HUMAN-COMPUTER INTERACTION, 1985, Volume 1, pp. 1-47

A Principled Design for an Integrated Computational Environment

Andrea A. diSessa
Massachusetts Institute of Technology

ABSTRACT

This paper aims at the *principled design* of a computational environment; it aims at being as explicit as possible about the space of possibilities and about the assumptions made in choosing from among them in the design process. The point is to develop a more systematic, if not yet scientific, basis for the design of complex but understandable artifacts. The particular object of design here is a simple but multifunctional system for naive and inexperienced users.

We begin theoretically by elaborating the notion of understandability, the key characteristic for which we must design. We present various models people can make of computational systems, each with its own learning curve, advantages, and disadvantages. Then we propose a pragmatic framework for a particular system. The framework includes the principle of naive realism: that users should be able to pretend that they see the system itself in the display. It also includes the pervasive use of a spatial metaphor whereby users' commonsense spatial knowledge is used to make the system easy to understand. The theoretical and pragmatic levels are linked, in that a number of important decisions about issues (such as reference, scoping and the meaning of evaluation) are based on the theoretical modeling considerations.

Author's present address: Andrea A. diSessa, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

CONTENTS

1. INTRODUCTION

It is certain that in the future most computer users will be people who are not computer specialists, people for whom the computer must be a useful tool for their own interests without requiring inordinate computational sophistication or effort. Secretaries, financial analysts, scientists of all kinds, teachers, students and trainees, hobbyists and homemakers will be among these users. We are convinced that most of these people will be best served by providing an integrated environment that has broad functionality, not via a great number of special subsystems, but via common facilities accomplishing basic tasks such as the following with a uniform, easily understood computational scheme:

Text-editing, including structured filing and retrieving
Using and modifying prewritten programs
Writing programs, including text-editing macros

Searching and manipulating databases
Producing graphics with a flexible, programmable graphics facility

Although we shall not pursue the argument for integrated computational environments in detail here, the dominant point is that even if novices *need* not deal with several of the above functional categories, it is still clear they could often benefit from doing so. Given that, there is an advantage to any system in which learning any one function automatically carries competence into other areas. In our view, it is so obvious that such synergistic effects can be accomplished that the only surprising thing is how little work has been done on integrated computational environments suitable for naive and inexperienced users. The Xerox Star and Apple Lisa begin to approach the goals stated here but fall short of the generality and flexibility we intend, most notably in user programmability. We refer the reader to the Smalltalk, Lisp machine, PIE and Interlisp projects (Goldberg, 1983; Goldstein & Bobrow, 1984; Greenblatt, Knight, Holloway, Moon, & Weinreb, 1984; Teitelman & Masinter, 1984) for other work on integrated environments, work which, with the exception of Smalltalk, has had little concern for unsophisticated users.

What issues arise in the design of an integrated computational environment for novices? One stands out above all others. That is the *understandability* of such a system as perceived by its user. We must therefore try to understand the mental models people form of a complex system, such as a computational environment, in order to design an effective one.

Unfortunately, cognitive science and psychology have not yet provided the well-elaborated theory and empirical studies of understandability one would like to have before beginning a design exercise. There do exist some promising theoretical beginnings (Gentner & Stevens, 1983; Rumelhart & Norman, 1981; Young, 1981), a small but growing empirical base (Ehrlich & Soloway, 1983; Mayer, 1981), and a few designs with an articulated set of principles (Eisenstadt, 1983; Ingalls, 1981; Smith, Irby, Kimball, & Verplank, 1982). Still, designing for understandability is more art than science. This paper derives from the conviction that trying to articulate principles in the process of design can advance the state of the art by developing explicit and testable general ideas in a context where the actual impact of those ideas in selecting and generating design alternatives is visible.

The first part of this paper sets out some understandability principles for integrated computational environments, largely by identifying paradigmatic models that users make of complex systems. Each model has its own strengths and weaknesses. The view of understanding provided by this typology highlights some important tradeoffs one makes in designing a system to be understandable in one way or another. In particular, it suggests the possibility, even the necessity, of using different models for different purposes and of a gradual shift in the kind of model employed as a user becomes more experienced.

The second half of the paper applies these principles to the design of a computational environment called Boxer, which is being developed by the Educational Computing Group in MIT's Laboratory for Computer Science.[1] Besides combining many capabilities for nonexperts, Boxer has several other novel features. Foremost, it integrates the user interface much more tightly than usual into the meaning of the system. This allows the user to take the stance of a *naive realist*—that what he sees and manipulates on the display screen is the system itself rather than simply an interface to actions which manipulate the for-the-most-part invisible system. Such a strong form of "what you see is what you have" enhances the communication of a model of the system. Boxer also makes use of a pervasive spatial metaphor in which language structures and relations are expressed by the spatial relations one sees on the screen. The intent is to tap the well-developed pool of knowledge about space that humans already possess in order to facilitate their making a model of the computational environment.

Readers who wish a preview of Boxer may look ahead to Section 5 which gives some scenarios of its use.

2. PRINCIPLES OF DESIGN

In Subsection 2.1. we lay out an important distinction—that between structure and function; and, as a basis for further discussion, we use that distinction to take the first few common-sensical steps toward principles of designing an integrated computational environment. Later subsections refine this initial distinction by presenting various *mental models*.

2.1. Structure and Function

The distinction between structure and function, long considered fundamental to the process of design, is particularly central to the design of an integrated computational environment. The distinction focuses on either (1) characteristics of an object or action which are defining and independent of specific use (structure) or (2) characteristics which have to do with specific use, consequences, or intent (function). The point is to separate descriptions according to whether the implied descriptive frame applies universally (structure) or not (function).

For example, the structural aspects of a variable in a computer language are given primarily by setting and accessing protocols, and they apply in all contexts. In contrast, a variable's functions might sometimes be described as *a flag*

[1] Boxer has been partially implemented, but to simplify exposition and more clearly highlight the modeling issues, what is described here is neither the same as implemented, nor precisely what we intend to implement. The differences are, however, mostly cosmetic.

or, more generally, as a communications device. In other cases, a variable might function as a *counter, data,* or *input.* In the last case, input, there is only partial structural overlap with the generic variable; the function known as input requires local scoping and flow-of-control organization to allow the procedure to have its inputs bound when it executes.

The most naive attempt at integration, that is, simply arranging for separately designed functions, such as system monitor, editor, programming language interpreter, and so forth, to be accessible from each other, leads to difficulties that are understandable in terms of function and structure. Structural elements in each area may differ because they are tuned to the functions of that particular area. Or worse, the structural elements may differ for no principled reason at all. Thus the syntax for typing a command may be different in the system monitor than in the programming language. Commands in a text-editor, for example, single keystroke activated commands, may have no written representation in the sense that there is no way to write and later reactivate useful combinations of commands. Only slightly better, a separate macro language may be provided for the editor. Such practices result in more to learn for the novice and confusion between similar but not identical structures. Worse case scenarios involve the modality of such systems where identical actions serve radically different functions: a "d" keystroke may delete a file rather than begin the insertion of "dog" because the user was in the file handler rather than the insert-mode editor. More seriously, one is deprived of the power of structures considered important in one area but not salient enough in another to warrant implementation: Should one be deprived of the power to write a new program built out of a few primitives in the text-editor simply because it is not a part of the usual functional specification of an editor? Is instant action on keystroke command useless for a programming environment?

We identify a basic design heuristic for integrating different function areas so as to avoid the above difficulties. *Detuning* means having general structures underlying the computational environment that are broadly applicable, less highly tuned to any specific function, and always available for use. The designer must seek to abstract the essence of several contexts into common structures available in all contexts and must expect any highly tuned application to be built out of provided structures rather than directly provided. In Boxer this will mean such things as the existence of exactly one kind of data object; there will be no essential difference visible to the user between strings, numbers, lists, records, and so forth, except the way in which the data object is used. Even variables and files will be essentially the same structure. Furthermore, text-editing commands will be a part of the programming language and hence usable in the same way other primitives of the language are. Any user-defined procedure can be given function key status in the same way editor commands are.

Detuning often entails and can be aided by *diffusing functionality* across several structures. That is, instead of having any one construct serve a particular

goal, several constructs can be involved. Comments document code, but so do mnemonic names. In Boxer, things may be named just for documentation purposes, not only for the usual purpose of defining an object for future computational reference. In addition to aiming at a simple syntax, Boxer will have interactive parsing and prompting to help novices avoid the need to remember parsing conventions.

Detuning and diffusing functionality aim at structural parsimony, reducing the number of structures to a small, common core. They are directed against what we call the *hacker bug* of implementing a structure for each identified function. By nature, integrated computational environments are systems in which we cannot afford a one-to-one connection between structures and function. Needless to say, this applies within as well as across large-scale functional domains. Reducing the number of structures within programming itself is an important goal.

Detuning, of course, can be carried to an extreme. Assembly language is both excessively detuned and useless to the novice. The topic of knowing and negotiating the limits of detuning will occupy a good deal of attention later in the paper. At this stage, we begin dealing with the problems by proposing the heuristic method of *shallow structuring*; anything the novice is likely to need to use or modify must be near the surface of the environment, that is, it must not require inordinate skills or knowledge to access. The problem with assembly language is that it is too far from ordinary functionality; large numbers of instructions are needed to implement simple functional units such as saving a file.

That a system composed of a large number of similar but subtly different structures is hard to learn and prompts mistakes is an assessment that motivates detuning. Counting structures is a plausible method of assessing complexity, but it has clear limits. Shallow structuring reminds us that the way structures need to be combined to achieve common goals must also be included in assessing complexity. But counting structures or even combinations of structures fails to recognize the fundamental fact that some individual ideas or combinations of ideas are much easier to learn or apply than others. To deal with this, we must have a more elaborate definition of learning than acquiring an idea. We must look more carefully at a hypothetical user's mental models of a system, that is, how a user comes to understand in order to control the behavior of a system. In this way we will get a clearer picture of the limits of detuning and structural simplicity in general as a way of achieving an understandable system.

2.2. Mental Models and Surrogates

The words *mental model* often conjure up the image of a sort of replacement machine located in the mind on which one can run experiments and envision results without touching the actual machine. Young (1983) calls coherent,

runnable conceptions *surrogate models*. For example, one typically models a "push-down" list as a physical stack on which objects may be piled and removed. A push-down list, of course, does not behave identically to a pile of objects, for example, overflow versus gravitational instability; but the image of a pile manipulated with put and remove operations does allow one to simulate its important behavior.

A surrogate model is intended to capture the computational mechanism in such a way as to offer explanation and correct predictions in uniform terms. It establishes the bottom line for the user for saying what will happen, given the present state, and for saying what state must have existed, given a particular behavior. Surrogate models are almost always explicit and taught, and one typically sees them invoked by a tutor when a novice's expectations have gone awry. Two of the most elaborate surrogate models, which attempt to encompass a large part of the operation of a computer language, are the actor model of Smalltalk (Tesler, 1981) and the versions of Papert's little-man model of Logo done by the Edinburgh Logo Project (du Boulay, O'Shea, & Monk, 1981). One possible surrogate model of a language is a specification of its implementation, though this is hopelessly inadequate pedagogically for technically unsophisticated users.

Surrogate models are close to what many people think it means really to understand a system. Surrogate models are archetypically detuned, dealing with a system independently of how it is used; they are prone, however, though not inherently, to certain problems as a way of giving users effective control of a system. Problems arise in the following areas:

1. *Learnability:* Since surrogate models aim at being uniform views of rather complex systems, they themselves tend to be complex. When pushed to cover all behaviors of a system, they can lose their simplicity because of ad hoc elements for dealing with loose ends. For example, though Smalltalk's actor model demands an object destination for each message, there are occasions when, for good reasons, a message must be assumed to be sent to the interpreter, a nonstandard Smalltalk object.

More particularly, because of their compact, tightly interconnected nature, surrogate models require a good deal of learning before they can be applied to even simple, everyday events. Many of the little-man model's behaviors are not precisely what one would expect, without a fair amount of coaching, of a little man. For example, a little man (procedure invocation) must "sleep" and call another little man for a subprocedure call. A simpler interpretation which is good enough for most purposes is that the subprocedure just gets done. Thinking in terms of the model actually complicates understanding early elements of the language. Incremental learnability is sacrificed for the sake of uniformity and completeness.

2. *Styles of use:* Surrogate models are typically slow and time-consuming to run. They are good for debugging, that is, for tracing an unexpected behavior step-by-step; but routine, relatively fluid interaction with the system cannot be

expected as a direct result of acquiring a surrogate model. At the very least, degrees of automation, compiling frequently used operations and the like, occur.

More fundamentally, the very kind of knowledge available with surrogate models makes them unsuited to certain tasks. Consider: A surrogate model is almost always unfortunately far from the task of inventing a way to effect some intended result. For example, in planning a program the specification of the goal out of which the plan must arise is typically made only in rough outline, and it is almost always functional. In contrast, the perspective of a surrogate model with its aim of comprehensive prediction is structural. Indeed, de Kleer and Brown (1981) argue convincingly that a model aimed at unfailing and comprehensive explanation must be structural and not functional. Hence, a fundamental gap exists between the functional level of description needed for planning and the structural level of the surrogate. Some examples of this gap follow.

