THE PROGRAMMING LOGIC COURSE IN THE DATA PROCESSING CURRICULUM

by Laura E. Saret
Associate Professor of Data Processing
Oakton Community College
DesPlaines, Illinois

Abstract: Learning the rules of a programming language is much easier than learning how to program. Learning how to program, that is, developing the logic necessary to solve a problem, is difficult for most people. Data processing students must unlearn the complicated thought processes they have become used to in dealing with everyday life and learn to tell the computer, in simplified terms, exactly what to do, step by step. This article discusses the role of the programming logic course in the data processing curriculum including reasons for teaching programming logic as a separate course, where the programming logic course fits in the data processing curriculum, appropriate course content, and teaching hints and strategies.

INTRODUCTION: WHAT IS PROGRAMMING LOGIC?

Learning how to program is not the same as learning a programming language. Learning a programming language is learning a set of rules. Learning how to program is learning how to formulate logical solutions to problems. Formulating solutions to problems is what programming logic is all about.

Learning the rules of a programming language is easy. Learning how to program, that is, developing the logic necessary to solve a problem, is difficult for most people. Data processing students must learn to "think like a computer." They must unlearn the complicated thought processes they have become used to in dealing with everyday life and learn to tell the computer, in simplified terms, exactly what to do, step by step.

This article discusses reasons for teaching programming logic as a separate course, where the programming logic course fits in the data processing curriculum, appropriate course content, and teaching hints and strategies.

WHY TEACH PROGRAMMING LOGIC AS A SEPARATE COURSE?

From my own experience as well as that of others I have spoken to, it seems that the first programming language is very difficult to learn regardless of whether that language is COBOL, BASIC, FORTRAN, PASCAL, etc. Subsequent languages are considerably easier to learn. The only explanation for this seems to be that when students learn their first language, they are also learning "how to program" that is, programming logic.

The purpose of the programming logic course is for the student to learn how to solve problems using a structured approach without being distracted by learning a programming language at the same time. Just as we learn to speak English before learning the rules of the English language, it is important to learn how to solve programming problems before learning the syntax rules of a programming language. If, for example, when you first learned to talk, someone corrected every grammatical error made, you would soon have become frustrated and stopped talking. First you concentrated on associating specific words with objects. Then you worried about such things as whether the verb matched the noun or the correct

pronoun was used. Similarly when first learning how to program, it is much less frustrating for a student to learn the logic involved before being corrected by an interpreter or compiler with regard to language syntax.

Many beginning students have difficulty distinguishing between syntax and logic errors. The first semester I taught COBOL, a student turned in an assignment which compiled with no errors. The output, however, was totally wrong. Since the compiler indicated that there were no errors, the student assumed that the program was correct. This is equivalent to an English student turning in a paper with no grammatical or spelling errors and insisting that the paper deserves an A even if it doesn't contain the content specified by the instructor. By separating the teaching of logic from the teaching of language syntax (at least at first), the student starts to appreciate that programming is not merely coding and that logic and syntax are not the same thing.

The programming logic course does not require the student to work on the computer. This has two advantages. First, if you do not have enough computer facilities for all your students, you have a course to offer the beginner which does not require a machine. Second, many students (particularly older students) are somewhat intimidated by operating a computer. They tend to worry about such things as breaking it, permanently destroying data, or some other catastrophe happening when they attempt to use it. If the student has to learn logic, syntax, how to operate a computer, and use text editors, etc. in the same course, it can seem overwhelming. By having the student take programming logic as a separate course, you have given the student some confidence in his or her ability to learn about computers and programming without using lab facilities.

If you have students transfer into your curriculum after having taken programming courses at an institution which does not teach structured techniques or after having been a programmer for a period of time, the programming logic course will teach structured techniques without forcing the student to repeat programming language courses. For example, we have had students enter our curriculum after taking COBOL or BASIC at another institution which did not teach structured techniques. Rather than have the student repeat the COBOL or BASIC class, we have him or her take programming logic and then start in the second COBOL or BASIC class.

