

Boxer: The Programming Language

by

Leigh Leach Klotz, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
at the Massachusetts Institute of Technology

January 1989

© Leigh L. Klotz, Jr., 1989
All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
and distribute copies of this thesis document in whole or in part.

Author _____
Department of Electrical Engineering and Computer Science
January 20, 1989

Certified by _____
Harold Abelson
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by _____
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Boxer: The Programming Language

by

Leigh Leach Klotz, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science

January 20, 1989

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

ABSTRACT

Boxer is an integrated computing environment for naïve computer users. It supports “what you see is what you have” both in the editor and in the semantics of the language: objects behave as if they are their screen representations. This constraint and related guiding principles place strong demands on the implementation.

This thesis describes the design and implementation of the programming language for Boxer, with particular emphasis on the efficient solutions to problems imposed by the design constraints. It also discusses the principles underlying the programming language, the resulting programming model, and its evolution.

Boxer has been implemented in Common Lisp on Symbolics 3600, Sun, and Hewlett-Packard computers.

Thesis Supervisor: Harold Abelson

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

The work described in this paper was performed in the Educational Computing Group of the Laboratory for Computer Science at the Massachusetts Institute of Technology, and in the Boxer Group of the Program in Education, Math, Science, and Technology of the School of Education at the University of California, Berkeley.

I would like to thank Prof. Harold Abelson, who has been my advisor during the many years I was a student at M.I.T., and also during the years when I was not. I would also like to thank Prof. Andrea diSessa, with whom I have also worked for those same years, both at M.I.T. and at Berkeley. These two men have taken a profound interest in my education, and have given me the opportunity to participate in their research for my entire undergraduate career. Prof. Gerald J. Sussman has often offered me important guidance, and I wish also to thank him.

My friends and colleagues Patrick Sobalvarro, Laurel Simmons, Deborah Tatar, and Caryl Clark have encouraged and helped me in more than ways than I can express, perhaps in more ways than I can understand.

Marie Crowley, Michael Eisenberg, and Hal Abelson were careful readers of this paper, and each made important contributions.

I would like to thank Ed Lay and Gregor Kiczales with whom I have worked under Profs. Abelson and diSessa to make Boxer a reality.

Outside my family, my friends and teachers Bob Lee, Jonah Ford, and Ben Weathersby contributed more than anyone to the development of curiosity and interests which led me to come to M.I.T. They deserve special, though belated, thanks.

Finally, I would like to thank my parents, Leigh Klotz, Sr. and the late Alyne Klotz, and my sisters, Leah McNeill and Carmen Roberson, for an uncountable number of things.

Contents

Acknowledgements	i
1 Introduction	1
2 What Is Boxer?	2
2.1 A Simple Example	2
2.2 Further Examples	4
3 The Goals and Principles Behind Boxer	13
3.1 Who Is Boxer For?	13
3.2 The Principles of Boxer	14
3.2.1 Boxer Is Integrated	14
3.2.2 Boxer Exploits Spatial Reasoning	14
3.2.3 Boxer is Concrete	15
3.2.4 Boxer Offers Diffused Functionality	16
3.2.5 Boxer Conforms to Naïve Realism	16
3.2.6 All of the Above	17
3.3 The History of Boxer	18
3.3.1 Boxer and Logo	18
3.3.2 Problems in the Logo Model	19
4 The Boxer Model	21
4.1 The Detailed Model	21
4.2 The Movie Stepper	23
4.3 Variable Scoping	23
4.4 Bugs in the Model	25
5 Language Implementation Issues	26
5.1 Implementing Boxer: An Apologia	26
5.2 Boxer and the Interpreter	29
5.2.1 State Machine	29
5.2.2 Stacks	31
5.2.3 Variable Reference	33

5.3	The Interpreter and the Editor	36
5.3.1	Editor Top Level	36
5.3.2	Polling	36
5.3.3	Naming	37
5.3.4	Transparent Boxes	37
5.3.5	Prompting	37
5.3.6	Stepping	38
5.3.7	Triggers	38
6	Implementation Efficiency Issues	40
6.1	Mechanisms for Efficiency	40
6.1.1	Explicit Control Evaluator Data-type Case Analysis System	40
6.1.2	Restriction of Evaluator Data-Types	40
6.2	Use of the Efficiency Mechanisms	41
6.2.1	Copying	41
6.2.2	Flavored Inputs	42
6.2.3	Arithmetic	42
6.2.4	Variable Lookup and Cache	43
7	Data-Manipulation Primitives	46
7.1	BUILD	46
7.2	@	48
7.3	TELL	49
8	Conclusion	51

Chapter 1

Introduction

This paper describes the programming language portion of Boxer, an integrated computing environment for naïve users. Boxer supports “what you see is what you have” both in the editor and in the semantics of the language: objects behave as if they were their screen representations. This constraint and related guiding principles place strong demands on the implementation.

This paper describes the design and implementation of the programming language, with particular emphasis on the efficient solutions to problems imposed by the design constraints. It also discusses the principles underlying the programming language, the resulting programming model, and its evolution.

Chapters Two, Three, and Four comprise a discussion of the language design issues, presenting a view of the Boxer system as a whole, and showing the influence of high-level design principles on the evolution of the language.

In Chapter Five, the emphasis shifts to language implementation issues. Chapter Five gives an overview of the implementation of Boxer, shows how the evaluator which realizes the language fits in with the rest of the Boxer system, and explains features in the evaluator that were dictated by the Boxer design principles. Chapters Six and Seven explore issues of efficiency in the implementation of features described in Chapter Two through Four.

List of Figures

2.1	A Sample Boxer World	3
2.2	The Journal box, expanded.	4
2.3	The CHANGE command alters variables.	5
2.4	Placing the cursor in HERE-TOO and typing alters both port and target.	6
2.5	Examples of PORT-TO, DATAFY and normal input flavors.	7
2.6	Some examples of BUILD.	8
2.7	Examples of the TELL command.	9
2.8	The various uses of the @ primitive.	11
2.9	Automatic calculation program using the modification-trigger feature.	12
2.10	The assets box with its closet shown.	12
4.1	How to execute a line of a Boxer program.	22
4.2	The procedures to be stepped:	24
4.3	Successive frames from the movie stepper:	24
5.1	Roadmap of language design, implementation, and efficiency issues.	27
5.2	Overall Structure of the Boxer System	30
7.1	Examples of BUILD.	47

Chapter 2

What Is Boxer?

Boxer is an integrated computing environment designed for naïve computer users. Boxer was first developed at the MIT Educational Computing Group of the Laboratory for Computer Science, under the direction of Prof. Harold Abelson and Prof. Andrea diSessa. Boxer is currently being developed at the U. C. Berkeley School of Education, in the Education in Math, Science, and Technology program, under the direction of Prof. diSessa.

Boxer's predecessor, or perhaps its godfather, is Logo; yet Boxer differs from Logo in profound ways. Boxer is best explained by example.

