A modern beginning programming course*

by ROY F. KELLER
Iowa State University
Ames, Iowa

ABSTRACT

This paper describes a beginning programming course. It represents an approach to the "right way" to teach programming independent of any programming language. This is accomplished by thinking of programming as a two part process—(1) constructing an algorithm and (2) translating the algorithm into a program in some chosen programming language. Basic structured programming constructs are used for constructing an algorithm and translation is demonstrated by translation of control constructs into FORTRAN.

INTRODUCTION

The following quotations are a good way to introduce this paper:

(1) "Over the past decade, computer science has suffered a loss of innocence. No longer can a programmer be a gullible optimist, convinced of a program's perfection by success with a few chosen data. A program must be seen to be correct, and clarity has become essential. One of the keys to clarity is the set of control structures used, and the debate over the choice has been lively." Ledgard and Marcotty.1

(2) "One point should be made clear. We must distinguish between the programming language and the notation used while programming. One doesn't program in a language but into it." Gries.2

(3) "Yet it is fair to say that almost none of the elementary programming books say anything about problem solving, about orderly thinking, about expressing algorithms clearly and simply. The only conclusion to draw is that the students are not being taught how to program; they are being taught a language." Gries.2

(4) "It is insufficient to present the endproduct and expect the beholder to perceive its structure by inspection or even deep meditation. Instead, the beholder must also see at least part of the programmer's thought process, starting from the original (very abstract) version and proceeding to the end product via a clearly presented sequence of clear transformations and refinements." Denning.3

(5) "We have outlined three views of what a program should be. The final form of development is suitable for communication to a computer; a carefully documented sequence of forms is suitable for communication between programmers; a proven sequence of forms is suitable for communication to posterity." McKeeman.4

(6) "I now believe that it was a fundamental error to feel that we could think in a programming language." McKeeman.4

(7) "I would like to see the S.P. forum redirected towards a less hopeless task than finding the perfect programming language or formally defining S.P. itself. If we take the definition to be simply the construction of efficient, readable, understandable, modifiable, and verifiable programs, we can discuss ways to globally reach these goals by educating the people who do programming. Since it follows from the axiom that no amount of construct-clipping will make the typical graduate of a "data processing" school even a potentially good programmer, we must find something that will. What can be said to be the proper use of goto's, conditionals, and global variables (and block structures, pointers, etc.)? What are general guidelines to follow with respect to procedures? How does one go about modularizing a task?" Flon.5

The ideas in these quotes along with those expressed in the many papers dealing with programming and structured programming6-12 form the basis for the following ideas in developing a beginning programming course:

a. The primary issue in programming is one of constructing a solution to a given problem to be programmed.

b. The solution should be "correct" and "readable." (Where correct means the solution solves the problem intended and readable means the form of the solution is "easily" understood by people.) Concepts of efficiency and reliability in programs come after correctness and readability. These ideas should come later in a more advanced course. However, a beginning course

* This work was supported by the U.S. Energy Research and Development Administration under contract no. W-7405-eng-82.
should promote practices which lead naturally to higher level considerations.

c. Constructing a correct and readable problem solution can best be done through a sequence of refinement steps starting with an abstract form and ending with a form easily translated into most any high level programming language.

d. In the refinement process problem solving constructs for constructing well structured algorithms are essential.

e. The purpose of a programming language is so that a problem solution can be expressed in a form understandable by a computer. A programming language is not often a good language in which to think or to document. It may be that expecting a programming language to serve both as a language (1) in which to think and (2) in which to make a computer understand is expecting too much.

A beginning programming course based on the above ideas has been taught since Fall 1974. This paper reports on the course contents and to some extent on its success as reflected by student and instructor surveys.

BASIC PHILOSOPHY FOR A BEGINNING PROGRAMMING COURSE

Based on the above, our approach to teaching programming was to focus carefully on what is essential to constructing a “well structured” solution to a problem in the form of an algorithm. What problem solving constructs are needed? Is there some hierarchy in such constructs so they can be developed naturally?

We divide programming into two parts:

Part I: constructing a “well structured” algorithm,

Part II: translating the algorithm into some programming language. (a program)

An algorithm is defined to be a problem solution, in the form of a sequence of statements which can be executed by a human using any devices at his disposal. A program is a problem solution in a form understandable by a computer. A “well structured” algorithm is to be a correct and readable one.

Dividing programming into the above two parts makes it possible to teach much of programming independent of any particular programming language. Part I is problem dependent and Part II is programming language dependent. In addition Part II can be taught by showing how to translate a particular Part I construct into appropriate statements in a chosen programming language. Thus the details of a language can be studied in the context of how to use it.

