CMU-ITC-87-055

Design and Implementation of a Computer-Based Tutor

David Trowbridge
Jill Larkin
Carol Scheftic

Center for Design
of Educational Computing

Carnegie Mellon University
Pittsburgh, PA 15213-3890

(412) 268-7641

Paper submitted to the 8th National Educational Computing Conference,
NECC '87

Design and Implementation of a Computer-Based Tutor

David Trowbridge
Jill Larkin
Carol Scheftic

Abstract

*We describe the development of a computer-based tutor concerned with the graphing of algebraic expressions. The program has some of the attributes of an "intelligent" tutor, including a built-in model for problem solving, a coach for helping students to apply the problem solving model, a facility for entering arbitrary new problems within a class, and the capability of solving any problem in that class. Known as **Sketch**, it teaches a systematic approach to curve-sketching, emphasizing a step-by-step procedure for transforming a simple expression into a more complex one and then transforming the graph accordingly. This paper details our steps in designing, implementing, testing and improving the program.*

Introduction

Computer Assisted Instruction today presents new options and opportunities: powerful machines, advanced computer science techniques, and input from cognitive psychology. Using these tools well is a challenge for which there are few guidelines or examples. The purpose of this paper is to describe how we used some simple artificial intelligence techniques in a powerful educational programming environment to capture some of the expertise of experienced teachers for an instructional program that runs on advanced workstations.

Overview of the Program

Sketch helps students to develop skills of visualizing and quickly sketching graphs of simple algebraic expressions. It teaches a systematic approach to the sketching of curves, emphasizing transformations of shape and location, rather than the plotting of individual points. At present, **Sketch** just handles simple expressions (e.g., $y(x) = 2\sin(3x)$ or $y(x) = 2x^3-4$) that contain a single non-arithmetic function (which we call the *base function*), but it can guide the graphing of any such expression. Within that class of expressions, **Sketch** provides appropriate instruction for any problem entered by a student or teacher. **Sketch** is a program of considerably greater flexibility than is usual in computer based instruction. It features:

A model for problem solving that is suitable for a class of problems -- the utility of the model is not limited to any particular collection of problems.

A coach for helping students to apply the problem solving model -- the coach is able to give appropriate help at each step.

A collection of instructive examples which is readily expandable by the teacher -- problem sets can be designed to reinforce the current lesson or to review previous lessons.

A facility for entering arbitrary new problems -- students can enter their own problems, provided those problems follow certain conventions of form and limited complexity.

