MIT/LCS/TM-223

A PRINCIPLED DESIGN

FOR AN INTEGRATED COMPUTATIONAL ENVIRONMENT

Andrea A. diSessa

July 1982

A Principled Design for an Integrated Computational Environment

Andrea A. diSessa

July 12, 1982

Abstract

Boxer is a computer language designed to be the base of an integrated computational environment providing a broad array of functionality -- from text editing to programming -- for naive and novice users. It stands in the line of Lisp inspired languages (Lisp, Logo, Scheme), but differs from these in achieving much of its understandability from pervasive use of a spatial metaphor reinforced through suitable graphics. This paper describes a set of learnability and understandability issues first and then uses them to motivate design decisions made concerning Boxer and the environment in which it is embedded.

KEYWORDS: spatial metaphor, user models, understandability, computer language, text editor, data base language, integrated computational environment, novice programming.

This work was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract NOOO14-75-C0661.

1. Introduction

It is certain that in the future most computer users will be people who are non-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, but also 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 which has broad functionality, not via a great number of special subsystems, but via common facilities which allow specialization if necessary, but accomplish basic tasks such as the following list with a uniform, easily understood computational scheme.

- Text editing, including structured filing and retrieving

- Using and modifying pre-written programs

- Writing programs (including text editing "macros")

- Searching and manipulating data bases

- Producing graphics with a flexible, programmable graphics facility

Though we will not pursue the argument for integrated computational environments in detail here, the dominant point is that even if naive and novice users <u>need</u> 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 functionality 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 that there do not yet exist any examples of integrated computational environments suitable for naive and novice users. Instead, research has aimed largely at separate systems for separate functions, apparently mainly because of the association of these areas with different pre-computer technologies and because groups have narrowly modeled their own interest on what functionality has traditionally found a place in individual "job categories." We refer the reader to the Smalltalk, Lisp machine, PIE and Interlisp projects [Tessler 81, Weinreb and Moon 82, Goldstein and Bobrow 80, Teitelman *et al* 75] for previous 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 naive and novice users? One stands out above all others. That is the understandability and simplicity of such a system as perceived by its user. While efficiency or power can serve as the primary measures for systems intended for experts, learnability and understandability are paramount for the people we aim at. We must therefore try to understand the mental models people form of complex systems 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. Though there are beginnings [Gentner and Stevens 82, Rumelhart and Norman 81, Young 81], for the most part we are at the stage of having to announce principles just before applying them, and in fact, to some extent explaining those principles through their application. This paper must be understood as an attempt to begin to explore the principles of *engineering understandability* as much as it is to lay out a design for a particular system.

It is not entirely a disadvantage to have to work in this way, with theory so close to application. It is not often that one gets to design a system from scratch so as to be understandable according to some set of principles. Other things we might try to make intelligible, physics and mathematics for example, are much less malleable, less subject to invention and mutation than a computational system. Newton's Laws do not allow tinkering in the way one can tinker with the appearance and workings of computational systems. So in this sense, engineering these systems can be a much more direct embodiment of the learnability principles we propose, and offer more direct and rich feedback on the principles themselves. The possibility of freely designing computational systems may well force and motivate theories of understandability the same way the technology for fabricating steam engines prompted the development of thermodynamics.

The first part of this paper sets out understandability principles for integrated computational environments, largely by identifying paradigmatic classes of models users make of complex systems, each with its own strengths and weaknesses. One of the main results of this will be that the central notion of simplicity will take on a much more textured and complex character than might be the case with less developed notions of what it is to understand. Fortunately, this complexity has returns in highlighting the tradeoffs one makes in designing a system to be understandable in one way or another, tradeoffs which must be negotiated to effect a useful design. It also suggests the possibility (even necessity) of using different models for different purposes, or for a gradual shift in kind of model employed as a user becomes more experienced.

The second half of the paper applies these principles to the design of a computer language, Boxer, being developed by the Educational Computing Group in MIT's Laboratory for Computer Science as the basis for an integrated computational environment. 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. Readers interested in what has been implemented are referred to [Neves 82].

The Boxer language stands in the line of Lisp inspired languages (Lisp, Logo, Scheme) with some crucial distinctions. In particular, Boxer makes the user interface much more integral to the meaning of the system. This allows the user's stance toward the system to be one of "naive realism," what you see is what you have, and thus enhances communication of the 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 well-developed spatial schemata that humans already possess in order to facilitate modeling the computational environment. Rather than serve as a specification of the language, the present paper is intended to show by example how understandability considerations can play a significant role in the design process.

2. Principles of Design

2.1 Structure and Function

The distinction between structure and function, long considered fundamental to the process of design in any form, is particularly vital to the design of an integrated computational environment. The distinction focuses on either characteristics of an object or action which are defining and independent of specific use (structure) or on 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 universally applies (structure) or not (function).

One can contrast the structural aspects of a variable in a computer language which are given primarily by setting and accessing protocols, with a variable's functions which might be described as "a flag" or more generally, as "a communications device." Some other functional descriptions of a variable which may or may not be applied according to context are "counter," "data" or "input." In the last case, input, there is only partial structural overlap with a generic variable; the function "input" requires some additional structure, namely local scoping and flow-of-control organization to allow the procedure to have its input bound when it executes. Yet, by and large, one treats inputs as variables. The issue of partial structural overlap, which variables and inputs evidence, will be a recurring topic.

The structure/function distinction is particularly important to the design of a computational environment because not only the designer, but also the user must come to grips with the issues involved. Much more than the user of a designed artifact such as a building, the user of a computational system must understand how the system operates and how it achieves its goals in order to make it do what he intends. In terms of function and structure one can understand the obvious failure of the most naive attempts at integration -- namely, simply arranging for all separately designed function areas to be accessible from each other. (We are ignoring another potential difficulty of this tactic, that such mutual access must be trivial for the user.) The problem is that structural elements in each area will 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 monitor" as compared to "in the programming language." Commands in a text editor may have no written representation at all for no particular reason except that data objects (text) dramatically dominate procedural objects in that function domain. As a result, there is more to learn for the novice, and confusions are constantly caused by similar but not identical structures. Just as much, one is deprived of the power of structures considered important in one area but not salient enough in the functionality of 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's not part of the usual functional specification of an editor? Is instant action on keystroke command entirely useless for a programming environment?

We identify two principle design heuristics for integrating different function areas so as to avoid the above difficulties. First, design the underlying, general structures of the computational environment in such a way that specific function can be built out of them. In this way we imagine that one would often use structures not specifically tuned to some function -- or if specific functionality is important, one would build it. Single keystroke actions, for example, could be the result of a general control construct. Thus editing commands ordinarily executed with a keystroke would also have written out names, follow the syntax of the programming language and would be editable and augmentable with any user-defined procedure. This example follows a pattern, "shallow structuring," that we think must become typical: Any functionality that is important for the user to construct or modify can only be a rather simple combination of basic structures. For naive and novice programmers, that would be the limit of understandability and personalization. If the gap between basic structures and usable functionality is large, it might as well be infinite.