Suppose one wants to communicate some information from one procedure to another. One thinks of using a variable not because one knows how a variable works but because one knows a variable can be and often is intended to have the effect of *information transmission.* In Pascal one often uses the PRINTLN command because one wants the side effect of moving to the next line, not because one wants to print a null line. Not surprisingly, novices must usually be taught this "hack"; it is a potential function not easily seen in the meaning of the command. An example more germane to later discussion is that either a variable or a function might perform precisely the same role in an expression, to provide a needed value. The considerations which dictate whether one chooses to implement that role with one or the other structure may be totally invisible to the functional semantics the programmer attributes to the symbol. For this reason it is troublesome, at the least an unnecessary burden on the programmer, if the syntax of the language requires distinctive visual form for structures that can have the same function. We generalize this argument later to argue for syntax in which structure is minimally intrusive.

Learning a command or construct entails learning important side effects of that command which can be exploited to attain particular ends as well as learning a context-free specification of the meaning of the command. It involves learning typical uses of the construct, what one might call *teleology.* It involves learning plan fragments, that is, sketches of ways to accomplish things using the construct, such as the counter paradigm for the use of a variable, possibly in paradigmatic combinations with other constructs. All of this functional knowledge is not germane to a surrogate model but is clearly part of understanding a system. See Ehrlich and Soloway (1983) and Waters (1984) for work on determining this functional vocabulary and for references to related work. Although we have concentrated on planning and related activities, it should be evident that a functional vocabulary is important to other aspects of programming as well, for example, understanding already written programs.

Overall, then, there are two fundamental problems with surrogate models.

The first involves their complexity, which leads to a related problem, that is, that they are slow to run and hard to learn. A design heuristic which concentrates on the problem of their complexity is to construct a language deliberately to have a simple surrogate model by selecting the outline of the model and building structure and syntax around it in such a way as not to lose any necessary functionalities. Smalltalk has followed this line, beginning with the root actor and message-passing model. We shall use this heuristic method, albeit carefully, since the second fundamental problem remains even if we achieve the basic goal of a simple surrogate. Specifically, the construction of a broad class of functions out of a tiny set of structural elements almost necessarily involves great cleverness. While systems designers may be very fond of these hacks, the novice is generally less appreciative. More generally, functional understanding is not well supported by a surrogate model. This side of a system will require specific tutoring; thus, the advantage of a small number of universal structures over a larger set more specifically tuned to important functions is not clear. The simpler surrogate may overall be at a distinct disadvantage.

Earlier we warned against the obviously problematic hacker bug of providing a structure for every function. In this section we have seen how considerations of understandability demand caution against the opposite extreme, the *formalist bug* of providing a sparse set of primitives out of which to build all functions. Since even a simple surrogate cannot ensure a learnable, understandable system, we turn now to complementary methods.

2.3. Functional Models

The fact that surrogate models are removed from application, that is, detuned, makes them attractive as universal, mechanistic ways to understand a system, but such a perspective slights functional understanding. This section briefly explores an alternative to surrogate models. It is a *modified hacker strategy*—not providing a structure for each function, but at least making some of the basic structures emerge directly from functions that are already understood or are easy to learn. As we remarked earlier, in making such *functional models* we must take care not to (1) have too many; (2) tune them tightly to traditional functional areas; (3) forfeit the possibility of using an effective surrogate when that is likely to be needed, like debugging. The pattern of learning, then, would facilitate acquiring a few important and generally useful aspects of the language as solutions to specific problems.

To consider the advantages and disadvantages of this possibility, we turn to a case study. Young (1981) details an archetypical functional model. It happens to be a model of algebraic calculators and works as follows. By typing in a problem, say 3 + 5, in a way which maps trivially to doing the same thing with paper and pencil, one has set up a context where what should happen is obvious: one wants the answer. Pressing = does precisely what one wants in

that context, namely, gives the answer. "Doing what one wants" in a prototypical situation is a sufficient model of the system for many purposes. The general schematic of functional models is that one has a descriptive frame, in this case doing arithmetic, that includes recognizable objects and actions such as "writing the problem" and "getting the answer." Then the user can understand computational constructs and actions as they function in this frame. + is part of writing the problem and = means "give me the answer." Functional models might be described simply as rules, for example, "to get the answer, press =." But to do so ignores the fact that such a rule is memorable precisely because it fits into a previously understood schema of goals and means, a descriptive frame like written arithmetic.

Functional models obviously have some defects. Unlike surrogate models, one cannot expect the general behavior of the system to be evident in a specific context. For example, the state of the system if one were to type 1 + + is not constrained by the prototypical writing-the-problem model of +. If one expected novices to need to interpret situations equivalent to 1 + +, such as understanding some prewritten code, or if the functionality achieved through 1 + + were important and not conveniently achieved through other means, one would certainly beware relying on this functional method of giving users a model of the keystroke +. Functional models provide restricted understanding. The descriptive frame will typically be weak with respect to structural aspects of the situation, for example, the internal state of the calculator after pressing 1 +. This knowledge is important for debugging and similar tasks, such as knowing what to do to correct a mistaken + when − was intended. More generally, certain combinations of structures (1 + + is an example) will be entirely meaningless in the functional frame, even though such a sequence of keystrokes might be not only legal, but useful. With some calculators 1 + + defines a constant calculation. If a structure is to serve several functions, therefore, a single functional projection often will not be sufficient to allow the user to understand or generate the other functional descriptions.

Functional models provide a view only of part of the system and that only with respect to a specific frame of analysis. Obviously, one needs a repertoire of models; the notion of a single one is tenable structurally (surrogate) but not functionally. With respect to the strengths of a surrogate, this fragmenting of understanding shows weaknesses. On the other hand, from the point of view of incremental learnability, teleology, and other important aspects of understanding, functional models can be superior precisely because of their contextual specificity.

2.4. Distributed Models

In this section we describe a kind of learning similar in some ways to that described for a calculator in the previous section. But in this case, the model is

accumulated through a spectrum of partial understandings, not by virtue of a single functional frame. We think such learning and the models derived from it are vital to understanding complex systems, even if they appear at first to be even less tractable as a basis for design.

The following example comes from an early stage in learning Logo. It is striking because it shows a quite successful learning sequence that cannot be accounted for in terms of either simple surrogates or functional models.

Logo beginners almost always start by driving the Logo turtle (a graphics cursor) around with commands like FORWARD 100. It seems certain that elementary school students interpret FORWARD 100 essentially as an abbreviation for an English sentence like "go forward 100 units." The need for input to the command FORWARD is not, therefore, understood structurally but according to the semantic need to complete the sentence FORWARD <how far?>. Linguistic and semantic function here come first and provide a preliminary model of the structure *command plus input.*

When students are taught to define their own procedures, the metaphor of teaching the computer how to do a new thing is invoked. One types TO SQUARE :SIDELENGTH followed by the list of commands defining square, as in the following recursive program:

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

In structural terms, TO SQUARE :SIDELENGTH is part of the syntax for defining a procedure, but it is easy to see that the syntax is intended to continue the interpretation of a procedure as a verb. The English infinitive is frequently used definitionally, a function that the Logo procedure-definition syntax aims at inheriting. Furthermore, input specification follows the same form as the FORWARD 100 sentence, which is also acceptable English, something like "to go far." Abstractly, one sees a problem (teaching a new verb) and a solution (TO SQUARE ...), all of which relies heavily on a knowledge frame, English, that is rather systematically used to help Logo beginners.[2]

Not all aspects of the syntax for definition are meaningful within the linguistic perspective or within the functional frame of "teaching a new word." In par-

[2] Some indication of the importance of these linguistic considerations comes from teaching Logo to non-English speaking students. In Portuguese where it is much less natural to use a word like *square* as a verb, difficulties are encountered (personal communication, José Valente, April, 1983). In Japanese, verbs usually come at the end of a sentence, and the FORWARD 100 command form seems harder to appropriate (personal communication, Leigh Klotz, April, 1983).

ticular, the use of : deserves attention. In Logo the : (pronounced "dots") denote the value of a variable and are included in the definition syntax to parallel and reinforce the pattern of invocation of procedures with variables as inputs, for example, FORWARD :SIDELENGTH, or, more particularly, to parallel recursive-call format, such as, SQUARE :SIDELENGTH in the final line of the above procedure. Another visual metaphor is provided by the pattern of commands making up the definition of SQUARE, which is the same line format one would see on the screen if one just typed them in for direct execution. These consonances are subtle but contribute to learnability without interfering with the linguistic frame.

It is important to note that the : marker as part of definition syntax has support other than from visually matching the pattern of typical invocation; namely, it has a simple rationalization—to distinguish variable inputs from the procedure's name. Note also that variables are distinguished by the form most characteristic of them, getting a value. Novices will often respond directly to queries about the definition syntax with such rationalizations: "SIDELENGTH is a variable," or "It's just like when you write SQUARE :SIDELENGTH" (recursive call). Implementations of Logo that changed the syntax to TO SQUARE SIDELENGTH have prompted complaints from novices whose rationalizations were violated.

The learnability of the procedure-definition process in Logo is due to its naturalness as a solution to a particular problem when interpreted in a number of frames, each of which partially explains the solution. We refer to models accumulated from multiple, partial explanations as *distributed models*. The notion of a distributed model is derived from ideas we have developed about understanding complex systems in other domains (See diSessa, 1982; diSessa, 1983). The list of frames for procedure-definition syntax includes:

a clear functional frame (the problem involves making a new procedure; the solution is TO ...),
English,
visual pattern matching, and
rationalizations.

It is easy to emphasize how far distributed models depart from what one would expect if a simple surrogate accounted for all of learnability and if coherence were measured only in structural terms. Logo is a descendant of Lisp, and, as a consequence, function application is the standard control organization for procedures with inputs. Assimilation to that standard would require TO to be a function and SQUARE and SIDELENGTH to be inputs. But then one would have to quote SQUARE and SIDELENGTH to denote the fact that they should not be evaluated as part of executing TO SQUARE SIDELENGTH. Thus one should write something like TO "SQUARE "SIDELENGTH followed by the

body of the procedure as a list of lists (the lines of the procedure definition). Not only is there loss of template matching to a typical use of the defined object, but there is no problem-specific rationalization for the syntactic markers. The different functions of SQUARE and SIDELENGTH are not marked, and TO is separated by syntactic marks from its close "English" partner, SQUARE. Beginners would need to memorize the syntax with essentially no semantic or experiential support. Of course, for the computer experienced, the syntax would have a great deal of meaning having to do mainly with the advantages of uniform, context-independent structures. But that doesn't help the unsophisticated user. For comparison, we note the functionally opaque, but structurally unexceptional, form that is actually provided by many Logos as an advanced version of procedure definition.

```
DEFINE "SQUARE [[SIDELENGTH] [FD :SIDELENGTH]
               [RT 90] [SQUARE]]
```

Distributed models involve learning by prototype in the sense that users learn a construct primarily by example. An instructor often simply shows the student how to do something, much like a parent teaches a word like *dog* to a young child by pointing to one. One does not, at least initially, expect to provide the user with elaborate explanations of the details of functional context, why each symbol appears, and so forth. Instead, rationalizations and other partial understandings provide a backbone of reasonableness that allows the user to remember the form of the construct and understand some of the variation possible.

These are aspects of procedure definition that make it particularly apt for use with a distributed model. The problem context for defining a procedure is easily understood in naive terms. As important, the problem solution is frequently enough used that the model will not be dangerously undermined by other experiences. For example, once one understands the fundamental structures of Logo, procedure definition is clearly exceptional. Yet because it is used so often, the syntax never comes to feel unnatural.

One must consider long-term effects of particular functional and distributed models. Some will remain and be integrated as special cases. Visual pattern matching is an example. Some will fade away naturally and be replaced where appropriate by surrogate models. No learner believes Logo is English for very long. But we must be aware that globally destructive misconceptions may be fostered as well as misconceptions which, ironically, are profitable.

In summary, a structurally simple language (one with a simple surrogate model) is in principle ideal for post hoc explanation, debugging, and prediction, but can fail to be generally useful by not being incrementally learnable and sufficiently close to the functional terms in which problems are phrased. Our discussion has not only mapped out these typical failure modes but also

proposed building, at least at early stages, less coherent, but still effective, models based on function or on compatibility with a collection of partially explanatory frames. The hoped for pattern is that the few initial structures which a beginner encounters have the following properties: (1) They provide sufficiently broad functionality through simple variation on the prototype to support many activities. (2) Those structures will be understandable on the basis of naive functional and distributed models. (3) The initial models lead unproblematically, through teaching and experience, to the appreciation of a moderately simple, relatively complete surrogate model.