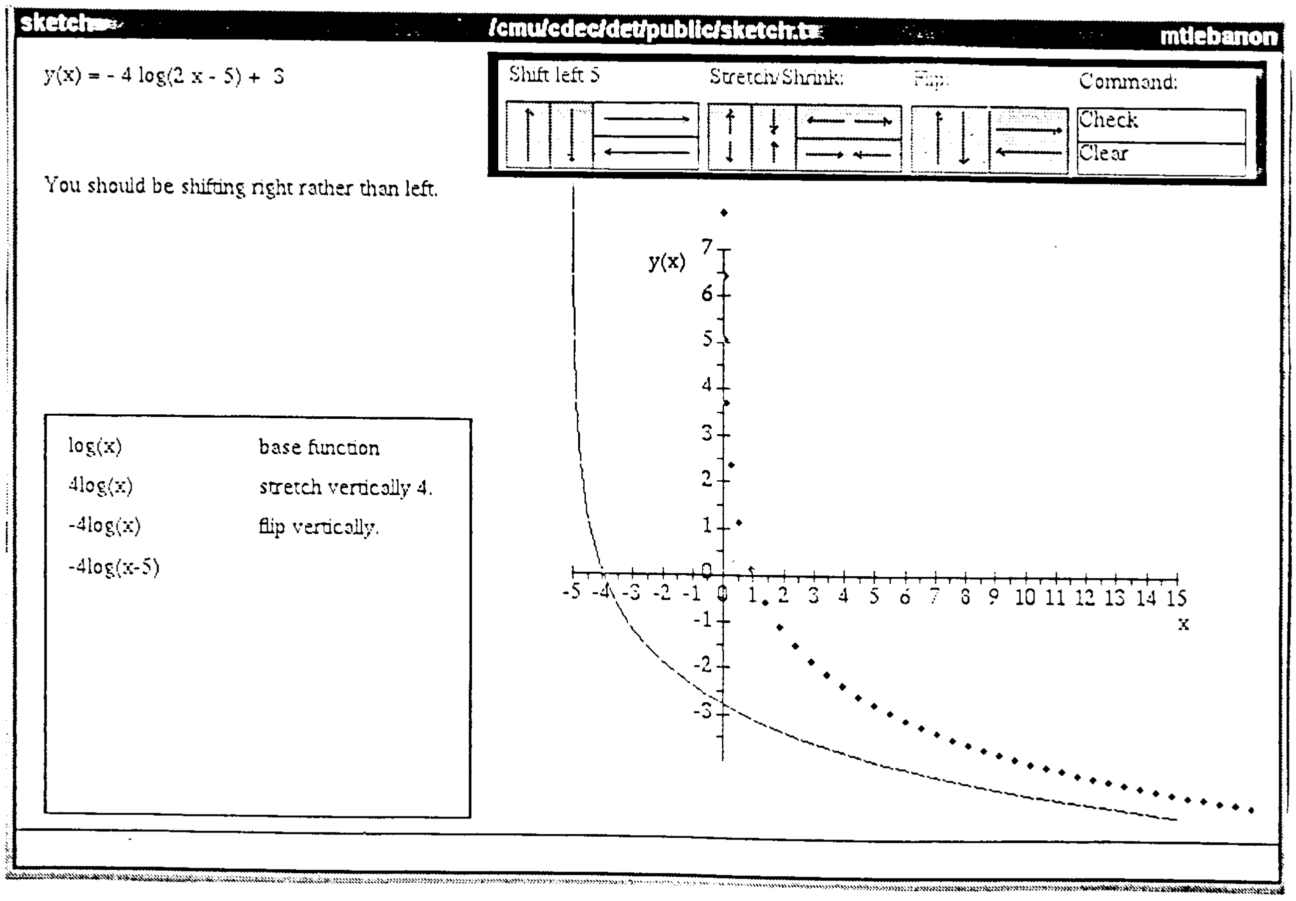
An interactive guide to using the program -- no separate instructions or documentation are required for effective use.

The same problem solving model and coaching strategies are used with the original examples provided by the program's authors and with additional problems posed by the program's users. All problem expressions use an internal representation that is suitable for any problem within a class, so that the program is able to diagnose errors and give detailed suggestions, regardless of whether the problem was among the set of original examples provided with the program, added to the program by the teacher, or interactively entered by students while using the program.

Description of the Program

The program is divided into three major parts. An interactive introduction to the program uses a simple example to introduce the problem solving strategy to the student. Upon entry, the student who has never used the program before is walked through a problem, following the problem solving model in a highly constrained manner which does not allow any deviations from the correct series of steps. Extensive guidance is given, concentrating on how to use the program, rather than on the concepts that will be taught later.

After finishing the walk-through problem, students are on their own. They may work problems from an existing set or enter their own problems. After selecting a problem, they are free to choose any steps they like. A coach watches each step of the student's solution, allowing several valid alternative paths at each step and giving context-specific help whenever the student starts down an invalid path. The program is able to diagnose errors and give detailed suggestions, regardless of whether the problem was among the set of "canned" examples or whether it was entered by the student. Figure 1 shows the appearance of the screen during a typical interaction.



$y(x) = -4 \log(2x - 5) + 3$

You should be shifting right rather than left.

log(x)	base function
4log(x)	stretch vertically 4.
-4log(x)	flip vertically.
-4log(x-5)	

Figure 1. Screen display midway through a student's solution to a problem.

Problem Solving Model

The problem solving model is the explicit method used by the program to solve a problem itself. The program teaches this model by advising or correcting students when their work fails to follow the model. The major intelligent feature of the program is its ability to independently solve any problem of the specified type, and to use the resulting solution as a basis for coaching.

The first step in the problem solving model is to examine the problem expression and identify the *base function*. For example, the base function of the expression, $2x^3-5$ is x^3.

(Although input of this function is currently in the form, x^3, future system developments will let the student use more natural notation.) The program currently recognizes the following base functions: x, x^2, x^3, sqrt(x), $\log_{10}(x)$, ln(x), exp(x), abs(x), sin(x), cos(x), tan(x), arcsin(x), arccos(x), and arctan(x), and its architecture could readily support additional functions if desired. Once the student identifies the base function, the computer displays a graph of it.

