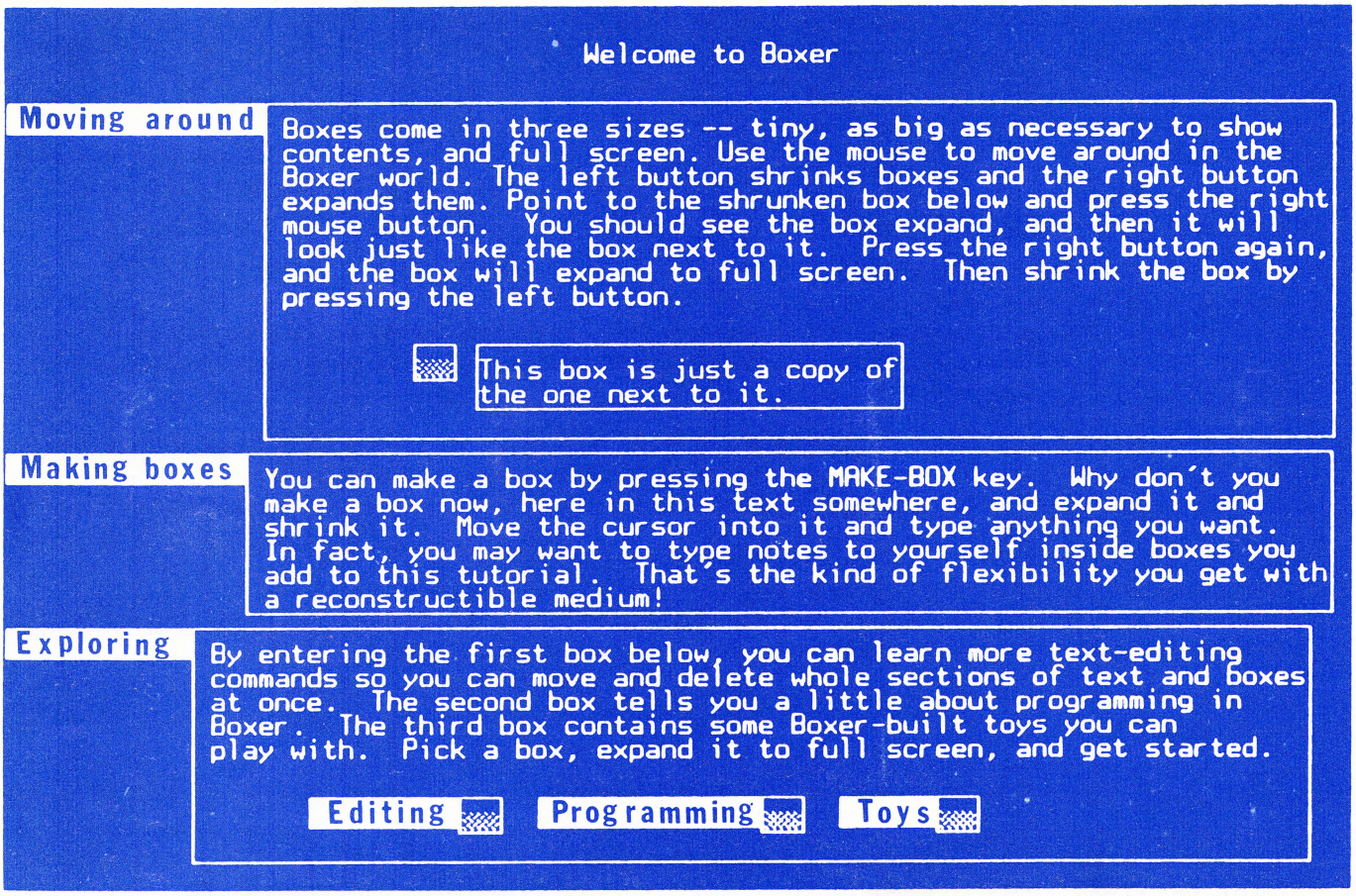


Figure 1. This "page" from Boxer Tutorial shows how boxes can structure text and hide details. A mouse is used as a pointing device to move around the system and to shrink and expand boxes.

FROM LOGO TO BOXER, A NEW COMPUTATIONAL ENVIRONMENT

Andrea A. diSessa

UNIVERSITY OF CALIFORNIA, BERKELEY, U.S.A.

(Edited from a transcript of a lecture given at Monash University, June 1985)

I want to begin by trying to impress upon you that the use of computers in education, while it has come on us all rather suddenly, is only at the beginning of what I think will be a rather long process of changing education. This process will even change the things that we teach in response to the new capabilities that we have with computation. If you look back in history at events that are parallel to the advent of computation in terms of the effect that we can expect on intellectual development, we have to go back to things like the invention of the printing press and the invention of writing. Herb Simon has said this very well. He says that the computer is a "once in several centuries innovation." If you begin thinking about an innovation of that magnitude, you really have to appreciate that it's going to be decades at least before we realize the full implications in terms of our teaching practice. I could spend an entire talk telling you about the possibilities of new curricular subjects, not just new methods of teaching, but teaching entirely new things. I'm not going to do that today because my emphasis is on technology, but I think you should realize that that's very much in the background. We really need to take a long time scale view, and the new environment that I'm going to talk about is dedicated to that view.