Just as we learn to speak English before learning the rules of the English language, it is important to learn how to solve programming problems before learning the syntax rules of a programming language.

The programming logic course is a good measure of both aptitude and interest in the programming field. A student can learn very early in his or her coursework whether he or she really wants to be a programmer and can learn the proper thought processes that go along with it.

WHERE DOES THE PROGRAMMING LOGIC COURSE FIT IN THE CURRICULUM?

The programming logic course is a first-semester course in a two-year data processing curriculum. It is also appropriate as a first or second semester course in a four-year information systems curriculum. Where I teach, there is no prerequisite to the programming logic course. It is assumed that the student can use simple arithmetic tools, but no training in advanced mathematics is required. Logic is a required prerequisite for our COBOL courses, a recommended pre or corequisite for our BASIC courses, and a requirement for graduation in the microcomputer specialist curriculum. In fact, I have had students who had already completed two semesters of BASIC take the programming logic course and tell me that the logic course really helped to improve their programming skills and that they wished they had taken it sooner.

Students who have had an introduction to data processing course or some computer experience tend to be more successful in the logic course. However, with a 2-year community college curriculum, we don't have the luxury of postponing the logic course until the second semester. If you are teaching in a four-year school, that would be something to consider. For students with no data processing background, it takes about a month to "catch up" to those students who have had an introductory course. In a class that has students both with and without a data processing background, extra encouragement must be provided to the beginners so that they don't get frustrated and quit.

COURSE CONTENT

The course I teach at Oakton Community College has a practical, business emphasis. For this reason, it is not intended for the computer science student. It is geared for the student who will be either working as a business programmer or transferring into a four-year information systems program after graduation.

Topics include data processing terminology, introduction to program flowcharting, structured techniques, report and display headings, control codes and control breaks, totals, input validation, interactive input, extract programs, table processing, sequential file processing, nonsequential file processing, sorting, and use of multiple input and output files. The following describes each of these topics in greater depth.

Data Processing Terminology.

About half of our students have taken an introduction to data processing course prior to enrolling in programming logic. As a review for those students and an introduction for students with no data processing background, the first week is spent on introductory terminology. I discuss what data processing is, describe what a computer can and cannot do, discuss the

parts of the central processing unit and their functions, the role of primary and secondary storage, and the role of input/output devices. The data organization concepts of files, records, fields, and characters, and the differences among alphabetic, numeric, and alphanumeric fields are also covered. In addition, the definition of a computer program, and the steps in the programming process are explained at this time.

Introduction to Program Flowcharting.

Since many students have had some experience with flowcharts, and students tend to have an easier time learning a graphic technique to start with, program flowcharts are used to introduce programming logic. After discussing the definition, purpose, advantages and disadvantages of flowcharting, I cover the flowchart symbols and basic rules of flowchart construction. The approach at the beginning is unstructured, because students find structured concepts to be too difficult at this point. Introductory topics include input, output, and work areas in storage, reading records into storage, checking for end-of-file, simple looping, using counters and accumulators, naming fields in storage, field names vs. literals, printer spacing charts, the need to clear output areas, arithmetic, output editing and the difference between branching and subroutines.

In order to keep topics at an introductory level at this point, accumulators are only used for final totals, and report output ignores the need for headings and page breaks. In addition, opening and closing files is ignored. In other words, this introduction only teaches the basic concepts of input, output, and data manipulation.

Structured Techniques.

After briefly discussing the history, advantages, and disadvantages of structured programming, the various structures are introduced. Techniques such as flowcharts, pseudocode, HIPO, hierarchy (structure) charts, decision tables, and Nassi-Schneiderman charts are described. Students are expected to become familiar with and use hierarchy charts, flowcharts, and pseudocode. Some people feel strongly that only pseudocode should be taught. Others feel that only flowcharts should be taught. My feeling is that in the "real world", because organizations have different requirements, students will be expected to know a variety of techniques. Since it is impossible to master all of the structured techniques in a single semester, I have singled out the more popular ones.

