SYSTEMATIC INSTRUCTION IN SIMPLE PROGRAMMING GAMBITS

Michael P. Barnett
Department of Computer and Information Science
Brooklyn College of the City University of New York

I. Introduction

This paper describes an approach to teaching programming that the writer follows in his sections of the main introductory PL/I programming course at Brooklyn College, in a forthcoming text - An Introduction to PL/I Programming, and in material being prepared for use in the local public schools. The subject is developed by reference to carefully paced examples of programs that use different programming "gambits", in isolation and in combination. The programs all relate to plausible applications. The gambits have been systematized, and an order of presentation established that matches the developing needs of professionals in other disciplines, whose potential use of computers is dominated by information processing rather than numerical computation.

The writer has been surprised by the extent to which gambits that seemed self-evident to people who learned to program in the past must now be pin-pointed and made the subject of individual explicit explanation. An example is given in Section II. The depths of non-comprehension are not always evident when a class of today's high school graduates are taught by methods that were viable a few years ago. The writer believes that his observations are not a local phenomenon, and that corresponding non-comprehension would be found quite widely if it was sought, among students and practicing programmers, with the potential for correction once it has been recognized.

Major areas of software development today require no numerical computation, and further areas need just the simplest arithmetical processes. The mainstream of the writer's approach to programming instruction requires no mathematical knowledge beyond the simplest rules of arithmetic, and uses even these minimally, leaving success open to the many students of negligible mathematical skill and experience (though possibly good but unfulfilled potential).

Sections III-V describe the successive modules of the writer's course, with the elements that can be added for the strong self-motivated student to allow immediate practical application of the knowledge that has been acquired, and to gain early insights into more advanced tactics of computer utilization. Section VI describes the kind of testing that the instructor uses at present, and comments on the results

Although numeracy need not be a technical prerequisite to much programming, the writer is seriously concerned by the problem of innumeracy, and hopes that the programming approach may reactivate interest in numbers, by students previously "turned off" by poor math instruction. An even greater problem, however, that may even subsume that of real mathematical comprehension is the area of poor verbal skill. In this regard, the writer places heavy emphasis on word problems and verbal explanations of the actions of individual programs. The mixed attitudes of instructors (whose training was math or math education) at the college and high school levels, to a verbal as opposed to a purely symbolic approach raises the question whether there exists a section of the math education community tending to project its own unease with words onto the new student generation. This in turn raises the question of whether computer programming in the schools should be just an extension and preserve of mathematics as it is taught and possibly compartmentalized at present.

Drill and reinforcement seem the key to success for many students, The writer is developing an approach to the mechanized production of instructional materials that are personalized to the interests and background of different target groups with an elimination of routine clerical effort. These methods will be applied to the production of materials to teach several subjects that include programming, supported in part by a grant recently awarded by the National Science Foundation CAUSE (Comprehensive Assistance to Undergraduate Science Education) program.

II. An example of incremental explanation

Introductory courses and books on programming usually concentrate on the structural rules of a programming language. Most of the individual examples and exercises in a typical account of programming make conjoint use of several concepts that, for many students today, really require separate explanation, illustration and reinforcement. It can no longer be assumed that a student will rediscover or reinvent the multitudinous gambits which occurred "naturally" in the past to the

typical person who learned the commands of a language, illustrated by a few programs that contained branches and loops.

The example of the need for incremental explanation that has surprised the writer most in the early part of his course follows the introduction of the assignment statement. It is necessary to explain in succession:

(1) initializing and incrementing the line number, in a program that reads a deck of cards, and prints one line for each card, consisting of the line number and card image;
(2) writing a program to read a deck of cards and print just the number of cards which were read;
(3) writing a program to print the total of the items in a single field of successive cards of the input deck;
(4) writing a program to print the average of these items.

Many students understand the programming concepts used for (1) to (4) when these are explained in turn, but cannot extrapolate from one to the next. For these students, the ideas are separable and distinct that:

(1) a line count can be initialized and incremented in each cycle of a program that reads a data card and prints what is in it;
(2) the same incrementing tactic can be used, omitting printout from the main cycle, and triggering printout at the end, to print what would have been the last line number, had the line numbers been printed;
(3) a variable can be incremented by a value read from an input card, instead of being incremented by 1 as it was for the line number;
(4) totalling and card counting tactics can be combined, using two variables that are incremented concurrently, to provide the two numbers needed to compute the average.

