# STUDENT SUPPORT IN AN INTRODUCTORY PROGRAMMING COURSE

ELIZABETH LEVINE
BEVERLY WOOLF

COINS TECHNICAL REPORT 85 - 49

# Student Support in an Introductory Programming Course

Submitted to:

National Educational Computing Conference (NECC)

30 October 1985

Submitted by:

Elizabeth Levine
Academic Computing
Smith College
Northampton, Massachusetts, 01060

Beverly Woolf
Computer and Information Science
University of Massachusetts
Amherst, Massachusetts, 01003

Contact:

Liz Levine

Telephone:
(413) 584 2700 x3088

## 1. Abstract

One of the greatest weaknesses of introductory programming courses is the lack of cohesion within the course structure. Typically, students attend lectures with two to three hundred people, spend time searching for available terminals and programming assistance, and approach the assignments with a "do it the night before attitude". Not only is this self defeating but it is also pedagogically intractable. Given staggering enrollments, archaic hardware, and a lack of challenging tasks in the introductory programming curriculum, we have been motivated to develop inspiring screen displays, friendly operating systems, and a way for novice programmers to custom-tailor their assignments toward their own individual interests.

This paper describes an ongoing research effort to develop software, hardware, and courseware solutions to the conditions of the introductory programming class. Our solution includes graphic software embedded in Pascal, frequent structured assignments, and a microcomputer laboratory supervised by trained assistants. We described our basic approach elsewhere (Levine and Woolf 1984); here we report on progress made in the current year.

## 2. Support for Modular Thinking

We don't believe that the typical first year course is too hard for novice programmers, but rather that the environment is tedious and uninteresting. Assignments are often long and spaced too far apart and deny a sense of continuity in the student's acquisition of knowledge. Another problem is that the lab is often unsupervised and

confounded by a slow moving mainframe offering opaque system messages.