Report and Display Headings.

When introducing flowcharting, the concepts of opening and closing files are ignored. This is the time to discuss those concepts as well as the role of initialization (housekeeping) and termination (end-of-job) routines. The various types of lines (heading, detail, and total) shown on reports need to be described. Methods for making the printer skip to the top of a new page and clearing screens prior to displaying headings are illustrated. In addition, the use of a lines counter to keep track of the number of lines printed or displayed, the use of a page counter for page numbers, describing heading lines as constants within the program, and various methods used to obtain the current date must be covered.

Control Codes and Control Breaks.

These topics are extremely important, because they are the basis for the use of report totals. When discussing codes, it is important to discuss why codes are used and why they are usually numeric. Control breaks that don't generate totals, such as breaks that cause a line to be printed or displayed, a heading to change, or a new page to be started are illustrated.

Totals.

I start with the use of a single total at the end of a report. At this point, students should be fairly comfortable with the use of counters. Single level totals are presented as a simple extension of counters. Next, single level totals with control breaks are discussed. Finally, the use of multiple level totals is explained. The steps to be followed at any level control break are listed and explained to make it easy for a student to learn to accumulate and print totals.

Two methods are used to prevent totals from being printed when processing the first input record. The first method involves updating the control field compare area with the contents of the control field of the input record as an initialization function. When processing the first record and comparing the content of its control field with the compare area, an equal comparison results, so no totals are printed. The second and less desirable method involves using a first-record indicator. I discourage students from using first-record indicators because they are inefficient. When using a first record indicator, every time a control break occurs, it is necessary to see if the first record is being processed. Since the first record is only processed one time, subsequent checks are a waste of time.

Input Validation.

This topic begins with a discussion of the types of errors that occur when translating and executing computer programs. This is an opportunity both to review the difference between translation and execution errors and to emphasize the need to test programs. After talking about what a validation program can do, the various types of validation are covered. It is important to remind students that they must check for all possible errors on a record, rather than stopping after the first error is found. Other topics covered here include the use of indicators to determine if a particular record or file has errors, and creating a file containing only valid records.

Interactive Input.

This topic covers programs which input data interactively and data validation of interactive input. When discussing

this topic, I discuss the difference between batch and online input and validation methods.

Extract Programs.

Extract programs are used to select certain records from a file to create a new file or report. Since extracts can be based on one or more criteria, ways to make the selection process an efficient one are discussed. Rules are given for determining the most efficient order for checking the conditions based on both AND and OR comparisons. Since my students are generally not very sophisticated mathematically, I have them memorize the rules without explaining the statistical theory behind them.

This topic is important to the database student in specifying selection criteria in database searches. It is important that the student understand that the order of specifying conditions will affect the efficiency of a search. For a large database, search time may be greatly reduced by correctly choosing the search order.

I cover the use and creation of parameter files to specify the selection criteria. Students should understand that if the selection criteria are specified in the program, every time the criteria change, a programmer must make the changes and the program retranslated. If the selection criteria are specified in parameter files, the user can change the criteria without getting a programmer involved.

Table Processing.

I begin with a discussion of tables used outside of data processing. I motivate students to learn table processing by demonstrating how the use of tables will make a flowchart or pseudocode considerably shorter by reducing the number of field names required. At this point I discuss the use of subscripts. Students have trouble using subscripts which are field names rather than constants. Both one- and two-dimensional tables are covered. Table initialization and methods for sequential and binary search are both explained. I discuss the use of tables in both validation and extract programs. Finally, I cover how tables are handled by various programming languages.

Sequential File Processing.

This topic starts with an explanation of sequential file organization. Then the various types of sequential files, such as master and transaction files, are discussed. The first type of sequential file processing covered is merging. In a merging program, two or more files are combined into a single file. At this point, we talk about the need to organize records in the same sorted sequence on both the master and transaction files prior to processing. The second type of sequential file processing covered is matching, that is, using information from the master file to process the transaction file. Finally, we discuss sequential file updating and the use of exception reports to show those transactions in which an update error occurred.

