Reference and Data Construction in Boxer[*]

Andrea A. diSessa

Graduate School of Education
University of California
Berkeley, CA 94720

Boxer is an integrated computational environment encompassing a broad range of functionality, from programming to text editing, interactive graphics and data base activity. It is currently in the process of design and implementation at the University of California, Berkeley, and is intended largely for educational computing at all levels, from early elementary school into university. This note describes features from two sets of mechanisms in Boxer: The first specifies how one can refer to computational objects in Boxer, and the second specifies how one constructs compound objects. The common thread is that both of these exemplify important tradeoffs that one must make in designing systems for unsophisticated users, for example, easing the first stages of learning at the cost of formal simplicity of the system as perceived by experts.

[*] This work is supported by the National Science Foundation under grant numbers MDR-85-96025 and MDR-86-42177.

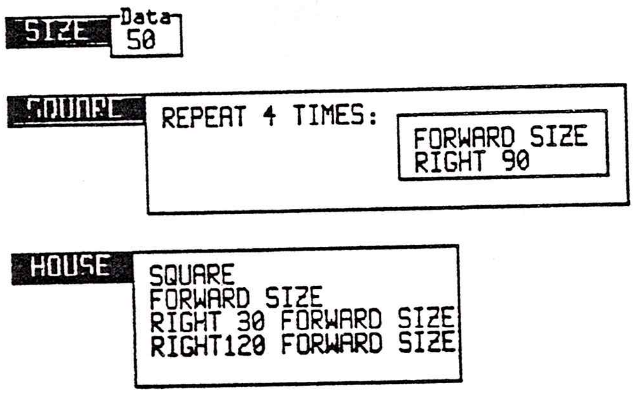
Introduction

In recent years the importance of finding principles that aid in designing easier-to-use computational systems has come to be more and more appreciated. Particularly in educational circles, it is apparent that, while there can be significant benefits from programming, the task of learning that skill is far from trivial. Even such seemingly simple ideas as variable and procedure take surprisingly long to master. Luckily, the increasing availability of more computer power at less expense means that we can consider substantially altering the appearance, structure and means of interacting with systems in addition to looking for better methods of teaching and hoping (vainly) for smarter students.

In this paper we will take a brief look at two pieces of design in Boxer to suggest ways to think about the design of complex systems so as to be maximally understandable and useful for unsophisticated users. In the next section we will briefly describe the fundamental notions behind Boxer that are intended to lead to learnability. These, however, are only to introduce Boxer and have been described in greater depth elsewhere [diSessa 85]. The more particular concern of this paper is a discussion of two of the more specific features of Boxer's structural design that contribute in smaller, but not insignificant ways to understandability. The first of these is the way Boxer handles the problem of referring to computational objects. This involves some straightforward means of using computational objects that are not permitted in many conventional languages because of limitations in their user interface. It also involves some more sophisticated notions that allow a broader range of reference types to be used more simply in Boxer than in many other languages. The second feature of Boxer's structural design that interests us here is the way in which complex data objects are constructed .

Basic Boxer

The most basic notion behind Boxer is to increase understandability by increasing the bandwidth between the user and the computational system. We do this by insisting that (essentially) all the state of the system be directly viewable on the display. Indeed, the user should be able to pretend that what is on the screen is the computational system itself. Additionally, we want the user to have direct, immediate and fine-grained access to adding onto or changing the system. In Boxer, the full capabilities of a text editing system is constantly available to alter the system as one might alter a textual document. Together, visibility and manipulability mark a major shift away from standard "conversational" paradigms of interacting with computational systems where one sees the system only in bits and pieces by requesting part of it to be displayed, and where one alters the system only by sending commands that, invisibly, alter the state of the system.[2]



```
50

REPEAT 4 TIMES:
    FORWARD SIZE
    RIGHT 90

SQUARE
FORWARD SIZE
RIGHT 30 FORWARD SIZE
RIGHT120 FORWARD SIZE
```

Figure 1. Named procedures appear in Boxer simply as boxes with a name tab attached. Variables are similar, but are marked as data.