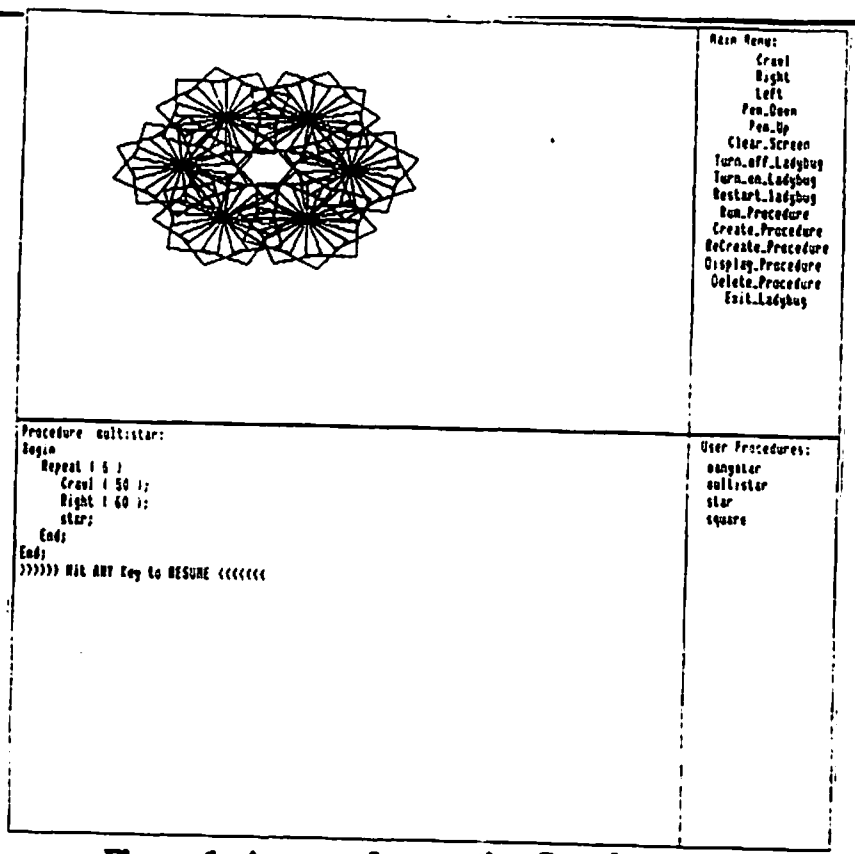


**Figure 1: A menu for creating Pascal programs.**

In response to these conditions, we have developed software packages designed to promote modular thinking via graphics examples. The packages have been tested and developed in a controlled environment for three years (Levine and Woolf 1984). They center around a programmable icon in the shape of a ladybug whose drawing capabilities are available at both an interactive and programming level. Primitives such as *Crawl*, *Right*, and *Pendown* are available through layered menus, Figure 1 displays the commands available at the initial stage of programming. Once a student is comfortable with these commands at an interactive level, embedded menus enable her to create procedures

interactively and to observe execution of these modules without the constraints of a formal programming language.

The screen display is split into three working parts: a menu, a scratch pad, and a programming tablet. In a typical session the student selects commands from the menu, and might, for instance, request that they be used in the construction of a "procedure". The system responds by writing a procedure on the programming tablet consisting of a header chosen by the student and a series of commands formatted according to typical programming convention. As the student selects commands, she is prompted for parameters (such as degrees or pixels) by an open parenthesis. The parameter list is automatically closed once an argument is provided, and the line of code is then formatted.

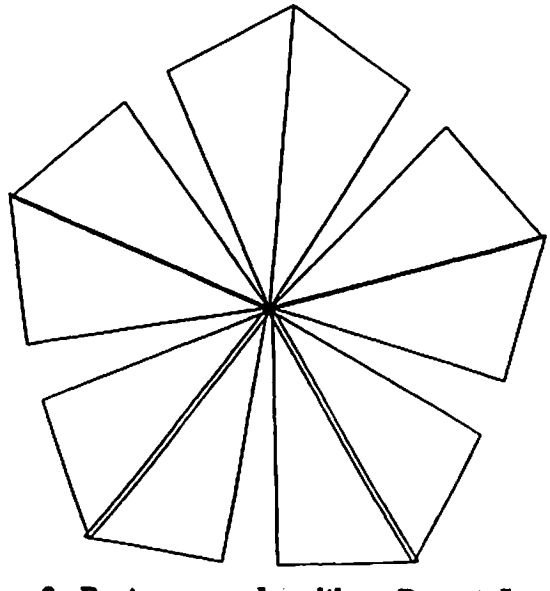

**Figure 2: Pentagon made with a Repeat Loop.**

Using this software embedded in Pascal, the student is exposed to Pascal formalism and indentation techniques while still residing in a relatively friendly error-free environment. For instance, the student can choose to "run" a procedure from the top-level menu, or nest it in a larger procedure, which she selects by invoking the command "Run_procedure". Figure 2 is an illustration of a graphic designed with a REPEAT loop. The results of this and other programming tasks appear on the scratch pad.

Although the simple graphics may use nothing more complex than a simple repeat loop, for the beginnning programmer this exposure to stepwise programming provides a solid foundation to the basic theory behind structured coding. The split-screen display mode facilitates viewing a program's execution in chunks, prior to the synthesis of the complete program. Thus the student is exposed to the benefits of driving procedures before learning the pre-knowledge that is often required in a traditional curriculum. This enables the student to relate top-down programming to the execution of complex tasks and to use procedures before becoming enveloped in the precision required for writing syntactically correct code.

An additional goal of this course is to facilitate individual expression in programming. Rather than have all the students solve the same problem, we have created lessons that allow a novice programmer to custom-tailor her assignments toward her own individual interests. Figures 3 to 5 are examples of solutions to such assignments.