Having sketched in general terms a set of issues that we see as important to understandability, we turn now to more detailed assessments based on particular knowledge—model-building material, which users may or may not possess. Given the general strategic decisions made for Boxer, we shall find that static organization of the system lends itself to a good deal of structural collapse to a small core. But for dynamic aspects of computation, other strategies are necessary, including proliferating structures to allow tighter functional match.

3. A PROPOSED FRAME FOR INTEGRATION

The visual medium has served a more and more important role at the interface between man and machine, particularly since the advent of bitmap displays. But surprisingly little use has been made of the medium to develop and support user models rather than simply to expand the bandwidth of the interface in the amount of data available at any given time or to facilitate the operation of the system for already comprehending users. We would like to use the video screen, in contrast to pop-up menus and iconic mnemonics, to attack the fundamental problem of understandability of the basic organization and operation of the computational environment.

The means we intend to use are twofold. First, all computational objects will be created, represented, and manipulated in essentially the same way, and the user will be able to pretend that the objects themselves are their visual representation. This useful fiction we call *naive realism.* What we want the user to think he sees on the screen is the computational system itself rather than a multiply-filtered view or one dominated by side effects, for example, having a window occur in some place and size because of what was available on the screen when the window was created. Not only does it provide a high-bandwidth communications channel to the system, generally enhancing rationalizations, visual metaphors, and so forth, but the means of modifying the visual representation assumes the role of a universal language of constructing or changing, the equivalent of manipulating the physical world through the universal interface of touching, pushing, and pulling. This characteristic provides a strong structural base for understanding the system. Taking naive real-

ism this seriously, in fact, separates this proposal most strongly from previous computer-language designs. Even those designers who are willing to divert resources like the display screen from highly tuned functionality to understandability have almost universally opted for user interfaces which act as buffers or façades to hide system complexities from the user rather than to search for a simplicity which could be shown. See, for example, Innocent (1982) or Goldberg and Robson (1979), who propose producing understandable systems by filtering genuine complexity into simplified visual forms.

Our second general means of using the video screen to enhance understandability is a comprehensive *spatial metaphor.* Spatial organization will have strong semantic content: Elements of the environment will have or be places, and their visible spatial relationships will have structural meaning. Humans have a great deal of knowledge and a broad collection of skills for dealing with space. We happen to live in a world that is profoundly geometric in the sense that objects and places are salient and fundamentally important. The dominant mode of interaction with the world is to rearrange objects, including oneself as an extremely important special case, into different configurations rather than, for example, to pass messages between abstract, placeless entities. It is not that we are not impressed with the power of actor-based languages, but the amount of work that the actor metaphor does in promoting understandability, besides providing a uniform syntax, is problematic. Things might be different if our primary sense were not vision but hearing, where messages are more salient. As things stand, spatially carried meaning is much richer as a way of making a system understandable to unsophisticated users.

Use of the spatial metaphor is dependent on the fact that spatial structure can be extremely compatible with essential computational structures. In particular, it will become apparent how two-dimensional configurations with containment representing hierarchy can subsume an important core structure to things like program and calling structure, hierarchical data, and file systems.

3.1. Static Structures and Functions

The Box

Essentially all static objects and configurations will be derived from a single object called a *box.* A box appears on the screen as a rectangular region with the interior containing the box's contents, predominantly text. The choice of text as the main surface stems from the idea of explicitly importing some natural language familiarity into computation and from the fact that text manipulation is itself a goal of an integrated environment. Boxer is "editor-top-level." One always talks to the system through the editor; it is the universal interface to the system. As details emerge, it will become apparent how text manipulation and program writing are intended to be mutually supportive activities;

and, from the point of view of learning, either can serve as a good introduction to the other. Our editor is Emacs-like (Stallman, 1984).

Boxes may contain subboxes, either named or not (Figure 1). Boxes are logically, as well as visually, two-dimensional arrays in the sense that they are a sequence of text lines, each of which is a sequence of words or boxes. The abstract structure of a box, a hierarchical two-dimensional array, is what allows us to build almost everything needed in Boxer without violating shallow structuring. Names of boxes must be words. This : meaning *name of,* should not be confused with Logo's :, meaning *variable value.*

We intend one of a beginning user's first activities to be wandering around in the system itself, inspecting it. This can be accomplished simply by moving a cursor around (our prototype system, built on a Symbolics 3600 Lisp Machine, uses a mouse), expanding and shrinking boxes. Boxes have three display sizes: (1) fully shrunk so that no detail shows, (2) normal (as big as necessary to show all its contents), and (3) full screen. A fully shrunk box, expanded twice, fills the full screen so that no part of its containing box can be seen. Two buttons on the mouse specify "expand" and "contract." Creating, deleting, and moving boxes are simple functions of the editor. The easiest way to create a box is with a "make box" key. Boxes behave as large characters; for example, the delete key erases a box as if it were a character. Typing on the left side of a box causes it to move to the right, as any character accommodates to an insertion in screen-oriented editors. Section 5.1. gives an extended example of some basic functions of the editor and boxes.

Boxes as Procedures

Procedures appear as boxes. Subprocedures like SIDE in Figure 2 may be written directly into procedures as subboxes, giving the functionality of visible block structuring. These subprocedures may be named for mnemonic purposes, as can any box. This is especially useful when one wishes to leave a box shrunken, suppressing detail for clarity. In many of the examples to follow we use turtle graphics and essentially the syntax of Logo. We have not settled the issue of syntax, but Logo is a reasonable approximation to our current best guess. In particular, Logo is line oriented rather than expression oriented, as is Lisp; and we have appropriated line orientation for Boxer.

To make all aspects of a procedure concrete and spatially accessible, in particular local data such as inputs, we need an additional structuring of a box. In fact, having data local to a box, but other than its contents, is so generally useful we declare that every box has a *local library* located in its upper right hand corner. This contains definitions of any local symbols, which may be used within the box and in any box contained (recursively) in that box. Containment implies inheritance. That is essentially all there is to scoping in Boxer, but details will be treated in Section 3.2. The procedure in Figure 3 has an input, NUMBER, and draws a polygon of NUMBER sides and sidelength LENGTH.

Figure 1. A box with contained boxes.

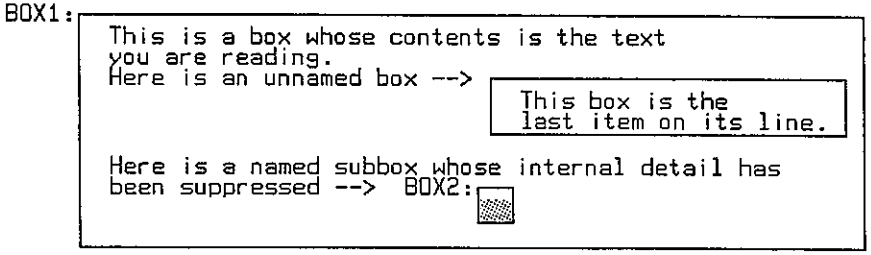


Figure 2. A box representing a procedure.

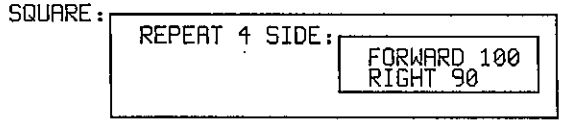
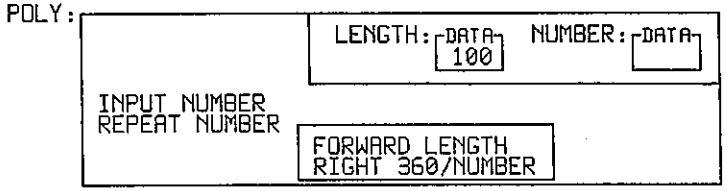


Figure 3. The local library appears in the upper-right corner of a box and contains definitions generally useful in the box.



The value of inputs and local variables will be available for inspection in the library during debugging. It is important that since the local state of a procedure represented in its library may contain procedures and data as well as inputs, one may place such items at more appropriate levels in the hierarchy of a procedure-subprocedure system than at the highest, "global" level. This makes systems of procedures easier to inspect and understand than the unorganized piles of Lisp. Section 5.2. gives an extended example of using box-and-library organization to structure programs for intelligibility and modifiability.

Although the local library is special in that it is not part of the contents of the box in the ordinary sense (for example, it is not executed as part of the procedure), it is structured, inspectable, and editable exactly as all other boxes are. Detail may be suppressed by shrinking the library or parts of it. One is free to arrange the contents of a local library spatially so that the most important procedures occur near the top and so that related procedures appear together. The library's meaning as container of generally useful information about the box makes it a natural place for annotation, documentation, and other help.

It is worth remarking that naive realism means we could entirely do away

with a separate procedure-definition mode, or special form. Instead, users can choose from various concrete methods of construction and modification. One may type a procedure directly into the local library. One may assemble a procedure out of previously written text, for example, out of commands typed in the course of experimentation, then try it out and later move it to the local library. Such fluid interaction between trying out pieces of a procedure and defining it is especially important for beginners. Of course one can also have some procedure (TO) do the procedure-defining work by side effect. But it is not wise to identify the process of constructing a procedure with its static representation, as with Logo's TO or Lisp's DEFUN. That identification is a remnant of teletype interaction, where object creation by side effect as opposed to piece-by-piece assembly, is a necessity. Especially for environments and large data objects, we conjecture that the concrete access provided by our spatial/naive realist approach will prove natural and functionally quite adequate for most purposes.

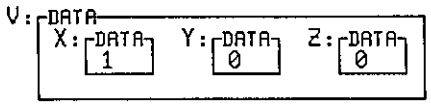
Boxes as Data Objects

The boxes NUMBER and LENGTH in Figure 3 function as variables, and generally boxes will serve to define data as well as procedural objects. Data boxes are a distinct type of box and are marked as such. In contrast to traditional languages which have a number of different structures for handling compound data (strings, arrays, lists, records, and so forth), Boxer's text and box structure is intended to be universal. This structural universality, as with lists in Lisp, should be a source of great power and simplicity. Any arbitrarily large box can be named or passed as an input or output of a procedure.

Lisp's universal compound data structure suffers some of the same problems as a simple surrogate, namely, that list structure is too far from important classes of functionality to be easily appropriated and used. We think the two-dimensional, line-oriented form of a box is better adapted to a broad range of functions than a simple ordered sequence. For example, a box can contain some text laid out in the usual way—words organized into lines. We will not tamper with the box structure per se in tuning even more to specific data functions. Instead, we will add a number of different access routes to parts of the structure that are aimed toward particularly important classes of functionality (described below). We expect these to be learned largely functionally, as solutions to particular problems encountered as the user advances.

One of the most important of compound-data functions is the ability to deal with named subparts easily. Most Lisps have property lists that are often used for this purpose. Pascal has records. Boxes have the capability implicit in the fact that any box or subbox may be tagged with a name (Figure 4). All one needs is an appropriate syntax for selection. We shall use an index notation here; V.X specifies the X subpart of V, and for assignment MAKE V.X 1 means set V.X to 1 in the same way any variable is set; MAKE NUMBER 5 sets NUM-

Figure 4. A vector with labeled subparts.



BER's contents to 5. One can specify any number of levels, for example, VECTORS.V.X. One can access variables contained in a local library as POLY.LIBRARY.LENGTH (Figure 3).

It is important to have *address* names for elements of compound data objects in cases where individual names are inconvenient or require too much overhead. Correspondingly, we will have an alternate vocabulary for specifying parts of a box based on location. It is extremely natural to use array indices into the two-dimensional structure (rows and columns) of a box; for example, RC 1 2 JOE retrieves the item at Row 1 Column 2 of JOE. One would also like to reference rows because of their important meaning (visually, in procedures, and so forth), and to reference elements by their sequence number (reading, as text, left to right, top to bottom), for example, ROW 1 ABOX or ITEM N BBOX.

Boxes as Environments

Boxes and local libraries provide a function that has been much neglected in computer languages, providing an environment. The point is to make available to the user a particular set of actions and objects, but in other ways to minimize the constraints on what to do with them and how to combine them. An environment must provide the ability to select and execute easily any of a set of built-in operations or to define a new operation. Logo's turtle graphics provide an archetypical example where the turtle's behaviors define the territory, but the full power of the programming language is available to combine those basic actions, adding to the environment to satisfy a broad range of student and teacher goals. A contrasting view is that programming is an activity for a programmer (usually a professional), who constructs a program and then gives it to users to run. Whether one views programming in this way or as an opportunity for users also to tinker and create determines whether environments are an expedient or a fundamental. In our view, environments are fundamental.

A box used as an environment—a place to go where a specific set of commands and data are defined—has some advantages over programs or even workspaces which might otherwise serve the same purposes. (A *workspace* is a cluster of related procedures and data usually saved as a single file and loaded together into an interpretive environment.) In contrast to a program with specific I/O, boxes/environments save the programmer from creating (and the user from needing to learn) a special interface. (Section 5.1. shows a mail facility with this property.) Environments allow flexibility in terms of program-

mining on top of what's given, easily accepting a very general class of user-initiated modifications.