The second design heuristic is to exploit the possibility of achieving some intended goal functionality not by having any dedicated structure or even a simple combination of structures, but by letting several aspects of the environment take over some appropriate part of the goal. A simple "motherhood" example is allowing the use of arbitrary, mnemonic names to aid documentation. A more complex example is to help solve parsing problems not only with a simple syntax, but through interactive parsing and prompting when an expression is typed in.

These two design principles aim at structural parsimony, "collapsing" structures to a small, common core. They are directed against what we call the *hacker bug* of implementing a structure for each identified functionality. By nature, integrated computational environments are systems in which we cannot afford a one-to-one connection between structures and functions. Needless to say, this applies within as well as across large-scale functional domains. Reducing the number of structures within programming in itself is an important goal. Shallow structuring is the first of an important series of refinements and constraints on the urge to structural simplicity.

2.2 Mental Models and Surrogates

One cannot approach the design of an understandable computational environment without saying something about understanding. To this point we have relied on intuitive assessments of simplicity such as "a system composed of a large number of similar but subtly different structures is hard to learn and prompts mistakes and confusions." While we believe these assessments, it is important to begin to take steps beyond them. We must look more carefully at a hypothetical user's "mental models" of a system, that is, how a user understands in order to control the behavior of a system.

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. Such coherent, runnable conceptions have been called *surrogate models* by Young [Young 82].[1] 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 (e.g. "overflow" versus gravitational instability), but the image of a pile does allow one to simulate its important behavior.

The notion of a surrogate model is as much functional as structural. It is the intended use of the model as much as any possible generic classification on the basis of mental operations which defines it. A surrogate model is intended to capture the causal mechanism in such a way as to offer explanation and correct predictions in uniform terms. As such, it is the ultimate arbiter (for the user) for tracing entailments forward or backward in time. (The behavior of the machine itself could serve this role, except that that behavior in itself provides no parsimony of understanding.) Surrogate models are almost always explicit and taught, and one would typically see them invoked by a tutor when a novice's expectations have gone awry. Two of the most elaborate surrogate models that are attempts to encompass a large part of the operation of a computer language are the actor model of Smalltalk and the elaborated versions of Papert's little man model of Logo done by the Edinburgh Logo Project [du Boulay *et al* 81]. One possible surrogate model of a language would be a specification of its implementation, though this is hopelessly inadequate pedagogically for technically unsophisticated users.

Though it is not inherent in the concept, surrogate models are prone to certain problems as a way of giving users effective control of a system.

[1] Our use of the term differs subtly from Young's, though not enough to warrant another label. Some of our discussion below follows this same reference as well.

- Learnability: Since they 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 often lose their air of coherence with *ad hoc* elements for dealing with loose ends. 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. Incremental learnability is sacrificed for the sake of uniformity and completeness.

- Styles of use: Surrogate models, though very good for debugging (that is in cases when one has a given input to the system and needs to afford the time to, step-by-step, trace down an unexpected behavior), are typically rather slow and time consuming to "run." Routine, relatively fluid interaction with the system cannot be expected to occur as a direct result of acquiring a surrogate model. At the very least, degrees of automation, "compiling" frequently used operations, etc., occur. Moreover, surrogate models can be unfortunately far from the task of inventing a way to effect some intended result. For example, in planning situations the specification of the goal out of which the plan must arise is typically roughly outlined and almost always functional, referring to a context frame different from the surrogate. In contrast, the perspective of a surrogate model with its aim of comprehensive prediction is clearly structural.[2] Brown and de Kleer [de Kleer and Brown 81] argue convincingly that a model aimed at unfailing and comprehensive explanation must be structural and not functional. Here, some examples of the gap between functional and structural specification must suffice.

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 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 the fact 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 semantics the programmer attributes to the symbol used.[3]

More generally, learning a command 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

[2]The relation between structural descriptions and surrogate models is interesting. It is probably true that "structural" is only well-defined if an explicit surrogate, or more generally, a "well-developed theory" is referenced.

[3]For this reason it seems 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.

"meaning" of the command. Teleology, plan fragments (such as the counter paradigm for the use of a variable) -- not to mention hacks -- are not germane to a surrogate model, but are clearly part of understanding a system.

We have already mentioned that surrogate models are often complex, especially for powerful, high-level languages. A design heuristic, probably originated in response to this problem, 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. While we do not mean to broadly criticize this heuristic, (we shall use it, albeit carefully) the above considerations suggest some dangers in this route, even if it achieves the basic goal of a simple surrogate. Specifically, the construction of a broad class of functionalities out of a tiny set of structural elements is almost bound to involve great cleverness. While systems designers may be very fond of these hacks, the novice user is generally less appreciative. One must expect many of these to require specific tutoring, in which case the advantage of a small number of universal structures over a larger set more specifically tuned to important functionalities is not clear. Indeed, depending on the naturalness of the hacks (more on naturalness below), the simpler surrogate may be at a distinct disadvantage.

In summary, we add a new heuristic to our arsenal: In addition to the obviously problematic hacker bug of providing a structure for every function, considerations of understandability warn against the opposite extreme, the *formalist bug* of providing a sparse set of primitives out of which to build all functions. If typical learning of the language will require a number of special functional tricks, the advantage of a very sparse structural vocabulary with correspondingly simple surrogate model is much reduced. An alternative strategy is to focus on some relatively small set of special cases corresponding to basic functionalities and to tune the language's structures to those cases. Indeed, we think learning prototypical cases in functional terms will almost always precede understanding a surrogate in any case. Though we must still expect coherence of the special cases to be important (and will explain how we expect our proposals to achieve coherence), we will follow this line to avoid the formalist bug.

2.3 Functional Models

Some readers may, understandably, be uneasy at our criticism of achieving integration with a uniform, simple surrogate model. What can we offer in replacement? To begin, we suggested above the technique of linking structures more closely with functions which are already understood or are easy to learn. Following earlier remarks, in making such *functional models* we must (1) take care not to have too large a set of these nor (2) tune them tightly to specific "traditional" functional areas, nor (3) foreclose the possibility of an effective surrogate in the cases that is likely to be needed, like debugging. The pattern of learning, then, would be that a *few*, *important* and *generally useful* aspects of the language, say, particular constructs, would be learned as solutions to specific problems.

These points deserve elaboration. Young [Young 81] details what he describes as a task-action mapping model of algebraic calculators 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 quite a sufficient model of the system for many purposes.

Models acquired in this way 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 prototype "problem insertion" 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 should 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, e.g. the internal state of the calculator after pressing 1 +. This knowledge is important for debugging and similar tasks, for example, to know what to do to correct a mistaken + when - was intended. It is also important in that if a structure is to serve several functions, a single functional projection may not be sufficient to allow the user to understand or generate the other functional descriptions. Indeed, because of the semantic constraints within the functional descriptive frame, certain combinations of structures (1 + + is an example) will lie entirely outside the capabilities of the frame to describe.