2.1 A Simple Example

This section presents a sample Boxer world, containing some programs, some non-programs, and some Logo-like turtle graphics.

The box is the basic unit of Boxer. There are two major kinds of boxes: DATA boxes and DOIT boxes. DATA boxes contain text, graphics, and other boxes. DOIT boxes contain program text and other DATA and DOIT boxes. The text within each box is arrayed in rows, with each character or box occupying one space in the row. Figure 2.1 shows a sample Boxer screen. The initial box, called the Boxer *world*, is a DATA box which occupies the entire screen.

Boxer uses the keyboard and mouse commands to create text and boxes. For example, the second arithmetic expression on the screen was made by typing `12 * (count + 3)`, and the answer was obtained by pressing the *doit* (execute) key. The DOIT box containing `count + 3` was constructed by typing the parentheses, which serve to group the addition expression the box contains as do parentheses in Logo or other programming languages.

Notice that both the whole expression and its result remain in place in the box, and do not scroll away or otherwise disappear. Pressing the *doit* key again will execute the same command once more.

The DATA box named `count` is a variable, whose value is 49.

The GRAPHICS is a special kind of DATA box that displays graphics information in addition to text.

WORLD

This world contains

- Some explanations
DATA
- Some data
DATA
- A graphics box
DATA
- A short program
DATA
- some named, shrunken boxes
DATA

DATA

count
49
DATA

12 * count | 588
DATA

12 * count + 3 | 624
DATA

Graphics


square
input side
repeat 4 forward side right 90

square 10 |

Microworlds
DATA
DATA

Journal
DATA
DATA

Thesis
DATA
DATA

Figure 2.1: A Sample Boxer World

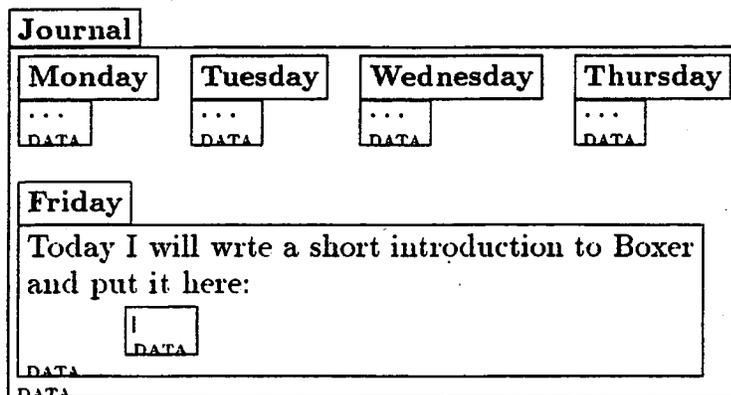


Figure 2.2: The Journal box, expanded.

Below the GRAPHICS box is a set of Boxer commands grouped together inside a DOIT box and given a name; in other words, it is a procedure named `square`. The row below the DOIT box contains an invocation of the procedure. A Boxer user has just executed the commands on that row by placing the cursor there and pressing the *doit* key. The `square` procedure has made the turtle in the graphics box draw a square on the screen, as a similar procedure would in Logo.

Below the call to the turtle graphics procedure are three shrunken boxes. Each might be a complicated world in its own right, containing many programs, much text and data, and perhaps a number of similarly shrunken boxes constituting further divisions.

Let's expand the box named `Journal` to the full screen size and examine its contents, as shown in Figure 2.2.

The journal contains a number of entries, each in a separate DATA box. All of the entries are shrunken, except for one. Notice that the entry box just surrounds the text and an empty DATA box. Placing the cursor inside the empty box and typing will cause both that box and the surrounding box to expand in size automatically. Pressing the right parenthesis key moves the cursor out of the box and back into the "Friday" box. Pressing it again places the cursor just after the entry box, and pressing the parenthesis once more will exit the `Journal` box and return it to its former, shrunken size. The screen will once again look like Figure 2.1, with the cursor after the `Journal` box.

These static pictures cannot truly convey what it is like to use Boxer; the reader without access to Boxer is referred to the video-tape presentation [25].

2.2 Further Examples

This section provides examples of some of Boxer's features whose design and implementation is discussed elsewhere in this thesis. It is not intended to be a complete introduction to Boxer. The interested reader is referred to [10], [23], and the video-tape presentation [25] for an introduction to other parts of Boxer.

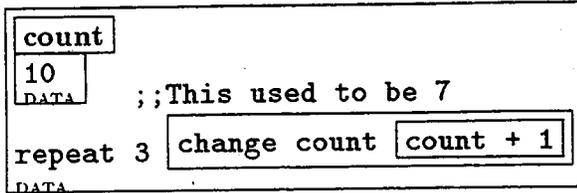


Figure 2.3: The CHANGE command alters variables.

Programs and Variables

Boxer has a simple rule for program evaluation, embodied in the phrase “copy and execute.” The expression `12 * count` which appears near the top of Figure 2.1 is a short Boxer program. In this case, the meaning of “execute” is fairly clear: the normal rules of arithmetic apply.

The treatment of the name `count` requires further explanation. It is here that the “copy” comes in. The box named `count` is just that, a box named “count.” In other words, it is a variable. The contents of the variable are the contents of the box. This means that variables in Boxer are named DATA boxes. When the Boxer evaluator sees a word that is the name of a variable in an expression, it finds the named box, makes a copy of it, and replaces the word with the new box. The primitive commands are invoked when all their inputs have been reduced to DATA boxes.

Finding a named box means starting outwards from the expression being executed and looking for a box with the appropriate name. Named boxes, whether they are variables or procedures, are accessible by name inside the box where the named box resides, and inside any boxes inside that box.

CHANGE

The contents of variables may be changed with the CHANGE command. The CHANGE command takes two arguments: a box to change, and the new contents. It replaces the contents of the box with a copy of the new contents.

As Figure 2.3 shows, the first argument to the CHANGE command may be a variable. By the Boxer evaluation rules, the named box should be copied before the CHANGE operation begins; hence it would appear to be impossible to mutate any data under program control! How can CHANGE possibly work? The answer is that CHANGE uses *port-flavored* inputs. Two somewhat advanced Boxer concepts must be introduced to explain how CHANGE works: *ports*, and *flavored inputs*. However, to use CHANGE, one need only see that it does indeed work.