The next step is to identify a single arithmetic operation that, when applied to the function or to the argument of the function, will result in a slightly more complex expression that lies directly along the path toward the desired (complex) problem expression.

When the student has chosen an appropriate arithmetic operation (e.g., replacing the base function x^3 with $2x^3$), the program asks the student to perform a corresponding geometric operation on the displayed base function graph. For example, replacing the base function x^3 with $2x^3$ corresponds to stretching the base function in the y-direction by a factor of 2.

The model is flexible in that at many points in a solution there are at least two steps that can appropriately be taken. Because the program is based on a general problem-solving model, and not on any recorded sequence of steps, these alternative orders are automatically acceptable to the program. The problem sclving model merely requires that the student make the algebraic expression more complex, one step at a time, and then, for each step, apply the corresponding geometric transformation to the curve.

The model can be described algorithmically by:

```
(1) Identify a base function.

(2) Plot the curve for the current expression. (This is done automatically.)

(3) Repeat until the current expression is equivalent to the given problem expression:

        (a) modify the expression by exactly one step:

                {[(Add, Subtract, Multiply, Divide) a number (to, from, by)],
                Change the sign of} the {current function, independent variable}

        (b) transform the curve in a way that corresponds to the algebraic
        transformation:

                {[Shift (up, down, left, right), Stretch (vertically, horizontally)] by
                a number}, {Flip (vertically, horizontally)}
```

Suppose the student goal is to sketch the expression: $y(x) = -4\log(2x-5) + 3$. Following the problem solving model, the student identifies the variable and the base function, log(x), and the computer plots it. In subsequent steps the student modifies either the base function or the variable, by adding, subtracting, multiplying or dividing by numbers, or by changing the sign, one step at a time. At each step, the student must perform the corresponding geometric transformation of the curve, shifting it up, down, left or right, stretching it or flipping it about one axis or the other. One possible sequence of steps appears in Figure 1. Other orders of steps (after identifying the base function) would be equally acceptable, as long as each algebraic step corresponds to a single, simple geometric transformation.

When we give demonstrations of this program, we are often asked how we developed the strategy it incorporates. The next section describes that process.

Design Process

The first development step was an extensive effort to import the knowledge of teachers with experience in mathematics and science and with an interest in innovative approaches to teaching and learning. We began by planning an open house, at which we would display both the environment in which we would be working and some demonstration programs developed for use with university undergraduates. We asked the superintendents of local public school districts to nominate their top mathematics and science teachers at the secondary level for participation in this event. After their introduction to our resources, we invited interested teachers to submit applications for participation in the project.

We identified six teachers who appeared particularly qualified, and grouped them into pairs. Two pairs consisted of one mathematics and one science teacher each, while the third pair was made up of two mathematics teachers. Our selection criteria included a combination of classroom teaching and individualized tutoring experiences, curriculum planning experience, and evidence of the use of innovative approaches. Although it is unlikely that an outstanding mathematics or science teacher today would have no experience with computers, specific experience in that area was not a determining factor in our selection of participants.

Each pair of teachers met individually with the authors to specify and plan a project. The project reported here is the result of the sessions with the group containing two mathematics teachers.

Preliminary Design Session

The two teachers shared a variety of experience teaching geometry, advanced algebra, trigonometry, and calculus. We spent the first morning brainstorming areas where students might benefit from an additional learning tool, without regard to any technical details of how such a tool might be implemented. Topics suggested during this session ranged from *graphing skills* to *the meaning of the derivative*, but there appeared to be a common theme that emphasized the relationship between the algebraic and the graphical representations of a particular problem or expression.

By mid-day, we had reached agreement that we would focus on a tool concerned with the rapid sketching of algebraic expressions, where the emphasis would be on the general position and shape of the curve, rather than on the details of plotting points. We were thus able to spend the afternoon in determining a vocabulary of relevant issues, a description of the problem solving strategy to be implemented, an overview of necessary modules, and a diagram of the relationships among the components. We selected a couple of sample problems and considered specifically how we would handle them in this type of context, concentrating on the pedagogical issues of the interaction between the teacher (human *or* machine) and the student. Again, we deliberately refused to limit our thinking by concern over technical issues about how this would actually be implemented. We did, however, take extensive notes on the teachers' comments and suggestions.