In a general-purpose computational environment, one cannot expect to model a very large part of the system with functional models based purely on common knowledge or previously understood media.[4] But the same strategy can be applied after some initial experience to bootstrap on learned

[4]The Xerox Star [Smith *et al* 82] uses ordinary office procedure itself as a model for understanding that system. This is an impressive expansion of modeling on the basis of previously understood media over modeling a calculator on paper and pencil calculation. But it falls short of being a general purpose computational environment.

capabilities. We expect beginners to start by learning relatively trivial actions which will allow them to inspect, generate and use simple programs. Many of the structures of the language will be understandable as solutions to specific problems such as creating and using in new contexts objects whose form and function are already understood.

Functional models provide only a view of part of the system, and that only with respect to a non-universal frame of analysis. Obviously, one needs a repertoire of them; the notion of a single model 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

Knowing a friend is not much like having a surrogate or even a collection of functional models. A moment's thought about the learning process reveals why. Learning experiences are more typically discovering actions and reactions of the friend in many different contexts, very few of which have simple, evident or generalizable functional frames. In this section we describe a kind of episodic learning similar to that described for a calculator above, but within which the acquired model is due to a spectrum of partial understandings -- not to 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.

This example comes from a rather early stage in learning Logo. Beginners almost always start by driving the Logo turtle (a graphics cursor) around with commands like FORWARD 100 (meaning move forward 100 steps). While structurally one could describe this as a function (in the Lisp or mathematical sense)[5] with its input, it seems certain elementary school students must be interpreting FORWARD 100 essentially as an abbreviation for an English sentence like "go forward 100 units." The need for input, therefore, is semantic and situation specific, not structural.

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 example.

[5]The ambiguity of language between function in this sense and function, the partner to structure, is troublesome, but unavoidable.

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

In structural terms, TO SQUARE :SIDELENGTH is the syntax for defining a function, but it is easy to see the syntax is intended to continue the interpretation of a command as a verb. The English infinite form 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, e.g. "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.[6]

But 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 particular, the use of : deserves attention. In Logo the : (pronounced "dots") denote the value of a variable and is included in the definition syntax to parallel and reinforce the pattern of invocation of procedures with variables as inputs, e.g. FORWARD :SIDELENGTH, or more particularly, to parallel recursive call format, e.g. SQUARE :SIDELENGTH in the final line of the above procedure. These consonances are subtle, but worthwhile, and do not interfere 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 name, and to distinguish them in the form one most frequently uses a variable, getting its value. Thus the whole syntax package is more easily adopted and used for several distinct reasons having to do with different channels of coherence with the rest of the system, channels such as a broad interpretive frame like "English," common-sense reasoning (in this case, rationalization, ": is used to distinguish input variables from the name of the

[6]The root meaning of Logo is Greek "word;" this use of natural language in making Logo learnable was quite deliberate on the designers' part. Incidently, assimilating procedure definition to the natural language structure takes advantage of the fact that, in English, the imperative form (SQUARE) and the infinitive form (TO SQUARE) use the same word, as well as of the fact that it is not outlandish in English to use a noun like "square" as a verb. That this is not the case in many other languages has proved problematical in devising effective translations of Logo.

procedure"), or even visual pattern matching.[7]

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*.[8] It is easy to emphasize how far these depart from what one would expect if a simple surrogate accounted for all of learnability, 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. 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 naive and novice user.

There are limits to the usefulness of such situation-specific distributed models. Procedure definition is special in several respects. The problem context 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 such as the structure of ordinary function application which implies that PRINT SIN 5 does not print "SIN 5" and that PRINT HELLO gives an error if HELLO is not a defined procedure. Only on such occasions (well-defined context, rich and frequent support for use of the context dependent "solution") can one expect specific semantics to reliably dominate uniform syntax through the entire course of coming to understand a system.

It goes without saying that one must consider long term effects of particular functional and

[7]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 which changed the syntax to TO SQUARE SIDELENGTH have prompted complaints from novices whose rationalizations were violated and, it seemed to us, prompted more mistakes from beginners.

[8]The notion of a distributed model is derived from ideas we have developed about understanding complex systems in other domains. See [diSessa 82a, diSessa 82b].

distributed models. Some will remain and be integrated as "special case" models. 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 are aware that globally destructive misconceptions may be fostered as well as "profitable misconceptions."

The force of this example is due mainly to the fact that defining a procedure is an extremely early activity in learning Logo. It is a great defect in past language design that even the grossest features of learning sequence are ignored in considering "simplicity." In our proposal, coarse features of a learning sequence will be at least implicit and sometimes explicit.

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 not sufficiently close to the functional terms in which problems to be solved are phrased. Our discussion has not only mapped out these typical failure modes but also has proposed building less coherent, but still effective models based on function or on compatibility with a collection of partially explanatory frames. A typical 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 need to be understandable on the basis of naive functional and distributed models. 3. The initial models should 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 we see as important to understandability, we turn now to more detailed assessments based on particular knowledge, "model-building material," users might have or lack. Given the general strategic decisions made for Boxer, we shall find that static organization 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 terms of amount of data available at any given time or to facilitate the operation of the system for already comprehending users. In contrast to pop-up menus and iconic mnemonics, we would like to use the video screen to attack the fundamental problem of understandability of the basic organization and operation of the computational environment.

The means we intend to use is a comprehensive spatial metaphor. In particular, 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. Perhaps most important, 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 are their visual representation. What we want the user to see on the screen is, as close as we can arrange it, the computational system itself rather than a multiply-filtered or side-effect dominated view of it (e.g. a window occurs in some place and size because that was what was available on the screen when the window was created). Taking naive realism so seriously, in fact, separates this proposal most strongly from all previous computer language designs. Even those designers who are willing to divert resources like the display screen from "tuned-to-a-t" functionality to understandability have almost universally opted for user interfaces which act as buffers or facades to hide system complexities from the user rather than to search for a simplicity which could be shown.[9]

There are two driving forces behind use of a spatial metaphor. The first is that humans have a great deal of knowledge and a broad collection of skills for dealing with space. Humans happen to live in a world which is profoundly geometric in the sense that objects and places are salient and tightly interrelated. The dominant mode of interaction with the world is to move objects (including oneself as an extremely important special case) around into different configurations rather than, for example, to mutate the objects or even pass messages between abstract, placeless entities.[10] Things might be different if our dominant sense were not vision, but instead, hearing, which is

[9]See, for example, [Innocent 82] or [Goldberg and Robson 79].

[10]It is not that we are not impressed with the power of actor based languages, but the amount of work the actor metaphor does in promoting understandability, besides providing a uniform syntax, is problematic.

dominated by messages.

The second driving force behind use of a spatial metaphor is the character of spatial knowledge, which we find extremely compatible with the structure of Lisp-like languages, of which Scheme is the closest to what we need. It appears to us that in many ways we are simply uniformizing, extending and concretizing many fragments of a typical user's understanding of these languages. This should essentially guarantee that what we make will be a usable, practical language.