Like a workspace, a box/environment simplifies the construction of what might otherwise be a complex, monolithic program by allowing one to build and try out smaller pieces. But an environment in Boxer is both more general (for example, one can nest environments) and better integrated (for example, constructable and editable in the same way data and procedures are). In Boxer one can even try out a subprocedure in context by moving to the place that it appears and executing it (after possibly assigning typical values to the inputs of the superprocedure, which defines the context of the to-be-tested subprocedure).

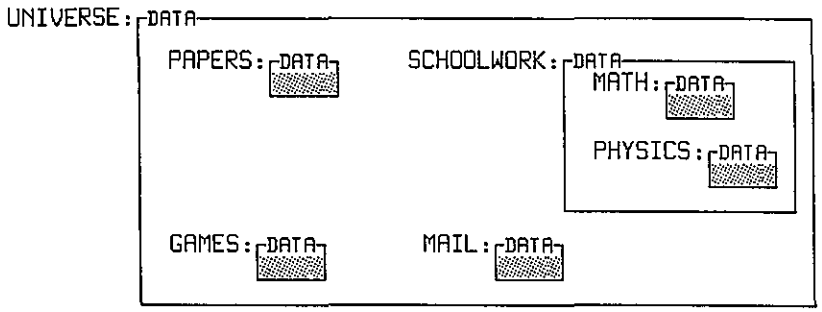
Considering that nested boxes offer a choice of where in the hierarchy to place needed objects, Boxer environments are also more controllable and self-annotating. Lisp and Logo workspaces often get so cluttered with utility procedures that the ones intended to be used at top level are not at all apparent. In Boxer, the local library of an environment may contain exactly those procedures intended for users' direct consumption, with all lower-level procedures appropriately hidden in the interior of the visible procedures. Again, see Section 5.2. for an example.

Time modularity, how one creates natural and stable boundaries in time between sets of activities in a project, is one function of the file-workspace organization that is not taken over by box structure. An example of state-of-the-art thinking in this area is Goldstein's PIE system (Goldstein & Bobrow, 1984), which provides sophisticated control of versions and alterations in software design. We project a minimal capability for Boxer of saving and restoring named versions of any box. Note that this gives much finer control over time modularity than workspace files. One can save versions of a procedure within an environment.

At still larger scales than environments, boxes can serve to organize an entire personal computational environment. One needs nothing more for a hierarchical file system. At the most global level, a box we might label UNIVERSE (Figure 5), the local library can contain documentation on all the system primitives. The contents of UNIVERSE would be the top-level view of the organization which the user chooses for his entire environment.

This use of box structure duplicates much of the functionality of one of the most successful aspects of the Smalltalk programming environment, the Browser, allowing leisurely perusal of the system. However, naive realism means no separate and special subsystem is needed. Browsing amounts to moving the cursor around on the screen, expanding and shrinking boxes, that is, only the most elementary editorial functions. There are no dedicated structures or procedures to do the browsing or to link a new object or change into the Browser. In contrast the Smalltalk Browser, as well as all others that we know of, is a distinct part of the system. Much of the visible structure of Browser is

Figure 5. Box structure can organize the whole computational environment.



special to the Browser, not easily modifiable by users, and the parts that are must be modified through the use of specially learned procedures. It is also true that part of the system organization seen in the Smalltalk Browser exists only for the Browser and does not reflect system semantics in a fundamental way.

Kinds of Boxes

We have been discussing a very wide class of functions deriving from a single structure, the generic box. Although this simplification is appealing, our implementation and other considerations have convinced us that boxes need to be labeled as to type and have slightly different behavior accordingly. We have already mentioned that a *data box* is a type of box distinct from procedure (*doit*) boxes. (Doit type is the default and is unmarked in this paper.) We have graphics boxes in which users can draw and save arbitrary pictures without pretense that these boxes are either as concretely accessible or as uniformly structured as an ordinary box. Currently we use data boxes for environments, but it is plausible that this distinct function is important enough to deserve a separate label to guide users concerning the intended use of such boxes. For several reasons, boxes that contain only text deserve a label and special behavior. One would almost certainly want slightly different behavior for the text-editing facilities, such as sentence and paragraph orientation and automatic justification. Special status for text allows one to use it in the midst of a procedure as annotation without danger of executing it.

It should be clear how much of the structural backbone of a computational environment can be supplied in concrete form by a two-dimensional hierarchical array—the box. We are convinced that the strong identification of "things" with "places" and "organization" with "spatial relationship" (in particular, containment implies inheritance) provides a firm foundation for easy, incremental learning of the system through inspection and through a uniform method of interpreting, modifying, and expanding what one sees. Nonetheless, identification of this sort is a very strong constraint on system organization and possi-

ble interpretations of "running a program" (to be discussed below). In particular, boxes are strictly hierarchical, and each box exists in precisely one place. This hierarchical structure means it is impossible to share in Boxer, as in Lisp (via multiple pointers to the shared object); a Boxer object is part of only one other object. It also means one has only a single view of any object in the system—that provided by the spatial context where the object exists. In contrast, one may sometimes want to see things on the screen that are related in some way other than with respect to their system organization. While running a program in some environment, one might wish to view the changing contents of some distant data box. Or one might want to be looking at some part of the system while constructing another part, say constructing a program in analogy with another from a different context. Window systems were invented partially to serve this kind of function (Kay & Goldberg, 1977).

We do not consider sharing or multiple views of highest importance to novices; still, they are important enough that, for more advanced users, we would like to incorporate them. To meet this need we propose a single structure that provides many of these functions, but which we consider minimally subversive of box semantics. This structure is called a *port* ("view port") and has most of the properties of a box. It appears as a rectangular region that can be named and is constructed and erased in essentially the same way that a box is. But its meaning is a passageway to another part of the system. What one sees in a port is a part of the system located in another place. Thus one can inspect and even change remote objects. The operational semantics of a port are the same as the viewed object. If some mutation is performed on the port, for example, setting a variable viewed through a port, then the object (variable) is changed. In general, one can pretend that another part of the system is in the place of viewing without changing the "real" organization of the system. The primary difference between a port and a window is that a port is a legitimate object in the programming language and is spatially located in the system hierarchy, not attached to the screen. A port appearing in a data structure indicates that the contents of the port are shared; the same data occur in some other object. Section 5.3. gives an extended example showing how ports can be used to provide multiple views, cross referencing, and so forth.

In the context of sharing, one can see a subtle but important shift in the meaning of variables from Lisp and Smalltalk to Boxer. The meaning of setting a variable in Boxer is to change the contents of a box, so that any port to that box sees the change. In Lisp or Smalltalk one cannot share in this way. A second object can indeed point to the value of a variable, but changing the setting of that variable creates a new pointer from the name to the new value, leaving the object that shared the old value still pointing to that old value. This all means an extra layer of indirection in implementing Boxer variables. But that layer corresponds to a key idea—it represents place: If a variable is to have

a place, that place must remain invariant in the process of setting the variable, and that fact, in turn, must be represented in the implementation.

Summary

In terms of understandability, we believe Boxer's static organization fares well. There is a small structural core of spatially organized textual objects, and the main associated functions do not radically change, either semantically or visually, that core. Even the variations needed to provide specific functionality are accomplished by means of a weak sort of typing, based on what we expect users to find natural functional categories: procedures (things that do something), data, text, and graphics. There are other ways of achieving functional variation of the core structures, for example, by adding syntax to specify use instead of types or using modular, special-duty parts of a box (such as doit or data parts); but types appear simplest. (The following section amplifies on the simplicity of types.) To be sure, the ties between initially perceived functionality and these types will be loosened as the behaviors of these different boxes come to be better understood in context-invariant terms, but this freedom is precisely the right thing to hope for when functional models are used. The real test, naturally, is empirical in terms of effective, long-term use of the system.

3.2. Dynamic Structures and Functions

Now we turn to dynamic structure and function, issues of control and change in the system. When one thinks of control in a computer language, typically what comes to mind is iteration and conditional structures like REPEAT <number of times> <things to repeat>, and IF <condition> THEN <action>. Our choices in this area neither reflect major innovation, nor do they deeply reflect our design heuristics. So we shall not discuss them here. Instead, we turn to more fundamental issues having to do with what a procedure does when executed and how names are made to refer to objects.

Reference

Reference in the context of computer languages is usually restricted to discussions about distinctions like "call by name" versus "call by value." Readers assuming that context should be aware that the discussion here involves a much broader construction of the issues involved.

A useful noncomputer context for introducing these issues is reference in natural language. Humans have an elaborate set of mechanisms for determining and verifying the reference of any utterance. The striking fact about this is that these mechanisms are almost totally invisible. In retelling a simple story the expression "the man who . . ." is apt to be replaced by "Joe" or who-

ever is understood on the basis of contextual information to be the man referred to. If there was an ambiguity of reference, the usual case is that, unless it was noticed at the time, that ambiguity will be unretrievable—how one established Joe to be the referent is not long stored, if it is ever recognized. In a similar way, elementary school students will respond to the joke: "Antidisestablishmentarianism. Bet you can't spell that." 'T' 'H' 'A' 'T'! But they are very unlikely to be able to describe or productively use the shift in reference of "that." The cues that prompt type/token or use/mention distinctions in reference are not well understood by linguists, let alone by "common folk." Even the fact of such distinctions is not available to most people. In short, establishing reference, though a complex process, is perceived as though it involved totally transparent pointers to referents.

The problem for computer languages is clear. Efficient reference mechanisms (to date) have been extremely simple, some version of lookup based on large-scale syntactic rules and/or type indexing. Lisp, as an extreme case, does a lookup on the basis of a universal syntactic form. Such schemes have understandability problems. They are not sensitive to the contexts that users will spontaneously apply, nor will a naive user be able to educe the cleverness needed to make the context-free mechanisms find the appropriate reference.

To elaborate the issues, we make another short case study of Logo. The designers of Logo took an apparently schizophrenic approach to the problems of reference. On the one hand they granted special status to functions like ERASE (clear a procedure from workspace), PRINTOUT, and even TO, so that one writes TO SQUARE and ERASE SQUARE rather than TO "SQUARE, and so forth, simplifying this semantically clear reference. As mentioned earlier, literal reference mode, specified by quote, would be necessary if these commands followed usual function evaluation rules for its input. On the other hand Logo chose to leave the distinction between function and variable lookup to the user, specifying variable lookup with : as in :X. Apparently the rationale was that the functional classes variable and procedure are sufficiently distinct on naive criteria to "allow" (read "require") users to be responsible for the distinction. In fact, experience has shown this to be relatively unproblematic. "Kinds-of-things" distinctions of this sort seem to be rather natural. Their naturalness can even occasionally be a source of minor problems: Some beginners seeing inputs in procedure definitions for the first time evidently rationalize the : to mean input in a kind-of-thing sense. Then they type SQUARE :100.

What has proved more seriously problematic is that : in Logo truly denote a structural reference mechanism and not a kind-of-thing as the functional distinction procedure/variable might imply. The difficulty is that assigning a variable a value involves two kinds of reference, a *named object* type (like ERASE <named object>) to specify which object is being set, and a *value* type to specify the new value. To simulate these out of its structures (which, recall, are

largely inherited from Lisp's function application standard) Logo writes MAKE "X :Y (X gets Y's value), even though X and Y are both variables.

Experience suggests that learnability is complicated; the variable assignment syntax is not as susceptible to learning based on germane rationalizations as one might have hoped. Thus it is a burden without significant advantage for beginning users who cannot be expected to see the structural significance to the markers and must rationalize on purely functional grounds—: denote a variable, except in MAKE, the latter part of which is without any generalizable import. Another problematic rationalization of the same functional-kind-of-thing type is to think that the two character string "X is the name of the variable and that :X is its value. A more profitable rationalization is that : denote a value-of operation, which would lead one to expect that ::X should give the value of the variable-name accessed by :X. Some instances of implementing Logo have supported this.

In fairness, there are things to be said for the syntax: (1) MAKE is then a function in the ordinary sense, which uses value reference for each of its inputs. (2) Because of this, variations of standard usage are relatively easy to achieve as in MAKE PROCEDURE.WHICH.COMPUTES.A.NAME or MAKE :VARIABLE. SET.TO.A.NAME. (3) A judgment was made that it is not only possible to teach the name/thing distinction, but that this could be a valuable gain from learning the language. We have already argued that (1) is a consideration for advanced users, not beginners, and (2) is as well: Computed names are almost never useful for novices. Moreover, novices find them strange and remarkable when they do encounter them. Even if the flexibility is there, it may not be seen or spontaneously used (formalist bug). One can have more sympathy for (3) except that it makes little sense to complicate very early use of a language with issues that will eventually arise in other contexts anyway. If one can choose the learning sequence in a language, it makes little sense to have some of the most difficult issues encountered at the earliest stages.