Figure 1 shows two of the basic structures of Boxer, two small *programs* using Logo turtle graphics commands (HOUSE and the subprocedure, SQUARE, used in HOUSE) and a *variable* (SIZE, which is also used in HOUSE to set the scale of the drawing). These structures are simply typed into Boxer. Just like the alphabetic characters, boxes are created with a single keystroke and thereafter expand as text is typed into them. In contrast to seeing the name of a variable or its value, or the text of a procedure in a definition, what is depicted in Figure 1 are the variable and the programs themselves. Within the conversational paradigm, there is no notation for the *fact* of a variable having a particular value, but only for the *action* of giving it one, or the action of reserving space for one (declaring a variable). The difference between

[2] Compiled languages are usually even worse than "conversational" interpreters. The state of the system is affected only in large chunks, e.g., a complete program, with long turn-around times between user interventions. After the program is compiled, except for watching its side effects, there may exist only very indirect "debugging systems" to make finer grained inspections and changes.

this and Boxer's state orientation is not only in visibility, but also in manipulability. Thus, if one wanted to change the value of the variable, one could simply edit what appears on the screen. If some command or program alters the value of the variable, what appears on the screen will automatically change. Pressing the delete key would cause the variable to disappear and to become undefined. Changing the text in the label of the variable "undefines" the old variable and "defines" the new one, to use conventional conversational terms. With regard to programs, it is simple in Boxer to point to each of the lines in a procedure and to execute them, one at a time, to see their effect. This is the sort of feature that is important to understanding the complex state transitions in the unfolding of procedure execution.

Having such direct, full and fine-grained access to computational objects is at present essentially unknown in computational systems. The conjecture is, of course, that having this kind of access greatly increases the perceived reality and ease of learning about programming.

Besides increased visibility and manipulability, Boxer strives to use spatial arrangement to express important system semantics. The aim is to capitalize on every learner's capability to see, understand and manipulate spatial structures, in order to make computation more "familiar" and easy to learn. For example, boxes are strictly hierarchical; they may appear inside one another but may not overlap. Containment thus makes an adequate representation of the important relation of "part of." If you want some variable to become part of a program, if you want a variable to have some substructure (so that it becomes a record with fields), or if you want a particular procedure to be a subprocedure, you simply put the appropriate pieces inside the desired object.