As difficult as programming logic is to learn, it is also difficult to teach.

Nonsequential File Processing.

Nonsequential processing is introduced to show students that not all processing is done sequentially (even though that's probably what students will be doing in introductory programming classes). It isn't necessary that beginning students have an in-depth understanding of all types of nonsequential processing. Since every language and access method handles nonsequential processing in a slightly different way, specifics for doing nonsequential processing will have to be learned when your students learn a particular programming language.

I begin with background information on why nonsequential processing is desirable. I then go on to explain access of records using their addresses. Both random and indexed file organization and processing are covered with an emphasis on indexed. Time permitting, I cover the various types of database organization as well.

Sorts.

Students can't assume that files are always in the correct sorted order. This topic discusses several methods of sorting including embedded (internal) sorts, utility sorts, and various programmer-written sorts. Indexing is introduced as an alternative to sorting.

Multiple Input and Output Files.

This topic is integrated throughout the course when discussing topics such as the creation of both report and file output in validation programs, spooling when more than one report is created as output, and use of multiple input files in file processing.

HINTS FOR TEACHING PROGRAMMING LOGIC

As difficult as programming logic is to learn, it is also difficult to teach. Teaching the logic course requires the patience to repeat the same explanation many times in many different ways. It requires seemingly endless paper grading. It requires putting together numerous examples and exercises to support the explanation of concepts. Because I believe that the best way to learn programming logic is to see many completely solved examples and that students need to learn by doing, I have written the book Data Processing Logic, which is now in its second edition, published by McGraw-Hill Book Company.

Grading, Exams, and Assignments.

Those of us who have programmed for a number of years find it very difficult to remember what it was like when we first started out. Concepts that seem second nature to a professional are very difficult for the beginner. Because students need alot of feedback and reinforcement in programming logic, I give 5 tests during a semester and assign 11-12 logic assignments to be graded. Equal

weight is given to assignments and exams. I also give "pop" quizzes. Many of the students who do not succeed in the logic course do so, not because they don't have the aptitude, but rather because they are not keeping up with the course on a daily basis. By giving "pop" quizzes (which cannot be made up), I am hoping to improve student success by encouraging students to both attend class and keep up their studying on a regular basis.

I never ask a student to construct a program's logic from scratch on a test, because I find that most students cannot do that given the time and other pressures of an exam. I do, however, include flowcharts or pseudocode for students to modify and/or answer questions about. Basically, I use exams to test terminology and concepts. All exams are closed book and give me a good indication of whether a student is doing his or her own assignments.

When determining final grades, I allow the student to drop his or her two lowest assignment grades. I include a mix of pseudocode and flowcharts on assignments and usually require construction of hierarchy charts. Two assignments are usually designated as group assignments. Students are required to work in groups of 2-4 students which encourages them to share ideas and examine alternative solutions as well as simulates working on projects with other programmers.

Classroom Activities.

I divide class time among lecture, discussion, small group work, and student presentations of logic solutions. A substantial amount of class time is spent discussing examples presented in the book and going through exercises at the end of chapters. Many exercises are assigned to be discussed in class and are not graded. Students find that "playing computer" using sample data to show exactly how the computer will process the data helps in understanding examples and exercises.

Several times during the semester, I divide the students into groups and have them work on problems during class time. These exercises are not graded, but serve to build confidence prior to doing assignments that will be graded. In addition, they show students that there is more than one way to solve a problem and improve the group problem solving skills which are so important in a business environment.

When reviewing assignment solutions in class, I ask students to present their solutions and request class members to make suggestions and criticisms. Students learn programming logic by being asked to "defend" their logic solutions. These presentations also allow students to see more than one solution to a problem and improve oral communication skills.