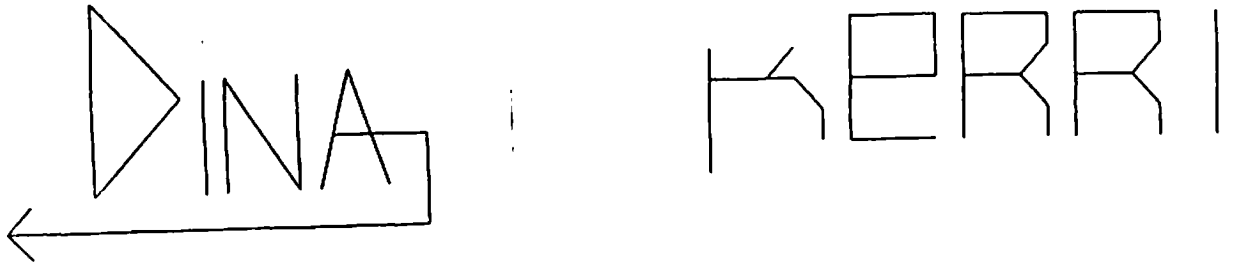


**Figure 3: A signature created with procedures.**

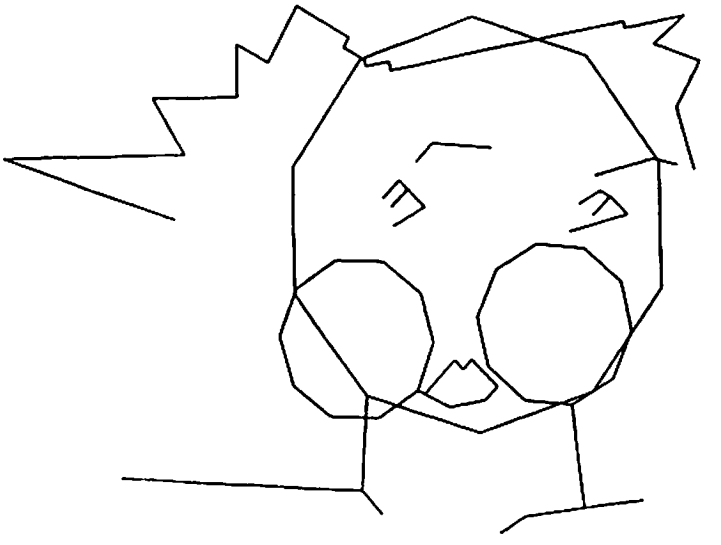

**Figure 4: A programmer's self portrait.**

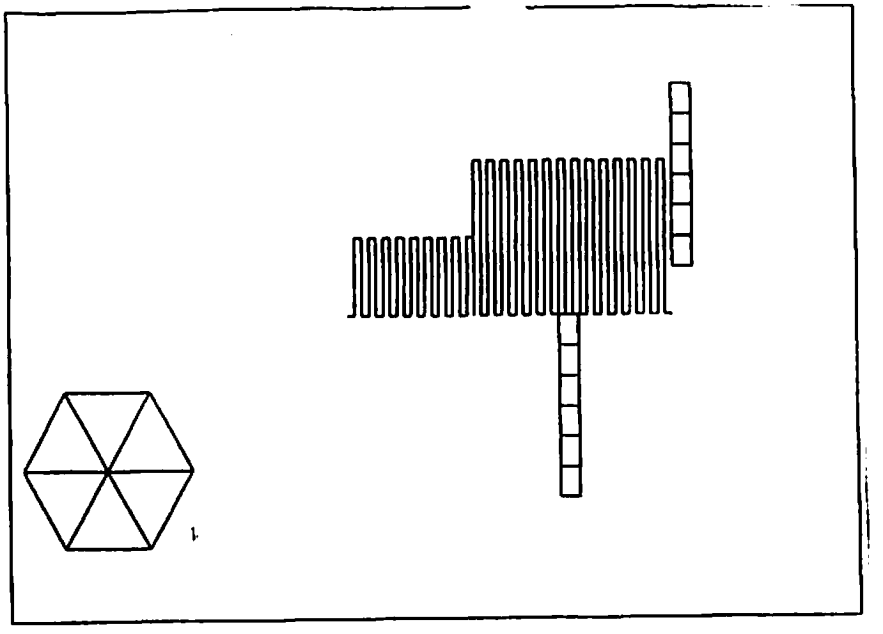

**Figure 5: Playground Created with Procedures.**

## 3. Support for Structured Teaching

Just as software created for this course stresses modularity and structured programming, the process of assigning student programs stresses modularity and structure. Each assignment specifies an early design date, by which a comprehensive design must be submitted. This design requires the student to consider such issues as:

- What variables will be needed in this program?
  Give Type and Var declarations
- Write an algorithm to solve this problem using mnemonic procedure names
- What do you want the user's interaction to be when accessing this program?
- Describe the data structure that will represent the user's responses.

The design task is worth 25 percent of the project grade, thus stressing the importance of completing the requirement. Once the design date has passed, students are given access to a SAMPLE DESIGN disk, which contains the design specications for the current assignment. Students who submitted weak designs, or no design are required to utilize the design on this disk. In this way, a student is not penalized but *supported* when her design for a particular assignment is not appropriate to the task. It is notable that those students who completed designs performed better on assignments than those who wrote the programs without prior design work. In fact, 75% of those students who worked on designs, received grades of 80% or better on the programming assignment. Help was available for students during the design process in the form of trained consultants and informal tutorials.

Along with structured programming assignments, we impose a structured calendar of tasks on students who are learning to program. Explicit tasks are needed because students come from a wide variety of academic backrounds and frequently, those from a non-technical backround have difficulty adhering to the deadlines and study schedules inherent in a programming course. Consequently, a parcel of material is distributed to students every two weeks containing a programming assignment, associated definitions for the current topics, a schematic overview of the structures under discussion, and a collection of examples which the student can study and alter if she wishes. The regularity of this parcel enables the student to recognize that solving a problem is a well-defined process, dependent upon a methodical approach. For a student, just preparing mentally for each new set of materials helps her become a part of this process.

