ED 396 698 IR 017 874

AUTHOR	Kurtz, Barry L.; O'Neal, Micheal B.
TITLE	A Software Laboratory Environment for Computer-Based Problem Solving.
PUB DATE	94
NOTE	8p.; In: Recreating the Revolution. Proceedings of the Annual National Educational Computing Conference (15th, Boston, Massachusetts, June 13-15, 1994); see IR 017 841.
PUB TYPE	Reports - Descriptive (141) -- Speeches/Conference Papers (150)
EDRS PRICE	MF01/PC01 Plus Postage.
DESCRIPTORS	*Computer Assisted Instruction; Computer Graphics; *Computer Science; *Computer Simulation; Databases; *Educational Technology; Engineers; Higher Education; *Learning Laboratories; Problem Solving; Programming; Spreadsheets
IDENTIFIERS	Louisiana Technological University

ABSTRACT

This paper describes a National Science Foundation-sponsored project at Louisiana Technological University to develop computer-based laboratories for "hands-on" introductions to major topics of computer science. The underlying strategy is to develop structured laboratory environments that present abstract concepts through the use of computer simulations. These simulations allow students to explore meaningful, but domain limited, problems that are representative of real problems solved by computer scientists and computer engineers. The laboratories focus on: spreadsheets; relational databases; data structures; graphics; the imperative, functional, and logical programming paradigms; and digital logic. Types of laboratories that are candidates for future development include finite state automata; automatic theorem proving; and machine organization and assembly language. In order to insure that the laboratory experiences are useful from a pedagogical standpoint, a rigorous evaluation program will be conducted. Pre- and posttests will be used to measure the changes in, and generalization of, problem solving abilities. Standardized instruments for measuring student attitudes will also be used. (Contains 11 references.) (AEF)

Paper (W4-202B)

A Software Laboratory Environment for Computer-based Problem Solving

Barry L. Kurtz
kurtz@engr.latech.edu

Micheal B. O'Neal
Computer Science Department
Louisiana Tech University
Box 10348
Ruston, LA 71272
(318) 257-2436
mike@engr.latech.edu

Key words: computer literacy, computer science, simulations, problem-solving

Abstract

This paper describes an NSF sponsored project to develop computer-based laboratory experiences for "hands-on" introductions to many of the major topics appropriate for an overview of computer science. Our underlying strategy is to develop structured laboratory environments that are designed to present abstract concepts through the use of computer-based simulations. These simulations will allow students to explore meaningful, but domain limited, problems that are representative of real problems solved by computer scientists and computer engineers.

Introduction

The current student profile in our "computer literacy" course (CS100) is quite heterogeneous. About 40-50% of the students are computer science majors, while the remainder are drawn from programs throughout the University. Typically, 30-40% of the students in CS100 are women, while approximately 25% of the class is composed of ethnic minorities (predominately African American). These students represent a wide range of interests and abilities. Most have little or no background in college-level mathematics.

In the spring of 1992 we began an effort to adapt this introductory computer science course to a breadth-first approach. The goal of this approach is to give students an overview of the entire computing milieu. The challenge is to develop a course that is rigorous enough to prepare the computer science majors for the follow-on courses, yet, at the same time, be both meaningful and accessible to the large number of non-computer science majors who take the course.

There are several good textbooks available that present the major aspects of computing such as computer architecture, operating systems, algorithm development, programming language paradigms, databases, and networks. However, most texts lack an integrated laboratory component. To overcome this deficiency we have embarked on a National Science Foundation funded project (DUE 9254317) to develop a laboratory environment for a breadth-first computer science overview course. This environment is composed of a collection of software modules, collectively known as "Watson." We chose the name Watson to emphasize that the environment is to act as an assistant that helps the student explore various aspects of computing.

In the past, we attempted to have students complete small assignments using off-the-shelf software in an open lab environment. We found that our open laboratories, which are successful for more advanced computer science courses, did not work well for CS100. The students were overwhelmed by the syntactic details of the various software tools they were expected to use and many students ended up very frustrated. It does not have to be this way. With the proper hardware and software, students can be presented with a positive learning experience that increases their understanding of, and appreciation for, computing.