Ports

The *port* in Boxer is a special kind of box. It is a window or door to the contents of another box. Ports provide for arbitrary connections to supplement the Boxer hierarchy. Thus, ports are used to connect disparate sections of Boxer. For example, we might add to the

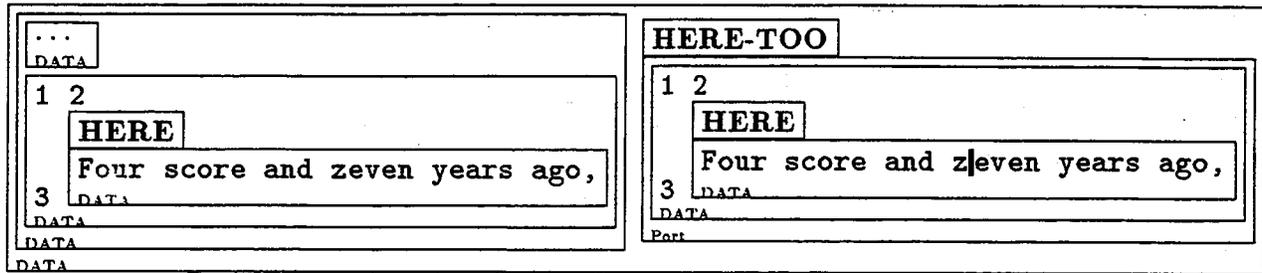


Figure 2.4: Placing the cursor in HERE-TOO and typing alters both port and target.

Journal box a box that would contain a series of ports to different journal entry boxes, organized in alphabetical order by subject. This new box would then form an index to the journal.

Figure 2.4 shows a box and a port to that box. The black rectangle after the “z” represents the cursor. The user has placed the cursor in the port, and has changed the “s” in “seven” to a “z.” The change in the port is immediately reflected in the original box.

Similarly, altering a port box under program control makes changes to the original box. It is this ability which allows the CHANGE primitive to work. Ports can be made under program control, using the PORT-TO command.

That the first input to CHANGE must be a port is only part of the answer to the puzzle. The user does not normally use the PORT-TO command in conjunction with CHANGE; Boxer creates the port automatically because the first input to CHANGE is *port flavored*.

Flavored Inputs

As described on page 5, the Boxer evaluator normally makes copies of Boxes that are destined to become inputs to procedures and primitives. Boxer provides a mechanism for giving inputs a certain “flavor,” to indicate that the corresponding boxes should be handled differently. When gathering a port-flavored input, the Boxer evaluator makes a port to the box, instead of making a copy.

Figure 2.5 shows the ONE-PLUS procedure. Its input is not port flavored, and so calling the procedure does not change the box, even though the ONE-PLUS procedure itself calls CHANGE. There is no lasting effect because the input variable of ONE-PLUS is a *copy* of the original box.

The similar INCREMENT takes a port-flavored input. When it alters the input with the CHANGE command, the original box is affected because the local variable in the INCREMENT procedure is a port to the original box, not a copy of it.

The simple TEST procedure in the Figure shows the DATAFY input flavor, which allows words or DOIT boxes to be put in DATA boxes and passed as parameters to procedures. The @ primitive (see Section 2.2) is the complementary de-reference operation, allowing use of the word or DOIT box. In the figure, DATAFY is used simply to allow an unboxed

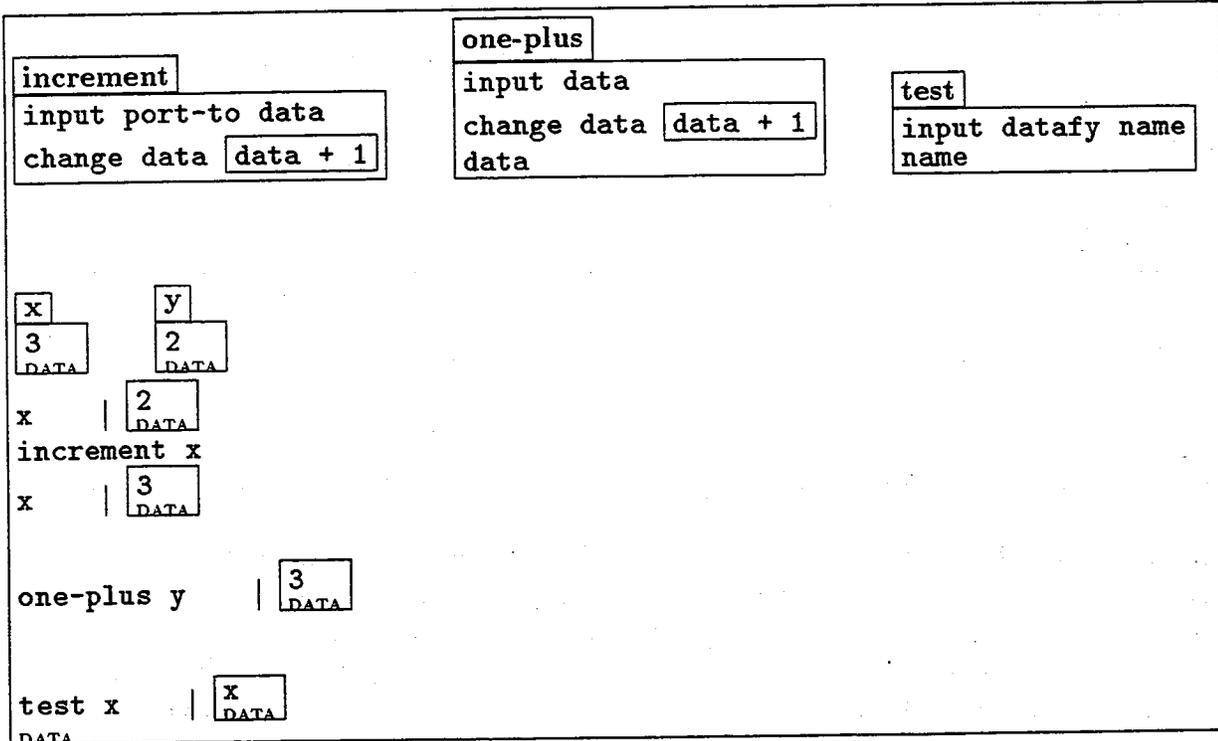


Figure 2.5: Examples of PORT-TO, DATAFY and normal input flavors.

keyword-style argument.

BUILD

Boxer provides a rich set of primitive commands for operating on DATA boxes (and ports), including data predicates, selectors, mutators, and constructors. The standard Logo-like operations on lists and arrays are generalized to operate on boxes, and are not described here.

Unique to Boxer is a primitive which exploits the spatial metaphor for constructing boxes. The BUILD command is a template-based constructor similar to the Lisp “back-quote” operator. BUILD takes one input (a template DATA box) and outputs a DATA box which looks largely like a copy of the template, with the exception of expressions indicated by the special characters “!” and “@.”

When BUILD encounters an expression beginning with the ! character, it evaluates the expression and uses the resulting box in place of the original expression in constructing the BUILD output box. We refer to this character as the *doit* character.

The effect of the @ character is like that of the ! character, except that the *contents* of the resulting box are taken out of the box and inserted into the BUILD output box; we say that the result of the expression is “un-boxed.”