3.1 Static Structures and Functions

3.1.1 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*, which is dominantly text. The root meaning of a box is a "thing" and its contents are its parts. The choice of text as the main surface form stems from the Logo idea to explicitly import 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. As details emerge, it will become apparent how text manipulation and program writing are intended to be mutually supportive activities. Either can serve as a good introduction to the other.

Boxes may contain subboxes, either named or not. Boxes are logically as well as visually two-dimensional arrays in the sense that they are a sequence of lines, each of which is a sequence of words (Lisp atoms) or boxes. Names of boxes must also be words.

```
BOX1---------------------------------------------------------
     |  This is a box whose contents is the text            |
     |  you are reading.                                    |
     |  Here is an unnamed box:  -------------------------  |
     |                           | This box is the       |  |
     |                           | last item on its line. | |
     |                           -------------------------  |
     |  Here is a named subbox whose internal detail has    |
     |  been suppressed:    BOX2--------                    |
     |                          |//////|                    |
     |                          --------                    |
     --------------------------------------------------------
```

Figure 3-1: A box with contained boxes.

The abstract structure of a box, a hierarchical two-dimensional array, is enough to build almost everything needed in BOXER without violating shallow structuring. 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 Lisp machine [Greenblatt *et al* 80], uses a mouse) and the keystroke commands ENTER and EXIT. ENTER, when the cursor has been placed in a subbox, visually causes that box to be the top level of display and EXIT "pops" up to the next higher level. Creating, deleting and moving boxes are simple functions of the editor.

3.1.2 Boxes as Procedures

Procedures appear as boxes. (See figure below.) Subprocedures 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 suppress detail for clarity. In many of the examples to follow we use turtle graphics and essentially the syntax of Logo.[11] In particular, Logo is line oriented rather than Lisp's expression orientation. Not only is this line orientation consistent with the basic box's visual organization, but it encourages a kind of modularity that is important for non-professional programmers; the effect of anything typed in a line is confined to that line.

```
SQUARE-------------------------------------
       |                                  |
       |  REPEAT 4 SIDE----------------   |
       |               | FORWARD 100  |   |
       |               | RIGHT 90     |   |
       |               ----------------   |
       ------------------------------------
```

Figure 3-2: A box represents a procedure.

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 usable 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 interior to the box and in any box contained (recursively) in that box. Containment implies inheritance. The details of scoping will be treated in section 3.2. The procedure below has an input, NUMBER, and draws a polygon of NUMBER sides and sidelength LENGTH.

---

[11] We have not settled the issue of syntax but Logo is a close approximation to our current best guess.

```
POLY-------------------------------------------------------------
    | INPUT NUMBER        LIBRARY| LENGTH------- NUMBER------- |
    |                            |        | 100 |       |     | |
    |                            |        -------       ------- |
    |                            --------------------------------
    |  REPEAT NUMBER  --------------------                      |
    |                 | FORWARD LENGTH   |                      |
    |                 | RIGHT 360/NUMBER |                      |
    |                 --------------------                      |
    -------------------------------------------------------------
```

Figure 3-3: The local library contains definitions generally useful in the box.

The value of inputs and local variables will be inspectable in the library during debugging. A value written into the contents of an input will serve as a default value. 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 of 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. Since subprocedures may appear in place, single-purpose subprocedures need never appear in any library at all.

Though the local library is special in the sense that it is not part of the contents of the box in the ordinary way (e.g. 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. 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 semantic as container of generally useful information about the box makes it a natural place for annotation, documentation and other help.

3.1.3 Boxes as Data Objects

The boxes NUMBER and LENGTH above function as variables, and generally boxes will serve to define data as well as procedural objects. In contrast to traditional languages which have a number of different structures for handling compound data (strings, arrays, lists, records, etc.) box structure is intended to be universal. This structural universality, as with lists in Lisp, should be a source of great power and simplicity. To achieve the full benefit, boxes must be first-class objects. Boxes already have names, and we will deal with details of other aspects of first class status later.

Lisp's universal compound structure suffers some of the same problems as a simple surrogate -- namely, 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 simply contain some text in the usual sense. And we will not tamper with the box structure *per se* in tuning even more to specific data functionalities, but instead we will add a number of different access routes to parts of the structure which are aimed toward particularly important classes of functionality. We expect these to be learned as solutions to particular problems as the user advances.

One of the most important of compound data functionalities is the ability to deal with named subparts easily. Most Lisps have property lists that are often used for this purpose. Boxes have the capability implicit in the fact that any box or subbox may be tagged with a name. 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 NUMBER's contents to 5. One can specify any number of levels, e.g. VECTORS.V.X.

```
V---------------------------
| X-----   Y-----   Z-----  |
|  | 1 |    | 0 |    | 0 |  |
|  -----    -----    -----  |
 ---------------------------
```

Figure 3-4: A vector with labeled subparts.

It is nearly as 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 would be extremely natural to use array indices into the two-dimensional structure (rows and columns) of a box. One would also like to reference rows because of their important meaning (visually, in procedures, etc.) and elements by their sequence number (reading, as text, left to right, top to bottom), e.g., ROW 1 ABOX or ITEM N BBOX.

3.1.4 Boxes as Environments

Boxes and a local library provide a functionality that has been much neglected in computer languages, that of an environment. One can arrange a place in which a particular set of procedures and data are available for a user to employ in an unconstrained way. The construction of environments has proved to be an invaluable method for teachers to provide students with domains to explore [Abelson and diSessa 81] or with generally useful "tool boxes." A box used as an environment has some advantages over programs or even workspaces which might otherwise serve the same purposes. In contrast to a program with specific I.O., box-environments save the programmer from creating (and the user from needing to learn) a special interface. Environments allow flexibility in terms of simple programming on top of what's given, easily accepting a very general class of user-initiated modifications. Like a workspace, an 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 (e.g. one can nest environments) and better integrated (e.g. constructible and editable in the same way data and procedures are). 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 "helping" procedures that the ones intended to be used at top-level are not at all apparent.

"Time modularity," how one creates natural and stable boundaries in time between sets of activities, is one functionality of the file-workspace organization that is not taken over by box structure. The simplest Boxer structure to handle this is the ability to save and restore named state versions of any box. Note this gives much finer control over time modularity than workspace-files. One can save versions of a procedure within an environment. With this simple method, coordinated changes in an environment must be handled at a level of the hierarchy which is sufficiently high to contain all such changes.

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, the local library can contain documentation on all the system primitives. The contents of UNIVERSE would contain the top level view of the organization the user chooses for his entire environment. This use of box structure duplicates most of the functionality of one of the most successful aspect of the Smalltalk programming environment, the Browser, allowing leisurely perusal of the entire system.

Boxer's advantage in this is that it is all Browser! There is no need for any dedicated structures with extra work in learning to construct or modify the "shape" of the system.[12] The long time-scale, global personal organizational capabilities of Boxer are one of its most important advances over previous programming languages.