Finally, one could argue that a syntax which hides the difference between kinds of references is bound to be confusing. But first note that if our earlier claim is true—that reference mechanisms are generally invisible—the user will experience both references in MAKE X Y as simple references. In addition, while the literal marker might be rationalized to represent named-object reference, in fact it represents only a mechanism of achieving that reference. (Even this rationalization is not likely to be made by beginners.) Although it is typical of that kind of reference, quotes are used for other purposes as well. If precision were the issue, one might better use some more specific marker. More directly to the point, there is an important semantic component of the reference associated with MAKE not captured by the literal reference; by MAKE "X <whatever>, one does not mean to replace the literal symbol X by some value. X must be understood to be a variable which happens in this instance to

be exhibiting the setable half of its set-and-get protocol, independent of what mechanisms and syntax cause that to happen. If a user understands that, there seems little point in a nonspecific syntactic reminder, that is, quote. Indeed, later we shall propose a semantic reminder in the form of a prompt, which has more attractive features.

Reference in Boxer

What, then, is Boxer's approach to reference? In general, the assumption is that a structural understanding of reference should not be the goal in early stages of learning the language. Early models must approximate the simplest functional model of reference possible—that a word refers to whatever the user intends it to refer to.

More specifically, we propose the following threefold strategy. (1) We generally favor reference mechanisms that are strongly linked to kinds of objects, rather than weakly linked (like : and variables) or unlinked (pure reference mechanisms, like quote). (2) We broaden the context sensitivities of the language, accepting the assumption that most commands in the language carry an almost unique semantically determined "natural" reference mechanism for their input, which we simulate with appropriate but syntactically invisible variations in lookup. (More detail on this assumption comes later.) So we would write MAKE X Y, even though the structural reference mechanisms for the symbols X and Y are different. The rest of our strategy follows from the observation that the first two parts only postpone many issues which will certainly arise as naive users stray farther from patterned imitation of prototypes and wish to program more complex operations, such as setting variables with computed names, and so forth. Thus, (3) we would like to ease the transition to structural understanding of reference. To do this we propose (a) to improve the understandability of the underlying reference mechanisms by developing better surrogate models for them and (b) to improve debugging aids to the point where even if a surrogate model fails (most likely by not being used!), the error is easy to locate. In particular, we wish to implement a method of watching a program in action to spot the error. Debugging, of course, is important in its own right. But perhaps most importantly, the visual method we've chosen to implement will aid the acquisition of intended models, as well as simply the catching of bugs. We expect episodes of watching the behavior of the system to lead to a rich set of rationalizations and other partial understandings important to incremental learnability. We elaborate these points starting with (a), an underlying surrogate model, on which the others depend.

A Surrogate Model for Boxer

The key idea in producing a surrogate for Boxer is to produce visualizable, hence, depictable, intermediate states in the execution process. Here we are building on a number of previous efforts. Baeker (1975) has an early reference

to program visualization, and Lieberman (1982) represents a more modern context. Smith's Pygmalion (1975) is especially notable in that it not only shows program execution, but attempts to make programming be the manipulation of that visual representation—a form of naive realism. In what follows, recall our goal is not only to produce a relatively clean surrogate model, but provide *handles* for functional and distributed models as well.

We mentioned that the distinction between variable and procedure, obviously natural to computer languages, is clear enough in naive terms to be adopted as a fundamental. Hence, Boxer has data and procedure boxes. A data box's function is to contain data in literal form. A data box appearing in place, for example, in a procedure, marks the contents as literally referenced (Figure 6). A named data box in a library is a variable, data waiting to be referenced. Thus, we have collapsed the two structures of literal reference and variable into one.

In contrast to Lisp's quote, which is an active function returning an unquoted object, evaluating a data box results in a data box. More precisely, it results in the same object—evaluation is trivial on data. This result reflects the shift from using a pure reference mechanism, quote, to a kind-of-object mechanism, with reference built in. It allows the user to begin with a more functional, less structural model of literals. As far as a novice user is concerned, evaluation simply doesn't happen at all on data. Data are inactive stuff.

The surrogate model for evaluating an expression involving a variable—a data box referenced by name—entails retrieving a copy of the data box from the most immediate superior box whose local library contains a box by that name. Then, execution proceeds as if the data had been written in place. Lookup and copy for a procedure are identical, but the execution stage is recursive, that is, it will in general involve copying and executing elements of the contents of the procedure. In short, this copy-and-execute model consists of optional copy (in case of reference by name) followed by execution, which is recursive in the case of a procedure, terminating at the action of language primitives. One difference between a surrogate and what really happens is clear here; no respectable implementation would literally do such copying. It is only important that the user be able to pretend that that is what is happening.

Perhaps the strongest argument for this surrogate model of the dynamics of Boxer is its visualizability. Copying a procedure or data box in some location is concretely realizable in the overall Boxer spatial frame. We are implementing a stepper as part of Boxer's debugging facilities in which one sees this copying of procedures on the screen, building the dynamic stack, and sees the replacement of a name reference to a variable by its value.

Such a stepper would reinforce, if not teach, the underlying surrogate model. One would expect watching simple programs executing to be a part of naive users' early introduction to the system. The user could pause to inspect the calling hierarchy and the state of local variables, including inputs, at any

Figure 6. A data box marks literal reference.

stage. The Logo little-man-model becomes concrete. In addition to stepping, such inspection would be extremely useful after an error occurs. We imagine that in addition to an error message, one could enter and inspect the stack via a port down to the level of the error.

These are not new functions to programming systems. Smalltalk, various Lisp implementations, and demonstration systems for other languages allow one to inspect the stack. However, the advance in Boxer is that the mode of inspection is identical to the concrete mode even beginners use to inspect any part of the system, and the meaning of what one sees is a direct embodiment of the fundamental dynamic surrogate of the system.

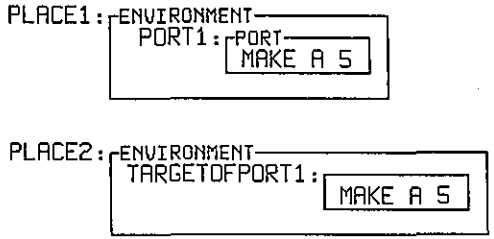
It is not hard to extend this surrogate to ports. For this we use the functional characterization of a port as imitating the presence of a box that exists in some distant part of the system. A port to a data box behaves just like the data box in accessing that data. Mutating data in a port, say, MAKE X Y, where X is a named port to some target databox, changes the target. A port to a procedure behaves precisely as if that procedure were in the place of the port, with the exception that the lookup environment for free variables in the procedure is established by the procedure, not by the location of the port. This scoping rule, while not strictly entailed by the meaning of ports, is consistent with it and provides a useful function, that of lexical scoping. (Scoping per se is the topic of Section 3.3.) In the example in Figure 7, executing PORT1 will set the variable A in PLACE2, which contains the target of the port. Dynamically, as well as statically, ports give a mechanism for breaking the strict hierarchy of box structure, and we consider it a minimal and natural extension. The power of ports is demonstrated by the fact that one can program without names, in that every reference is wired in with port connectors. This characteristic provides a very strong form of referential transparency, removing issues of scoping through the technique of reference by pointing.

The meaning of using a port by name is completely defined by what it means to copy a port in the copy and execute model. We propose that a copy of a port be equivalent to the original port, that is, the copy is a port linking directly to the target of the copied port. This maintains all the functionality of a port when it is referenced by name.

Inputs

We return to the issue of varied and unobtrusive reference mechanisms. The idea is to let the procedure establish context, how the text which constitutes an input is to be treated. We propose three types of input which parallel each of

Figure 7. Ports provide access to other environments for dynamic purposes such as setting a variable. Here the expression MAKE A 5 resides in PLACE2 but is visible (and could be executed from) PLACE1.



the three ways—procedure, data, and port—in which execution treats the contents of a box. Procedure evaluates the input text according to the standard Boxer rules and installs the result in the input's data box in the inputing procedure's local library. This matches Logo's input structure. The second kind of input treats the input text as data and transfers it unevaluated into the input's box in the local library. It is appropriate for messages and other textual data. One need not bother with literal markings.

The final kind of input uses port semantics and therefore will probably be used only by advanced users. For this reason we can use this structure for expert-appropriate functionality. We propose that the port version of input create a port (in the inputing procedure's local library) to the box used as input, or to the box named in case a word rather than a box appears as input. This has an important implication: Suppose procedure CALLER has port type input parameter IN and is called with another procedure FUNARG as input. Then FUNARG will not be executed on invocation of CALLER but only where IN appears in the body of CALLER. Thus, the port type of input allows procedures to be passed as inputs. Furthermore, the environment available to FUNARG when it is executed will be its actual location, rather than the interior of CALLER. This scoping is the most appropriate for function arguments according to Steele and Sussman (1978a).

As far as learning sequence goes, one expects that users will use the procedural (*value* makes a better mnemonic) version for their own definitions for quite a while. Value inputs are the default when no type is specified. During that early time, the other types serve to relieve the need to understand the subtleties of referencing in using system primitives or any procedures added to the system, presumably by more experienced programmers, for the user.

Difficulty with versions of inputs will occur if the procedure's perceived domain of applicability overlaps into situations where another reference mechanism is appropriate. For example, a misunderstanding may result if a procedure's semantics allows either name or number as an input. A data-input

structure will work in typical situations; however, if the user expects to use a variable set to a number in place of the number, an error will result. Advanced users, of course, should be able to change the input-reference mechanism with explicit markers at the place of invocation. Eval and quote are used this way in Lisp, though ideally one would prefer a cleaner relationship between control and reference mechanisms. Section 4. provides a suggestion for a transparent way to accomplish such a relationship.

Two Proposals for Non-Lisp Structures

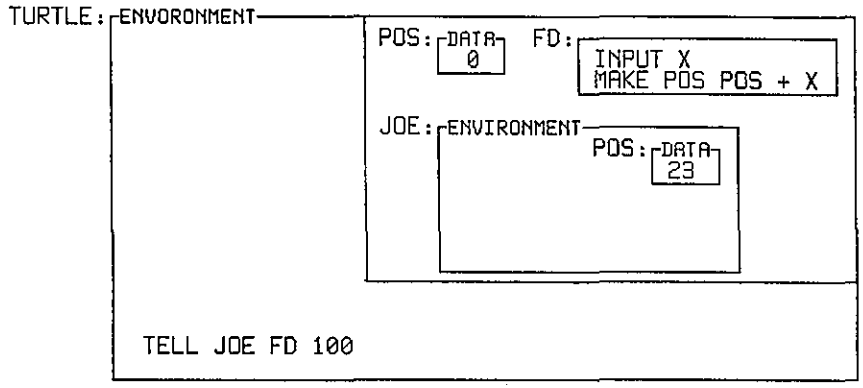
In this section we treat two functions not well served by stuctures in Lisp and Logo; accordingly, we make propoals for Boxer. The first of these functions is message passing.

Consider the concretely realizable process of moving to a distant environment, executing a procedure, and returning with the result. This is the basis of message passing in Boxer. In particular, we have special syntax tuned to this function, TELL <environment> <whatever>. Since Boxer's structure has fully developed environments, we believe syntax is all the dedicated structure message passing needs. Instances and subclasses can be made by copying and nesting environments. Figure 8 shows how to make an instance. The environment TURTLE has a state variable, POS (position), which is manipulated functionally, by FD. A turtle instance, JOE, is created by making a subenvironment containing its own state variable. But JOE can use TURTLE's manipulator code, FD, on its own POS.

The second neglected function is the construction of compound objects out of evaluated parts. Lisp and Logo use constructor functions for this purpose, functions which evaluate their arguments and output a compound structure constructed of the values. We consider this an overextension of the control structure of function to an area in which it is not well adapted, at least in the perception of beginning programmers. The reason is simple. Even in Logo, spatial organization is part of a typical user's model of a compound object. A list is a series of elements in a row. Why then can one not use such an organization to specify the *shape* of a compound object to be constructed? In Logo if the value of :X is "A, LPUT :X[B C] produces the counter-visual result [B C A]. Instead, one would like to write something like [B C :X]. Boxer's intent to make spatial organization pay dividends suggests we should try to do better than Logo. Many Lisps now have a "back-quote" structure to serve this function, and what we want for Boxer is a cleaner, better integrated implementation of the motivating concerns.

One of the designs we have implemented collects pieces of data using a single constructor function which operates on a template databox. It is natural to have a number of versions which specify whether or not to unbox the data in collecting it and whether or not the default treatment of items inside the template box is to evaluate them. Additional markers would be useful to alter de-

Figure 8. Sending a message to JOE by executing a TELL within the TURTLE environment. JOE behaves as an instance of TURTLE, using TURTLE's FD on its own state variable, POS.



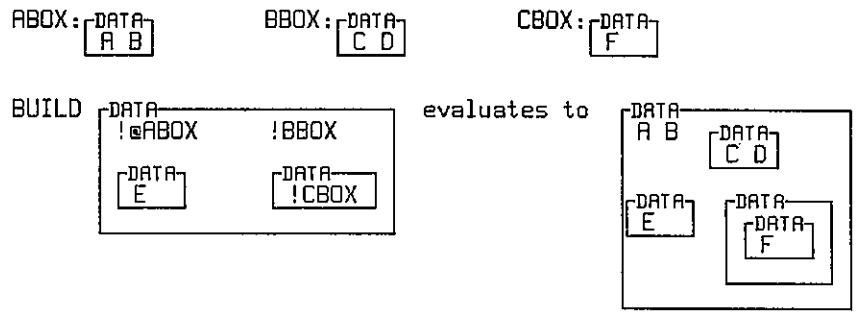
fault treatment. Figure 9 gives the gist. Here we have nonevaluation as a default, prefix ! flags items to be evaluated, and @ marks "unbox."