Once these ideas have been explained explicity, illustrated, and reinforced with a few variations, the students can use them effectively. The writer would like to believe that the inability of students to progress from (1) to (4) without explicit instruction results from a prior lack of challenge in imaginative extension of ideas, and that presenting examples of this kind of progression develops the ability of at least some students to extend ideas. There is some evidence for this. Later in the course, the students are given a simple program that reads the name of a customer, and the catalog number of a commodity, from each card of an input deck, and tallies the number of orders for each commodity by adding 1 to the element of an array that is indexed by the catalog number. Several students who took some time to grasp the progression from (2) to (3) were able to "invent" for themselves the tactic of reading the size of each order from its data card, and adding this to the appropriate element of the array, to compute the total quantities ordered.

It is quite easy, when teaching programming, to be unaware of the limited comprehension of large numbers of students. Many instructors will assume that a student who has grasped the example (1) will be able to deal with (2) to (4) without further instruction, and never find that this was not the case. Programming assignments tend to test just a small selection of concepts and, quite often, only the ability of the student to find someone else who can write the program. Proctored examinations and quizzes that go to an appreciable level of detail are time consuming to prepare, conduct and grade.

III. Strings, numbers, arithmetic and conditions

1. Getting started: Students are given a prototype job to keypunch and run on an IBM 370 from an RJE station, that contains a PL/C program to read and print the contents of a data card. This introduces the PROCEDURE, DECLARE, GET and PUT EDIT statements, the idea of an identifier, the idea of attributes, and the specific details of the CHARACTER attribute and the A format item.

EDIT I/O is used at the outset instead of LIST I/O for several reasons. The facility to format the output using PUT EDIT is satisfying and useful to students with even the minimum of programming knowledge. The appearance of floating point output of PUT LIST statements discourages many students. Making further progress contingent on an acceptance of floating point output causes a time consuming hangup over a detail that can be circumscribed and side-stepped, by use of PUT EDIT, for students to grasp independently, without limiting their exposure to other ideas. The further details of format lists can be presented for stronger students, without imposing this information on students who might be confused. Designing simple reference files for information processing is a widespread need among applications oriented personnel in practically every area of human endeavor. GET EDIT starts students in the right direction for this, and becomes effective for a surprising range of applications with little further instruction. From this strictly utilitarian standpoint, students will have to use EDIT I/O in the "real world", and the writer sees no evidence that an initial exposure to EDIT rather than LIST I/O slows the progress of a class, or creates significant debugging problems.

2. Producing printouts: The one-card printout is extended to listing a deck with one complete card image per output line. This introduces statement labels, the GO TO, ON ENDFILE and STOP statements, and flow diagrams. Printing from individual card fields and reformatting the data for output requires explicit and reinforced explanation that the order of items on the input cards must be matched by the order of items in the input data list; that the order in the output must be matched by the order in the output data list; that the orders of input and output can be different; that data items can be contiguous on input cards; and that punctuation can be elided in the input and inserted by the program in the output. Printing data from one card on successive lines of output; combining data from several cards in the same output line; and treating data from successive cards in other cyclic ways require explanation - and the writer has found this with sophisticated practitioners of humanistic research, as well as with urban undergraduates.

The provision of headings, footings, and connective phrases in detail lines introduces string constants in data lists, the INITIAL option and assignment statements. The concept of cyclically updated predecessor values can be introduced by reading a chronologically ordered file about monarchs, and printing one line for each that contains his name and his predecessor's.

The idea of a parameter card can be introduced, e.g. carrying a slogan to be printed after each message from successive data cards. Simple form document production, inclusion of page headings (introducing the ON ENDPAGE statement), and non trivial cyclic processes such as printing a seating arrangement in which spouses have two people between them, from a deck in which each card contains the names of a married couple, can be presented at this point for students with stronger motivation.