A good way of getting started is by talking about some of the things that we think we did right when we designed Logo. The first of these is our very serious concern for what I call "democratizing technology," that is, taking computational power and trying to put it in the hands of common folk, of teachers and students and

curriculum developers, anybody who is not a computer specialist. We think that was one of the most important ideas that we had with Logo, of taking agency away from the computer scientists and the programmers and, for educational purposes, putting it in the hands of the educators and those who want to be educated. From my point of view, sitting in a Laboratory for Computer Science, I must emphasize that it's not very easy to get those kind of folk, computer scientists, to think about human issues. They tend to be preoccupied with making systems that have a nice crystalline structure from a mathematical point of view but which don't necessarily do anything good for education and aren't necessarily terribly understandable. Democratizing technology is a political issue. It's a very important conviction that we need to spread: technology belongs in the hands of the ordinary people. That's a theme I think you'll see in my motivation for designing this new system. It's something that we thought about very carefully with Logo, and it's something that we are trying to continue with our new system, Boxer.

There is a second idea that I want to pick out that grew up with the design of Logo, and that we are really trying to develop further. It is the idea of building learning environments that we call microworlds. The notion of a microworld is to build a relatively unconstrained computational environment where there is, nonetheless, a pedagogical topic, an aim, such as teaching physics. It's important that, in terms of the activities of the students, we are not constructing a complete, prescribed step-by-step curriculum; we need room for the learner to find his own goals, means and ways of thinking. The model that most people know is Logo and turtle geometry, as a synergistic pair that define a microworld of geometry and a certain slice of mathematics. It doesn't cover all possible mathematics; in fact, it covers subject matter that many are not familiar with as mathematics. But, though a student may not see it, the microworld is "about" that mathematics.

If you watch what a child does when he tries to draw a picture with a Logo turtle, you see a lot of encounters with very important mathematical ideas, ideas such as the concept of an angle, such as estimation. It's amazing to sit down with older elementary school students and have them begin working with the turtle, telling it to go forward 1 000 units, forward 200 units. You see that many of these children are for the first time using numbers in a meaningful way, in a goal directed way, where they get feedback on the relative sizes of numbers that they never get doing school arithmetic. So we design the turtle and Logo as an environment that is about mathematics, but it is also about drawing. That's the other side, that microworlds always have to have a motivational or perceived value to the student, motivating him to get on with working in that environment.

The turtle, and the Logo programming language as a general purpose utility, are pretty well known. But we have done a lot of work with developing other sorts of microworlds. We firmly believe that this is a way of achieving goals that we've had for centuries in education, goals that for technological reasons, we haven't systematically attained. The top level goal is returning agency, returning the centre of activity, to the learner in the work that he or she is going to do. So you'll see in Boxer the attempt to make a system with which it is easier to build microworlds of all sorts.

One ought to think about the computer as a new kind of medium, something like writing or verbal communication.

I would like to spend a couple of minutes talking about the role of programming in all of this. I don't know really where things stand in Australia, but in the United States there is a lot of debate these days about whether one should turn kids into programmers. I think the image that people have when they protest against turning kids into programmers is of programming as an esoteric activity that is done by specialists, where you learn some arcane language, and you sit in your basement for many hours and crank out a program that grinds through an incredible algorithm to give you some numbers in the end. That image of programming should be a false one; it is one that will have to change. People should not think of programming as something of that order, but instead one ought to think about the computer as a new kind of medium, something like writing or verbal communication. It has important powers beyond writing, to take an example, in that it's interactive; it's expressive in a very different way. But, basically realize that the same thing is going on. Computation is (should be) an expressive medium. One should now think of programming simply as the way of constructing and reconstructing in this new medium. And that means that every time you sit down and poke this medium, you make a little change to an existing piece of it, you are a programmer. I'm hoping that people's intuitive image of programming changes profoundly when we have developed the technology sufficiently so that it is easy and profitable for even young elementary school kids to build their own microworlds in the computational medium.

With that introduction to some of what I think are the important ideas in Logo, let me spend just a couple of minutes talking about what we think is wrong with Logo, what are the shortcomings, what we want to go beyond in the design of our new environment.

The first item is the issue of functionality, what you do with your computational system. I think it has become clear to us that our image of what one did with a computational system in those very early days of Logo was restricted. The image was programming more like the arcane activity I criticized a minute ago. We need to have the computer become useful in many other areas in addition to that kind of programming. Programming will always have a central role, but text editing, becoming popular in educational computing, really ought to be contained in exactly the same environment. We're trying to build a computational environment that has many different functionalities. So Logo was aimed, we think, a little too narrowly towards programming, slighting things like personal databases, text editing, etc.