3.3. Scoping

Boxer's basic scoping rule is that variables and procedures are accessible by name within the box in which they are defined or in any subbox, recursively. This accessibility, along with the copy-and-execute model, implies dynamic scoping—a procedure invocation alters the name-space available to subprocedures. There are cogent reasons to be wary of this scoping. Most prominently, it can lead to a dangerous nonmodularity where what a procedure does depends on where and when it is called. For example, a local variable in a procedure call may shield a variable reference in a subprocedure from a global value that would be available if that subprocedure were used at top level. The alternative advocated by many is to use lexical scoping, where a free variable refers to the environment in which the procedure is defined rather than where it is called. Since Boxer has a static environment structure above and beyond the dynamic one (procedure calls), the issue is doubly important. So it is particularly appropriate to look at the considerations of functionality and learnability which led to the choice of dynamic scoping.

1. Sometimes one really wants dynamic scoping. Consider environments in the sense of workspaces discussed earlier. One may take a procedure to another environment or create an intermediate environment between the procedure's environment and UNIVERSE explicitly for the purpose of altering the meaning of the terms making up the procedure definition. The method of creating actor-

Figure 9. A BUILD function assembles data spatially out of pieces. ! means evaluate, @ means unbox.



style instances shown in Figure 8 depends on dynamic scoping. With lexical scoping the POS manipulated by FD would always be the one available in the environment of the definition of FD, TURTLE, not the one available in the subenvironment, JOE. Other arguments for the usefulness of dynamic scoping based on dynamic environment (procedure call) considerations only are contained in Steele and Sussman (1978b).

2. Boxer will have most of the functionality of lexical scoping available for advanced users. Some of this functionality can be carried by the local library, which is copied with the procedure body to the environment of execution. By putting subprocedures and local data in the procedure's library, that part of the procedure's environment will always be available on invocation.

Ports can take over more of the functionality of lexical scoping. This can happen in two ways. If one uses a port to a procedure instead of its name to reference it, one gets lexical scoping for that procedure, according to the declared rules for scoping ports. Also, a port-type input creates a port to the input, which gives lexical scoping for procedures that are taken as parameters to other procedures in this way.

3. Dynamic scoping is more natural to Boxer than to any nonspatially organized language. This is a judgment on how the experience of using a system supports one or another model of its actions. The overt experience of a Boxer user is performing operations in environments that define the meaning of those operations. In a Lisp or Logo experience, environments are transitory—set up for function calls and destroyed on exit. Lexically scoped languages have environments as a basic fact of life; however, these environments are hardly concrete and manipulable in the transparent way they are in Boxer, that is, picked up and moved around with the editor. Without such concreteness it is a more appropriate aesthetic to avoid dependence on environments as they are nearly invisible and hard to manipulate things; functions really ought to do the same thing on each invocation. In Boxer, that procedures operate in environments is

the fundamental, concretely represented metaphor of the system. We need to worry less about potential false expectations of modularity and problems in debugging them if problems occur. To some extent the problem is also ameliorated by the fact that novices will not be constructing the deep and complex programs that experts do, which poses stricter modularity problems.

4. What may be the most important criticism is that lexical scoping simply does not have as simple a surrogate model as dynamic scoping, at least in spatial terms. The overt signs of this are that one must distinguish the text of a procedure from the procedure itself (Steele & Sussman, 1978a). This runs directly counter to the principle of naive realism we have adopted for Boxer. Procedures and containing environments are separately represented in Boxer and need not—probably should not—be strongly linked in the way lexical scoping does.

Continuing the last point, consider the changes needed to the copy-and-execute surrogate of Boxer for lexical scoping. When a procedure is called, one cannot set up an environment at that place in which to observe the actions of the procedure since the free variables in the procedure refer to nonlocal entities, entities that exist in the environment in which the procedure was defined. So after binding inputs (which do, in fact, come from the procedure invocation location), one must shift geographical focus to the defining environment for the execution phase. After execution, one must return control and any resulting value to the calling environment. (The alternative to these shifts in locus is to give up the identification of containment with "environmentness" basic to our spatial metaphor.) Imagining or actually watching a procedure execute would be considerably complicated by constantly switching environments. The topology of the calling structure of a procedure stopped in midstream could wind tortuously through the spatial hierarchy. Although the surrogate per se is not immensely more complex for lexical scoping, more of it is invisible and not amendable to learning by episodes of interpreting what one sees happen. How should one represent, for example, return pointers? In the dynamic copy-and-execute model return pointers are unnecessary; procedures return in place.

Of course, one may argue that it is the spatial copy-and-execute model that one should abandon, not lexical scoping. But with lexical scoping, it seems one will always be faced with representing two hierarchies—the calling hierarchy, which should not be ignored, and the lexical one. Simple models embodying both hierarchies seem hard to come by.

We have a priori excluded from discussion possibilities such as using strictly local (no inheritance) scoping in the special case of input parameters or using dynamic scoping for static environment structure at the same time as lexical scoping for dynamic environments. The judgment is that, from the point of view of understandability, one scoping rule is quite enough for inexperienced users.

4. THE USER INTERFACE

The user interface of a system is important since it is the part that the user directly perceives (mainly on the video screen) and operates (keyboard, mouse, and so forth). One must decide what parts of the system are shown, when and how they are portrayed, and how one selects and enacts actions. For our primary purpose—understandability—the important interface issue is how the interface relates to the abstract objects in the computational environment and to the actions that can be performed with or on them. For Boxer and, indeed, for any naive realist system, there should not be much to say about this relationship; it ought to be very simple. The objects of the system (boxes, text) should be directly visible and manipulable in their own terms (the Boxer text-editor). Thus, the descriptions we have made of Boxer's computational semantics are also, to a large extent, descriptions of the user interface.

Because of the close connection between interface and programming language, each can serve to augment the other. For example, some of the functionality that one usually needs to have as part of the programming language can be taken over by the interface. One can construct procedures, and even the global organization of the system itself, concretely with the editor rather than needing all procedure- and structure-creating commands to be a part of the language. Conversely, language constructs can often be used directly to support the user interface. Making a region of the screen that responds in some special way to typed characters is a trivial operation, making a box and binding local procedures to keys. Making a part of an interface that continuously shows the values of certain variables amounts to no more than making those variables visible on the screen, say, via a port. Overall, then, naive realism implies a high degree of diffusing functionality both ways across the user interface/programming language boundary.

In more detail, however, our learnability principles were not designed to deal with convenience or with the fine structure of keystroke-by-keystroke interaction. Nonetheless, we will discuss two issues usually associated with the user interface as they relate to Boxer. As far as *perception* is concerned, we focus on screen organization. As far as *operation* is concerned, we shall focus on facilitating recall and rapid-command enaction.

Screen Organization. Boxes may appear at first to be very inefficient in terms of the use of screen space. For example, our use of box structure to identify objects and inheritance in the system means we are not free to have overlapping boxes as one can overlap windows. But Boxer has a number of compensating capabilities.

In the first instance, by moving around in the system, one can choose what part of the system to see. Recall that any box can be chosen to fill the screen. To save clutter, any or all subboxes can be shrunk to basically one character space.

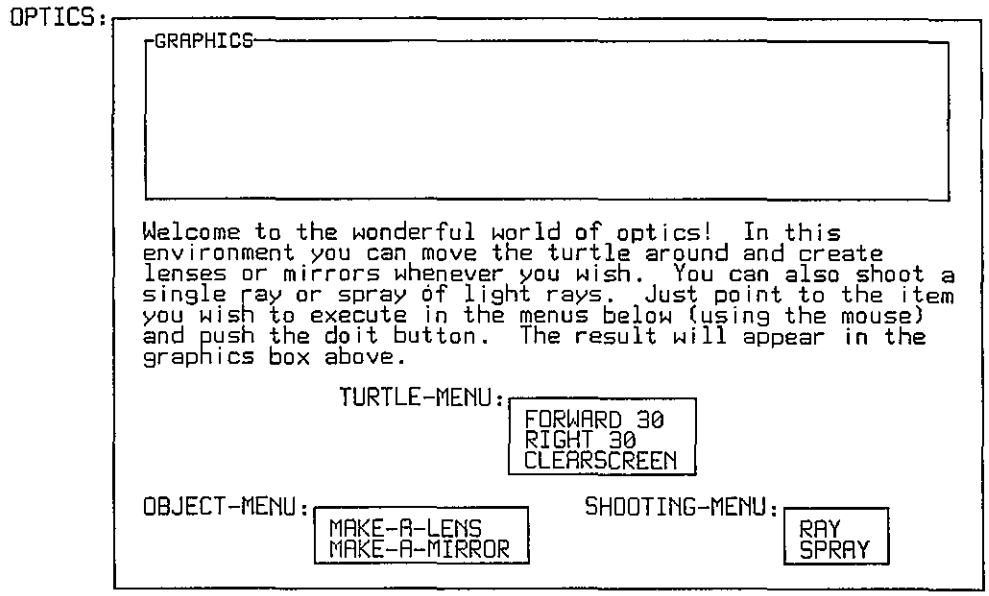
Indeed, one can use boxes purely for this detail suppression mechanism, and we expect users to build their own worlds with viewability of the world as a major consideration. The display presentation of a box is remembered so that when one returns to that box, subboxes are shrunk or not and in general everything is displayed according to the form in which it was left. For more advanced users, boxes can be frozen in any size between shrunk and full screen. The frozen size can be overriden by expand or shrink commands, but it is the size in which one will always see the box when it first comes into view. In order to handle boxes whose contents are bigger than the frozen size (or bigger than full screen), boxes can scroll vertically and horizontally.

Facilitating Recall and Rapid-Command Enaction. Menus relieve users of the need always to remember available commands, and they also facilitate rapid use of the system. Luckily Boxer has much of the usual menu functionality built in. Anything the user types is a usable artifact that may be selected and executed. We have a line-oriented default for selection, compatible with the line-oriented substructuring of boxes. So all a user has to do to use some text as a menu is to point at a line and press the DOIT key. Users will undoubtedly gradually build their own menus in various environments out of what they type to try things out. Some might wish to put frequently used commands in a box labeled MENU, and anyone who makes a subenvironment for others' use should leave such artifacts around (Figure 10). The important thing is that essentially all the functionality of a menu is available without the overhead of learning to construct or change special structures.

Rapid command activation may also be accomplished by attaching commands to keypresses. In Boxer, as previously noted, one can add instant keystroke activation of any command, including user-defined commands, to any key. Naturally, this power can be abused, for example, by having keys whose definition changes frequently in moving from one environment to another. But our judgement was that the ability to define and redefine keystrokes is important to assure adaptability of the interface, for example adding or changing text-editor commands. See Section 5.1. for an example.

As another memory aid, Boxer will have an interactive prompter. We have implemented the following: If a user wants help with a function, he types the name just as if preparing to execute the function, or he can move to a menu line where the name already exists. Pressing HELP causes input prompts to appear following the command as boxes labeled by the names chosen as input parameters for the program. After the user fills in the inputs, the DOIT key executes as usual. It is important that these prompts are in every way ordinary screen objects: They can be typed by the user instead of by the prompter, and they may be deleted as may any Boxer objects. In addition, there is no special prompt mode. When the prompts appear, the user is not obligated to follow up in any particular way.

Figure 10. A page from an *interactive workbook* includes a set of menus allowing readers to activate available procedures.



The prompt boxes have a type marker which displays the type of input they represent: value (labelled simply INPUT), literal (labelled DATAINPUT), or port (PORTINPUT). (Recall that these are the three types of input reference mechanisms.) Aside from documenting the define-time choice at call time, a user can change the type for any particular call by editing the type shown by the prompt. Prompt boxes also serve to parse expressions to an arbitrary depth based on the same box hierarchy used generally in the system.

5. BOXER SCENARIOS

What does it feel like to use Boxer? This section attempts to give that sense while at the same time illustrating more concretely some of the main points already made about the system.

5.1. The Morning Mail—Getting Around the System

This example shows how to use basic cursor motion and text-editing commands to move around the system and accomplish an everyday task without the need for any special program devoted to the task. Secondarily, it demonstrates the kind of simple tuning one can do to adapt the environment to ease such tasks.

On entering his personal Boxer universe, a user would see whatever he constructed to be the top level of his world, say, something like Figure 5. Using the mouse to move the cursor into the MAIL box, one then presses the expand button on the mouse to cause the box gradually to expand, revealing what is inside, Figure 11. At this size, as opposed to the shrunken- or full-screen sizes, a box automatically expands and shrinks to accommodate whatever is inserted or removed. A second press of the same button expands the box to full-screen size, effectively entering the mail subenvironment (Figure 12). Note that a reminder to return mail to John, which the user simply typed in this appropriate place, appears at the top of the environment. Ignoring the reminder for the moment, the user can enter the NEWMAIL box by moving the cursor with the mouse again and pressing the mouse expand button.