3. Introducing numbers: The module that has just been described allows the student who is uneasy about computing a 5% discount on the price of a case of liquor or cosmetics to survive, uncompounded by arithmetical woes, the initial interaction with keypunches, JCL and overabundant printout and to start on some numerical work with a degree of confidence in being able to get programs to run. The FIXED attribute, the F format item, and the + and - symbols are enough for a range of applications that include (i) performing a simple sum with the data from each input card, without holding data or results from one card to the next; (ii) totalling data from an entire deck; (iii) line numbering and card counting operations; (iv) using data from a parameter card in each of the contexts (i)-(iii); (v) using predecessor values, e.g. to compute the distance along a highway of each town from the town before it. Multiplying and dividing lead into the use of decimals, consideration of size, accuracy and order of operations, and examples that are a little more involved computationally, again using the concepts of (i)-(v) explicitly. The simple built-in functions give a good return on time invested in illustrating their application.

Most students can grasp the ideas of floating point numbers without too much trouble. One class of exercise that interests the math oriented student, and which can also arouse the interest of other students in math, is experimental demonstration of the validity of mathematical identities, using exponentiation and, for the stronger students, elementary functions. Many students respond well to the use of numbers in format items, e.g. to print pictures or to plot graphs using coordinates read from data cards in LINE and COLUMN items. ON condition statements, triggered by overflow and other arithmetic conditions can also be introduced here for the math oriented student, to teach their use as part of computation from the outset, rather than as a special item "left over" from the formal programming course, and possibly never learned as a result.

4. The IF statement: The simple IF...THEN statement (without the ELSE clause) can be developed in the context of the earlier modules, with explicit attention to (i) uniform action for each input record (e.g. printing it conditionally) independent of other records; (ii) counting records that satisfy different criteria; (iii) matching an item read from the card with a value set by the program; (iv) comparing sets of values read from each detail card; (v) getting critical values from a parameter card; (vi) comparing values read from successive detail cards. Finding an extreme value and explicit loop control can be introduced here. The ELSE clause and comparison of strings then follow in the contexts used for the simple IF statement, and also allow variant actions on a final result.

The null DO group allows multi-statement success and fail actions. Simple examples include selection of data associated with extreme values of other items (setting the stage for later introduction of structures); use of group header cards; and the formation of group totals and cross footing, triggered by header cards and control breaks. Multiple IF statements open up further classes of application, including the use of propagating data in hierarchically structured files, and the interpretation of externally specified operations.

IV Loops and arrays

The DO-range statement is introduced with some inoffensively arithmetical examples (e.g.) printing a table of squares) and examples of cyclic formatting, (e.g. leaving blank lines at uniform intervals). The construction of a table of squares from second differences often intrigues an active class of even nonmathematical students. Using limits of a DO loop that are read from cards helps reinforce the unified role of numerical constants and variables. Using DO loops to draw squares, circles and other shapes catches the interest of many students, and motivates their attention to simple formulas.

The nesting of loops attunes many students to the power of generalization in a programming language. Cyclic spacing, formation of conversion tables (e.g. from miles, furlongs and yards into yards) without multiplication are well received. Computing tables of physical phenomena from simple scientific formulas can raise the interest of non-science majors in technical matters, particularly when these relate to environment, nutrition, energy and other topics of social concern.

The use of variable limits in a DO loop is illustrated by printing number combinations (e.g. for a lottery to select finalists in a competition) and by Cantor enumeration, that also holds the interest of many nonmathematicians. The use of DO loops in data lists should not be concealed from students who can benefit; neither should it be required knowledge. The DO WHILE statement provides an opportunity to demonstrate iteration - e.g. to computer a square root, to apply Euclid's algorithm, to find when increased frequency of compounding interest has no further significant

effect and, for the math oriented student, to find when numerical differentiation and integration converge to preset accuracy.

Identifying and systematizing the gambits made possible by arrays is a challenge. The writer begins with examples that simply read a subscript value from each data card, and print either the corresponding element of an array of numbers, or a quantity computed from this element (e.g. the price of a commodity identified by a catalog number, the social security number of an employee identified by a badge number, the Gregorian day number corresponding to given date). The introduction of arrays in this fashion needs to be illustrated liberally by examples from diverse situations.

Array I/O using DO loops around the GET and PUT statements are described and illustrated next. For the stronger students, the use of DO loops in the data lists, and unsubscripted array names, are described. Arrays of strings are illustrated by simple indexing examples, e.g. printing dates containing month names from purely numerical dates, printing verbalized car license information from numerical codes, and longer examples, e.g. verbalizing numbers up to a million.