[12]The Browser in Smalltalk is not part of the programming language, but part of the user interface. This means, among other things, that much Browser structure is special to the Browser, not easily modifiable by users, and the parts that are modifiable must be affected through specially learned procedures. It is also true that part of the system organization seen in the Browser exists only for the Browser and does not reflect system semantics in a fundamental way.

```
UNIVERSE-------------------------------------------------------
        |                                                     |
        |  PAPERS-------------  SCHOOLWORK------------------  |
        |        |///////////|             |  MATH------   |  |
        |        -------------             |      |////|   |  |
        |                                  |      ------   |  |
        |                                  |               |  |
        |                                  |  PHYSICS----  |  |
        |                                  |         |//|  |  |
        |                                  |         ----  |  |
        |                                  -----------------  |
        |                                                     |
        |   GAMES-------           MAIL-------                |
        |        |//////|              |//////|               |
        |        -------               -------                |
        -------------------------------------------------------
```

Figure 3-5: Box structure can organize the whole computational environment.

Some of the flexibility and ease of learning in Boxer comes from the fact that the process of constructing a box is separated from the static representation of the box once made. This allows users to choose method of construction and modification. One may type a procedure or assemble it out of previously written text (for example, out of commands typed in an environment in the course of experimentation), then move it to the local library; one may type a procedure directly into the local library; or one can have some procedure (TO) do that work by side effect. Identifying the process of constructing a procedure with its static representation, as with Logo's TO or Lisp's defun, is a relatively clean hack, but one we consider a remnant of teletype interaction, where object creation by side effect is a necessity. Especially for environments and large data objects, the concrete access provided by our spatial/naive realist approach is natural and functionally superior to creation by side effect and mutation by magic.

3.1.5 Kinds of Boxes

We have been discussing a very wide class of functionalities deriving from a single structure. Though we would like this to be precisely true, our implementation and other considerations have convinced us that boxes need to be labeled as to type, and have slightly different behavior accordingly. Most of this has to do with control (next section). Briefly, the kinds of boxes will be something like the following list: We will have *data* (variable) and *doit* (procedure) boxes. *Environments* should be labeled to help an exploring user locate the levels of the system intended for use in that way and to keep one from the meaningless act of executing an environment. For several reasons, boxes which 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 it being executed. Finally, we will want *graphics* boxes which are saved as bitmap images.

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 learnability of the system through inspection and through a uniform method of interpreting, modifying and expanding what one sees. Nonetheless, these identifications are a very strong constraint on system organization and possible interpretations of "running a program" (to be discussed below). There are very cogent arguments that simple hierarchies are not optimal in, for example, scoping rules, though the sense of optimal is usually phrased in terms of reliability and flexibility -- not in terms of our chief concern, understandability. A context in which we do accept the force of these arguments is the "user interface." In Boxer what one sees on the screen, how one interprets it and how one manipulates it is dominated, if not entirely appropriated by a uniform semantic and method of manipulation. One sees the system. On the other hand what one wants to see on the screen at any given time might be things which 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.

To meet this need and a few others we have been experimenting with a single structure which provides some of this functionality but which we consider minimally subversive of the system semantic. It is called a *port* and has most of the properties of a box. It appears as a rectangular region, though specially marked, can be named and is constructed and erased in the same way that a box is. But its meaning is a passway 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. 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 the port itself is spatially located in the system hierarchy, not attached to the screen. A port appearing in a data structure indicates the object contained is shared in basically the Lisp sense. The difference between Boxer sharing and Lisp sharing is that any object really belongs to (is contained in) a unique other object and can only be "used" in other places. In the section below we will give some examples of functionalities other than "screen organization" that can be appropriated by the port.

In terms of understandability, we believe Boxer fares well as far as it's static organization is concerned. There is a small structural core of spatially organized textual objects, and the main associated functionalities do not appear to make radical changes, either semantically or visually, to 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 which 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. 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 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. We expect the results will bear not only on the judgments we have made about the feasibility of expecting users to appropriate the models outlined, but also they should bear non-trivially on the sketch of mental model building for systems of this sort which has served as a design heuristic.

3.2 Dynamic Structures and Functions

Now we turn to dynamic structure and function, the issue 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>. These can be dismissed here easily. While we are aware of some difficulties in modeling these, we do not think the problems are as serious as many others. And it is also true that we do not have much to add to what has already been said about them. So Boxer will simply appropriate some set of these which do not differ greatly from those available in Logo or even Pascal. Instead, we think the deep issues have to do with what a procedure does when executed, what it means to access and set a variable.

3.2.1 Reference

A useful non-computer context for introducing these issues is reference in natural language.[13] Humans have an extremely 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 man who ..." is apt to be replace by "Joe" or whoever 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 which prompt type/token or use/mention distinctions in classes of 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 comprehend the clevernesses needed to make the context-free mechanisms find the appropriate reference.

Logo took an apparently schizophrenic approach to the problem. On the one hand, it granted special status to functions like ERASE (clear a procedure from workspace), PRINTOUT and even TO so that one writes TO SQUARE rather than TO "SQUARE, to simplify this semantically clear reference.[14] 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"

[13]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 construal of the issues involved.

[14]As mentioned earlier, literal reference mode, specified by quote, would be necessary if TO followed usual function evaluation rules for its input. Also, we are concentrating on inputs here because the imperative (English) interpretation of function names is rather successful at providing a model for the reference (carrying out an action) of the first element of most Logo command strings.

(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.[15]

What has proved more problematic is that : in Logo truly denotes a structural reference mechanism and not a kind-of-thing as the functional distinction procedure/variable might imply. The problem 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 are largely inherited from Lisp and tuned to function application) Logo writes MAKE "X :Y (X gets Y's value), even though X and Y are both variables.[16] Experience suggests that learnability is problematic; the variable assignment syntax is not as susceptible to episodic learning based on germane rationalizations as one might have hoped. Thus it appears to be a burden without significant advantage for beginning users who cannot be expected to see the structural significance of the markers and must rationalize on purely functional grounds -- ": denotes a variable, except in MAKE," the latter part of which is without any generalizable import.[17]

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. Not only that, but novices find them strange and remarkable when they do encounter them. Even if the flexibility is there, that does not mean it will 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. In general our heuristic with Boxer is to simplify the lives of early users, even

---

[15] As an example, 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 following that assumption. Good rationalizations, like visual pattern matching in procedure definition syntax/invocation, do not spread to inappropriate contexts like this.

[16] This assignment syntax is a little like the $e^{i\pi} = -1$ of Logo in that it contains nearly all the important things in the world (namely, the three main reference types, function-procedure, literal and variable) to produce a simple effect, variable assignment.

[17] Another problematic rationalization is to think that the two character string "X is the name of the variable and :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 implementations of Logo have supported this.

if it means complicating slightly the lives of experts.

Finally, one could argue that a syntax which hides the difference between kinds of references is bound to be confusing. But in the first place, 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. Second, while the literal marker might be rationalized to represent named-object reference, in fact, it represents only a mechanism of achieving that reference. (Although it is typical of that kind of reference, quotes are used for other purposes as well.) As well, this is not a rationalization likely to be made by beginners. More 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 non-specific syntactic reminder, quote.[18] Indeed, later we will propose a semantic reminder in the form of a prompt, which has more attractive features.