Background

Watson is designed to support a breadth first approach to computer science. Serious discussion of this approach started with the Denning committee report [Denning, et. al., 1989] that describes a three course sequence for a breadth-first introduction to computer science. This report was the starting point for the development of Curricula 91 [Tucker, et. al., 1991] that described the core areas of an undergraduate computer science curriculum as a set of knowledge units within ten major topic areas. Exposure of students to the breadth of computing is a philosophy pervasive in these efforts and in our laboratories. We have also adopted a philosophy of closed lab sessions and provisions for group work, as recommended by the Curricula 91 report.

The breadth-first approach has influenced courses and textbooks for computer literacy. *Computer Science: An Overview*, by Glenn Brookshear [4th. Ed., 1994], is an excellent text aimed at the same student audience as our laboratories. We are currently using this text in our CS100 course. It avoids the syntactic complexities of any particular programming language or software package and, instead, focuses on the "bigger issues." However, Brookshear's text lacks an integrated laboratory component. Another text, *The Analytic Engine*, by Rick Decker and Stuart Hirshfield, comes the closest to our project since it contains an integrated laboratory environment. They state that "students are both relieved that the course de-emphasizes programming and are interested to find out that there is more to computer science." [Decker, 1990, p. 235]. Another interesting approach is provided by Alan Biermann in his text *Great Ideas in Computer Science: A Gentle Introduction* [1990]. These types of changes to computer literacy instruction are discussed in an excellent paper, "The New Generation of Computer Literacy," by Paul Myers [1989].

A recent report, *America's Academic Future*, issued by the National Science Foundation [NSF, 1992], identifies many of the larger issues facing American education. One of the primary recommendations is to "Encourage the development of discovery-oriented learning environments and technology-based instruction at all educational levels." [p. 4] There is a particular emphasis on the use of "new communication, information, and visualization technologies." We strongly believe the goals and objectives of our project are in concert with these recommendations.

The Laboratory Environment

While it would be tempting to simply search the Internet for public domain packages that cover the topics of interest, we believe that such an approach would be doomed to failure with freshman-level students. At this level, syntax and "look and feel" issues are major hurdles. A freshman student presented with many unrelated environments would spend the majority of the semester just learning how to use the software environments, rather than actually solving problems with them. It is clear that a single, consistent, flexible software environment, specifically designed to support computer-based problem solving, is needed.

Our laboratory modules incorporate a number of guidelines we have developed based on our experiences introducing freshmen to computer science. These guidelines include:

- Present a uniform environment that minimizes the need for keyboard input and prevents the introduction of syntax errors.

- Provide a consistent help system that includes information on how to use the environment, as well as high-level problem solving assistance.
- Allow for both "tutorial" and "guided discovery" laboratory experiences.
- Allow for flexibility and ease of modification.

In addition, as a practical matter, the software must be able to run on a variety of computer platforms such as PCs and UNIX workstations.

We have observed that many CS100 students feel uncomfortable in front of a keyboard. The mechanics of typing and entering commands and correcting mistakes are difficult for some. A more widely recognized problem is the difficulty students have with programming language syntax. Our approach to these problems is to use syntax directed editors so that only syntactically correct "programs" can be entered, and to limit keyboard input wherever practical.

The help system in each of the laboratory modules has been designed to provide two types of assistance: information about the software laboratory environment, and problem solving assistance. To invoke the first type of help the student presses a button marked "What is?" and then clicks on the item for which help is to be provided. A popup window appears with a description of that item. The second type of help involves problem solving advice. Initially, when laboratory exercises are written, the faculty members authoring those exercises provide a collection of helpful hints to avoid common pitfalls. This advice is incorporated into the software and displayed whenever the student indicates their need for assistance.