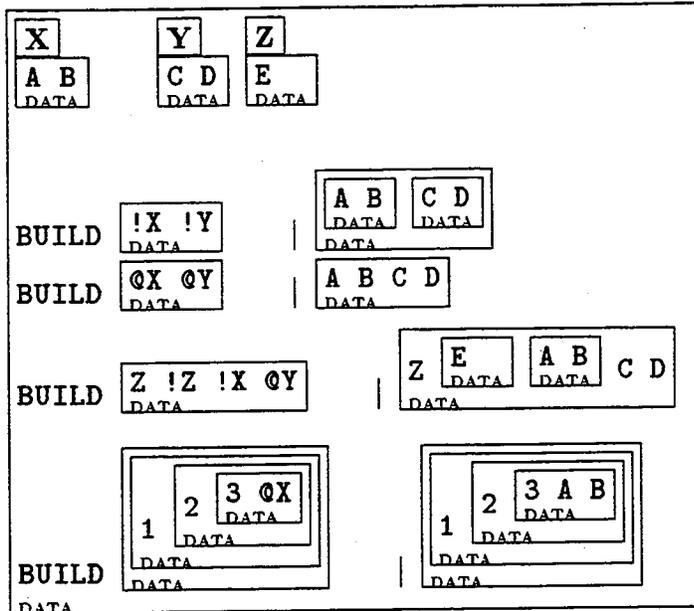


Figure 2.6: Some examples of BUILD.

Figure 2.6 shows some simple examples of BUILD.¹

TELL

In addition to ports, Boxer provides another means for non-hierarchical access to data. The TELL primitive evaluates expressions inside distant boxes.

As Figure 2.7 shows, the result of TELL BOX1 CHANGE COUNT 3 is the same as that of moving the cursor to the box BOX1 and evaluating the expression CHANGE COUNT 3. The expression may also return a value; TELL can thus be used for accessing variables inside other boxes.

Since TELL causes a change in the context in which expressions are evaluated, it may not be possible for the distant expression to access the variables which are available to the procedure which calls TELL.

The second argument to TELL is considered to be "build-flavored," so the ! character may be used inside the second argument to indicate expressions which are to be evaluated in the local environment, before the TELL operation takes place. Variable names not preceded by an ! character are evaluated in the distant environment. In the figure, the value of the amount variable is taken from the deposit procedure.

¹This example is taken from [11].

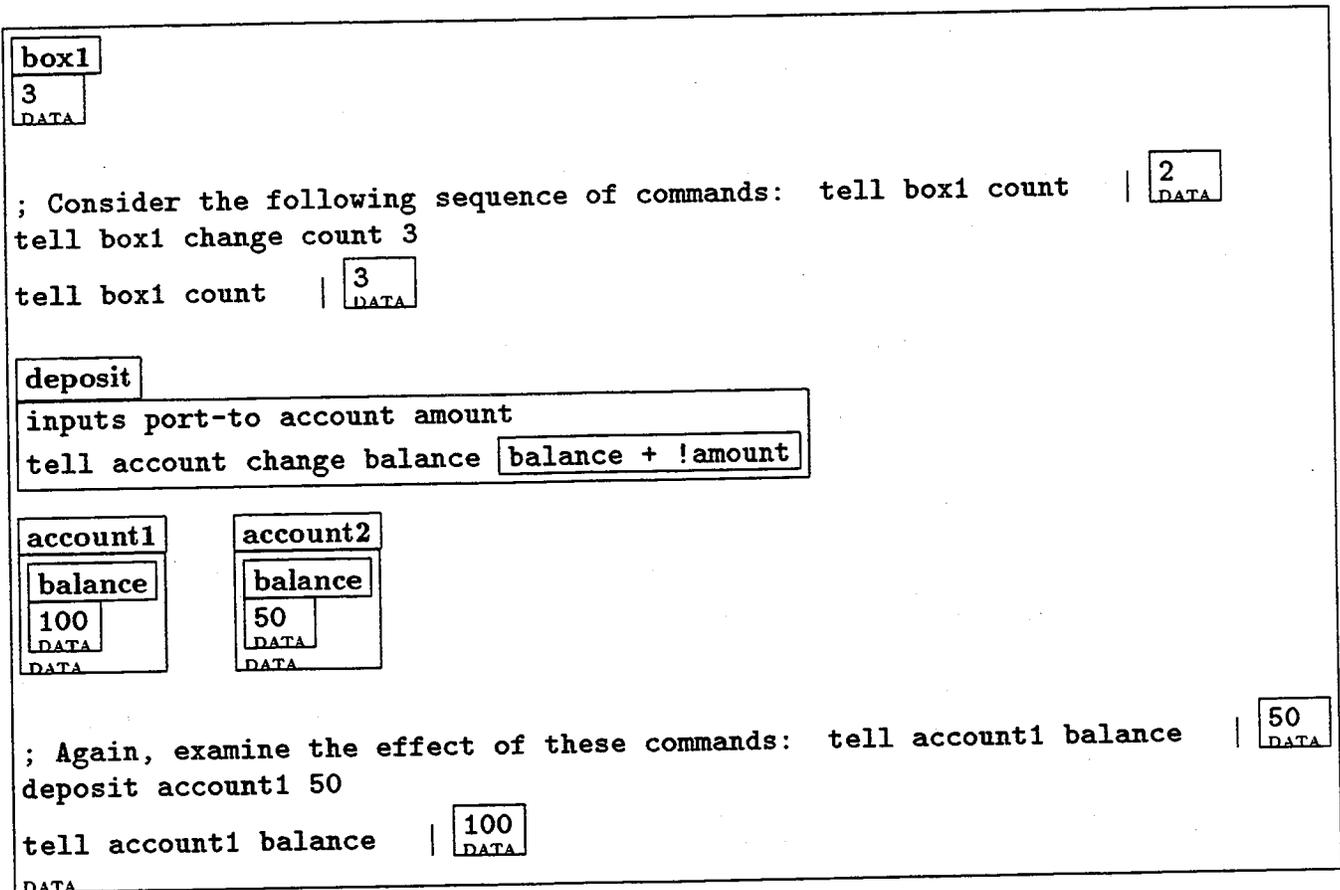


Figure 2.7: Examples of the TELL command.

@

The @ primitive is an appropriation of the @ character from the BUILD primitive. The difference in the two uses of @ is that, as a primitive command, @ may appear directly in lines of Boxer code. When Boxer encounters an @ character outside of a BUILD template, it evaluates the following expression, removes the contents from the resulting box, and places those contents on the line in place of the @ and its following expression, to be executed once more.

If the expression following the @ evaluates to a DATA box containing exactly one DATA box, then the effect of the @ operation is to open up one level of box hierarchy (since DATA boxes are self-evaluating). If the expression evaluates to a DATA box containing an expression, then that expression will be evaluated.

If the expression instead evaluates to a DATA box containing only part of an expression, then the remainder of the line will be used to complete the expression. Thus the @ character can be used to access procedures (DOIT boxes) which have been passed as arguments (inside DATA boxes).

Figure 2.8 shows some examples of these uses of the @ primitive.