Two techniques I have found to be very helpful in teaching are using everyday (as opposed to data processing) examples and asking students to manually solve problems. One of the first activities I do is ask students to build paper airplanes. Then I divide them into pairs and without showing the other person their airplane, one member of the pair writes instructions for building the airplane. Based on the instructions only (that is, they are not to see the finished airplane or ask questions of the person who gave them the instructions), the other person then builds the airplane designed by his or her teammate. This illustrates the importance of instructions being clear and in sequence, as well as provides some fun in class when everyone attempts to fly their airplanes.

Other beginning activities include asking students to draw flowcharts for such things as brushing their teeth (it's amazing how many students don't take the top off the toothpaste!), backing a car out of a garage (I have alot of students who go right through the garage door without ever opening it!), getting ready for school or work, preparing for an exam, giving a guest directions to get to their house, and so on.

When discussing constants (literals) vs. variables (field names), I bring an empty box to class. I then put various objects in it. The box is used to represent a variable; the objects are used to represent constants. I then talk about how the box can contain various objects, but that the objects never change.

When covering the MOVE or ASSIGNMENT instruction, I have the students actually move around the room. I demonstrate how a single chair (memory location or variable) can hold only one student. When I ask a student to move from one chair to another, we talk about how the student being moved will replace the person currently in the chair he or she is moving to, regardless of whether the person currently in the chair is larger or smaller than he or she is. This illustrates that the content of the sending field replaces the content of the receiving field regardless of how many characters the sending and receiving fields contain. This analogy falls short, however, in explaining that the MOVE instruction does not erase the data in the sending area. This would require that I make a clone of the student being moved, so that I can leave one of him where he started from and put one of him where he is moved to!

I illustrate subroutines by sending someone to my office to do some task. I then talk about how the student returns to the classroom after completing the task, thus illustrating that a subroutine returns back to the calling routine. I take this example one step further by talking about how the person with whom I share an office could send a student from the classroom where he is teaching to our office to do the same task. His student would return to his classroom after completing the task. The above shows that the same subroutine can be accessed from more than one place in a program and that the program will know where to return after completing the subroutine.

When introducing structured techniques, students will feel that structuring complicates rather than simplifies the logic (it does at the beginning!). Discuss their feelings at this point. After all, you are teaching them a new way of looking at

things which they probably will object to.

One non-data processing example I use to illustrate the advantage of dividing tasks into subroutines involves my family's annual trip to Colorado to visit my husband's family. Since the trip always takes place during the summer, the items to be packed for each family member are pretty much the same from year to year. The first time we went, I spent a long time putting together a list for each family member (a subroutine) of things to pack. For each subsequent year I am able to use the old lists with minor modifications. I do not have to recreate the lists every year. In addition, the lists have been "tested", so to speak, so I know nothing is forgotten. Similarly, after creating a subroutine, the student will be able to use the subroutine in subsequent assignments, because all programs share many of the same tasks.

To carry the above one step further, I discuss how many of the items on each person's "packing" list were the same for all members of the family (e.g. toothbrushes, swim suits, underwear, etc.). To simplify my task I made one master list containing common items for the whole family as well as lists for each family member containing those items that are unique to that individual (e.g. my son's favorite pillow, my daughter's Barbie dolls, my husband's camera, etc.). In effect, I created a list (subroutine) for each of the four people in my family and a fifth list (subroutine) to be used by each of us. This fifth list can be compared to a commom module or routine because it is accessed by each of the other lists.

Students seem to struggle with the concepts of testing for an EOF condition only following a READ instruction and the use of an end-of-file indicator. I illustrate this using the example of calling directory assistance. When you call directory assistance in my area, the operator (or computer) will give you the phone number once (compare this to checking for EOF only once). If you record the number in your personal phone directory (an indicator), you can check the number anytime you want.