### 3.1 Exams

Exams serve as logical breaking points for topics in the curriculum. We distribute a study guide ten days prior to the exam that serves as a pointer to those topics that will receive emphasis and to the specific *types* and program formats to be presented. We feel it is important to stress that exams should not confound the student by way of unexpected form or content. Regardless of the complexity of the material, testing materials should serve to elucidate problem areas not obfuscate them. In particular, the study guides ask students to solve short programming problems and to provide the pseudocode for longer ones.

The final exam is offered in such a way as to minimize student duress during a difficult time of the semester. All students are required to take the first hour of the exam. The second hour is optional. Those wishing to improve a possible grade outcome are encouraged to take this exam. Students who have performed well throughout the semester are free to concentrate on other coursework. The grade for the second exam is substituted for the poorest programming grade. We feel that this is appropriate only because the content of the exams stresses programming tasks rather than short answer questions.

### 3.2 On Line Courseware

The underlying principle behind the courseware is the concept of process. We want the student to recognize that programming involves several repeatable steps: design, revision, code, redesign, run etc. We support a student to engage in this design, redesign process on her own and to use tools that we have provided for independent use. Several banks of documented examples exist on a "class floppy" along with running solutions to programming

assignments and sample design specifications. Such courseware is on-line and available at all times.

The *examples bank* consists of executable programs demonstrating the use of a particular control or data structure. A source code directory exists on the same disk from which a student might print the source code. She can run the program with the code in hand. Examples serve as models for the student regarding documentation and formatting. New examples are added to the banks throughout the term; model student programs are used when appropriate.

Assignments are not distributed until a running solution (in executable-only format) is available on the SOLUTIONS disk. Students are encouraged to review these running solutions *prior* to coding the assignment. Grading specifications are particular regarding the characteristics of a completed assignment. A running solution provides the definitive example of these specifications. It is important for the student to watch a program run with code in hand, and to know that this is just one step in the structured process that is central to writing programs.