In order for Watson to gain the widest possible acceptance, we do not want to limit it to any one pedagogical style. Instead, our goal is to build enough flexibility into the software so that it can be used with a range of pedagogical techniques from fully scripted tutorials to directed discovery. Flexibility is also important in another sense. It should be possible for instructors who use this software to adapt it to their own style with as little pain as possible. For these reasons, our software is organized into three distinct "levels". The first level, the "message level," is a keyed text file. The existence of this level allows instructors with limited programming ability to quickly change certain aspects of a laboratory exercise, such as the definition of terms, or explanation of a problem. The second level, the "activity level", is a C source program that incorporates a number of specialized functions that allow this level to sense and control activities at the underlying level. Instructors with a working knowledge of C programming should be able to change most aspects of a laboratory exercise without needing to learn the details of the underlying software environment. The final level is the "laboratory environment level" which actually defines all of the software objects that make up a laboratory environment.

A final requirement, that we established early on, was that all software developed under this project must run on a variety of hardware platforms, especially UNIX workstations and Windows-based PCs. The ability to run on Macintosh computers was also considered important. To meet these requirements, we selected SUIT, the Simple User Interface Toolkit [Conway, 1992]. This product, developed at the University of Virginia, provides a flexible, easy to use, extensible environment that allowed for rapid prototyping of the user interface and provided for cross platform compatibility. SUIT is currently available under X-windows, Microsoft Windows, DOS, and Mac OS, and is free of charge for academic projects.

Description of the Laboratory Activities

We are using our first year funding to develop eight laboratories. We describe two laboratories in detail below and provide a brief description of the others. Each lab has two levels of presentation: a concrete, hands-on component where students manipulate objects and observe results and an abstract level where the underlying theory can be used to produce the same results.

Spreadsheet Laboratory

This simplified spreadsheet program allows students to enter arrays of data and to calculate additional data based on existing data. Sample calculations are summation, average, and projection of values based on specified growth rates. Unlike a real spreadsheet package, this environment is tightly constrained and monitored. If the student starts going far astray in completing an activity, the system provides interactive help to get him or her back on track. The concrete level is the data as it appears in the table and that can be manipulated through mouse input. The abstract level includes the mathematical equations that define the relationships between data values.

Relational Database Laboratory

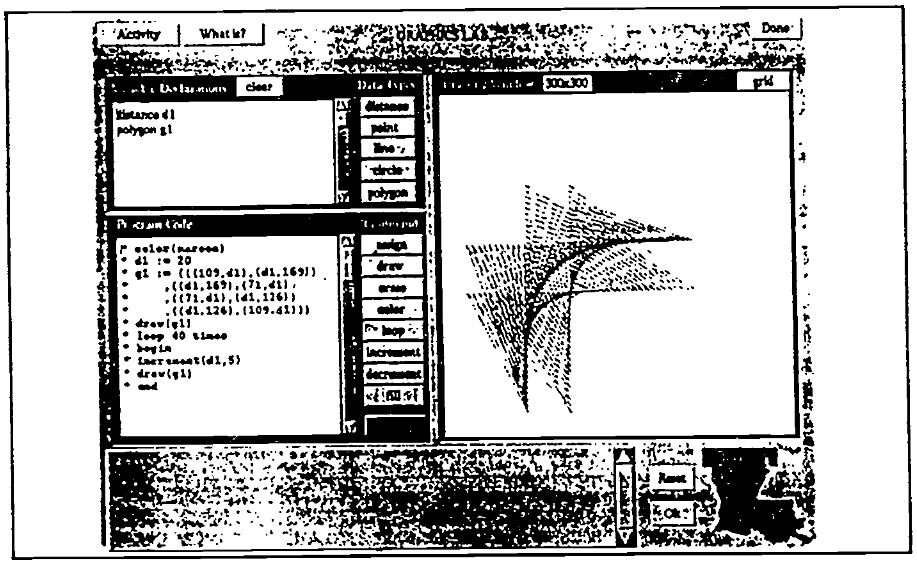
The main activities in this lab involve composing queries to answer specific questions concerning one or more tables of data. An initial academic database is provided with three tables: a student information table, a table of classes attended, and a faculty table that includes classes taught. The data has been simplified so that the student can focus on the primary activity: database queries. Queries involve one or more tables and use of the operations project, select and join. There are two modes of operation: query by example (the concrete level) and query by relational equation (the abstract level). With query by example selecting a column headings is equivalent to a project operation, matching two column headings from two tables is a join, and applying restrictions to values is a selection. As these concrete actions take place, the corresponding relational equations appear on the screen. During the second part of the assignment the student has to enter relational equations directly to obtain the desired results.