Our philosophy for a beginning course is to naturally develop problem solving constructs for constructing correct and readable algorithms and then show how to translate these into language statements in such a way to preserve the structure of the algorithm in the program. Thus we have “structured programming” in any programming language.

CONSTRUCTS FOR CONSTRUCTING ALGORITHMS

The details regarding the development of a set of constructs and pedagogically how to develop these can be found in Keller. Here we shall list them along with some explanation or reasons for the construct. (All constructs are developed in context of a problem to be programmed starting with problems requiring simple constructs and progressing to the more complex ones.)

a. Basic constructs (or concepts): value, constant, variable, identifiers (names), expressions (operators and operands), statements, assignment statement, read and print statements. The crucial concepts here are the “ideas” of a value, a constant, a variable and of value and name association. Initializing values and assigning values form the basis for programming.

b. Decision making constructs (control mechanisms): an action (execution of a statement); sequence of actions; if-then and if-then-else constructs; while-do and repeat-until constructs. These constructs can be represented by flow diagrams and developed naturally in the following order.

(1) action

(2) sequence of actions

(3) if-then: if C then S

(Where C is a condition and S is a statement or sequence of statements.) The concept is to do S zero times if C is not satisfied (F) or once if C is satisfied (T).
(4) if-then-else: $\text{if } C \text{ then } S_1 \text{ else } S_2$

The concept is to do $S_1$ if $C$ is satisfied; $S_2$ if $C$ is not satisfied. Always one, $S_1$ or $S_2$, is to be done.

(5) nested if-then-else: $\text{if } C_1 \text{ then } S_1 \text{ else if } C_2 \text{ then } S_2 \text{ else } \ldots \text{ else if } C_n \text{ then } S_n \text{ else } S_{n+1}$

The concept here is to select one sequence to be done from many (more than two). This represents a natural extension of if-then-else.

(6) while-do: $\text{while } C \text{ do } S$

The while-do concept develops naturally from if-then. If-then implies $S$ is done zero or one time and the while-do extends doing $S$ to more than once, i.e., iteration, while still preserving the idea of doing $S$ zero or one time.

(7) repeat-until: $\text{repeat } S \text{ until } C$

The concept is to do $S$ repeatedly until $C$ is satisfied. These are the $D$ and $D'$ structures (without the case construct) found in Ledgard and Marcotty that have been studied by Dijkstra and others (See Reference 1 and references therein for further details on control structures.)

(8) repeat-forever: $\text{repeat-forever } S$

The concept here is to iterate forever through the sequence $S$ which will contain one or more exit constructs of the form:

$$\text{if } C \text{ then exit or exit alone}$$

Exit is always to the statement following $S$. In a beginning programming course there is seldom a need for more than two exits. This construct is a natural extension of while-do and repeat-until and obviously includes them.

c. Modularizing constructs: (Blocks and procedures). This includes:

(1) the concepts of local and global variables and how to specify their scope using some form of a declare statement;
(2) procedures as independent blocks with information passed through the formal and actual parameters. (This includes function procedures.)

The essential point for the student to understand is that modularizing comes naturally when one constructs an algorithm in a refinement fashion.

d. Data structures and value types.

Throughout all of the previous subject matter simple data structures (simple variables) are implicit. Iteration is developed by introducing a one-dimensional array data structure, introduced by considering a problem to be programmed. Multiple dimensional arrays and multiple dimensional structures are next developed. At this point the goal is that the student see clearly that developing an algorithm starts with his visualizing the data structures which are appropriate to the problem and naming these structures in such a way that he has names for substructures. He then must write statements involving these structures which when executed will solve the problem intended. The above constructs are his means of getting the job done through a sequence of refinement steps.

Character and Boolean values are next introduced again by considering appropriate problems. These data values give rise to taking a closer look at expressions, in the sense of operators and operands, and expression evaluation.

e. Verification of algorithms.

Throughout the refinement process of developing an algorithm, verification of each refinement step is heavily emphasized. Most of this is "hand" verification, testing decision points and iteration termination. In some cases simple proofs can be inserted to show a part of an algorithm (module) is correct.

Each refinement step is followed by the programmer proving the refinement is correct either formally or experimentally by hand execution with trial data. If great care in verification is taken, then a minimum number of bugs are introduced in an algorithm and programmer time is used efficiently.

Refinement steps and verification are never tidy. Many trial refinements are tried, determined to be incorrect by trial verification and replaced by other trial refinements. This is what programming is about; the student learns by doing. This is how he learns to think about the process of constructing a program.

TRANSLATING ALGORITHMS TO PROGRAMS

All the previous ideas are based on problem solving constructs needed for constructing algorithms which are correct and readable. These are essentially independent of any major programming language.