The second item is an additional persepective on the first one. We really want to have all of these functionalities very closely knit together. We want it to be an "integrated" system so that there isn't any distinction between where you are in the system or what part of it you are using when you're writing a paper, as opposed to when you're writing a program, as opposed to when you're constructing a database, as opposed to when you're accessing a database. We want to have an entirely uniform environment, which means that you can easily construct microworlds that have combined capabilities from all of these function areas. One of the things that struck me about Logo is its limitation in building large

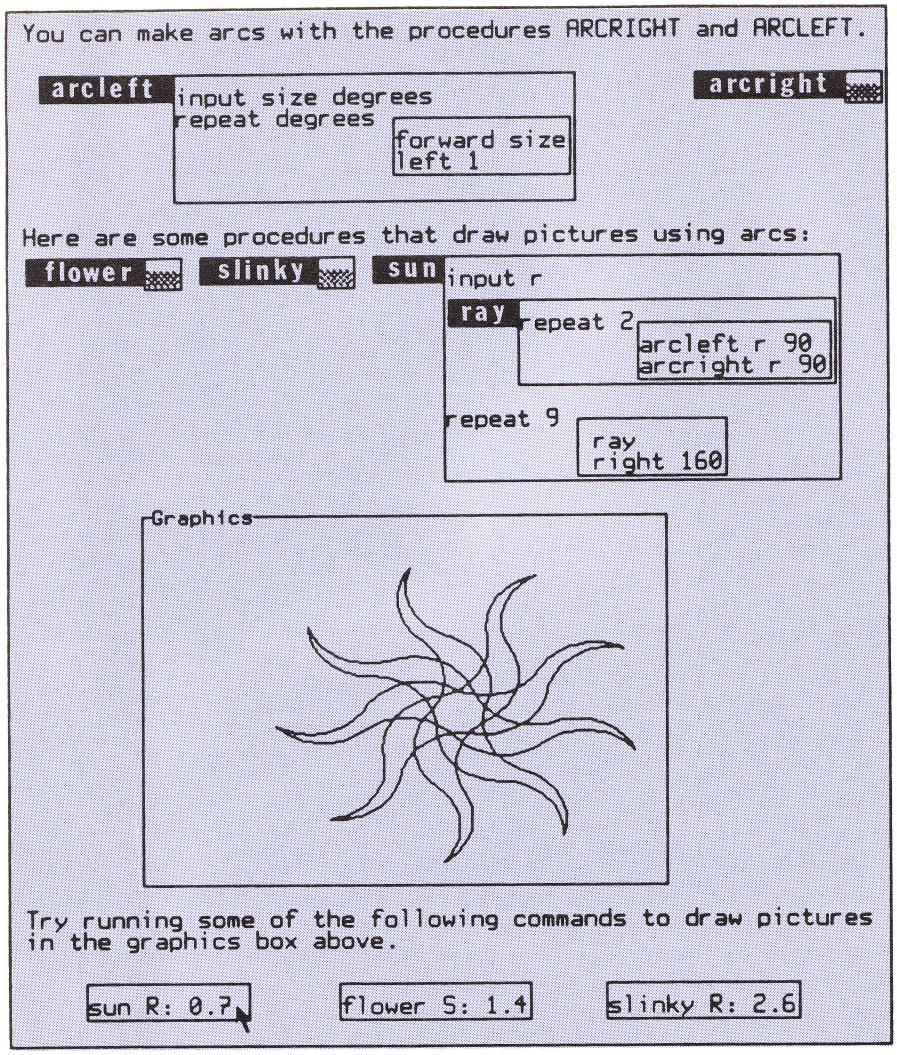


Figure 2. Graphics appear inside regions of the screen called graphics boxes. The box labelled SUN defines the procedure that drew the design in the graphics box above. Definitions (boxes with name tabs on them) can be invoked anywhere within the box in which they appear. The bottom line of the screen is a menu from which the user can select commands to be run.

structures, for example database. Similarly, in Logo I don't think we did terribly well in terms of things like files and work spaces. These turn out to be rather abstract objects that are difficult to manipulate. You can't see them in great detail, and you don't have detailed enough control so that you can develop complex but understood things over an extended period of time. Our new system aims at giving people the ability to create some object of large scale, not just a program but something more in the order of a database or a book or a big piece of hypertext, in a way that Logo simply doesn't.

I should say in defence of what went on in the design of Logo that it was constraints of the technology, constraints on the amount of memory, constraints on the processing power that, in part, forced us to slight some of these other areas. As well, I think we had a slightly over optimisitic image of how good a programmer you common ordinary everyday student or teacher would be. We have decided that we want to make our new system easier to use for everyday tasks. Instead of having somebody build a text editor in Logo, we really want to have quite an elaborate text editor already built in, so that functionality is automatic once you step into this computational environment. One can always do these things, one can always build the kind of functionality that we are talking about for our new system out of Logo; but we've decided that an important part of it should be prebuilt for people.