Rather than the radical step of abandoning any form of uniform input protocol, we propose the following two-fold strategy. (1) First 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 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 second arm of our strategy follows from the observation that this only postpones the issue, 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, etc. (2) We would therefore like to ease the transition to structural understanding of reference mechanisms. To do this we propose (2a) to improve the understandability of the underlying reference mechanisms by developing better surrogate models for them, and (2b) 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

[18]In this light, consider the proposal made earlier, that V.X should denote the X subpart of V. This is intended to allow users to think of V.X as a named object which exhibits *variable* behavior. Thus in most contexts V.X is an appropriate replacement for the contents of V.X, and in the case of MAKE V.X <whatever> one gets the "setting" behavior of this kind of named object. If a user understands what . means, it should be no concern of his that ordinary structural reference mechanisms must be augmented so that V.X can be interpreted properly and not as a character string which appears literally as the label of some box.

to implement a method of watching a program in action to spot the error.[19] Debugging, of course, is important in its own right for a host of other reasons. But perhaps most important, the visual method we've chosen to implement will aid the acquisition of the 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 2a, underlying surrogate models, on which the others depend.

3.2.2 A Surrogate Model for Boxer

The key ideas in producing a surrogate for Boxer are to start with reference mechanisms linked with kinds of objects, and to produce meaningful and visualizable (hence also depictable) intermediate states in the execution process.[20]

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. As such, we have collapsed the two structures of literal reference and variable into one. A data box appearing in place (e.g., in a procedure) marks the contents as literally referenced.

```
          <data>
PRINT   ---------
        | HELLO |
        ---------
```

Figure 3-6: A data box marks literal reference.

The surrogate model for evaluating an expression involving a data box referenced by name involves the process of 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 is identical, but the execution stage is recursive, i.e., will in general involve copying and executing elements of the contents of the procedure. In short, this "copy and execute" model involves optional copy (in case of reference by name) followed by execution,

[19]Ron Baeker [Baeker 75] and Henry Lieberman (personal communication) have implemented systems with this functionality.

[20]See [Lieberman 82], which discusses the importance of representing intermediate states.

which is recursive in the case of a procedure, terminating at the action of language primitives.[21]

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 imagine 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. It is important to realize that the hierarchical lookup scheme adopted as standard for Boxer along with the copy and execute surrogate causes dynamic scoping to be the rule for free variables, i.e., ones used in a procedure but not contained in that procedure's local library (which is copied with the procedure).

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. One could pause to inspect the calling hierarchy and the state of local variables (including inputs) at any 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 (via a port down to the level of the error) and inspect the stack.[22]

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 which actually exists in some distant part of the system. Thus control is passed to that distant place, in which execution proceeds in an entirely normal manner, except any result is passed back to the calling environment by virtue of being "visible in the port." One can retrieve and set variables in non-local environments and use procedures which have need of a different environment. In the example below, executing PORT1 will set the variable A in PLACE2 which contains the target of the port.[23]

The semantics of copying a port is unproblematic; the behavior of a copy is the same as if one had written the port directly in place. So one can even use ports by name. Dynamically as well as

---

[21] A 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's what's happening.

[22] These are not new functionalities to programming systems. Smalltalk and various Lisp implementations 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.

[23] We now consider it more likely that a port should be only a flavoring of reference type which indicates remote environment, not a reference type in itself. Thus ports will have to be marked explicitly *dataport* or *doitport*.

```
PLACE1----------------------
       |         <port>     |
       |  PORT1------------ |
       |       | MAKE A 5  ||
       |       ------------ |
       ----------------------

PLACE2-----------------------------
       | TARGETOFPORT1------------ |
       |              | MAKE A 5  ||
       |              ------------ |
       -----------------------------
```

Figure 3-7: 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.

statically, ports give a mechanism for breaking the strict hierarchy of box structure, and we consider it a minimal and natural extension.

It is important to note that, though their dynamic surrogate expressed in words is not complicated, ports are more difficult to watch than normally copied and executed objects.[24] This is the main reason we consider ports to be advanced elements in Boxer.

3.2.3 Inputs

We return to the issue of varied and invisible reference mechanisms for inputs. (Recall, procedures and variables when referenced by name are not syntactically distinguished.) The idea is to let the procedure establish context, how the text which constitutes an input is to be treated. We propose three flavors of input which parallel each of the three ways (data, procedure and port) execution treats the contents of a box. The procedure flavor evaluates the input text according to the standard Boxer rules and installs the result in a data box of the appropriate name which is located in the inputting procedure's local library. This matches closely the universal Lisp and Logo 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 (which is still a data box). This is Lisp's fexpr, except the "flavoring" is attached to the particular input, not to the function (Boxer procedure) as a whole. This is the right kind of input for messages and other textual data. One need not bother with literal markings of any sort.

[24] If the difficulty is not obvious, we will consider it in more detail in section 3.3. Incidentally, one of the "cute" things which can be done with ports is programming without names, where every reference is "wired in" with port connectors.

The final kind of input uses port semantics, and therefore will probably be used only by advanced users. The input box is a port to the text of the input. This is the appropriate way to input procedures, as it avoids certain funarg problems [Steele and Sussman 78a]. For example, if BAR is taken as a port input to a procedure FOO, the environment available to BAR when it is executed in FOO is the environment where the symbol BAR was typed, outside of FOO. If BAR is a port, naturally the environment available to it would be the target of the port.[25]

As far as learning sequence goes, it should be expected that users will use the procedure ("value" makes a better mnemonic) flavor for their own definitions for quite a while. Numbers predominate as inputs to early, user-defined procedures. Value inputs will be the default when no flavor is specified. During that early time, the other flavors 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 flavored 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 semantic 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.[26]

3.2.4 Two Proposals for Non-Lisp structures

In this section we treat two kinds of functionalities not well served by structures in Lisp, and accordingly we make proposals for Boxer. The first of these functionalities 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 intend to have special syntax tuned to this functionality, e.g. IN <environment> <do such and such>, or TELL <environment> <whatever>. Since Boxer has a fully developed environment structure, we believe syntax is all the dedicated structure message passing needs. Instances will be made by

[25]Note that inputs are always data object when used internally (within the inputting procedure). Passing procedures as inputs means having a procedure inside a data box. If it is to be executed, an explicit DO (meaning execute the contents of this databox) will be needed. Outputting procedures must proceed similarly. Scheme does not link type of object to type of reference, but uses syntax for the latter. This allows one to reference procedures as data objects more easily than with Boxer.

[26]Advanced users, of course, can change 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. A suggestion for a transparent way to do this in some important cases is given in Section 4.

copying environments, and subclassing by nesting environments. (An example of subclassing appears in the next section.) There are potentially important gains in having message passing functionality in Boxer, but also some issues about how convenient and natural such facility will actually be, which we are in the process of exploring.

The second neglected functionality 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 novice 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.