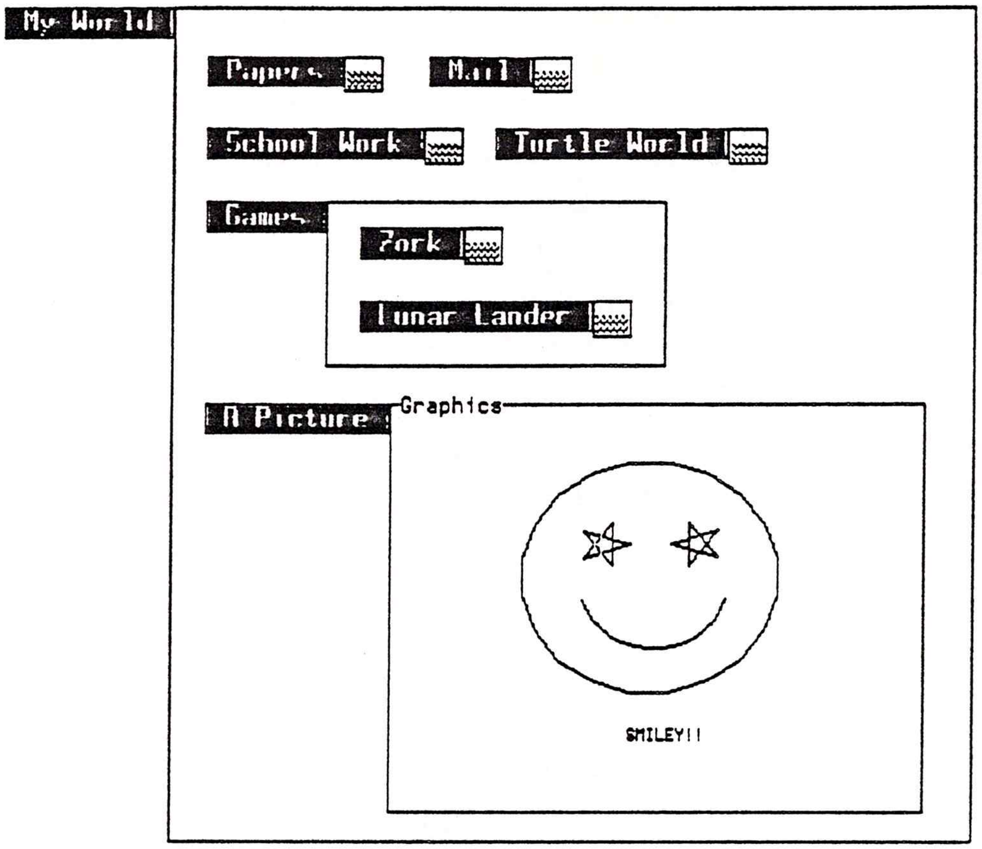
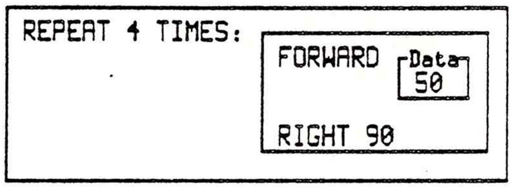


Figure 2. A user's entire environment can be arranged in boxes, and inspected and changed by moving around inside them.

Figure 2 shows an extreme form of the use of this spatial structuring in that a hypothetical user's entire computational world is organized by boxes. If he wishes to view or change the parts of his "papers" box, he merely points to that box and expands it with a keystroke. We defer to other writings [diSessa and Abelson 86, diSessa 86] for the details of these and other uses of spatial arrangement in Boxer to show, for example, how parts of drawings are box-parts of the boxes that represent the drawings.

Reference in Boxer

Many conventional languages never let you get your hands on computational objects themselves. Instead, you create them by name and always refer to them by name. So-called object-oriented languages like Lisp and Smalltalk improve the situation a bit by allowing users to have access to data objects through keeping and passing around pointers to those objects. Yet, even this ability is restricted in that both Smalltalk and older Lisps don't allow one to deal with procedure objects in the same way as data. Boxer's concreteness means that we have no choice but to let users handle all objects in essentially the same way. And, indeed, one can literally pick up and move with editing commands any computational object, data, procedure, graphical object, etc., without having to resort to the intermediary of a pointer. This implies particularly that procedures have more reality as objects in Boxer and are more parallel to data objects, lending a cleaner systematicity than in conventional languages. For example, while many languages do not have any mechanism for creating unnamed procedures, in Boxer, one can have them in the same way one has literal data objects. Visually, all this entails is a box without a name. Figure 3 shows a box that can be used in place of the name SQUARE in HOUSE (Figure 1). This procedure also contains a variable in place, rather than using the named variable SIZE. Figure 4 shows how to change such variables under program control without using names.



```
REPEAT 4 TIMES:
    FORWARD Data
            50
    RIGHT 90
```

Figure 3. The subprocedure (SQUARE) and variable (SIZE) of Figure 1 may be used directly in place rather than having a separate definition referenced by name. Specifically, this box can replace "SQUARE" in HOUSE.