I'm going to spend a lot of time later talking about understandability, so I'll just mention it now. I think Logo could do better in terms of being an understandable system. I was personally disappointed in how far particularly elementary school kids were able to go. They seem to reach a kind of plateau and then not go very far beyond that. With this new system we're explicity trying to develop a system that will bring students forward to the most advanced capabilities of the system one step at a time, in a way that I think Logo does not do very well. I won't say anything at all about the details here since I'm not going to give a talk on Logo syntax or anything like that. But we do think we have learned something about how to make understandable and useful programming languages at the level of details like syntax, and what kind of data objects you want to have, and so forth.

I also should mention that it's not just Logo that I'm criticising. I happen to think that for educational purposes Logo is the best thing we have today, so that my criticisms go double for other systems such as BASIC and Pascal and Lisp.

So here is what we are trying to produce in our new system. We're trying to produce an integrated computational environment, in which simple programming is the central kind of glue that holds it all together. That's the way you construct and modify and rebuild in this new medium. In particular, the activity of being given partially formed programs, which you will change and modify for your own purposes, will be something that this new system will accommodate much better than Logo. I've already mentioned things like text editing, having a file system that is a lot more visible, a lot more concrete. Databases: you want to have the capability of handling a lot of data very easily, building methods of accessing it in various ways. And of course you want a really nice programmable graphics capability; that almost goes without saying from somebody coming out of the Logo context.

Now let me spend just a few minutes talking about some of the more detailed principles of design that we had in our heads when we began designing our new system. These, if you like, are the technical methods that we have employed to make the system more understandable. The first of these principles is one that I call naive realism. The idea is that when a user steps up to a computational system, what he ought to see on the display screen is the computational system. Every part of the system has to be visible on the screen. If you change the appearance of the system

on the screen, then you have changed the system. That's the notion of naive realism, that what you see is your world, and you can touch it directly, and you can change it directly. Our new environment is really the first computational (programming) system that adheres rigorously to this principle, that the computational system is what you see on the screen, as opposed to something that you just think about.

Using Logo or BASIC, you type some things on the screen. Those are messages that have to go off to the interpreter (be executed) to really change anything. Maybe the interpreter prints something back out on the screen. But you don't see the system itself; you can see the name or value of a variable, but you don't see variables themselves. You can enter the text editor (why leave Logo in the first place?), and you can poke at a written representation of a program, but if you exit the editor in the wrong way you haven't really changed the program. There are all sorts of subtle and some not so subtle ways the system shows you there is a distinction between what you see on the screen and really what the system is. In this new system we are trying to entirely eliminate the distinction. I think that's a very powerful principle, to make something understandable, make it visible, and make it touchable; make it manipulate in uniform terms.

The basic appearance for this new system is text, augmented by a single structure that we call a box, hence the name Boxer. It consists only of text and boxes in which you can put more text or more boxes in arbitrary combinations. The way that you manipulate the system is to point to various parts of it and use the editor that is part of Boxer one hundred percent of the time. You are always using the editor to change the system; you change the system by changing what you see on the screen. That's the way you "talk to" the system.

The second principle of understandibility is something that I call the spatial metaphor. When you think about how you are going to make something understandable to somebody, you ask what they already know that could be of use to you in explaining the new system. In this particular case, we've taken people's common sense notions of space, of moving around, of being inside, of moving outside and so forth. I happen to believe that humans' understanding of space is perhaps the most profound and well-developed intelligence that people have. We're building the computational semantics around that spatial metaphor, so that people can see spatial relations on the screen and interpret those immediately in terms of the computational ideas.

The third principle that I want to talk about in a little detail is the principle of building multiple models. The notion here comes from a lot of work that I've been doing, and other people have been doing, looking at the intuitive knowledge that people seem to acquire naturally about the physical world: about space, about pushing things around, the common sense notion of causality, that sort of thing. You discover that the knowledge system that people build up is a very fragmented kind of patchwork system. I don't take that to be a deficit; I take that to be a fact of life that we have to work with. People's cognitive systems are meant to deal with systems that are learned a fragment at a time. You learn this way of looking at it; you learn that way of looking at it. You don't learn the five rules that define your computational system, but, instead, your learn a lot of guesses, a lot of rationalizations, a lot of things to do with your system; that's the way you come to understand the system.

With this new system we're explicitly trying to develop a system that will bring students forward to the most advanced capabilities of the system one step at a time, in a way that I think Logo does not do very well,you change the system by changing what you see on the screen. That's the way you "talk to" the system.