At this point any programming language could be introduced and studied from the standpoint of translating problem solving constructs into programming language statements. Of course it would be great if the major programming languages had these constructs. Unfortunately they do not. Someday major programming languages will contain these constructs, however, there will still remain some translation effort necessary for syntax reasons alone. In constructing an algorithm the programmer can ignore many details that a computer requires, but, a person does not. He can use indentation, one statement per line, etc. to make the algorithm readable for a person. Many of these human readable forms may never be machine readable. So some translation will be required for machine readability. The point is that people read algorithms and computers read programs.

The translation process is really not very difficult even for a language similar to FORTRAN. We illustrate translation to FORTRAN of the decision constructs using the FORTRAN logical IF statement: (These and other translations can be found in one of References 13-17.)

a. if C then S

\[
\text{IF} (\text{C}) \text{ GO TO then} \\
\text{GO TO next} \\
\text{then S} \\
\text{next CONTINUE}
\]

Where \text{then} and \text{next} represent statement numbers and C must be translated to a logical expression in FORTRAN.

b. if C then \text{S}_1, else \text{S}_2

\[
\text{IF} (\text{C}) \text{ GO TO then} \\
\text{GO TO else} \\
\text{then \text{S}_1} \\
\text{else \text{S}_2} \\
\text{next CONTINUE}
\]

(Where \text{then}, \text{else} and \text{next} are statement numbers.)
c. while C do S

while IF (C) GO TO do
  GO TO enddo
  S
GO TO while
enddo CONTINUE

(Where while, do and enddo are the statement numbers.)

d. repeat S until C

repeat S
  IF (C) GO TO next
  S
  GO TO repeat
next CONTINUE

(Where repeat and next are statement numbers.)
e. Repeat-forever translation is obvious. Exit translates to a GO TO exit.

Translations for variables, constants arithmetic expressions and Boolean expressions are straightforward. Procedures and functions procedures translate to subroutines and functions respectively in FORTRAN.

Blocks do not translate directly into FORTRAN; local variables must translate into global ones unless a subroutine or function is used for translation. Other programming languages avoid many of the FORTRAN translation problems.

While the above translations represent good ones they are by no means the only ones possible. These translations are the most natural and keep the structure of the algorithm present in the program. By use of comments in the FORTRAN program the translations can mirror even clearer the structure of the algorithm. Our experience has shown this is not of much use provided the final algorithm and selected refinements are maintained along with the program for documentation purposes; again the idea is that people read an algorithm and the computer reads a program.

SUMMARY AND CONCLUSIONS

This type of course in programming using FORTRAN has been taught to approximately two thousand students mostly non-computer science majors by large lectures (150 plus students) and small (25 students) recitations. Lectures met twice a week; recitation once a week. Fall 1974 was the first time.

General reaction from students has been favorable. For the first time in our experience students feel they are learning to program and feel they can. About sixty percent of the students reported in a survey that they felt "competent" to construct a program in FORTRAN. In a survey reported on by Krietzberg and Shneiderman, the prefacce of their book, only ten percent reported they felt "competent" to construct a program. It seems clear that the emphasis on constructing algorithms by refinement and verification is what makes the difference.

A total of five faculty members and more than twenty graduate assistants have been involved. With few exceptions all agree that algorithm construction is most crucial and should be the emphasis in a beginning course. Most feel that regular size classes (about 30 students) would be much more effective than large lecture sections. It is very difficult to teach problem solving via large lectures.

Both students and staff feel that two languages are involved and this seems to create some difficulties for the student. The top students overcome this easily, but, others have troubles. It takes a while before the student realizes that the problem solving constructs are for thinking about the problem and its solution and the programming language is for translating into.

Further support for this type of programming course can be found in the conclusions of the paper by Ledgard and Marcotty. We list their four points here. More detail can be found in their paper.

1. “From the programmer’s viewpoint, theoretical results based on the conversion of one program form to another under restrictive conditions may not be practical significance.”
2. “The need for higher level (above D and D’) control structures remains unproven.”
3. “The utility of the goto is seriously questioned.”
4. “The utility of $D'$ structures over $D$-structures is supported.” ($D$-structures are: actions, compositions, if-then-else and while-do. $D'$ structures are: any $D$-structure, plus if-then, repeat-until and case statements.)

We feel all of these except the case statement are important for the beginning programming course. Case statements and other control structures (BJ, RE, etc. See Reference 1 for definition of these and other control constructs.) should be introduced in a more advanced programming course.

In our approach the argument for or against the goto is quieted for the goto never appears in an algorithm. If used in a translation the goto mechanically causes the correct transfer of control specified by the algorithm. Thus program structure and correctness are independent of the goto. Readability is present in the algorithm
refinement levels kept with the program for documentation purposes.

REFERENCES