Concrete accessibility of all computational objects in Boxer means it is often simple to create non-standard objects out of standard pieces. For example, one can simply insert a procedure box into a data box and, by doing that, make all of the capabilities of a variable available to the procedure (e.g., the ability to change it simply, and the ability to pass it as an argument into and return it as a value from a function). Even more foreign to most languages, one can package clusters of definitions in data boxes, and thus create and make it possible to pass around entire environments of procedures and data. A standard way of using this capability is to think of a data box containing a set of procedures and data definitions as an "actor" with a set of attributes (local variables) and private capabilities (procedures). In Boxer, one may tell such an actor to exercise its capabilities or return part of the information it holds with a generic command TELL, as in TELL JOE FORWARD 100 (Joe is a turtle), TELL BANK WITHDRAW $100, or TELL BANK BALANCE-OF-ANDY'S-ACCOUNT (this last TELL has more the meaning "ask"). Smalltalk and other message passing languages have this capability by virtue of special features and built-in structure. Boxer has it mainly by virtue of concrete accessibility and the unconstrained combinability of any sort of computational objects.

When a data object is referenced by name in Boxer, the default meaning that we support is "information transfer." That is, a user that types SIZE expects to get the information contained in the variable SIZE. This meaning is supported structurally in declaring that on accessing a variable, the value is copied. (In fact, our implementation does not literally copy in these instances, but operates so as to maintain the illusion for the user that it has done so.) Lisp and most similar languages typically pass information by passing pointers to that information, not copies of it. While that is sometimes useful, and it is certainly efficient, it is frequently confusing to novices who, on changing part of an object, discover they have changed another object which happened to share structure via pointers. Thus, the copying discipline in Boxer is really a modularity principle to simplify the lives of novice computer programmers by preserving the main effect of "information flow" of accessing variables, without the potentially confusing aspects that full-blown sharing via pointers can bring. Without going into detail, Boxer also maintains a standard copying semantics in the use of procedures so that one won't have the untoward possibility that the execution of a procedure might, as a side-effect, change its definition.

For more advanced applications, copying semantics can be limiting. Boxer provides three methods of getting around such limits. In the first instance, TELL, as mentioned above, allows one to reference and change a genuine object, not always simply the information (i.e., a copy of the contents) of an object.[3] More profoundly, we have introduced into the language as an advanced construct the concept of a *port* that directly embodies a "reference to particular computational objects" rather than the default "reference to information." Ports are discussed in the next section, and following that, we look at the final mechanism for advanced reference, flavored inputs.

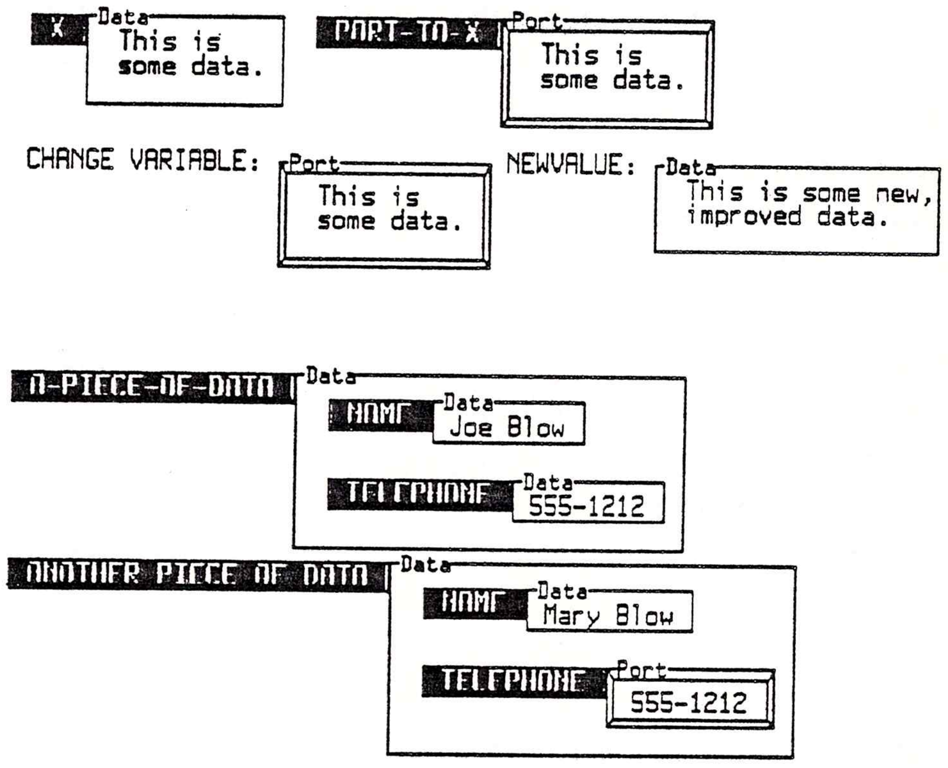
[3] Actually, simply setting a variable is also a limited form of this.