We are exploring two possibilities. The first we call *geometric outputting*. It uses the procedure box control structure along with the image of a dereferenced data box (or the output of an executed procedure box) leaving its value in the place where it is executed. Thus, a Boxer procedure will output all values returned from subordinate procedures or data boxes whose values are not used by another procedure, and the geometric relationship of those values will be preserved. The figure below gives an example. A procedure box, none of whose subboxes return a value, itself will not return a value. We are assuming that, like Logo, many of the built-in primitives don't output.

```
                  <proc>                                    <data>
 ------------------------------                          -------------
 |      <data>                 |                         | HELLO 2   |
 |      ----------             |     evaluates to        -------------
 |      | HELLO |    (1 + 1)   |
 |      ----------             |
 ------------------------------
```

Figure 3-8: Geometric outputting

There are limits to the usefulness of this idea. It is tuned to "flat" structures both because it flattens (e.g. HELLO above is "unboxed") and because the use of procedure-invocation control does not allow unevaluated levels between the top-level and terminal evaluated nodes. If one wants to insert an evaluated object at a low level of the to-be-constructed object, one will have to, for example, construct the "shape" of the object first, and then assign the computed value.[27] But many simple cases will be taken care of easily.

Geometric outputting may pose implementation efficiency problems. A fallback position would be to use conventional Lisp outputting (last subform supplies value), and have a new *constructor* type box which takes over the function of geometric outputting in compound data construction, but without overhead for usual outputting situations.

3.3 Lexical versus Dynamic Scoping

We have been implementing Boxer on top of a Scheme interpreter, and in general have found Scheme a congenial implementation language for the semantics of boxes. A natural question is why doesn't Boxer appropriate as its dominant scoping technique one of Scheme's distinguishing characteristics, lexical scoping? Deciding on a scoping mechanism has been a difficult task, and the decision process makes a good case study because, in the end, modeling considerations have been pivotal. Below is the case for dynamic scoping.

1. Sometimes one really wants dynamic scoping. Consider environments in the sense of workspaces discussed earlier. If one takes a procedure to another environment, or creates an intermediate environment between the procedure's environment and UNIVERSE, it may well be explicitly for the purpose of altering the meaning of the terms making up the procedure definition. The figure below shows how dynamic scoping can be used as a mechanism to create actor-style instances. The environment TURTLE has a set of state variables, X Y and HEADING, which are manipulated by functions FD (forward), RT (right turn) and LT (left turn). A turtle instance JOE is created by making a subenvironment containing its own state variables, but it uses TURTLE's manipulator code on these. If one wanted JOE to have a different FD behavior, a new FD could just be added to his library, shielding TURTLE's FD. Other arguments for the usefulness of dynamic scoping, which are not dependent on Boxer's environment structure, are contained in [Steele and Sussman 78b].

2. Dynamic scoping is more natural to Boxer than to non-spatially organized languages like Scheme. 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 one of performing

[27] In some cases, this "hack" will be easier and more obvious than others. For example, one might concretely construct the shape of the compound object once and for all in some local library, and assign the computed values to subordinate boxes as the values become available.

```
TURTLE---------------------------------------------------
      |                              | X Y HEADING       |
      |                              | FD LT RT          |
      |                              |                   |
      |                              | JOE-------------- |
      |                              |    |   | X Y    | |
      |                              |    |   | HEADING| |
      |                              |    |   ---------- |
      |                              |    |            | |
      |                              |    -------------- |
      |                              |                   |
      |                              ---------------------
      | TELL JOE FD 100                                  |
      |                                                  |
      ----------------------------------------------------
```

Figure 3-9: Sending a message to JOE, an instance of TURTLE.

operations in environments which define the meaning of those operations.[28] In a Lisp experience, environments are transitory, set up for function calls and destroyed on exit. Though Scheme has environments as a basic fact of life, environments are hardly concrete and manipulable in the transparent way they are in Boxer (e.g. picked up and moved around with the editor). In Lisp worlds, it is an appropriate aesthetic to avoid dependence on invisible and hard to manipulate things; functions really ought to do the same thing on each invocation. But if "functions operating in environments" is the fundamental, concretely represented metaphor of the system, we need worry less about potential false expectations of modularity and problems debugging them if they occur. To some extent the problem is also ameliorated by the fact that novices will not be constructing extremely deep and complex programs which pose stricter modularity problems.

3. Boxer, particularly with ports, will have most of the functionality of lexical scoping for advanced users. Some of the important functionality of lexical scoping can be taken over by the local library, which is copied with the procedure text to the environment of execution. If one wishes to do some work, one can have an arbitrary amount of the defining environment carried to the calling environment in this way. Ports can take over more of the functionality of lexical scoping. The copy-and-execute rule that ports are copied as ports to their original target, and the meaning of ports, that procedures viewed in a port are executed in their original environment, imply that ports used in this way cause precisely the effect of lexical scoping.

4. 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 and Sussman 78a]. This runs directly

[28] In fact, it can be argued that concrete experience in Boxer, e.g. typing a command in some environment and executing it, teaches dynamic scoping. This is because in such circumstances, there is no distinction between dynamic scoping and lexical scoping at all. One would expect users to generalize the simpler model, dynamic scoping. (Point 4 in the text above argues that it is simpler.)

counter to the principle of naive realism we have adopted for Boxer, what you see is what you have. 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 non-local 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) geographical focus must shift 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 mid-stream could wind tortuously through the spatial hierarchy. Though the surrogate *per se* is not immensely more complex for lexical scoping, more of it is invisible and not amenable 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 which 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.

4. The User Interface

We have taken a great deal of time explaining the computational semantic of Boxer and very little describing the user interface. The primary reason for this is that so much of the functionality of what is usually called user interface has been built directly into the fundamental meaning of the system. This seems inescapable in naive realist systems.

Beyond basic functionality, users also need to be able to tailor the interface to their own needs. This can be done to some extent with the basic structures proposed already: via expanding and changing the set of single keystroke commands, via port placement, and other examples to come.

Conversely, some of the functionality which one usually needs to have as part of the programming language can be taken over by the interface. In our case one constructs 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 part of the language. One may wish to have procedures created as side effects of special function calls, but that is not necessary for most novice programming.

There is another pair of user interface issues worth mentioning here. Menus are extremely useful to unsophisticated users, therefore it is important to retain some of their functionality for Boxer. Luckily this is not hard to do. Anything the user types is a usable artifact which 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 will have to do to use some text as a menu will be to point at a line and press the DOIT key. Users will undoubtedly gradually build their own menu interface in some environment from 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 sub-environment for others' use should leave such artifacts around. 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.