Linear searching of arrays of numbers or strings is illustrated by examples from a variety of situations, and followed by examples of translation by table lookup (e.g. from an English to a Spanish month word). Tallying (e.g. finding the number of people born in different years from cards containing their birthdates) requires careful explanation and reinforcement. Tactics to store data that occupies variable portions of an array are described next.

Combinatorial examples using nested DO loops are popular and provide a good medium for inventiveness - tourney rosters, menus and the like; and calendars to illustrate non-commensurable cycles. For stronger students, the algorithmic processes of binary searching, merging and bubble sorting; simple examples of pointer lists, and representations of trees are mentioned here.

Multi-dimensional arrays are introduced in a parallel way to that used for one dimensional arrays. Multi-dimensional tallying seems to be an abstraction that gives difficulty to a significant proportion of students, and needs extensive drill and reinforcement.

V. Strings, structures and subprograms

The concatenation operator is introduced as a way to construct a string in an assignment statement for repeated use, so reducing the programs length. The SUBSTR function opens up a range of text manipulating examples, for cosmetic and other purposes. The STRING function and STRING option of the GET and PUT EDIT statements are mentioned for stronger students. Simple scanning tactics are described using the substring function and DO loops. The INDEX function is then introduced.

Internal representations, and the TRANSLATE and UNSPEC functions are mentioned for stronger students. The uses of bit strings as profiles of binary attributes, requirements and prohibitions are mentioned.

The usefulness of structures, and the syntactic conventions for them, are illustrated by the reduction of effort that structures permit in a program that prints an extensive body of data, selected by one item having an extreme value within a file of similar records.

Subprograms are developed likewise as a means of making programs shorter. User defined functions are presented as an extension of the kind of convenience that built-in functions provide. Examples are given in turn of numeric functions of a single numeric argument, numeric functions of a single string argument, string functions of a single numeric or string argument, functions of a single array or structure, functions of several arguments, functions that alter one or more of their arguments, and subroutine subprograms.

Considerable attention must be given to reinforcing students comprehension of the relationship between arguments and parameters, and their ability to invoke subprograms. It is possible for students to demonstrate an apparent understanding of the grammatical rules and actual output of subprograms in artificial examples, without really understanding what subprograms are about (beyond their use being a sign of virtue).

VI. Reinforcement and testing

Most students of programming in previous years would have derived little or no benefit from examples and exercises that were repetitious and patently isomorphous. Once a gambit was grasped it was not likely to be forgotten. Many students today seem to need precisely the opposite approach - massive sets of examples and exercises that are close parallels, for a tactic to come through as a general principle applicable say to any set of words identified by a numerical code, rather than something specific to eye color on driving licenses.

For testing purposes, the writer tends at present to use questions that describe a simple "real world" problem (or a humorous prototype), display the skeleton of a program for it, and ask the students to fill in any details or statements that they believe were omitted. There is merit to making sure that a student can write a complete program that starts with a PROCEDURE statement and ends with an END statement, and exam conditions provide the only assurance. Unfortunately this penalizes the slow writer if too many complete programs are required. Questions that just require the completion of a skeleton program help in this respect but make the papers lengthy. Some students, however, are disinclined to tackle a paper if it seems lengthy, a problem better resolved by training them to improved attitudes rather than mandating short papers.

The use of verbal frameworks for such problems has also been questioned in relation to poor readers. Here again it seems that programming and literacy will both be helped by setting and main-

taining standards. In any event, programmers who cannot work to written directives or explain what they have done are of doubtful usefulness. Another style of question that the writer has used displays a simple program, asks the student to explain briefly what it does (in the hope that the student will state "print a conversion table from pounds and ounces to ounces" rather than trying to write the complete output) and then give the specifications of another program that uses the same theme, and ask the student to write it. This again does not work with a student who does not know how to write a simple description, which does not mean that it should be abandoned.

The writer hopes that in fact by pursuing this kind of approach in programming instruction, standards of literacy can be raised, and that parallel tactics in mathematics and other technical areas may help reverse the tide of creeping illiteracy which is perhaps the most serious educational problem of today.