Ports

Ports are basically views of an object from some remote part of the Boxer system. They function in most respects identically to the object viewed by the port. For instance, the ports in Figure 4a view the variable X. Changing X or either of the ports with the editor or under program control changes all of them. For example, either CHANGE X (CHANGE is Boxer's "set the variable" command) or CHANGE PORT-TO-X has the same result, that X and all ports to it are changed. The second line in the figure changes X without using its name, but only a port "reference" to it.

A

B



```
CHANGE VARIABLE: [Port: This is some data.] NEWVALUE: [Data: This is some new, improved data.]
```

Figure 4. a. A variable, X is "viewed" by the port PORT-TO-X. A change in either changes both. Executing the CHANGE statement, which contains another port to X, changes X and the two ports. b. A typical use of ports to share data. The telephone number of Joe and Mary Blow is shared; Mary's telephone data is a port to Joe's.

Ports are a powerful referencing mechanism that can, in fact, replace naming. One may generally use a port to a data object in the place where one would ordinarily use the name of a variable. Indeed, one can use a port to a procedure where one would ordinarily use the procedure name. Hence, one can program entirely without names! There are two reasons why this is not a beginner's mode of operation. First, as already mentioned, ports implement sharing. With shared structure, one must be quite careful about changing objects; other objects may see those changes via ports to the changed object or to parts of it. Logo, for example, deliberately avoided any appearance of sharing. Secondly, we conjecture that operating at the level of text, using names instead of visual reference (ports) to remote objects simplifies the life of the beginner by allowing him to apply his common linguistic sense in producing slightly stiff, but still reasonable approximations of English as a means of programming.

In general, we have tried to maintain the principle that one can always use the name of an object in the place of the object and still get the same effect. This dictates how named ports should operate; they should respect the "reference to an object" semantics (not "reference to some information"). Instead of a copy of the information in the port, one should get another port to the object viewed by the named port. Thus ports exhibit a kind of "stickiness" that propagates throughout the system when they are used, maintaining contact with the original data or procedure. This stickiness extends to ports that are contained in data boxes, so that, for example, any reference to the variable A-PIECE-OF-DATA in Figure 4b will maintain the fact that a sub-part of it, TELEPHONE, will always be the same as the corresponding subpart of ANOTHER-PIECE-OF-DATA, to which it is linked by a port relation. Stickiness is necessary in order to maintain the function of ports as referencers of computational objects, hence sharers that create fully operational links between particular objects in the system.

Flavored Inputs

When one knows in advance what kind of object one is creating, or when one only uses an object in a single way, the choice between making a data object (information reference) or a port (object reference) can support the distinctions one wants. Unfortunately, it is impossible always to know this in advance, and, as well, one often needs to treat objects in different ways according to circumstances.