To make this idea of multiple models a little more concrete I'd like to give you some examples of the different kinds of models that we have built into Boxer. When I say the word "model," some of you might get the idea of something like a replacement machine that people have in their head, that they can sort of crank through to get the answer. And to be sure in some circumstances we'd like to have people have that kind of model of a computational system in their head. I'll give you a good example of this, what Richard Young calls a surrogate model, or a replacement machine model. This is a model for a computational structure called a "push down list" also known as a "stack." A push down list is a thing that works in the following way: You take a piece of data and, like a block, you can put it down in a pile. That's called "pushing," so you can "push A." A is a piece of data you've "pushed onto the stack." You push item B; that means take another block, say it has B written on it, and put that on top of the

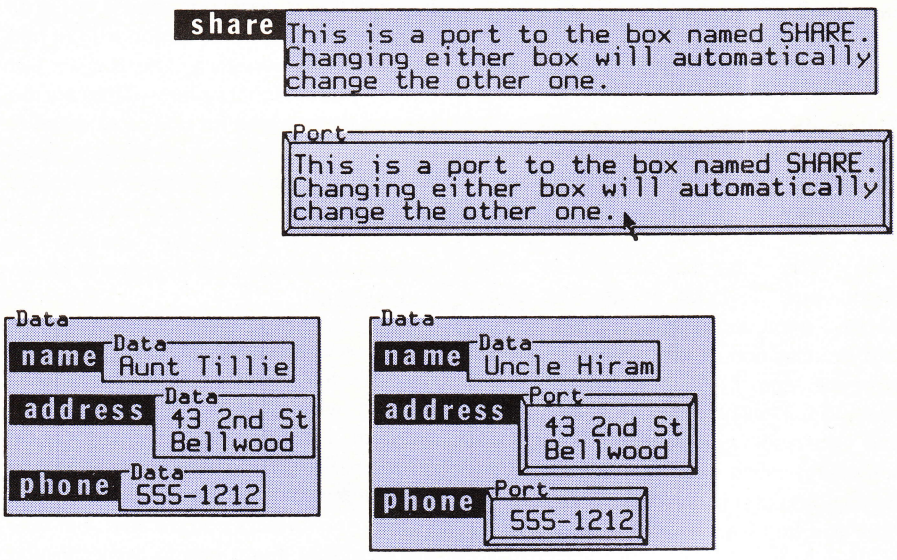


Figure 3. The second box in the figure is a port (alternative view) of the box named SHARE above. Text typed into the port at the arrow cursor has also automatically appeared in the SHARE box. The two lower boxes illustrate how ports can be used to implement shared data. Here, any change to the ADDRESS or PHONE fields in either box will also appear in the other box.