Triggers

A *trigger* is a DOIT box which is associated with a box and which is run when a certain thing happens to that box. There are three kinds of triggers: modification triggers, entry triggers, and exit triggers. The modification trigger is run whenever the box is changed directly by typing, or by a procedure. Entry and exit triggers, if present, are run whenever the cursor enters or leaves a box.

Trigger procedures are DOIT boxes with special names. For example, the exit trigger of a box is a DOIT box named *exit-trigger*. Triggers are generally placed in the box *closet*, which is a hidden part of a box. The CHANGE operation does not affect items in the box closet, and the items are not normally displayed. A special key opens the closet and makes its contents visible.

One typical use of the entry and exit triggers is making an object in a GRAPHICS box visible only while the cursor is in a particular box. The *modification-trigger* mechanism is useful for enforcing constraints on the contents of several boxes. For example, consider the Figure 2.2, which shows a spreadsheet-like program written in one line of Boxer code. The code is the user formula, and is run when the contents of the boxes change.

This simple example is not a true spreadsheet, since there is a distinction between inputs and outputs, and changing the outputs has no effect. However, a program which did use constraint propagation to perform the calculations would have the same trigger structure.

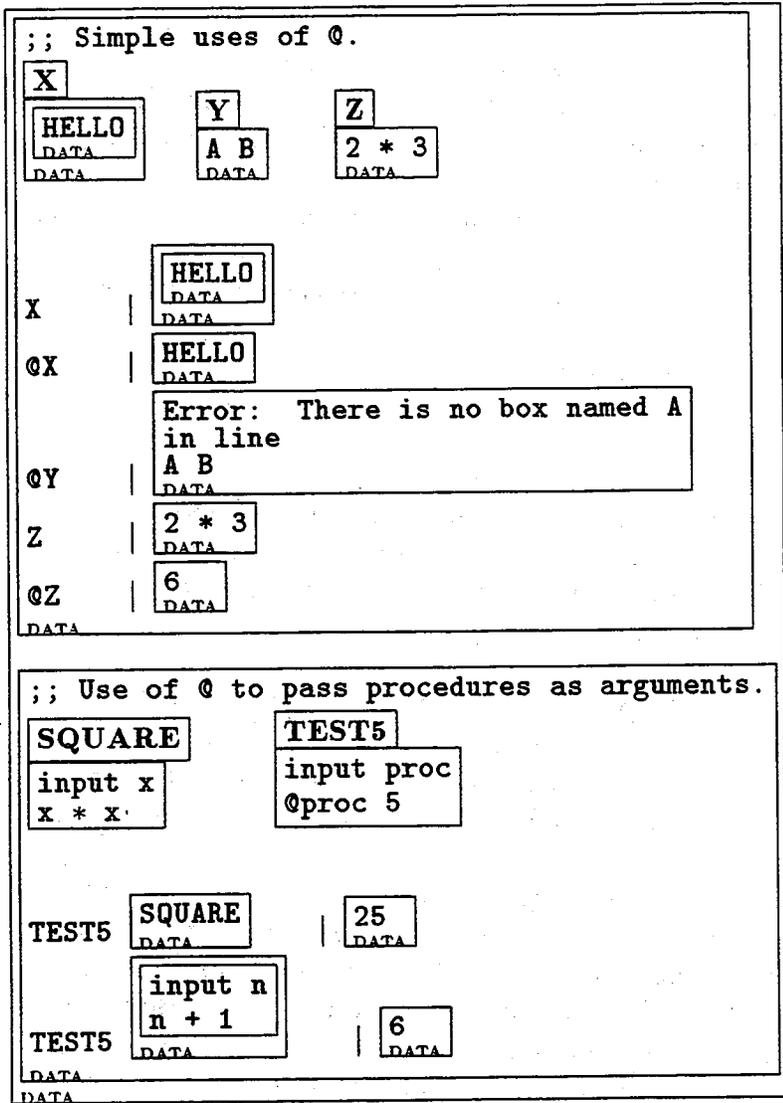


Figure 2.8: The various uses of the @ primitive.

Figure 2.9: Automatic calculation program using the modification-trigger feature.

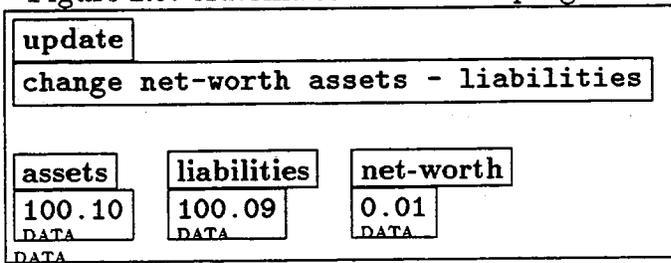
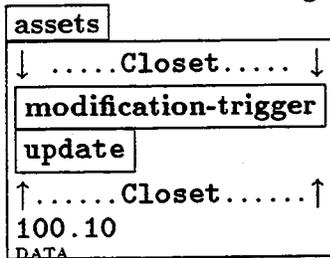


Figure 2.10: The assets box with its closet shown.



Chapter 3

The Goals and Principles Behind Boxer

The previous chapter presented Boxer from the user's point of view. This chapter discusses the users for whom Boxer was designed, and how they are expected to use Boxer. It describes the goals and principles which fostered the design and implementation of Boxer.

3.1 Who Is Boxer For?

Boxer is for naïve computer users. Whereas Logo was targeted at users of elementary school age and older, Boxer was designed with the assumption that the majority of people who could profitably use it would be at least ten to twelve years old.

Naïve computer users are people who are not programmers, but who need to use a computer with more processing capability than word-processing and graphics-design programs provide. Boxer users are envisioned as taking advantage of existing bodies of Boxer code, but still able to understand the internal workings of the tools they use enough to modify these tools or create new ones.

Outside of education, the typical Boxer user is seen as someone who needs to organize and store information, annotate it, operate on it, and retrieve it. Surely Boxer is more friendly and easier to use than the combination of the Unix file system, shell scripts, and `/usr/bin/vi` that even normal people are subjected to these days in the name of progress.

In the educational sphere, Boxer is used as a medium for *microworlds*.¹ Because Boxer can easily mimic the appearance of the printed page, it is sometimes helpful to think of Boxer microworlds as “interactive workbooks” — that is, explanatory text, suggestions for projects, and pictures. The difference is that the text, the suggestions, and the pictures might all be programs or under program control. In Boxer, most such programs are simple, easy to understand, and easy for users to modify.

¹See [1] for an introduction to the microworld concept.

The ease with which students are able to use and adapt Boxer programs was demonstrated in the summer of 1988, when a group of twelve- to fourteen-year-old students participating in a U. C. Berkeley summer program used a Boxer microworld on statistical sampling as an adjunct to their course on statistics. The experimenters found that students were able to use Boxer as a reconstructable medium, making a series of gradual reconstructions in refining their work. Furthermore, they were able to utilize the tool-rich environment by modifying existing tools and creating their own tools from scratch.[6]

3.2 The Principles of Boxer

Boxer is an *integrated, concrete* system that exploits *spatial reasoning* to provide an easy-to-understand computational resource for naïve computer users.