Boxer handles this by having explicit conversions, i.e., a PORT-TO operator that generates a port to an object, and a COPY operator that makes a copy and thus terminates the stickiness of ports. But in addition, Boxer has a more subtle mechanism that adds context sensitivity to reference type. One can specify the kind of reference for each input of a procedure. We call these "flavored inputs." The default type of input respects the type of reference specified by the object referenced. Data objects get copied, and ports become ports through which the procedure may manipulate the actual referenced object. Alternately, *port flavored inputs* force object reference. Thus, what might be referenced ordinarily as information (data), may be referenced as an object. In fact, the first input to TELL, which specifies the object to talk to, is port flavored, and one may talk to data objects that otherwise provide only copies of themselves. (It usually does no good to "talk to" a copy of an object. The reason is, for example, if you intend to change an object's internal state, talking to a copy of the object doesn't do any good.)

We also have a third type of input flavor that causes the input to be interpreted directly as data. The second input to TELL is of this flavor, and this causes TELL JOE FORWARD 100 to pass Joe the literal message FORWARD 100. A normal input would execute FORWARD 100, "expecting" FORWARD 100 to return some message data that would then be passed to Joe for execution. Note that, after all, TELL is not really an exceptional way to get object reference, but an example of the use of flavored inputs in order to invisibly get the appropriate reference.

The mechanisms described above are not the formally simplest ways of getting the range of functionalities envisaged. For example, it would almost certainly be formally simpler (1) not to distinguish data boxes from procedure boxes at all, (2) to have all named reference occur in the form of object reference (pointer- or port-like reference), (3) to have procedure activation specified by special markings, and (4) also to have copying specified as another special marking.[4] But our judgement is that this would be only a formal simplicity which beginners would not appreciate because of a plethora of initially incomprehensible syntactic marks in common expressions.

To take the example of an existing language, Scheme (a dialect of Lisp) references procedure objects in exactly the same way that it references data objects. But the unfortunate side of this simplicity is that one must use explicit syntax in order to cause a procedure to execute. The Scheme syntax is not complex, merely enclosing an expression in parentheses, but our intention has been to keep explicit structure out of the earliest expressions a user learns. Logo also maintains substantially more visible structure than

[4] In fact, we tried out a variation of this suggestion.

Boxer in simple commands such as "dots" and "quotes" in setting variables, e.g., MAKE "X: X + 1.

Flavored inputs mean that TELL can magically work properly in most instances without the user having to realize that a special kind of reference is needed, and thus having to add special markings. If the user does things much more complicated than stereotypical use of these invisible conventions, he is likely to run into trouble. But that is precisely the time to introduce the more advanced concepts and syntactic marks that explain the normal case and allow for exceptional cases to be handled, rather than to burden the beginner with difficult notions (as reference is) or initially incomprehensible syntactic markings.

We can summarize: Boxer has made an attempt to provide a very low threshold by simplifying the lives of beginners, while at the same time providing some of the important advanced functionality of modern programming languages like object-oriented capability and message passing. It has done this at the expense of a formal simplicity that might provide only general, context-independent mechanisms suitable for the range of functionality wanted. Instead, we chose to slightly complicate the language, from the point of view of an expert who knows it all, so that beginners can, for a long time, avoid thinking about things like reference and the complexities of sharing. The trick is to find slightly more complex structures than the minimal formal ones, but ones that can operate invisibly and correctly as long as beginners stay with simple uses of the commands they learn.

BUILD

This section briefly describes yet another kind of structure in Boxer that was invented for a particular functionality, though more generic structures could well suffice at the expense of additional cognitive load for beginners. The functionality is the construction of compound data objects out of literal or computed parts. In languages like Lisp or Logo, this is handled largely with generic kinds of operations, namely function invocation on parts that are, by default, assumed to be evaluated unless quoted (marked as literals). For example, Logo includes operations called LIST, SENTENCE, FPUT and LPUT that work as follows:

```
    :X is [A B],     :Y is [C D],     :Z is "E
```