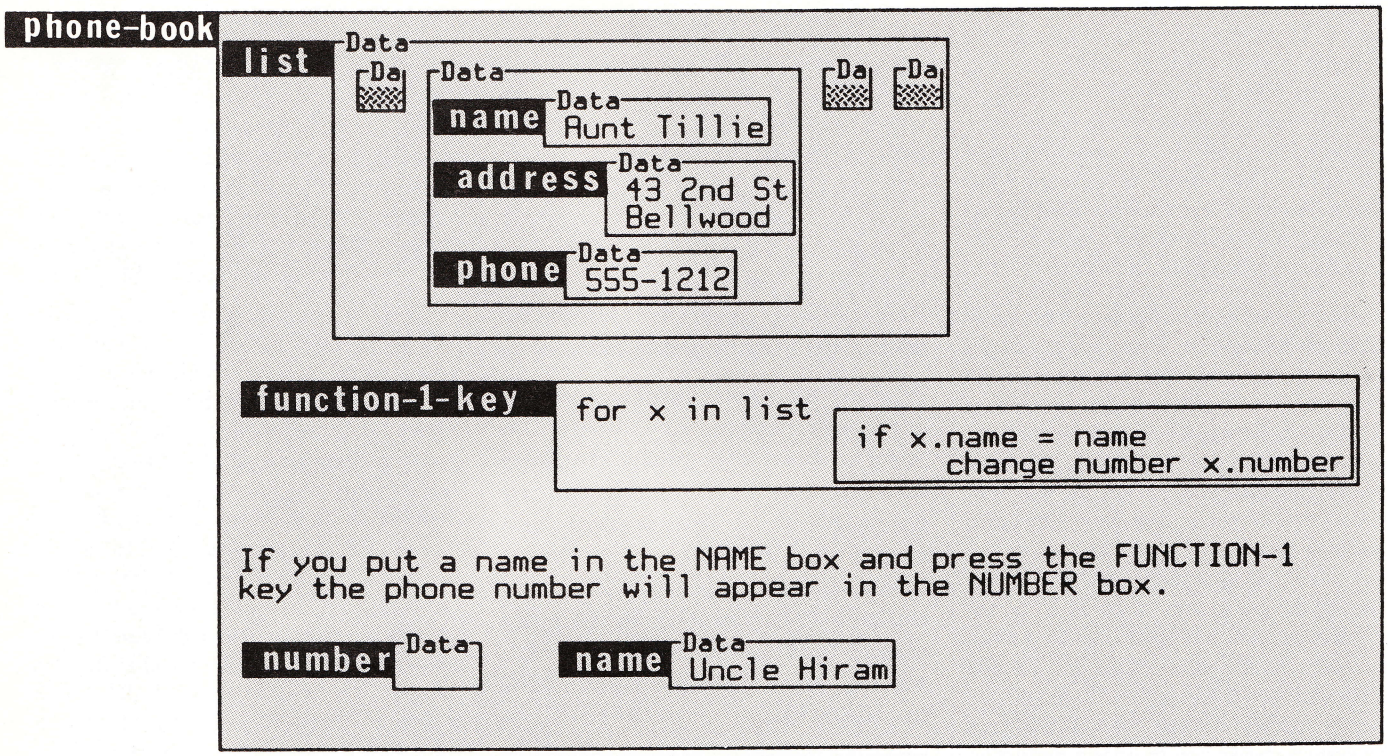


Figure 4. The PHONE-BOOK box is a utility for finding phone numbers. Its local data base is a box called LIST that contains sub-boxes with a standard format. The procedure named FUNCTION-1-KEY will be run whenever the FUNCTION-1 key is pressed.

stack. You push C, and you get a third block that has a C on it on top of the stack. The operation of access is called "pop;" you pop items, one at a time, off this push down list. So if I pop the push down list that I've created just now, what do I have in my hand? C. If I pop again, now what do I have in my hand? B. I pop again, and I have the first thing that I put on.

What you are doing is using your understanding of space, of piling blocks. You're using your ability to create an image, to create something in your head, which is really a perfectly good replacement for this abstract notion of a push down list. Now you can predict almost everything that there is to know about a push down list. In particular, if you pop too many times you are going to run into trouble, and at that point you get an error message. If you only have so much room to the ceiling, and you keep pushing and pushing and pushing, you get something called overflow; the stack gets too big, and you get another kind of error. Now you know a lot about a stack.

You can make neat clean models of certain things in a computational system. But there are limitations to this notion of a surrogate or replacement model for a system. In the first instance, these models are rather hard to learn. If you take something like a computational system, like Logo or BASIC or your favourite programming language, and you try to develop a replacement model for all of the actions in the language, it just turns out to be a very difficult thing to do. You develop a model that is very large, that is not at all learnable in the kind of session we just had on the push down stack. This is just a fact about computational systems. They are too big and complicated for you to sit down at the beginning and learn all there is about them, learn this replacement model, and then just go off and do your work on the system. The surrogate models or replacement models fail in terms of bit-by-bit learnability.

The second problem with surrogate models is that it would take forever to think through what is going to happen in a program. I'm not going to try to give you an example of a surrogate model of a programming language because I've already said that it's complicated, but take my word that if you had to think through every step in a model for everything you do in a programming language you'd simply never get through it. So people develop fractional, piece-wise models to deal with special cases and so forth, so that they just don't have to think through all the details of this uniform surrogate model.

Finally, the basic intention of this replacement machine kind of model is to be uniform, to be absolutely independent of whatever application that you use the computational system for. But that puts you in a real bind when it comes to inventing ways of doing something; you don't have any specific suggestions for how you should accomplish something merely by knowing how the machine works. What has been discovered empirically by looking at programmers is that they build up a very rich vocabulary of fragmentary plans, partial ways of doing things. A counter loop, for example, is a thing that programmers learn, or in Logo, tail recursion. These little techniques, little plans, are important parts of coming to understand a programming system.

Let me not belabor the point. Surrogate models are important to have, to ensure that you have a uniform understanding of the system that you can use when it becomes important, for example in debugging, to trace through every last step. If you don't have a surrogate model, if you don't have a way of understanding the nuts and bolts of the system, then you will fail

when something goes seriously wrong. But such models are not by any means universal replacements for other ways of understanding.

Now I want to give you a couple of other models of model, a couple of other ways of building understandability into a system as we've done for Boxer. The second class of model that we're explicitly designing into Boxer is what I call functional models. This takes up where the replacement machine models leave off. It tries to make you understand the constructs in the system according to what they do for you.

The best example of a functional model that I know is how people understand an ordinary calculator. Most people don't have the foggiest idea what is inside a calculator. They don't know that there's an operator register; they don't know that there are two data registers, and what you see on the display readout is one or the other of those numerical registers according to what you have typed on the system. They don't have any idea of the internals of the machine. Instead, they think of the calculator as something that does arithmetic. There's a very simple mapping between the things that you can do in this calculator domain and the things that you are used to doing with paper and pencil. When you type one plus one on your calculator, that is obviously "entering the problem." It isn't "entering the problem" to the calculator. You've put some stuff in various registers, and it's just sitting there waiting for the next keystroke. You haven't entered any problem, but in your mind, because you already understand what arithmetic is about, you think that you're entering the problem. And you press the equals button; what is that? That is "getting the answer." The calculator has no notion of "getting the answer." It does some internal things, and it shows a different register on the screen, but you think that you are getting an answer. That mapping that you do almost automatically between paper and pencil arithmetic and button pressing on a calculator is what I call a functional model. (Richard Young calls it a task/action mapping model.)