The First Prototype

It would be several weeks before we would again meet together as a group. The two teachers agreed to come up with twenty questions each, of the type that they would want this system to be able to handle. Using a highly interactive programming environment (see Programming Environment section of this paper), we quickly generated some technically feasible graphical displays that eventually might be used in our program and then, when we had received the teachers' sample questions, investigated how they might be presented in our displays.

Subsequent Design Sessions

Our next set of meetings covered two consecutive days. We began by reviewing our understanding of the the system to be designed, based on our previous meeting. We promptly moved into detailed discussions of the interactions that might take place as a student proceeded through several of the sample problems.

We started with one problem, y(x) = 2sqrt(x + 3), and spent most of the first day examining how our system might handle it. The teachers discussed how they would present the sketching of this equation if they were individually tutoring a student. We tried to relate their suggestions to our sample displays, modifying the visual representations as necessary to meet the conditions the teachers considered most important. We discussed the system's responses to both correct and incorrect input from the students, noting and diagramming each step.

Next, we went through the same steps for several additional problems. We tried to use the same process, the same form of input and responses, for several different types of problem. When we found a situation that was not covered adequately by our original plan, we modified the plan so that it would cover both the new problem and all previous ones.

Program Development

The design sheets from our three days of work with these expert teachers can be viewed as what cognitive psychologists call a protocol -- a detailed, on-line record of the thinking of these expert teachers as they designed a lesson (Newell & Simon, 1972; Anderson *et. al.*, 1984).

From such design sheets it is possible (and often appropriate) simply to write an instructional program that implements sequentially the interactions described in them (Bork, 1985). Viewing the design sheets as a record of expert teacher thinking, however, we took a somewhat different approach. We designed a program to capture the knowledge of the teachers in a way that could generate a lesson on any of a broad range of problems. The lessons on the design sheets were then naturally generated by this general program, but the program would be able to handle a large class of additional problems as well.

This approach also simplifies the task of providing appropriate feedback to the student. Since our system can generate its own multiple versions of possible solution sequences, the program is able to give very specific responses to a wide range of different steps that a student might propose. This is possible, furthermore, regardless of whether the problem under consideration is one of the system's standard examples or whether it was proposed by tbe student.

Field Testing

Initial testing was done by making the program available to a variety of users of CMU's Andrew system and then modifying the program based on the informal feedback thus obtained. A more systematic test was conducted in the Fall of 1986, when the program was made available to students at a local high school for two days. Detailed observations were made, followed by appropriate modifications.

As a result of both the formal and informal testing, we are revising some of the system responses, speeding up some of the interface features, and clarifying some segments of the walk-through portion. It is interesting to note that, due to the extremely short revision/review cycle (see the section below on the Programming Environment), it was possible to make some of the indicated changes in the five minutes between class periods at the high school, and to test the revised program immediately!

Future Plans

We look forward to having the **Sketch** program in regular use in several local high schools during the Spring of 1987. We are also considering several major enhancements to the program. These include permitting the use of non-integer factors and expanding the class of allowed problems to include combinations and compositions of functions (i.e., allowing more than one base function at a time).

Implementation

Implementation of these ideas required design of a suitable data structure. As is often the case, a good choice of data structure is central to the program's performance. In **Sketch**, expressions are represented internally as a tree. The nodes of the tree correspond to the base function, the independent variable, and the operators that act on them.

How the Data Structure is Used

The diagram in Figure 2 illustrates the tree as it would be set up for the expression -4log(2x-5) + 3. In the tree, the next appropriate steps are always the adjacent nodes either upward or downward from the current function. As can be seen in Figure 2, after sketching y = log(x), the student could modify it to produce either y = log(x-5) (one down in the tree) or y = 4log(x) (one up in the tree). If student chose to sketch 4log(x), then the following step could be either -4log(x) or 4log(2x-5), moving one up or one down from the subtree for the preceding expression. The system also recognizes the special case of unary minus (changing the sign, flipping the curve), where an operation two nodes away from the current function is considered a valid next step.