Boxer will have an interactive parser and prompter to aid users in constructing expressions which do what they intend. The key is not to make such help obtrusive by burdening the user when he does not want it, or by introducing modes and screen objects which do not behave like the normal Boxer mode (note the singular!) and objects. What we have in mind is something like the following: If a user wants help with a function, he types the name and presses HELP. What appears are input prompts as boxes labeled by the mnemonic names chosen by the programmer. If defaults were given by being written into the input box as it appears in the local library of the procedure definition, those will appear in the prompt boxes. One could have written additional commentary as a text box in the input in the same way as a default value. (Remember text boxes are not executed with the code in a box.) It is important that the prompt boxes are not special; they can be changed in the usual way, e.g., tc change the value which appears as a default.[29]

The prompt boxes have a natural meaning in the surrogate of the system. Namely, these are the boxes which determine which kind of input is being used, value (procedure), literal (data) or port. Aside from documenting that choice at call time, a user can change the type if his local purposes do not match the selection made at define time.

Prompt boxes also serve to parse expressions to an arbitrary depth based on the same box hierarchy used generally in the system. Outputting with no explicit markers allows the execution surrogate to be imported directly to this situation without loss of any functionality. One can type, for example, "sin x" in an input box rather than "output sin x" as would be necessary in Logo. In short, procedure boxes serve parsing functionality as well as procedure, function and (possibly) data-constructor ones. Because of this multiplicity of use we will likely have "procedure" be the default type of box, assumed if no other type is written in.

[29] In order to keep the system modeless, one would like to be able to create input prompts by hand as well as using HELP.

5. Summary

In designing an integrated computational environment the most basic heuristic is to combine functions, i.e., to try to generate a small set of structures out of which all necessary functionality can be built. An immediate caveat to this is shallow structuring, that common functionalities must not be difficult to express in the fundamental structural vocabulary. But deeper and more complex revisions to the basic heuristic are in order in view of the limiting resource in understanding and controlling a complex system, namely the materials and capability to construct mental models of such a system on the part of the human user.

We have found it important to consider a few paradigmatic kinds of models 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 predict and 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 class of models, distributed models, in which not only is there no global mechanistic frame (surrogate) but one may not even be able to identify a single functional frame which accounts for understanding and remembering 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 a language, Boxer, aimed at being the basis of an integrated computational environment. Boxer follows the following four principles:

- In order to minimize the need for abstract 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 a rather extreme form of 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 aptness 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 (flavors of inputs, syntax for message passing, spatial construction of compound data objects). In particular, flavored inputs allow the simulation of a broader range of naive reference mechanisms without intrusion into the surface appearance of the language. As well, 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 not discussed either the consistency or completeness of Boxer as a computational scheme.[30] 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 hardly 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, also on our own experiences and introspection). Even granted our general modeling considerstions, we have had to make decisions about specifically what knowledge we can count on users having and applying, e.g., what rationalizations will be made. Obviously a great deal more study in this area needs to be done, 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 as manifestations and tests of more systematic theories of design based on principles of learnability and understandability.

[30]We do not mean completeness in the Turing sense, but in the sense of treating the main functionalities contemporary aesthetics demand in a general-purpose computational environment. Of course, these aesthetics differ widely, which is one reason we have avoided the subject.

References

[Abelson and diSessa 81]
Abelson, H. and diSessa, A. A.
Turtle Geometry: The Computer as a Medium for Exploring Mathematics.
M.I.T. Press, Cambridge, MA, 1981.

[Baeker 75]
Baeker, R.
Two Systems which Produce Animated Representations of the Execution of Computer Programs.
ACM SIGCSE Bulletin 7(1), Feb., 1975.

[de Kleer and Brown 81]
de Kleer, J. and Brown, J. S.
Mental Models of Physical Mechanisms and their Acquisition.
In Anderson, J. R., editor, *Cognitive Skills and their Acquisition*. Lawrence Erlbaum, Hillsdale, NJ, 1981.

[diSessa 82a]
diSessa, A. A.
Unlearning Aristotelian Physics: A Study of Knowledge-Based Learning.
Cognitive Science 6:37-75, 1982.

[diSessa 82b]
diSessa, A. A.
Phenomenology and the Evolution of Intuition.
In Gentner, D. and Stevens, A., editors, *Mental Models*. Lawrence Erlbaum, Hillsdale, NJ, 1982.

[du Boulay *et al* 81]
du Boulay, B., O'Shea, T. and Monk, J.
The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices.
International Journal of Man-Machine Studies 14:237-250, 1981.

[Gentner and Stevens 82]
Gentner, D. and Stevens, A., editors.
Mental Models.
Lawrence Erlbaum, Hillsdale, NJ, 1982.

[Goldberg and Robson 79]
Goldberg, A. and Robson, D. A.
A Metaphor for User Interface Design.
In *Proceedings of the University of Hawaii Twelfth Annual Symposium on System Sciences.* Honolulu, January, 1979.

[Goldstein and Bobrow 80]
Goldstein, I. P. and Bobrow, D. G.
Extending Object-Oriented Programming in Smalltalk.
In *Proceedings of the Lisp Conference*. Stanford, CA, August, 1980.

[Greenblatt *et al* 80]
Greenblatt, R., Knight, T., Holloway, J. and Moon, D.
A Lisp Machine.
ACM SIGMOD Record 10(4), March, 1980.

[Innocent 82]
Innocent, P. R.
Towards Self-Adaptive Interface Systems.
International Journal of Man-Machine Studies 16:287-299, 1982.

[Lieberman 82]
Lieberman, Henry.
Watching What Your Programs are Doing.
Memo 656, M. I. T. Artificial Intelligence Laboratory, 1982.

[Neves 82]
Neves, D.
Boxer Manual.
M.I.T. Laboratory for Computer Science, 1982.

[Rumelhart and Norman 81]
Rumelhart, D. E. and Norman, D. A.
Analogical Processes in Learning.
In Anderson, J. R., editor, *Cognitive Skills and Their Acquisition*. Lawrence Erlbaum, Hillsdale, NJ, 1981.

[Smith *et al* 82]
Smith, D. C., Irby, C., Kimball, R. and Verplank, B.
Designing the Star User Interface.
Byte :242-282, April, 1982.

[Steele and Sussman 78a]
Steele, G. L. and Sussman, G. J.
The Revised Report on Scheme: A Dialect of Lisp.
Memo 452, M. I. T. Artificial Intelligence Laboratory, January, 1978.

[Steele and Sussman 78b]
Steele, G.L. and Sussman, G. J.
The Art of the Interpreter.
Memo 453, M. I. T. Artificial Intelligence Laboratory, May, 1978.

[Teitelman *et al* 75]
Teitelman, W., *et al*.
Interlisp Reference Manual.
Xerox Palo Alto Research Center, Palo Alto, CA, 1975.

[Tessler 81]
Tessler, Larry.
The Smalltalk Environment.
Byte :90-147, August, 1981.

[Weinreb and Moon 82]
Weinreb, D. and Moon, D.
Lisp Machine Manual.
M.I.T. Artificial Intelligence Laboratory, 1982.

[Young 81]
Young, R. M.
The Machine Inside the Machine: Users' Models of Pocket Calculators.
International Journal of Man-Machine Studies 15:51-85, 1981.

[Young 82]
Young, R. M.
Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices.
In Gentner, D. and Stevens, A., editors, *Mental Models*. Lawrence Erlbaum, Hillsdale, NJ, 1982.