Just to prove the point that that is not a complete model of the system, I wonder how many of you know what state the system is in if you press 1 + +. Now in terms of arithmetic that's a meaningless statement; you've said something dumb, and it just doesn't mean anything. In terms of the calculator, you have done something definite to it. It's in a certain state, but you have absolutely no way of understanding it. You don't have one of these replacement machine models. For the most part, you just don't need it. But, suppose it was important for you to understand 1 + + (and in fact many calculators use that 1 + + to do what's called constant calculations). You won't get that from the functional model here, the model that a calculator "is doing arithmetic."

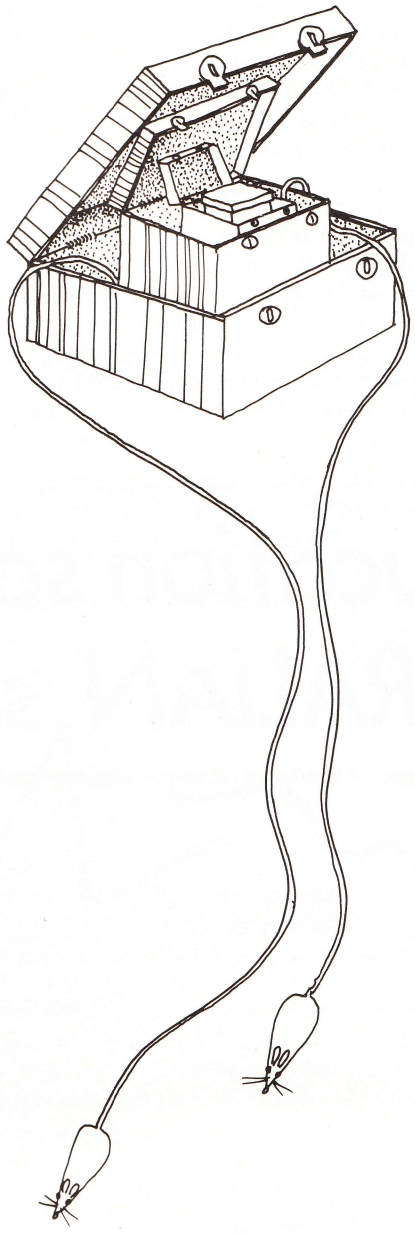
What's good about functional models? Certainly they allow you to understand a machine often times enough to get something done, and in many circumstances that's just enough for you. Secondly, functional models are useful when it comes time to design and build something. If you want to do arithmetic, it's good to have an arithmetic functional model of a calculator. In the case of a computational system, what will typically happen is that you will learn a portion of it, then you'll learn a functional model of a new part of that system, that this particular construct does that particular kind of thing. And that would give you entry, it will allow you to begin doing things before you understand any of the details of the mechanism.

Unfortunately, functional models have their down side too. They're not comprehensive, which shows up if you need to debug. For example, you typed 1 + +, and you meant minus. How do you fix that? Is it enough to press minus? Do you have to push the clear button? Do you have to push the clear button twice? What do you have to do to correct that mistake? Functional models have the property that you can get yourself into binds where you will need a more detailed model. Functional models are good at early stages, in order to get people going on a system, but they're no good as a final state of your understanding. They will need to be supplemented by other kinds of models, replacement machine models or the equivalent.

Incidentally, you should be reading this as a story of understanding other things, not just computational systems, understanding arithmetic or physics or something like that. You can understand arithmetic in an axiomatic way that's equivalent to having the kind of replacement machine, uniform, context independent, explaining every detail. An axiomatic system is, of course, one way of understanding arithmetic, but it is hopeless to begin there. Instead, you really ought to begin with functional models, with showing how arithmetic does things in the real world; to a certain extent, this is common sense. In designing Boxer we are, in part, trying to develop common sense into a more rigorous understanding of the ways that people can understand computational systems, and also other parts of the world.

Finally, I want to show you that there are even more subtle, more confusing ways that people give meaning to a computational environment. What I have done here is write the definition of a program in Logo. It defines a procedure called SQUARE.

```
TO SQUARE :SIZE
FORWARD :SIZE
RIGHT 90
SQUARE :SIZE
END
```



Now, one of the shocking things about this peculiar procedure definition syntax is that it's very easy for people to use; they hardly ever get confused about it. It's one of the most successful things that I know about the Logo language. Nobody complains "Gee, that's funny, that's very strange." Yet from the point of view of having a simple replacement machine for Logo, this is really quite awful. The understandability here comes from a totally different set of notions. For one, it comes from importing knowledge of the English language. The title line, TO SQUARE, the infinitive form of the verb, is something that people are used to dealing with as a definition. "To swim, wave your arms." So "to square, go forward" and so on. So language is an important mechanism that brings meaning to this operation.