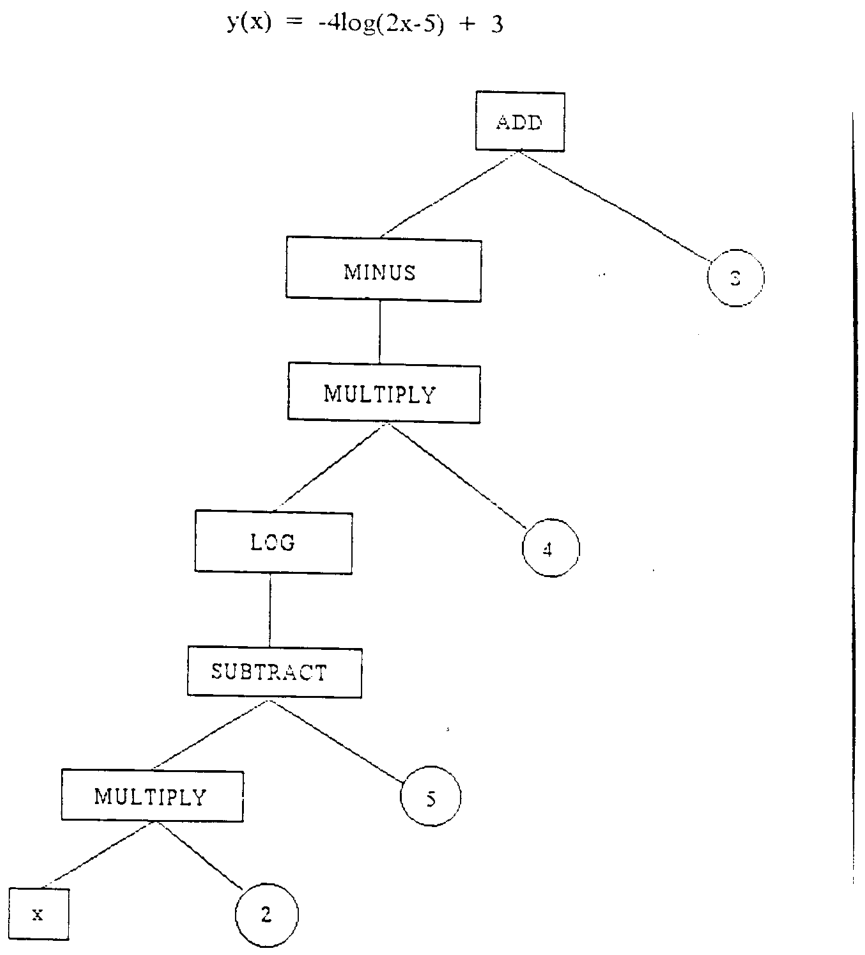


Figure 2. Problem representation using a tree data structure.

Representing the equations as trees has the following advantages. First, any equation with a single base function can be represented with equal ease simply by making a smaller or larger tree. Second, this data structure lets the program easily find the next step(s) in the problem-solving model. Therefore it can either offer suggestions or, if the student is really stuck, execute the step. Third, the tree naturally supports a variety of possible correct solutions because it always lets the student take any appropriate next step. Finally, by finding exact discrepancies between the tree and students' proposed algebraic steps, very specific feedback can be given.

The following sample interactions illustrate **Sketch**'s ability to respond to errors. The student is working toward the final equation $y(x) = -4\log(2x-5) + 3$, and is entering the step following the base equation $\log(x)$. The program is using the tree in Figure 2. Each interaction is followed by a summary of the program's process for recognizing the student's error. In all cases the program inserts numbers and variables appropriate to the current example.

Student entry:	-4log(2x-5) + 3
Program response:	That's the whole equation, but we're not ready to make such a complex change from the previous equation.
Process:	*It compares the student's expression with the complete problem expression.*
Student entry:	log(2x-5)
Program response:	You have indicated two operations on the variable: subtracting 5, and multiplying by 2. Please choose one or the other.
Process:	*It "walks the tree" down from the base equation node (log) two levels (subtract and multiply). The student expression is identical to the sub-tree of the problem expression that is spanned downward from the base function to the independent variable.*
Student entry:	log(2x)
Program response:	Your answer, replacing x with 2x, is a very tempting idea. But, if you do that, you will not be able to replace x with x-5 in order to get 2x-5.
Process:	*The step is out of order -- the student's entry corresponds to the node two down from the base equation rather than directly below it.*

The last of these errors is particularly common and tricky. The program is correct -- if the student first sketches log(2x), a later step would require subtracting 5/2 from the variable in order to get log[2(x-5/2)] = log(2x-5). This fractional shift is awkward for the student, since it requires additional algebraic steps of factoring and rewriting the expression. Furthermore, the graphic interface in **Sketch** currently allows only integer input. We hope, ultimately, to support both the explicit rewriting of expressions, and the transforming of graphs by fractional amounts, while maintaining the simplicity of the program's user interface.