3.2.1 Boxer Is Integrated

Boxer is not a programming language coupled with a variety of satellite systems for user interaction. Rather, it is a unified system with a single conceptual model spanning both the programming language and the environment. Unlike conventional languages, Boxer has no separate editor; the user can edit any procedure or data object visible on the screen. Debugging and error handling are also subsumed by the same framework that supports the programming language. The role normally played by a file system is subsumed by the Boxer editor hierarchy itself. The Boxer user need not learn a series of different commands or interaction styles for different modes or sub-systems.

3.2.2 Boxer Exploits Spatial Reasoning

The sense of space and depth inherent in a visual, two-and-a-half dimensional system like Boxer makes the use of physical metaphors more direct and inviting than the text-oriented interfaces of conventional programming languages.

The hierarchical organization inherent² in Boxer makes the organization of one's world into functional units natural. DOIT boxes visually delineate the procedures or expressions they contain. Subroutines are represented as DOIT boxes within other DOIT boxes. DATA boxes can be used to group other boxes into coherent units, expressing relationships in subject to form programs or microworlds, or expressing relationships in functionality within a program to make toolboxes or subroutine packages. Boxer represents these concepts in an intuitive, spatial way.

Moving from one box to another is like moving around in a physical space. People often worry that users will get lost inside Boxer worlds, and suggest that some sort of bird's-eye view map is necessary.³ Initially, Boxer had such a map; but people did not use it. In

²Heterarchies are also supported via *ports*. See page 5.

³In particular, Hypercard users seem to make this comment on first seeing Boxer.

general, Boxer users do not get lost. One reason might be that Boxer shows context at both higher and lower levels of detail, and makes it easy to pop up one or two levels to look around if necessary. Systems that do not spatially represent the connection between levels of information are difficult to navigate. Of eighteen systems supporting hypertext capability recently reviewed[7], Boxer's presentation was found to be the one which most naturally expressed hierarchical connections.

3.2.3 Boxer is Concrete

In conventional programming languages, and even in modern, interactive programming languages like Scheme, procedures are at best "abstract beings which inhabit the computer." [8] When a programmer edits a procedure, s/he does so in an editor, making changes not to the procedure, but to the printed representation of a procedure, which must be then compiled or loaded. The programmer deals only indirectly with abstract objects, through a computerized glove box. Data are treated in much the same fashion, though data structures are frequently not at all editable, and can be known only through some procedural abstraction.

In Boxer, every procedure and every data object is presented as if it actually existed on the screen. There is a one-to-one correspondence between objects in the Boxer world and objects on the screen. One consequence of Boxer being concrete is that one can change the value of a variable or the definition of a procedure merely by placing the cursor inside a box and typing. Conversely, changing the contents of a variable with a Boxer program causes the new contents to be displayed on the screen. Users can manipulate the procedures and data that fill the computer's memory directly, in a concrete way

This concept is sometimes difficult for users of conventional programming languages to grasp at an intuitive level. On the other hand, naïve users of computers *expect* computers to operate this way. After all, (outside of television) the physical world works the same way: if you can see something in front of you, then the thing you see *is* that something; and if you can touch it, you can also change it.

Variables in Boxer are also different from those of other languages. In Lisp-like languages, variables are associations between names and objects, and are best thought of as cells containing pointers; two variables may, for example, point to the same object. In Boxer, variables are simply boxes with names attached. The name is the variable name, and the contents of the box form the value. Rather than having a Lisp-like SET! or SETQ operator which alters the value pointer of a variable, Boxer offers direct mutation of box contents as its primitive assignment operator. The CHANGE primitive replaces the contents of one box with a copy of the contents of a second.⁴ The box that represents the variable remains the same box; only its contents are replaced by the new contents. This model of

⁴Boxer's concept of variable is more like that of FORTRAN than that of Lisp: the data are actually in the variable. As in FORTRAN assignment, it is the contents of variables that are changed. Indeed, in FORTRAN the contents of *constants* may be changed! This is quite Boxer-like, although in Boxer only copies of the constants are changed.

variables fits well with the naïve, physical understanding of variables as labeled boxes that contain their values.

Boxer also represents variable scoping in a visual, intuitive way. An expression may refer to variables defined in the box that contains it, or to those defined in a superior box. Thus, boxes (both DATA and DOIT), represent environment boundaries.

3.2.4 Boxer Offers Diffused Functionality

Related to the ideas of concreteness and integration are those of *tunability* and *diffused functionality*. DiSessa cites *diffused functionality* as the idea of one construct serving many functions. The box itself is a prime example: already we have seen boxes used to represent data and procedures, to form environments for variable binding, to group text and other boxes, and to act as parenthesized expressions. Boxer uses the box to all these ends in a consistent, natural way.⁵

The design of the programming language for Boxer, also, has been guided by the idea of diffused functionality and the related idea of *detuning*. As diSessa puts it,

Detuning means having general structures underlying the computational environment that are broadly applicable, less highly tuned to any specific function, and always available for use.[10]

The *Waring Bar Blender*, a consumer product,⁶ provides a more physical example of detuning. Most blenders made in the United States feature a long row of buttons, each corresponding to a different motor speed. The buttons are labeled with words, beginning with “stir,” proceeding through “blend,” and ending with “liquify.” A separate button, labeled “pulse,” offers pulse control of the motor. Blender users, however, ignore most of the speeds, and favor simply turning the blender on and off over using the pulse button. The Waring blender, by contrast, has a single toggle switch with three settings: off, high, and low. The blender, instead of offering a multitude of confusing options, presents a uniform interface which subsumes the functionality of other systems.

3.2.5 Boxer Conforms to Naïve Realism

People expect the new things they encounter to be like old things, only a little different. Computers, or more precisely, the abstractions that computers present, are frequently unlike old things in surprising and complicated ways. Without a doubt, some of the power of computers comes from this abstractness, and from their difference from things in the physical world. Even so, it should be possible to make this power available through some mechanism which more closely approaches the physical world and the assumptions people have about it.

⁵Michael Eisenberg offers an excellent discussion of these issues in his section on Boxer in his master's thesis on Boxer.[9]

⁶Waring, Inc., 475 Steamboat Road, Greenwich, Conn.

Boxer attempts to conform to models that people already have about computers. The *copy-and-execute* model of computation is chosen for Boxer because of its simple, unifying nature. In its barest form, it offers a two-step process for executing Boxer procedures: On encountering the name of a DOIT box, replace it with a copy of the box, and begin executing the contents of the procedure. Similarly, to evaluate a name which specifies a DATA box, replace the name with a copy of the box.

As the Boxer user progresses, s/he encounters more advanced features of Boxer that are not explained in this initial subset of the model. We have taken care that these concepts are not necessary in the early stages of Boxer use; nevertheless, they are explained in the full Boxer model. A more complete, yet still intuitive, explanation of the Boxer model of evaluation can be found in Chapter 4.