Let me mention another. If you just sat down and typed a series of commands in Logo you would have the same format as a procedure definition: one line, then another line, and then another line. So there's another means, a visual metaphor, if you like, for understanding what the procedure definition syntax is.

Why is there a : in front of SIZE? That's a good question to ask somebody who knows a lot of Logo because they can never give you the right answer. They will say something like, "Because SIZE is a variable, and variables have dots in front of them," and that's the wrong reason. The real reason is because when you ask people why that has dots in front of it, they will say, "Because it's a variable." We're setting things up for a rationalization. They will be happy with it. That will cause them to remember that they ought to put dots in front of the thing, and they'll go a long way before, if ever, they will realize that from another point of view, you really shouldn't have those dots there.

These are what I call distributed models. What you are doing is having a computational construct, and you are counting on people grabbing little pieces of knowledge to attach to this from all sorts of different places. So it's not a monolithic model. It's not even a functional model. It doesn't have to do particularly with getting something done. This is a very strange way of understanding in most psychologists' view, particularly since some parts of it may just be wrong. But it's one way of remembering and giving meaning that I think is very important and one that we really have to design in. We have to realize that if we are going to teach anything like physics or mathematics, that the way people come to understand things is by building models of this quality, as patchworky as this is. That's the way people think for the most part, and you have to deal with that.

Now I'll provide you some contrast. Suppose you didn't do it this way with

distributed, patchwork models. You don't have to do it that way. In fact, from another point of view you really ought to do it differently. So what else is there? Let me fix this procedure definition, piece by piece, so that it's "right" in Logo. Firstly, things in Logo that are not defined procedures ready to be executed really ought to be quoted. The quote mark tells you, don't execute this thing now, because, for example, maybe I haven't told you what it is yet. So in making a procedure definition you really should quote the title, "SQUARE, if you take a uniform view to Logo syntax. Secondly, you ought to quote the SIZE as well. The dots really mean fetch the value of this variable. People rationalize the dots as a marker for a variable-like thing, but that isn't what it is in Logo. It means "fetch the value of this thing," and in a procedure definition you don't have a value to fetch yet. If you put :SIZE, you are saying "give me back the value of this thing," and you haven't defined its value. So again you should not have the dots there, according to a uniform interpretation of Logo. Finally, the rest of the procedure definition is a series of lines. Well, a series of lines is not a data object in Logo. It's really no kind of regular Logo object at all. The only data objects that there are in Logo are lists. (There are words as well, but they don't do you too much good in this context.) Instead of having a series of lines, you really ought to have a list of lists.

So this is in fact what a procedure definition should look like if you have done things to be maximally consistent.

TO "SQUARE "SIZE [[FORWARD :SIZE] [RIGHT 90] [SQUARE :SIZE]]

You can realize that this would just be a mess; this would just be totally unrememberable, something that would just boggle beginners' minds. They would have no way of getting into that before they have learned all of the details of Logo syntax, data types and all the rest. Instead, we rely on rationalizations. We rely on partial understandings. We rely on visual metaphors. We rely on all kinds of other junky kinds of knowledge to get them going. I believe that it's very important in all sorts of educational domains to realize that the way people build understanding is with models of this quality, this kind of patchwork quality.

I'm not going to try and go into detail showing you how we implemented each of these kinds of models in Boxer. We did spend a lot of time making sure that we had a relatively consistent, relatively small replacement machine model, surrogate model. On the other hand, we made very sure that we have a lot of other kinds of ways of understanding, these distributed models and functional models.

I've told you about some of the methods that we've used to make Boxer a more understandable, more integrated, more powerful computational environment. We're not entirely done. In particular the machine that Boxer runs on currently is both too expensive and much too big to carry around so that I could show you the system. But this is our next project. In a couple of years, you should be able to see a Boxer and play with it for yourself. And a few years after that (we're shooting for 1990) we hope Boxer will be a mainstay for educational computing.

REFERENCES

"A Principled Design for an Integrated Computational Environment," **Human-Computer Interaction,** Vol. 1, No. 1 (1985), pp. 1-47.

"Notes on the Future of Programming: Breaking the Utility Barrier," in **User Centered System Design: New Perspectives on Human-Computer Interaction,** D. A. Norman and S. W. Draper (eds.), Hillsdale NJ: Lawrence Erlbaum, 1986. In the same volume: "Models of Computation.

Andrea diSessa has recently taken up a position in the Graduate School of Education at the University of California at Berkeley. Previously he worked in the Laboratory for Computer Science at M.I.T. with the developers of Logo. He devised the dynaturtle, a Logo-driven creature with which learners might explore some laws of physics and, with Hal Abelson, wrote Turtle Geometry, a substantial book on mathematics using Logo. As well as the development of Boxer, his current research work is concerned with learning processes, particularly in science.