In the beginning of the course we read entire records into the computer and print entire records. We quickly progress to needing to treat fields individually and naming them. I use the following non-data processing example to illustrate this. If I sent a party invitation addressed to the SACKS (neighbors of mine), their entire family of five people would come to the party. This is equivalent to treating all the fields in the input area (everyone in their home) as one unit. On the other hand, suppose the invitation was addressed to STEVEN SACKS. Only Steven would come to the party, leaving the rest of the family at home. This is equivalent to assigning unique names to fields in the input area and allowing one or more of the fields to be moved to the output area and be written.

When introducing housekeeping and termination functions, compare the tasks to what you must do when you're giving a party. Before the party begins, you must do housekeeping. That is, you must clean the house, prepare the food, and so on. After the party is over, you have to clean up. In a computer program, housekeeping refers to the tasks you must do to prepare for processing. Termination functions are done after processing the input.

In covering opening and closing files, compare the computer task to the manual task of opening and closing file cabinets when using a file contained within the cabinet.

When discussing the need to define report lines in work areas rather than in output areas, I use the analogy of the chair in my family room that everyone likes to sit in to watch television. The chair (output area) can only hold one person at a time (comfortably). If someone is sitting in the chair, the only way another family member can use the chair is if the first person leaves the chair. While waiting to use the favorite chair, family members sit on the sofa (work area). Similarly in a computer program, there is only one output area which must be shared by all the output lines for a particular report. The work area is used to hold the lines that are waiting to use the output area.

When discussing tables, arrange the chairs in your classroom so that they are in rows and columns. Have a class discussion in which students must address each other by their row and column location rather than by name.

It is important to have your students manually solve problems. You will find that most of your students can manually solve any example or problem you give them. The difficulty seems to be in translating manual solutions into instructions that the computer can follow. Throughout the entire course, I stress and remind students that they can manually solve the problem that we are providing a computer solution for. When manually solving a problem I have students write down the steps they used. I sometimes have them trade these solutions with a classmate to see if the classmate can solve the problem using the instructions. This teaches the students to be very careful to include all the steps in the solution.

When discussing programs that include totals, start with solving problems manually. Have the students write down the steps. Then talk about solving the problem on a computer. Most students will roll totals when they solve a problem manually. After all, it takes less effort to roll the totals than to accumulate them for each input record. Remind them that whatever saves time manually will also save time when programming and when the program is executing.

In the discussion of sequential file processing, have each student create a master file and a transaction file on index cards. Have students process the files using the rule that once a record is re!d from a file, previous records on that file are no longer available. Using manual processing, students will be able to understand the need for sorting files prior to

sequential processing and the logic involved in sequential matching, merging, and updating.

When sorting, have students physically sort items using the various algorithms.

SUMMARY

The difficult part about learning to program is learning programming logic. Logic must be taught either as part of a programming language or as a separate course. Even students who already have programming language skills need to be taught to solve data processing problems in a structured manner.

I have taught programming language courses both when programming logic is a prerequisite and when it is not. When students have been exposed to programming logic prior to being taught a language, the success rate in the language course is greater, and considerably more material can be covered in a semester.

If you are presently thinking about developing a programming logic course, I hope you will find the material presented in this paper helpful. If you are currently teaching a programming logic course, the teaching suggestions described above will assist you in improving your students' understanding.

AUTHOR'S BIOGRAPHY

Laura Saret is Associate Professor of Data Processing and Coordinator of Faculty Development at Oakton Community College. She has an MBA from the University of Chicago with an area of concentration in information systems. Her undergraduate degree is in mathematics and mathematics education.

She has authored a textbook entitled, *Data Processing Logic,* published by McGraw-Hill (1987) and coauthored two textbooks with Peter Dublin: *Using Software Tools (Word Perfect, VP Planner/LOTUS and dBASE III Plus*) and *Using Software Tools (Word Star, VP Planner/LOTUS and dBASE III Plus),* published by McGraw- Hill (1988).

Prior to teaching at Oakton College, Laura was a programmer, systems analyst and operations research analyst at a pharmaceutical company.

Information Systems & Computing
Academic Professionals

STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 1055-3096