3.2.6 All of the Above

Boxer, being integrated, is a synergistic whole. The naïve realism of the programming model is inter-related in design and implementation with the aspects grouped under “concreteness.” It is difficult to discuss these principles in isolation of each another, and difficult to ascribe a particular feature of Boxer to just one of them.

Boxer attempts to make computers available to people. In Boxer, “simple things should be easy; hard things should be possible.”⁷ Many things one writes programs for in other languages can be accomplished in Boxer with little or even no programming, because so much of what people want to do with computers is taken care of by Boxer’s integration.

In Boxer, it is simple to input, organize, manipulate, and display information. In this sense, Boxer is like the Alto computer of Xerox PARC, whose designers recognized that the most important tasks of a personal computer are the input and display of information. In an strategy unheard of at the time, the Alto devoted over half of its processor time to the display.[21] However, the designers of Boxer have also taken their Logo heritage and placed importance on the manipulation of that information under program control.

A loose categorization would place manipulation in the hands of the Boxer language itself, though certainly the rest of the system plays a role. Similarly, the language must support (or at least operate in consonance with) the input, output, and in particular organizational principles of the Boxer system.

While it would be possible to make an editor-only version of Boxer, to do so would be to miss the point of the system as a whole. The programming language itself comprises less than 20in Boxer, yet it is precisely the presence of the programming capability which makes Boxer an reconstructable, interactive medium.

⁷diSessa, personal communication.

3.3 The History of Boxer

Boxer has had a unique development history. It was not hammered out in a short series of intense committee meetings, nor was it an amalgam of the best (or worst) features of already existing languages. Instead, Boxer came about incrementally, through a cycle of design, discussion, implementation, and testing.

Boxer and Logo share the incremental design-and-test development cycle. Some of the ideas for Boxer were formed during the development of Logo itself, or arose from thinking about problems in Logo. My work on implementing Boxer is directly related to my previous experience at the M.I.T. Logo Laboratory. In this section, I present the history of Boxer as I have seen it.

The idea for Boxer is due to Profs. diSessa and Abelson. The work described in this thesis has all taken place under their guidance, and in the framework which they established. They formed and headed the Educational Computing Group, in the Laboratory for Computer Science, at M.I.T., where the Boxer project was centered from 1981 through 1985. Since 1985 the Boxer project has been a part of the U. C. Berkeley School of Education, Division of Education in Math, Science and Technology.

3.3.1 Boxer and Logo

Over the years, the designers of Logo expended much effort on the programming language, with less emphasis on the programming environment. Although the editor, interaction system, file system and graphics were always close to state-of-the-art for research computers, their design and their integration with the model of computation was not given the priority the language itself enjoyed.

With the development of MIT Logo for the Apple II and TI Logo, Logo received its first wide exposure. Problems in understanding and using Logo became prominent. As part of the team that implemented these last versions of Logo at MIT, I became concerned about some of these issues.

Logo offers little in the way of organizing *programs*. Logo programs much longer than ten or twelve procedures became complicated to deal with, because Logo offers no ability to group procedures textually. The “bag of procedures” methodology is the only one available. Within the procedure, Logo makes no provision for formatting lines of code. Logo offers only *lists* and *words* as data structures. Harvard’s PPL, which supported record types, seemed to have something to offer.[12]

In 1981, I began working on a specification for a new Logo, one that would solve these problems. My colleague Patrick Sobalvarro and I co-authored an internal Logo paper on a next-generation Logo to run on the new Motorola MC68000 architecture. The Logo system was to use my new Logo specification, and to offer an editing system integrated with Logo, along the lines of the MIT Lisp Machine CADR⁸ system. In short, we proposed to build a Logo machine.

⁸The CADR system integrates the programming language and the operating system. See [26].

When I presented the first of my ideas on formatting to Abelson, he told me that he and diSessa had been developing a concept for a completely new language called "Boxer", in which procedures and data would be unified and represented on the screen as boxes.

Abelson and diSessa envisioned a system that would be designed *a priori* with a simple, understandable model computation, which would appeal to spatial reasoning as a mode of understanding. The emphasis was on ease of use and understanding, with less importance placed on efficiency or difficulty of implementation. To this end, the authors proposed a system which integrated the editing, filing, interaction, and computation systems under a single model.

3.3.2 Problems in the Logo Model

In using Logo, children initially treat the language as if were English. The Logo expression `FORWARD 100` seems acceptably near the English sentence "Go forward 100 steps." Logo procedures were designed to take advantage of the similarity in English between the infinitive form of verbs used in definitions with the `TO` primitive and the imperative form used in programs. Procedures, if introduced at this stage, are seen as recipes, or some other convenient analogy. Young[13] refers to this kind of understanding as a *strong analogy*, and gives the example of a computer terminal being "like" a typewriter. Such models may make computers seem more familiar, but are of limited use in terms of actually using computer systems.

Young further discusses *surrogate models*, which he explains as simplified, mechanistic accounts of systems which are fairly accurate "cover stories." They need not explain every detail of a system, but they should allow the user to understand and predict system behavior. The Scheme substitution model[8] is an example of such a surrogate model.

Surrogate models are usually too complicated to consult for every decision, and novices often employ a more simple kind of model in the actual use of a system. Young calls these models "task-action models." These models explain only a small part of a system, by providing a mapping from what the user wants to do (the task) to what the user must do to the system (the action).⁹

Logo users start with a simple analogy or model: Logo is like English. The standard Logo route to understanding is a series of successively more nearly correct surrogate models. There is, however, no single "Logo model," other than the Lisp-based implementation.

At the Logo Laboratory, Seymour Papert and Cynthia Solomon developed a model of Logo called "Little Men Lines," in which procedures are seen to be little people, with one person per procedure. Procedure invocation is seen as the "cloning" of a new procedure man, who receives things (the inputs) from, and returns something (the output) to his caller.

⁹Much to the consternation of Logo designers, some young children, while failing to understand more useful parts of Logo, revel in certain obscurities, such as the syntax of the print command, or the numbers that represent particular colors. Perhaps their delight is in the mastery of collections of Young's task-action models.

Although the “Little Men Lines” surrogate model seems to help explain recursive procedures to children who can already use procedures, it does not help in other areas, such as understanding expressions that make complicated use of variables. Logo users adopt task-action models for dealing with such problems. For example, Logo uses “dots” (colon) to distinguish a reference to the value of variable from a call to a procedure of the same name. Not understanding the idea of reference, Logo users frequently interpret the dots character to mean simply “variable.” Although this task-action model works at one level, it does not explain multiple evaluation of variables, or the use of variable values as variable or procedure names.