Implications of Using a Tree

A tree data structure like that used here is common in computer science, but less common in CAI. Our tree was implemented using simple arrays (see Appendix), rather than lists, or records and pointers (as would be done in LISP or Pascal, respectively). Using the tree requires one other moderately advanced technique. Because the data structure (the tree) does not have a predefined size, but may be larger or smaller depending on the problem, getting information from the tree requires a recursive technique to do depth-first search. This technique, easily implemented in most languages (Winston & Horn, 1981), looks through a tree, to whatever depth is necessary, to find any specified item that the program currently needs.

Although the tree is a fine data structure for the program, this is not how people write equations. Therefore, to let both students and teachers define their own problems, we have built a parser that takes an equation in conventional form and builds the tree that the program needs.

User Interface

For **Sketch** to be an easy-to-use aid in thinking about curves, it requires a display that reflects the problem solving model and that makes the program's operation self-evident. The algebra interface is reasonably straightforward, with the student simply typing in successive expressions. The geometry interface, on the other hand, requires a special device that makes it easy to move or reshape curves.

On-screen *buttons* are provided. The student clicks once or more on a button to shift, stretch or flip the curve. (See Figure 1, upper right corner of the display) This seems to work satisfactorily, provided that students receive adequate explanation and practice during the initial walk-through problem. A two-column table records the history of algebraic steps and corresponding geometric transformations. The desired generality of the program required us to develop algorithms for producing nice looking axes with reasonable labels, numbers and tick marks. Designing the screen was aided by ports, rectangular regions within the window corresponding to subroutines containing information about how they behave under varying conditions. During design, ports can be rearranged on the screen to improve the display without losing their properties.

Workstations

Certain characteristics of advanced computing technology were central to the development of this program. Advanced workstations (e.g., the IBM RT Personal Computer, the DEC

VAXstationII, and Sun Microsystems Sun-2 and Sun-3) provide essential features including: (1) virtual memory, which allows the programmer to work largely as if the machine had almost unlimited memory; (2) a high-level operating system (in our case, BSD 4.2 UNIX $_{tm}$), which provides both development tools and the potential for easily moving programs from machine to machine without modification; (3) a large bit-mapped display (like that of the Macintosh, but having about four times as many dots), which makes it much easier to display a large amount of information in a clear, logical fashion. The value of this last feature is illustrated in the figures of our **Sketch** program, where the student has simultaneous views of a developing list of equations, an interactive graphing environment, and instructions and coaching from the program. Features like this require about 3 megabytes of memory and a processing speed of roughly a million instructions per second.

Programming Environment

This program was implemented using CMU Tutor, a programming environment for advanced-function workstations which includes a number of features especially valuable in the development of educational software (Sherwood and Sherwood, 1986). CMU Tutor is a descendent of the TUTOR and MicroTutor languages originally developed in the PLATO project at the University of Illinois. With this history, many of its features have had 20 years of testing for effectiveness and efficiency.

CMU Tutor gives authors full access to the Andrew system, an educational computing environment being built at Carnegie Mellon University. Andrew is the distributed personal computing environment being developed in a joint project between IBM and CMU (Morris, *et. al.*, 1986). Aspects of Andrew that are most relevant to the discussion here are the use of 1000 x 1000 pixel graphics screens, a windowing environment using a mouse, and a processing speed of roughly a million instructions per second.

The language is particularly well-suited for non-expert programmers. In addition to all of the usual control structures of a general-purpose programming language, CMU Tutor supports many capabilities which are particularly germane to the creation of educational applications. These include picture drawing, graph drawing, animation, answer judging, menus, mouse input, and display of fancy text.

CMU Tutor is an incrementally compiled language, making possible an extremely short revision/review cycle. Even with large programs, an author need never wait more than about 10 seconds to see the effects of changes just made.

The complex interactive interface developed for **Sketch** could not have been done in any reasonable manner without this set of tools. Implementation of **Sketch** has involved three people programming concurrently. Presently, the program consists of 7000 lines of code.