### 3.3 Student Centered Slides

Lectures in programming courses often do not describe the student's experience at the machine. We address this problem by using student centered slides that detail step-by-step responses the student can expect when she uses various commands. This is often a tedious chore for the teacher - yet, we feel that it helps to disabuse students of the "treading water" sensation that occurs later in the course if fundamentals are not clarified during the initial interface sessions.

As a student gains sophistication in programming, the lectures continue to address the actual laboratory situation; running programs are demonstrated and typical errors illustrated. Students view code through the overhead projector while watching the program's execution on an adjacent screen. Topics are usually motivated at the beginning of lectures with problems that necessitate a particular structure. Lectures conclude with the demonstrations.

## 4. Department Support

We have found it necessary to go to two departments for space, money and trained personnel needed to develop this course over a three year period. Initially the Department of Computer and Information Science (COINS), supported development of the courseware. The School of Education generously provided a temporary classroom and laboratory for the first semester until COINS could provide laboratory space. As in many computer science departments space is at a premium.

Both departments were generous in their support of an instructor for the course. The School of Education gave the instructor a grant for curriculum redevelopment. Subsequent programmers were given stipends by the COINS Department for specific tasks, such as porting the software to new machinery and building an on-line help system. Finally, COINS paid a head teaching assistant to supervise the other teaching assistants who were given course credit for their work. An additional position which has yet to be filled or funded is that of a systems and hardware person to handle the invariable repair and replacement of parts required by the use of microcomputers. The course has been given university status and made a regular Computer Science offering, thus obviating the need for support

from two distinct departments.

## 4.1 Computing Environment

The courseware and student operations have been moved to microcomputers to ensure quick response time. Students work on the same or related assignments in a laboratory supervised by trained assistants and are encouraged to present algorithms or flow charts to the assistant prior to beginning the first edition of an assignment.

Assistants can often detect misconceptions before extensive coding has taken place. They are screened for empathetic yet disciplined teaching skills and are trained using role playing sessions that simulate interactions with bewildered students. Familiarity with the machinery is acquired through hands on sessions as well as required reading of documentation. Assistants meet weekly with the course instructor to discuss current assignments, common questions and difficulties with current concepts. Also provided by the assistants are ongoing workshops dealing with advanced editing techniques and particular programming concepts such as recursion.

We have purchased microcomputers to support the graphics software on which the course is based. Within the environment there is an opportunity for graduate students interested in systems programming to transport and improve the software for use on the new microcomputers. In addition, students with teaching interests are invited to teach the course and to make suggestions toward incorporating more graphics systems into the curriculum. Future teachers of the course are pedagogically supported by accumulated on-line curriculum and parcels.

## 5. Unresolved Issues

Student attrition is still a major unresolved problem. Despite our efforts to make this a small, user-friendly course, programming itself seems to be a difficult discipline as displayed by a 25% attrition rate. We attribute this, in part, to a lack of cohesion on the part of the students regarding study habits and time management. In the past, our theory has been to support students through well-timed assignments, on-line courseware and supervised lab time. Now we believe we have to teach "tinkering" or the patience and discipline required to tinker and fix a program repeatedly before submitting it. Students often believe a program will work the first time it is run; the logical connective to this is that a program can be written the night before. Obviously students must be disabused of this idea. More importantly, they must realize that the actual running program is the last task in a lengthy learning process. We have not as yet developed curricula support that teaches this kind of perseverance and tinkering knowledge.

## 6. Future Projects

### 6.1 Software Extension to Data Structures

We envision an extension of our software package that will offer a graphical representation of arrays, records and files. For example a student will be able to access a record or array and observe the automatic loading of information into fields. The program will ask the student what type of structure she needs to create, and prompt for specific types of input. The record (or other structure), is then displayed on the screen, permitting

the student to observe the fields accepting values. A bibliography application is an obvious use of such a system. Such a program might prompt the student for the name of the book or article, the author, and the topic of the work. As the student enters the information, she will see a record displayed on the monitor, storing the information as she types it in. In this way the student experiences the feeling of constructing and loading a structure with information. This is important for the novice who frequently wonders "but how do you know it's there?", referring to the fields of a record, for example.

