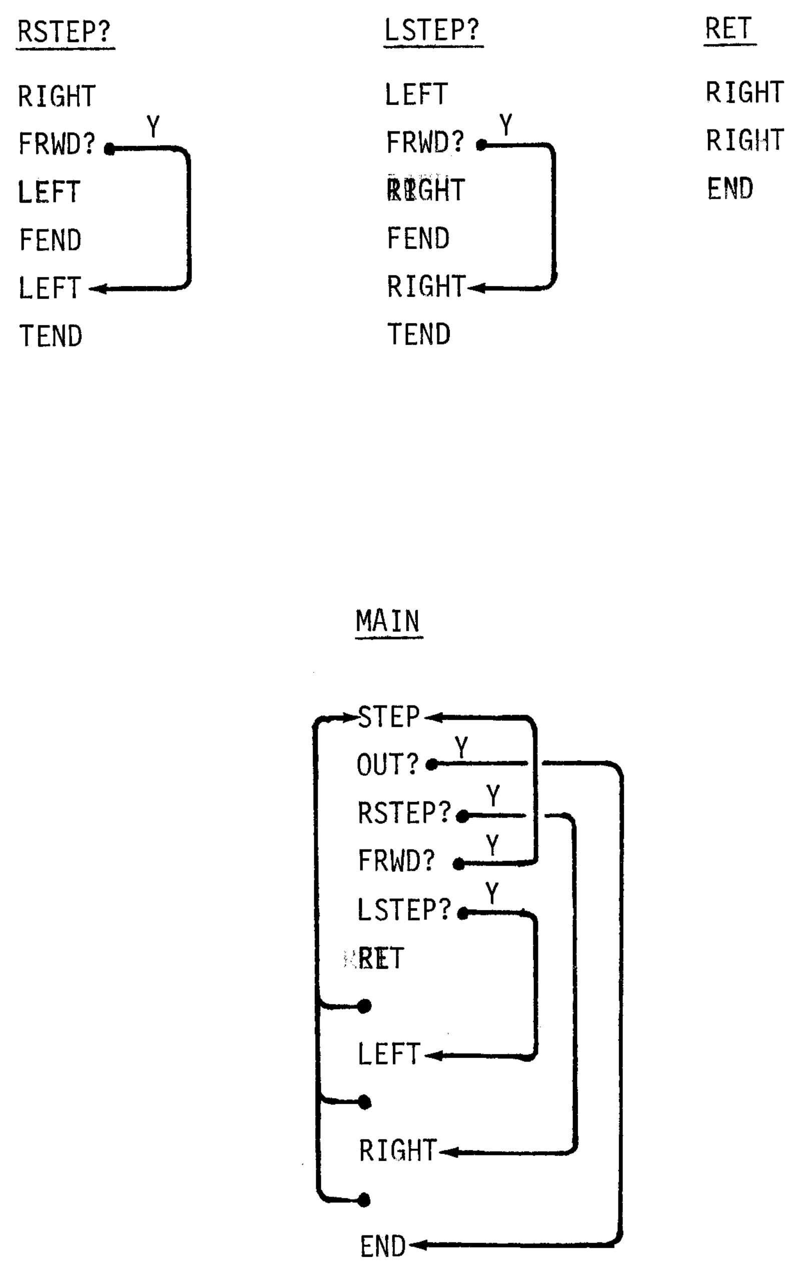

FIGURE (6) - The generalized maze algorithm