Acknowledgments

This program is part of a project funded by the National Science Foundation (Grant Number MDR-8470166) to explore the use of advanced function workstations for educational computing, with emphasis on teaching topics in science and mathematics to high school and college students. The project is based at Carnegie Mellon University; it represents a collaboration between CMU and the Allegheny Intermediate Unit (a regional educational service agency for schools in Allegheny County, PA). Additional support has been provided by a fellowship from the John Simon Guggenheim memorial foundation and the NSF program of Visiting Professorships for Women (Grant Number R11-8600412).

The authors wish to acknowledge the contributions of several additional people in the development of this program. David Yee, a CMU student, joined us as a programmer and turned out to also have a real talent for the design of the user interface. Bruce A. Sherwood and Judith N. Sherwood, our colleagues at CMU, offered both assistance and patience as we worked our way through the more technical details of CMU Tutor. Our two subject matter experts, Connie Radocay, a teacher in Plum Borough School District, and Andy Tiglio, a teacher in Chartiers Valley School District, provided both insight and inspiration.

Bibliography

Anderson, J.A., Boyle, F. C., Farrell, R. and Reiser, B., "Cognitive Principles in the Design of Computer Tutors," *Technical Report*, Department of Psychology, Carnegie Mellon University, Pittsburgh, 1984.

Bork, A., *Personal Computers for Education*, Harper and Row, 1985.

Erikson, K.A., & Simon, H. A., *Protocol Analysis: Verbal Reports as Data*, MIT Press, 1984.

Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H.. and Smith, F.D., "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, 29:3 March, 1986.

Newell, A. & Simon,H. A., *Human Problem Solving*, Prentice Hall, 1972.

Sherwood, B. A., and Sherwood, J. N., "CMU Tutor: An integrated programming environment for advanced-function workstations," *Proceedings of the IBM Academic Information Systems University AEP Conference*, IBM Academic Information Systems, San Diego, April, 1986.

Sherwood, B. A., and Sherwood, J. N., *The CMU Tutor Language, Preliminary Edition*, Stipes Publishing Company, 10 Chester Street, Champaign, Illinois 61820, 1986.

Sherwood, B. A., "Workstations at Carnegie Mellon," *Proceedings of the Fall Joint ACM-IEEE Computer Conference*, November, 1986.

Trowbridge, D., "Using Andrew for Development of Educational Applications," *Proceedings of the 1985 Academic Information Systems University AEP Conference*, IBM Academic Information Systems, Milford, CT, p. 85, June 23-26, 1985.

Trowbridge, D., "A Sampler of Educational Software at CMU," *Proceedings of NECC '86*, National Educational Computing Conference, International Council on Computers in Education, University of Oregon, Eugene, OR, p. 135 June 4-6, 1986

Winston, P. H. & Horn, B. K. P., *LISP*, Addison Wesley, 1981.

Appendix

The following declarations illustrate how the tree data structure was actually implemented using an early version of CMU Tutor (before it supported rich data types), how it would be done in Pascal, and how it might be done in an improved CMU Tutor having Pascal-like data types.

CMU Tutor version (uses numbered nodes and integers to link nodes in a linear tree; static allocation)

```
define              f:Operator(15),Arg1(15),Arg2(15),Up(15)
                    ADD=1, SUBTRACT=2, MULTIPLY=3, DIVIDE=4,
                    SQUARE=5,CUBE6, LOG=7,
                    FCN=8, VAR=9, NUMBER=10
```

Pascal version (uses pointers to link nodes in a linear tree; dynamic allocation)

```
TYPE                    OpType = [FCN, VAR,
                        ADD, SUBTRACT, MULTIPLY, DIVIDE,
                        SQUARE, CUBE, LOG];

                        NumPtr = ^INTEGER;

                        NodePtr = ^NodeType;

                        NodeType = RECORD
                        Operator : OpType
                        Arg1, Up : NodePtr;
                        Arg2 : NumPtr;
                        END;

VAR                     Node : NodeType;
```

Possible appearance of CMU Tutor declarations in the near future (uses Pascal-like enumeration types, records and pointers in a linear tree; dynamic allocation)

```
type                    enum: OpType = (FCN, VAR, ADD.......)
type                    integer: *NumPtr
record                  NodeType
                        OpType: Operator
                        NodeType: *Arg1, *Up
                        NumPtr: Arg2
define                  NodeType: Node, *NodePtr
```