then

```
LIST :X :Y  evaluates to  [[A B] [C D]]      SENTENCE :X :Y  evaluates to  [A B C D]
LIST :Z :X  evaluates to  [E [A B]]          SENTENCE :Z :X  evaluates to  [E A B]
LIST "Z :X  evaluates to  [Z [A B]]          SENTENCE "Z :X  evaluates to  [Z A B]
LIST [Z] :X  evaluates to  [[Z] [A B]]       SENTENCE [Z] :X  evaluates to  [[Z] A B]

FPUT :X :Y  evaluates to  [[A B] C D]        LPUT :X :Y  evaluates to  [C D [A B]]
FPUT :Z :X  evaluates to  [E A B]            LPUT :Z :Y  evaluates to  [C D E]
FPUT "Z :X  evaluates to  [Z A B]            LPUT "Z :Y  evaluates to  [C D Z]
FPUT [Z] :X  evaluates to  [[Z] A B]         LPUT [Z] :Y  evaluates to  [C D [Z]]
```

The problems with this paradigm of compound data construction are three-fold. First, the actual construction of the data object is the province of the internals of the constructor function, hence must be understood and imagined as an abstract operation without visual support on the screen. The second problem is actually an extension of this first: What is actually seen may be visually counter to what is produced. For example, LPUT places its <u>first</u> argument at the <u>end</u> of the object it constructs. Compare LPUT :X :Y to FPUT :X :Y above. Finally, construction by function invocation is very difficult for deep structures as evaluation must be propagated explicitly to the lowest levels. For example, if one wanted to produce

[[1 [2 [3 *<the value of X>*]]]],

one would have to type something like

LPUT LPUT LPUT LPUT :X [3] [2] [1] [].

In contrast, Boxer uses a spatially-oriented mechanism in which one makes a template of precisely the form that one wants. The parts that need to be evaluated are marked with a !, meaning evaluate, or @, meaning unbox. The single construction function is BUILD, which causes the ! and @ operations to be carried out. Figure 5 shows expressions in Boxer that carry out the equivalent to some of the Logo expressions above, including the particularly problematic LPUT LPUT LPUT LPUT :X [3] [2] [1] []. Note also that Boxer has only one data marker, a data box, not the " and [] of Logo.