Data Structures Laboratory

The data structures lab is intended to familiarize students with the behavior of common data structures, such as stacks, queues, and trees. For example, a stack data structure is presented on the screen as a graphical object that may be activated by clicking on buttons such as "push" or "pop," the concrete level of operation. At the same time, a sequence of instructions is displayed. At the abstract level the student must solve a problem by entering a correct list of instructions and executing them. A sample task might be creating the reverse of a given stack.

Graphics Laboratory

The graphics laboratory introduces students to several fundamental concepts in computer science: declaration of object types, assignment of values to objects, and interactive manipulation of objects. A complete screen layout appears in Figure 1. The upper left window contains variable declarations, where choices such as point, line, circle or polygon are selected from a menu of push buttons. Objects must be declared before they are assigned values in the program code window. Commands such as draw and color produce changes in the drawing window. The bottom tutor window can display a description of the problem to be solved or provide assistance when errors in the student's code are detected.



```
distance d1
polygon g1
```

```
• color(maroon)
• d1 := 20
• g1 := (((109,d1),(d1,169))
•       ,((d1,169),(71,d1))
•       ,((71,d1),(d1,126))
•       ,((d1,126),(109,d1)))
• draw(g1)
• loop 40 times
• begin
• increment(d1,5)
• draw(g1)
• end
```

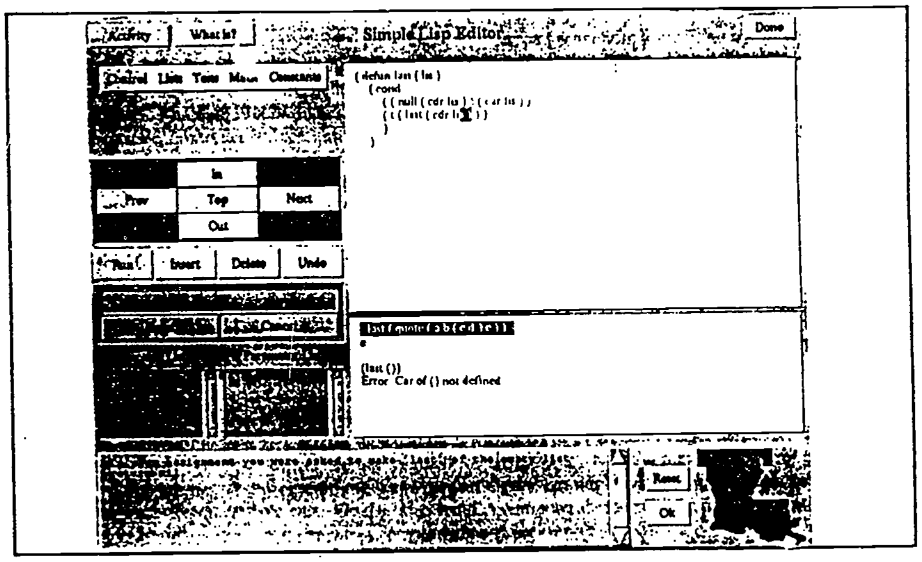
Figure 1. Screen Layout for the Graphics Activity

There is a duality between the program declarations and code (the abstract level), shown in the left windows, and the picture constructed in the drawing window (the concrete level). The student can "paint" a picture in the drawing window and watch the corresponding declarations and commands appear in the left windows, or the student can enter declarations and commands to cause a picture to be drawn in the right window. Students should be able to draw simple pictures using either approach.

The Imperative, Functional, and Logical Programming Paradigms

All three programming paradigms use syntax directed editors designed to allow students to enter syntactically correct programs and to minimize the use of keyboard input. Programming assignments are very short since the goal is not to teach the student to program well in any one language, rather it is to expose students to a variety of programming paradigms. The semantics of a program can be defined by its input/output behavior; we consider this the abstraction of the program. The actual implementation in a particular paradigm or using a particular technique (e.g., iteration versus recursion) represents the concrete level of understanding.