We are equally concerned with the support needed for more advanced students as they encounter the concept of the abstract data structure. A frame-based representation of abstract data structures, such as trees and linked lists, would facilitate the user's understanding of dynamic allocation and recursion. This representation would enable the user to construct a tree, observe the nodes being linked and view a binary search as it locates the item in question or stalls on a particular node. Frequently, the student new to abstract implementations has no concept of the term "independence of implementation", and insists upon coding deliberately without previous design work. Our graphics package would inform the student, using a selection of applications, how a linked list might be strung together dynamically for an ongoing library program, yet be better implemented with an array for a computerized research facility where the quantity of membership is known.

### 6.2 Networked Laboratory

Ideally, students and instructors should communicate via networked terminals. This enables students to access one another for help and examples; in a networked lab, they can depend upon rapid response. Instructors and students need to send assignments, questions and problems on the spur of the moment. A networked environment facilitates these tasks, as does distributed online tutorials. This means a student can access an example from a central account without utilizing a mainframe or waiting for the necessary floppy.

## 7. Distributed Teaching

The ladybug sSoftware will be utilized in the large (around 500 students per semester) introductory programming course that initially spawned its conception. We are not going in circles; rather this large course is going to transfer its base of operations to personal computers and will incorporate the laboratory based curriculum developed in the smaller graphics course. Each section of the course will strive to emulate the mechanics utilized in the prototypical experimental environment. This section details the methodology which we will use to maintain the flavor of a small microcomputer programming class within an administratively large programming course.

Our model of educational management is based on networks and distributed power. This approach differs from the hierarchical model used in many industries. Since teachers generally share the same goals of service and instruction, our model operates more like a network than a pyramid and we think of it as a collection of nodes (teachers) where each node has the problem solving knowledge to handle any teaching problem and each node is

fully capable of communicating with every other node. In this way, the pressure on one person, to minister to every difficulty is alleviated. In addition, if a student has a problem, she need not wait until the "manager" is available. Several people are equipped to help with the problem. We provide each teacher with enough information to manage the course and to communicate quickly via the computer with every other member of the team. In this way each member of the staff is able to diagnose deficiencies in the system, to anticipate disasters, and to design additional tools as needed. In the future, these tools will be rebuilt for a networked micro lab. Among the mainframe computer tools used to aid in the control of comunication are online practice exams, lesson assignments, messages about exams, help sessions, etc. examples, a grading program and a syllabus.

Several of these tools are designed to relieve instructors of purely routine bookeeping chores. In particular, a grading program allows each student to access her program and exam grades at any time and provides teaching assistants with the ability to keep track of class rosters and software libraries. The database of grades can be assembled to establish a meter or monitor for each teaching section.

## 8. Discussion

Our aim is to maintain an ongoing laboratory for research into software innovations for teaching, managing and supporting students. In addition, the laboratory setting is conducive to the collection of programming misconceptions and the formation of misconception classifications. Our lab is ideal for running half baked ideas and almost finished ones, for reworking software, for finding bugs, and for handing them again to the

students for evaluation.

We have described several software and courseware artifacts used to create cohesion and support in our introductory course. These artifacts include explicit graphic software, printed matter, interactive running solutions, student-centered overheads, online examples, and carefully trained consultants. By using the computer to assist in teaching and managing students we have created an environment that helps organize the learning process and helps us rethink our traditional definition of curriculum and textbooks. We have learned that consistent interaction between students, instructors and implementers can benefit development of sophisticated courseware and suggest that as future generations become increasingly dependent on software, more and improved uses of computers in education will be required.