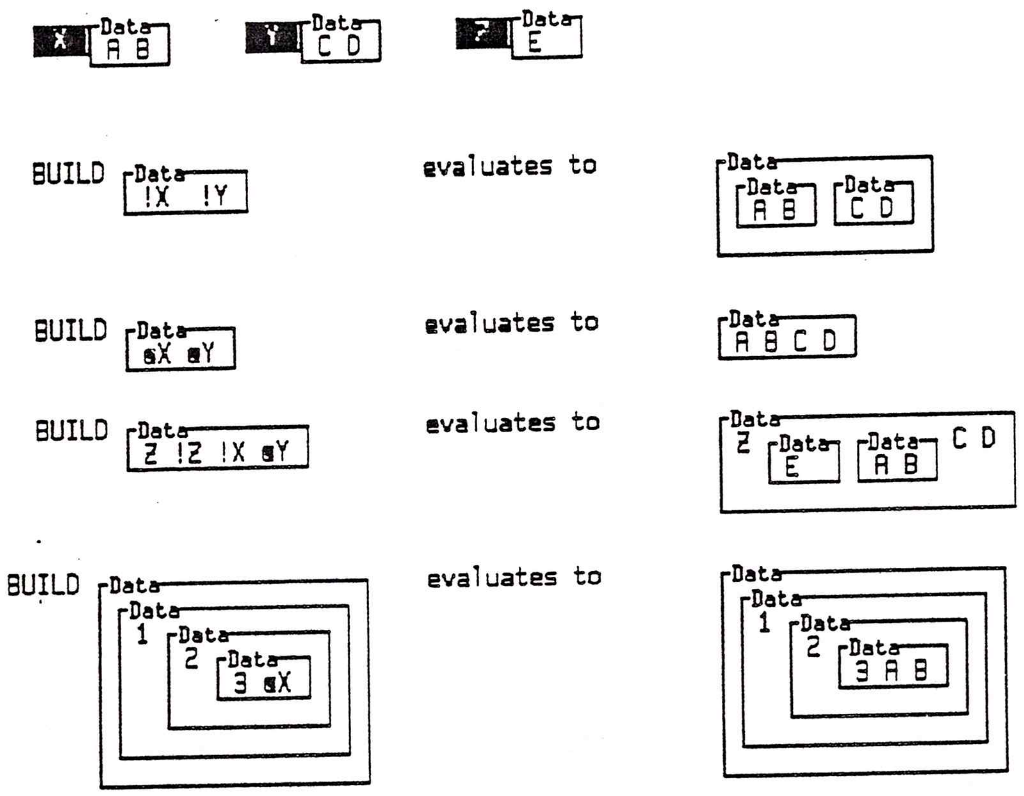

X Data A B
Y Data C D
Z Data E
BUILD Data !X !Y evaluates to Data Data A B Data C D
BUILD Data @X @Y evaluates to Data A B C D
BUILD Data Z !Z !X @Y evaluates to Data Z Data E Data A B C D
BUILD Data Data 1 Data 2 Data 3 @X evaluates to Data Data 1 Data 2 Data 3 A B

Figure 5. Some examples of BUILD.

Finally, we have extended the use of ! and @ to top-level execution. In code to be executed, these markers effect a preliminary pass that constructs a line of commands to be executed. This gives easy-to-visualize methods for doing things such as countering default reference types. For example, in order to compute the name of a box to be told, rather than specifying the name literally, one can use:

TELL @FUNCTION-TO-COMPUTE-THE-NAME-OF-A-BOX <whatever>
(The @ is used instead of ! here since functions return data in data boxes, which is not what TELL expects to see.)

Another example: Suppose in using TELL, one wanted to specify that some part of a message should be evaluated before the message is sent. For example, if one wanted to tell Joe to move forward the global value of X rather than his personal X, one would say:

TELL JOE FORWARD !X

rather than

```
        TELL JOE FORWARD X
```

Conclusion

In this paper I have described two kinds of mechanisms in Boxer and the rationale for their particular forms. I have described mechanisms that provide for a broad range of reference types, but which should not severely complicate the system for beginners who need only the simplest "information transfer" type of reference, or who use only simple cases of "object reference" for message passing (TELL). Indeed, the mechanisms are designed so that early use of these structures is entirely invisible, and just typing a textual command "does the right thing."

I have also described the mechanism we have designed in Boxer to eliminate some of the complexity of constructing compound structures. This also has a small cost to experts in that it introduces non-standard operators, like BUILD, ! and @.

Because of the tension between formal simplicity and providing a seemingly easier learning curve, the design decisions explained above are open to debate and to empirical testing. That testing will not be simple as it must seriously take into account long term use of the system. We must beware touting introductory simplicity that might cause long term difficulties or limitations. We must beware decrying transient incomprehensibilities that quickly evaporate with more experience. Despite such difficulty, the reason for providing our design rationale in papers like this is precisely to open discussion that can clarify and generalize, or, on the contrary, refute our claims with empirical results or better argument. We believe laying open one's design rationale for discussion is one of the best ways to proceed with the task of designing new and improved technological artifacts while at the same time advancing the state of our scientific understanding of principles of simplicity and understandability.

References

[diSessa 85]
diSessa, A. A. A Principled Design for an Integrated Computational Environment, *Human-Computer Interaction*, Vol. 1, No. 1, 1985.

[diSessa 86]
diSessa, A. A. Notes on the Future of Programming: Breaking the Utility Barrier, in *User-Centered Systems Design*, D. Norman and S. Draper (eds.), Lawrence Erlbaum Associates, Hillsdale, NJ: 1986.

[diSessa and Abelson 86]
diSessa, A. A. and Abelson, H. Boxer: An Expressive Reconstructible Medium, *Communications of the ACM*, in press for September, 1986.