The imperative programming language is subset of a Pascal-like language with type declarations and commands for selection, iteration, and procedure invocation. Procedures with or without parameters can be declared. All entry is via an editor that insures the syntax and static semantics of the program are correct. Laboratory assignments do not involve lengthy programs, rather most focus on a few very small procedures designed to work together to accomplish a particular task.



```
(defun last (lst)
  (cond
    ((null (cdr lst)) (car lst))
    (t (last (cdr lst)))
  )
)
```

```
last (quote (a b (c d) e))
e

(last ())
Error Car of () not defined
```

Figure 2. The Syntax Directed Editor for Lisp

We use the Lisp editor, as pictured in Figure 2, to illustrate a syntax directed editor; the other editors are of similar design. The following pull down menus are available: Control (defun, cond), Lists (car, cdr, cons, list, append), Tests (eq, null, atom, listp, zerop), Math (+, -, *, /), and Constants (t, nil, quote, (<list>)). A function that is defined for the first time is normally entered in a top down manner by selecting a defun option, naming the function, specifying the parameter list and specifying the function body. Once the function name and parameters are specified, these names appear in the windows called "Function List" and "Parameter List". From this point on, references to function names and parameters are through selection, not keyboard entry.

Lab activities involve writing simple functions that may manipulate lists, such as the "last" function shown, or perform simple arithmetic operations, such as a factorial function. We do not allow any setq operations and we depend on recursion to perform repetition. The intent of this lab is to expose students to the "look and feel" of a functional programming language.

The logic programming laboratory will use a syntax directed editor for a subset of Prolog that is similar to the on described above for Lisp. Logic programming is introduced from a relational perspective, similar to the academic database used in the Relational Database Laboratory. Next the list manipulation facilities of Prolog are introduced. Some of the simpler functional programming activities are replicated for Prolog.

Digital Logic Laboratory

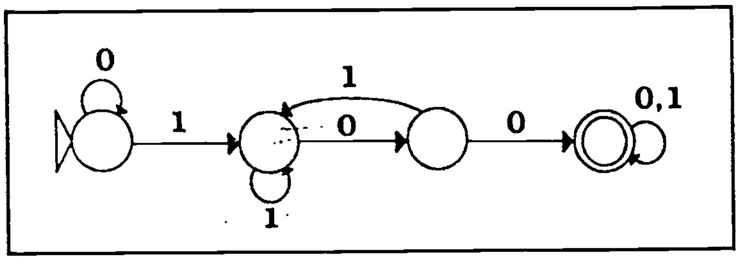
This laboratory covers two topics: combinatorial logic and sequential logic. The concrete level of operation are the gates and flip-flops used to build circuits; the abstract level is the Boolean equation that represents the behavior of the circuit (or the timing diagram for a sequential circuit). Students first build simple combinatorial circuits using AND, OR and NOT gates. The lab environment allows circuits to be named and then recalled as "black box" devices. For example, gates are used to build a half-adder which is stored in a user defined library. The half-adder is used to build a full-adder, which in turn can be used to build a four bit adder. A major goal of this lab is to be able to write a Boolean equation for any connection in the circuit or, given a Boolean equation, build the corresponding circuit. In the final combinatorial logic activity the student builds a latch in the form of a D flip-flop and then adds a clock signal to obtain the familiar edge-triggered D flip-flop. Shift registers, counters, ring counters, and parallel-to-serial converters are studied. Students are introduced to the use of timing diagrams to analyze sequential circuits.

Possible Laboratories for Future Development

We hope to develop additional laboratories to cover a broader range of topics. This would allow an instructor to pick and choose topics to meet the needs of a particular target audience. For example, a computer literacy course targeted for business or education majors may elect to cover the applications and programming language paradigms, but might not cover the hardware and theory oriented components. On the other hand, a computer literacy course for engineers may elect to de-emphasize the business components and include all the hardware-oriented components. Here we only briefly describe three of the laboratories that are candidates for future development and list some others.

Finite State Automata Laboratory

Students would work in a graphical environment where they can construct and test simple finite state automata. An example challenge might be: design an FSA to accept a string of 0's and 1's if the pattern 100 appears anywhere in the string. A sample solution appears below.