Here (Figure 13) a new message has arrived from Leigh. Ordinarily the user might simply read the mail and delete it. Deleting the box containing the message can be accomplished by moving the cursor to the position just to the right of the box and pressing the rubout key, as if the box were just a large character. Another option would be to move the message to OLDMAIL with editing functions or to have some built-in function do that.

Part of the newly arrived message is a program defining an instant-action keystroke command, CNTRL-M-KEY. In our present Boxer, the suffix KEY in the name of a box denotes that this program should be executed on pressing the named key. CNTRL-M-KEY itself uses the BUILD operator to construct a databox out of some literal data and an evaluated function. (! means *evaluate this* to BUILD). SYSTEMDATE is evaluated, providing the obvious information. The intended net effect is that when CNTRL-M is pressed, the command CNTRL-M-KEY is executed, returning the built template for a mail item at the position of the cursor.[3] The actual sending of mail could happen by typing SEND in front of the message and then pressing the DOIT key. In general, the DOIT key causes the current line to be executed. SEND might be the only primitive supplied with the mail system.

In order to install this CNTRL-M feature in the MAIL environment, the user must move the CNTRL-M-KEY program to an appropriate place of definition, probably the library of the MAIL box. Moving the program involves simple editing: (1) moving the cursor with the mouse to the CNTRL-M-KEY program, (2) picking the program up (one could use an editor command to "cut" the line), (3) moving to the place of definition using the mouse and its expand and shrink buttons and depositing the command (for example, with a "paste" editor command). In order to change the template at any future time, one has only to move back to the library and type in the changes.

[3] Returning values is done through a variation of Lisp's "last subform supplies value." In Boxer, procedures return a value only if the last subexpression provides it. A subexpression, of course, may simply be a data box.

Figure 11. MAIL has been expanded so that one might see its contents.

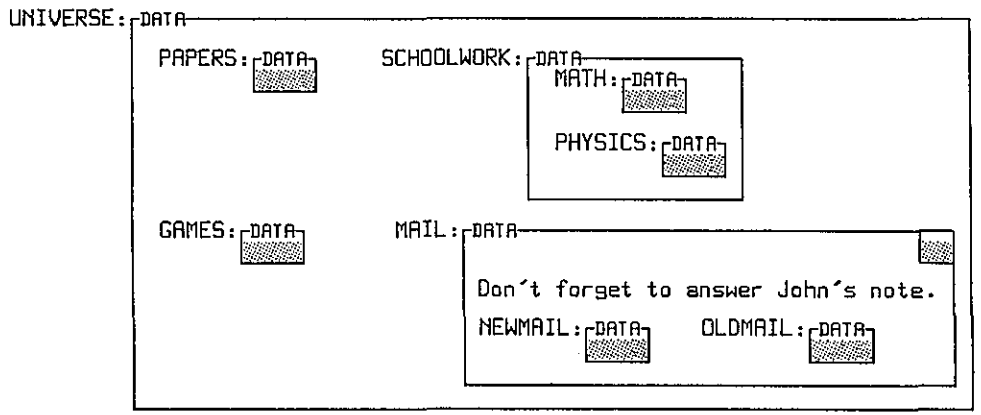
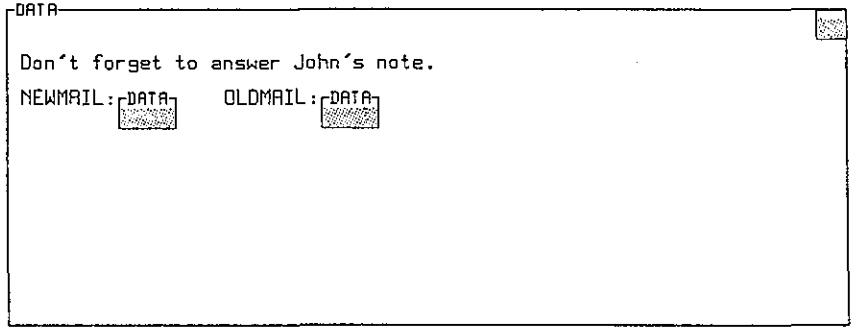


Figure 12. MAIL is expanded to full screen.

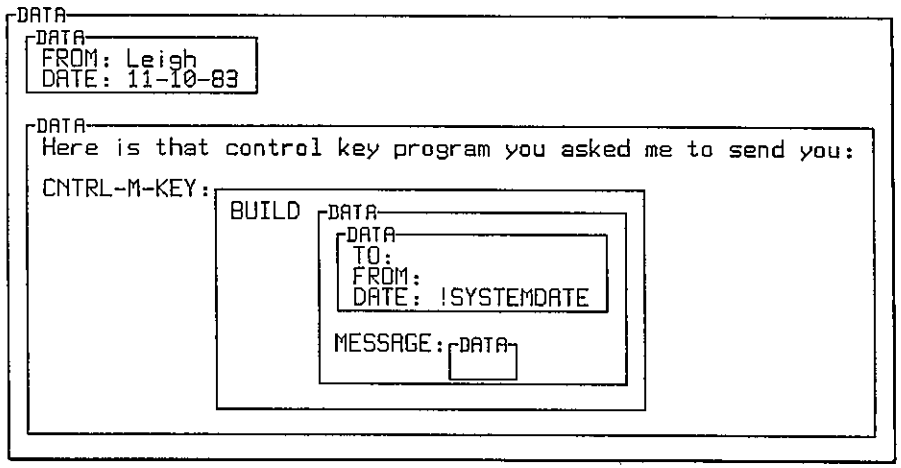


5.2. A Simple Program

Defining a program can be another task accomplished quite concretely with the always-available text-editing commands. One could, for example, push the make-box key, type the text of a procedure in the box, then attach a name, and, finally, move the program to an appropriate library. Typically, an appropriate library means the library of the environment you are currently working in. In the following example, we have chosen to illustrate the expressive possibilities of typing subprocedures directly into place or of putting subprocedures in the library of their main procedure rather than in some environment box library.

GRAM (Figure 14) assembles a randomly generated sentence from a simple grammar that specifies: A sentence consists of a noun phrase followed by a verb

Figure 13. A message from Leigh. Suppressing detail by shrinking the CNTRL-M-KEY box would make the message more readable.



phrase; a noun phrase consists of an article followed by ...; and so forth. Each of these rules is a program, and each occurs in the place in which it needs to be used in GRAM. The names of the programs are only to document their function. The library of GRAM contains the utilities SELECT-ONE and SELECT-SOME, which pick random elements from a box (RANDOM returns a random number between 1 and its input). The library also contains the lists of terminal nodes of the grammar, that is, the words, which one may wish to find near the surface of the program for easy inspection or modification. Of course, for various purposes, one might choose to write the GRAM program burying the terminal nodes and showing the rules at the top level. Figure 15 shows how NOUNPHRASE, VERBPHRASE, and a couple of their subordinates would appear if the shrunken boxes in Figure 14 were expanded, though we do not show the context for each.

5.3. A Journal

Keeping a personal database in Boxer is a trivial matter. Here, we have tuned the basic box structure by using ports to make available a second organization of entries in a journal. The top box in Figure 16, BY-CHRONOLOGY, contains all entries in chronological order. The bottom box, which is shown expanded in Figure 17, contains the same entries reorganized via ports according to topic. The intent is to allow the journal keeper to access an entry according to preference or how he remembers it: "I seem to remember writing somthing about that a week or so ago;" versus "Let me see what I have on the topic of ports." Because ports are views on objects which appear in another place, any change made in a port is instantly reflected in the original entry. If

Figure 14. The top level of a procedure to generate a random sentence. **BUILD** is the general constructor which recognizes !@ to mean *evaluate and unbox,* described in Section 3.2.4.

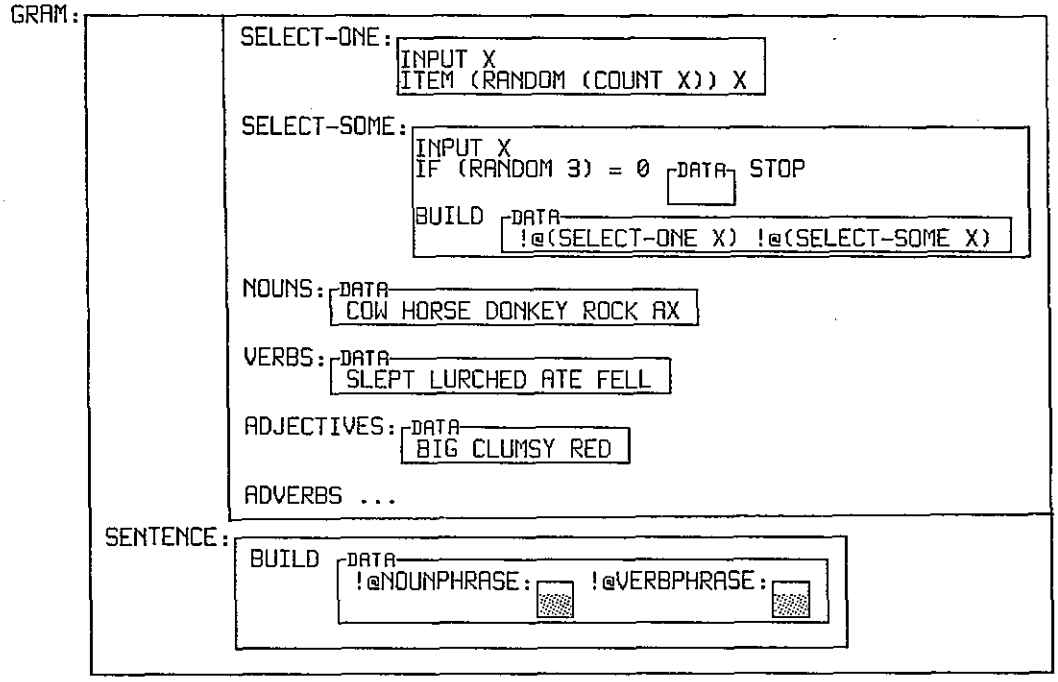
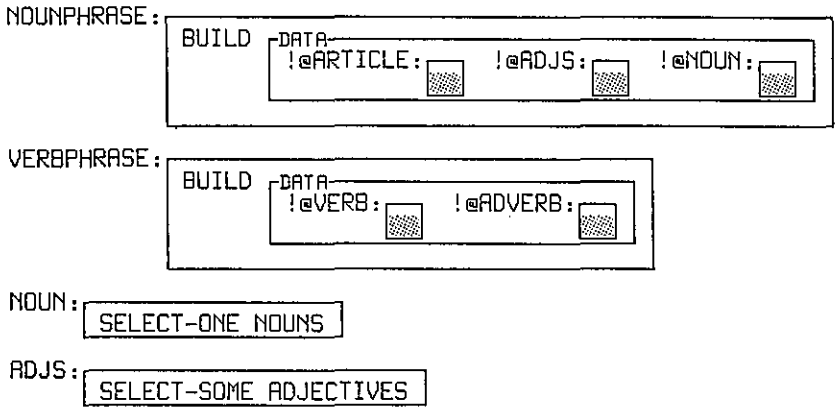


Figure 15. Some of the component subprocedures from GRAM, not in context.



both the port and the target of the port are on the screen, typing into either results in having the characters appear in both.

We imagine the protocol for using such a journal to be something like the following. One can make an entry by copying the form of a previous entry or using a template as described for mail above. Actually putting the entry in place could be handled with a FILE function, which would make sure to insert a port to the new entry under the appropriate topic in the BY-TOPIC listing. Of course, one could also have an UPDATE function, which, when executed, could place a port to any new, unported entries in the BY-TOPIC listing. These utilities, FILE and UPDATE, are actually not very complex Boxer procedures, though one would not expect novices to write them. They could be augmented with procedures, for example, to search the journal and return a box containing ports to entries that have specified key words—constructing another view at need, rather than incrementally as the BY-TOPIC view is constructed. One could even write a procedure to reorganize the journal completely.

6. SUMMARY

In designing an integrated computational environment the most basic heuristic demand is to try to generate a small set of structures out of which all necessary functionality can be built. An immediate caveat to that is shallow structuring, that common functions must not be difficult to express in the fundamental structural vocabulary. But deeper and more complex revisions to this aim are appropriate in view of a limiting resource in understanding and controlling a complex system. This limiting resource is the materials (knowl-

Figure 16. A journal contains two views of its entries, BY-CHRONOLOGY and BY-TOPIC.