Automatic Theorem Proving Laboratory

Automatic theorem proving is first introduced with the logic programming laboratory. This lab focuses on two main topics: translation from English to first order predicate calculus to clausal form and then mechanically constructing a resolution proof. Although this topic may appear to be beyond a typical student in a computer literacy class, we have had some success in the past given a well designed lab activity to introduce resolution theorem proving [Gasser, 1992].

Machine Organization and Assembly Language

We will devise a simple assembly language whose execution can be simulated with a "model" computer. This model will contain the instruction register, decoding logic, program counter, RAM memory, arithmetic-logic unit, and register set. Students will be able to enter assembly language programs and execute them, step-by-step. In this way students will learn about op codes, addressing modes, and data manipulation for a simple computer.

Other topics we may develop labs for are:

- Artificial Intelligence Laboratory
- Software Engineering Laboratory
- Programming Language Translation Laboratory
- Operating Systems Laboratory
- Distributed and Parallel Processing Laboratory
- Ethics and Computing

In addition to constructing more laboratory modules, we plan to extend the work described in this paper in a number of directions. One direction we wish to explore is the use of multimedia, including digitized voice and video. We are also interested in innovative input techniques such as speech recognition and pen-based input.

Assessment, Evaluation and Availability of Materials

We intend to insure that the laboratory experiences are useful from a pedagogical standpoint, by following a rigorous evaluation program. We are conducting our formative evaluation using the following approaches:

- the teaching is being monitored by laboratory developers who have not been assigned as the instructor
- examination results are scrutinized to indicate laboratory strengths and weaknesses

- the laboratory sessions are monitored by graduate students who report on general use of the laboratory software
- laboratory sessions are monitored electronically by keeping a history of activities for each student
- students are asked to evaluate the course content and laboratories using instruments specifically designed to evaluate these course materials
- students complete the standard evaluation forms

We plan to use pre and post tests to measure the changes in, and generalization of, problem solving abilities. Standardized instruments for measuring student attitudes will also used.

Once the laboratories are fully developed and refined at Louisiana Tech University, we plan to make them available over the Internet using ftp. We have initially targeted two platforms: Sun Sparcstations and IBM PCs. Almost any Sparcstation with a color monitor can be used. The IBM PCs require Windows 3.0 or higher and SVGA graphics support.

References

A. W. Biermann, Great Ideas in Computer Science—A Gentle Introduction, The MIT Press, Cambridge, MA, 1990

J. G. Brookshear, Computer Science — An Overview, 4th. Ed., Benjamin/Cummings Pub., Menlo Park, CA, 1994

M. J. Conway, The SUIT Version 2.3 Reference Manual, University of Virginia, 1992 (inquiries can be made at suit@uvacs.cs.Virginia.edu)

R. Decker, S. Hirshfield, "A Survey Course in Computer Science Using *HyperCard*", SIGCSE Bulletin, Vol. 22, No. 1, February 1990, pp. 229-235

R. Decker, S. Hirshfield, The Analytic Engine, Wadsworth, Belmont, CA, 1992

P. J. Denning, D. E. Comer, D. Gries, M. C. Melder, A. Tucker, A. J. Turner, P. R. Young, "Computing as a Discipline", CACM, Jan. 89, vol. 32, no. 1, pp. 9-23

R. Gasser, Logic Tutor: An Intelligent Tutoring System for Resolution Refutation Proofs, M.S. thesis, Louisiana Tech University, 1992

B. L. Kurtz, M. B. O'Neal, An Interdisciplinary, Laboratory-Oriented Course Sequence for Computer-Based Problem Solving, proposal funded by the National Science Foundation, DUE 9254317

J. P. Myers, "The New Generation of Computer Literacy", SIGCSE Bulletin, Vol. 21, No. 1, February 1989, pp. 177-181

National Science Foundation, "America's Academic Future: A Report of the Presidential Young Investigators Colloquium on U.S. Engineering, Mathematics, and Science Education for the Year 2010 and Beyond", J. Lohman and J. Stacey (co-chairs), Directorate for Education and Human Resources, January 1992

A. Tucker (ed.), et. al., Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, 1991