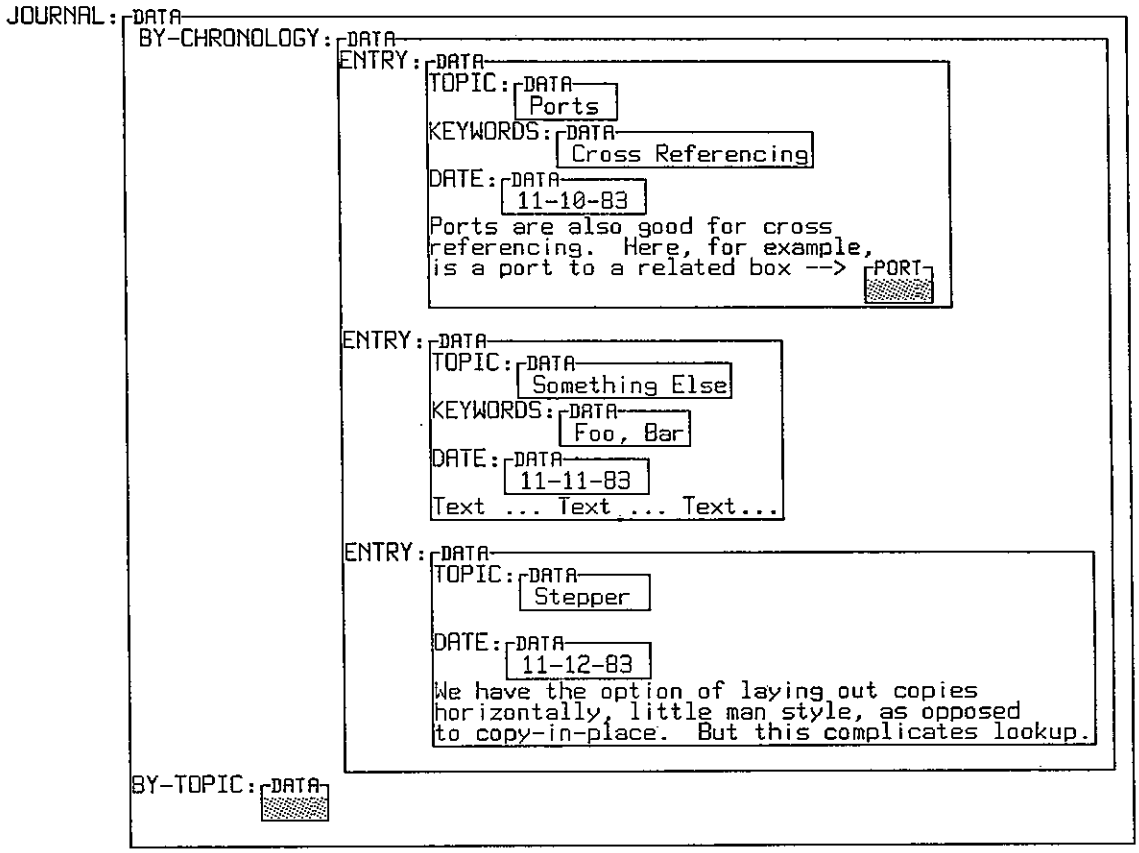
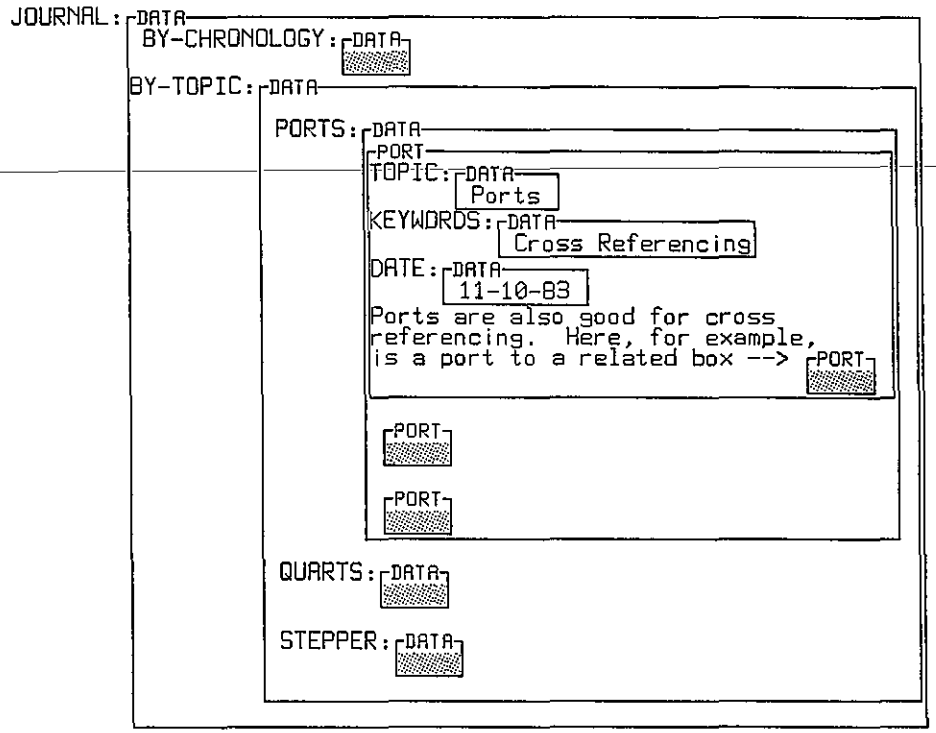


Figure 17. The first box in the PORTS section of BY-TOPIC is a port to the first entry shown in Figure 16. The other ports in that section are to other entries in BY-CHRONOLOGY.



edge) and capabilty to construct mental models of such a system on the part of the user.

We have found it important to consider a few paradigms to get proper purchase on the issues of understandability and learnability. Surrogate models are replacement machines which one can "run" in one's imagination to understand the actual machine. These are good for prediction and debugging but are not typically learnable in small increments and lack the kind of ties to functionality necessary to fluid interaction and to the invention of techniques to solve problems posed in solution-independent terms. Functional models in which, typically, a structure is learned as a solution to a particular problem—"it does the right thing"—create functional ties but are weak in terms of completeness and context-invariant application. We think it extremely important to consider a third paradigm, distributed models, in which not only is there no global mechanistic frame, but one may not even be able to identify a single functional frame that accounts for understanding and remem-

bering in terms of a simple mapping to previously understood situations. Instead, a number of situation-specific rationalizations, including visual metaphors and the inheritance of reasonableness from frames like natural language, altogether produce an account of some behavior of the system which makes that behavior generalizable, hence useful, as a model.

As an elaboration of these ideas we have sketched the design of Boxer. Boxer's key ideas are as follows:

> In order to minimize the need for invisible structures mediating between what one sees and how one understands it, and in order to promote modeling on the basis of visual rationalization, we have proposed naive realism as a guiding principle: All screen objects are real and manipulable in a uniform way. In this way most of what is usually thought of as user interface is integral to the system.
>
> In order to take advantage of the character of the video display and in order to link into an important class of pre-existing knowledge users have, Boxer employs a systematic spatial metaphor, using spatial relationships to express language semantics.
>
> Because of the strengths of the spatial metaphor and its appropriateness to computational systems, we have collapsed static structures to a small core, introducing functional multiplicity through variation of the basic object, the box, based on nearly naive functional categories such as procedures and data. All of the functional hierarchies in Boxer—procedure/subprocedure, hierarchical data, environment (file structure), and scoping—are organized with boxes.
>
> Because of the weakness of naive understanding of reference mechanisms, we have introduced an expanded set of functionally motivated dynamic structures (types of input, syntax for message passing, spatial construction of compound data objects). In particular, multiple types of input allow the simulation of a broader range of naive reference mechanisms without intrusion into the surface appearance of the language. Moreover, care has been taken to maintain a visualizable surrogate model to aid understanding dynamic aspects of the system.

What has been left out of this account of the design process? In order to focus clearly on issues of mental modeling we have discussed neither the consistency nor the completeness of Boxer as a computational scheme. Nor have we discussed the issue of efficient implementation or the heuristics we used to trade off implementation against functionality and user understandability. Naturally, we have proposed a system we think is consistent and efficiently implementable, but this has not been demonstrated.

Finally, our judgments about understandability are based on our assessment of both the difficulties and, occasionally, the surprising successes of students in understanding computational systems (and, to be fair, on our own experiences and introspection as well). Even granted our general modeling considerations, we have had to make decisions about specifically what knowledge we can count on users having and applying, for example, what rationalizations will be made. Obviously we need a great deal more study in this area, but we do not apologize for trying to use and systematize what we think we know already. There is no dispute that innovation, in terms of both computational structures and functions, is important to making progress in constructing powerful and usable computational environments. But we think it both possible and proper to begin to regard such innovations in the context of more systematic theories of design based on principles of learnability and understandability.

Acknowledgments. The design and construction of an extraordinarily complex piece of software like Boxer requires the dedicated efforts of many people. I herewith acknowledge the contributions and express my great appreciation to a wonderful group of MIT students and staff who have played this role. Among those who have shown the greatest dedication and influence are Ed Lay, Gregor Kiczales, Mike Eisenberg and Leigh Klotz. I must especially note the contributions of Hal Abelson, co-founder of the Boxer project, for guiding the implementation and assuring the computational integrity of the system with a constant supply of criticism and good ideas.

This paper has benefited immensely from incredibly extensive and thoughtful comments by Richard Young, Tom Moran, and an anonymous referee, for which I am also indebted.

Support. This work was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract NOOO1475-C0661.

REFERENCES

Baeker, R. (1975). Two systems which produce animated representations of the execution of computer programs. *ACM SIGCSE Bulletin, 7*(1), 158–167.

de Kleer, J., & Brown, J. S. (1981). Mental models of physical mechanisms and their acquisition. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

diSessa, A. A. (1982). Unlearning Aristotelian physics: A study of knowledge-based learning. *Cognitive Science, 6,* 37–75.

diSessa, A. A. (1983). Phenomenology and the evolution of intuition. In D. Gentner & A. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Lawrence Erlbaum Associates.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies, 14,* 237–250.

Erlich, K., & Soloway, E. (1983). An empirical investigation of the tacit plan knowledge in programming. In J. Thomas & M. L. Schneider (Eds.), *Human factors in computer systems.* Norwood, NJ: Ablex.

Eisenstadt, M. (1983). A user-friendly software environment for the novice programmer. *Communications of the ACM, 26,* 1058–1064.

Gentner, D., & Stevens, A. (Eds.). (1983). *Mental models.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Goldberg, A. (1983). *Smalltalk-80: The interactive programming environment.* Reading, MA: Addison-Wesley.

Goldberg, A., & Robson, D. A. (1979, January). *A metaphor for user interface design.* Proceedings of the University of Hawaii Twelfth Annual Symposium on System Sciences, Honolulu.

Goldstein, I. P., & Bobrow, D. G. (1984). A layered approach to software design. In D. R. Barstow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive programming environments.* New York: McGraw-Hill.

Greenblatt, R. D., Knight, T. F., Holloway, J., Moon, D. A., & Weinreb, D. L. (1984). The LISP machine. In D. R. Barstow, H. E. Shrobe, & E. Sandewall (Eds.). *Interactive programming environments.* New York: McGraw-Hill.

Ingalls, D. H. (1981, August). Design principles behind smalltalk. *Byte,* pp. 286–298.

Innocent, P. R. (1982). Towards self-adaptive interface systems. *International Journal of Man-Machine Studies, 16,* 287–299.

Kay, A., & Goldberg, A. (1977, March). Personal dynamic media. *Computer,* pp. 31–41.

Lieberman, H. (1982). *Watching what your programs are doing* (Tech. Rep. No. 656). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *ACM Computing Surveys, 13*(1), 121–141.

Rumelhart, D. E., & Norman, D. A. (1981). Analogical processes in learning. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Smith, D. C. (1975). *Pygmalion.* Boston: Birkhauser.

Smith, D. C., Irby, C., Kimball, R., & Verplank, B. (1982, April). Designing the star user interface. *Byte,* pp. 242–282.

Stallman, R. M. (1984). Emacs: The extensible, customizable, self-documenting display editor. In D. R. Barstow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive programming environments.* New York: McGraw-Hill.

Steele, G. L., & Sussman, G. J. (1978a, January). *The revised report on Scheme: A dialect of Lisp* (Tech. Rep. No. 452). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Steele, G. L., & Sussman, G. J. (1978b, May). *The art of the interpreter* (Tech. Rep. No. 453). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Teitelman, W., & Masinter, L. (1984). The interlisp programming environment. In D. R. Bastrow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive programming environments.* New York: McGraw-Hill.

Tesler, L. (1981, August). The Smalltalk environment. *Byte,* pp. 90–147.

Waters, R. C. (1984). The programmer's apprentice: Knowledge based program editing. In D. R. Barstow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive programming environments.* New York: McGraw-Hill.

Young, R. M. (1981). The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies, 15,* 51–85.

Young, R. M. (1983). Surrogates and mappings: Two kinds of conceptual models for interactive devices. In D. Gentner & A. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Lawrence Erlbaum Associates.

HCI Editorial Record. First manuscript received January 7, 1983. Revision received January 12, 1984. Accepted by Richard Young and Thomas Moran. Final manuscript received August 27, 1984. —*Editor*