Indeed, the only model of Logo that can explain everything that Logo does is the real, Lisp-like implementation. Unfortunately for the user, Logo at this level is even more complicated than most Lisps. Logo code is parsed, using precedences and parsing rules that are seldom documented. Logo only imperfectly hides some of its implementation details. Even the moderately advanced Logo programmer runs into problems that can only be solved — or at least understood — in terms of the details of the underlying implementation, and this is not subject to any cognitive model.¹⁰

Models and analogies can generally help students learn how to write procedures in Logo. For example, procedures with arguments (*inputs* in Logo parlance) are generally introduced by analogy to primitive commands with inputs. Introducing procedures that return values (*output*) is usually done with some sort of pictorial explanation, with input hoppers and output chutes. Global variables are seen as boxes with labeled tops.¹¹ Many children frequently do not master the use of inputs, and even fewer use outputs. Procedures in Logo are simply too abstract.

The Logo community developed a schism when Logo Computer Systems, Inc. introduced their own version of Logo in 1981. The developers of LCSi Logo, somewhat under the general direction of Prof. Seymour Papert, hypothesized that the difficulties in understanding Logo came from irregularities in its implementation. In particular, the conditional operator IF and the primitives that deal with printing and editing procedures (EDIT, PRINTOUT, ERASE) were all special forms in Logo. LCSi Logo developers felt that this was an impediment to true understanding of the Logo model.

The difference between the MIT and LCSi implementations is not so much about which is truer to some abstract model, but about which Logo presents a real, working system that is more easily modeled from a naïve standpoint. This means nothing more than asking which system is easier to understand.

The position of the Boxer developers was that a new language for naïve users should be designed from the ground up — with an explicit, spatial model of computation.

¹⁰For example, the FIRST and BUTFIRST primitives, corresponding to CAR and CDR in Lisp are extended to operate on symbols and numbers and are supplemented with the LAST and BUTLAST primitives; but LAST and BUTLAST are much slower than their counterparts, FIRST being $O(1)$ and LAST being $O(n)$.

¹¹That this appeal to visual modes of understanding works is one of the strongest arguments for a Boxer-type system.

Chapter 4

The Boxer Model

The previous chapters have presented Boxer from the point of view of users and designers. This chapter provides a transition, presenting a detailed explanation of the model used for evaluating Boxer programs. One description is textual and semi-formal. This description of Boxer is the definition to which the implementation must conform. The other description is a visual example of the process of evaluating a particular expression.

4.1 The Detailed Model

Section 3.2.5 presented a two-step model for the execution of Boxer programs. That model is correct, though incomplete. It is the first explanation of Boxer program execution given to a Boxer user, and for the cases novices encounter it is sufficient. Indeed, the phrase “copy and execute,” though merely an emblem of the entire evaluation process, is enough to predict much of how Boxer evaluates programs.

A fuller model of execution is presented below. It is not a contradiction with Boxer’s principles if this model looks daunting on paper: The spatial, intuitive aspects of understanding the Boxer model cannot be overstressed. The reader may wish to refer to Figure 4.2, which shows the successive frames that the Boxer movie stepper¹ presents in the evaluation of a simple expression.

The explanation of the model presented below explicitly handles every aspect of Boxer programs except for flavored inputs and the details of primitive commands such as TELL, REPEAT, and IF.

Flavored Inputs Input flavors offer control over the parameter passing mechanism. Normal Boxer “copy” semantics is call-by-text. The port-to input flavor changes the mechanism to call-by-reference for a given parameter. The presence of the datafy flavor allows access to the text of a parameter object, and rules out call-by-denotation as an implementation strategy.

¹See Section 4.2.

Figure 4.1: How to execute a line of a Boxer program.

1. Examine the first thing on the line.
 - If it is a word which is the name of a box:
 - (a) Look up through the levels of boxes until you find a box with that name.
 - (b) Copy the named box, sans name, and replace the word on the executing line.
 - (c) Go back to before the new box and continue executing the line.
 - If it is a DATA box, leave it as is.
 - If it is a number, replace it with a box containing that number.
 - If it is a port, replace it with a port to the same box.
 - If it is a DOIT box, examine the DOIT box and count the number of inputs. If the box requires inputs, continue executing the rest of the line until there are enough DATA boxes available on the rest of the line to serve as inputs. When done (or if there are no inputs), perform the following operations:
 - (a) Move the input DATA boxes into the DOIT box.
 - (b) Name them according to the names of the inputs of the DOIT box.
 - (c) Begin executing the lines of the DOIT box, one by one.
 - (d) When finished, replace the DOIT box with the last DATA box or port left on the last line of the DOIT box.
 - If it is an @ character followed by a Boxer object:
 - (a) Evaluate the Boxer object by itself, using these rules, and obtain a result box.
 - (b) Replace both the @ and the object with the contents of that result box.
 - (c) If it is a word which is the name of a primitive:

Treat it as if it were a DOIT box which produces its result automatically once given the input DATA boxes.
2. Step 1 should have produced one or more data boxes or ports. Continue using Step 1 to execute the line until it has nothing left but DATA boxes or ports.
3. When all names, DOIT boxes, and ports have been eliminated, the line is done. The result of evaluating a line is the last DATA box or port on that line.

Although input flavors are not described in the semi-formal model of Figure 4.1, they are a straightforward extension of it. The formalism necessary to write down the modifications is cumbersome and not instructive. Boxer users never see such a model written down; they are taught the Boxer model of execution in conjunction with the Boxer movie stepper, and learn it in a visual framework.

The underlying idea, however, is simple. For each expression which is expected to produce a value, the system must remember which input flavor is expected. This input flavor comes from the `INPUT` line of the `DOIT` box, or from the definition of the primitive. When it comes time to copy the result of the expression (or immediate datum), the evaluator must check the input flavor. If it is a `PORT-TO` flavor, the evaluator should construct a port instead of making a copy. Other flavors, such as `DATAFY` and `BOX-REST` (similar to the Lisp `&REST` argument declaration), are merely new cases of the “If it is...” form.

See Section 6.2.2 for a discussion of the implementation of flavored inputs.

4.2 The Movie Stepper

The Boxer *movie stepper* is a tool for understanding Boxer. Pressing the *step* key instead of the *doit* key causes Boxer to show the evaluation process on the screen. This stepper is called a “movie” stepper because the way it represents the process is by showing a movie; steppers in other languages are merely programs which print out tracing or debugging information, unrelated to the semantics of the language. In Boxer, the movie which the stepper shows is the actual process of evaluation. By the principle of concreteness, every object in Boxer has a place on the screen. The stepper shows the intermediate objects which are created during the evaluation of expressions.

Figure 4.2 shows the successive frames which the stepper presents in the evaluation of a simple expression.

Section 5.3.6 discusses a few implementation details of the stepper; however, a detailed analysis of the stepper is outside the scope of this paper.

4.3 Variable Scoping

The clever reader of the Boxer programming model will note that since procedures are copied textually into their callers, Boxer has dynamic scoping. The choice of dynamic scoping runs counter to modern trends in computer science, and requires some explanation. Part of the appeal of dynamic scoping is that it emerges from the simplicity of the model. Other models, almost as simple, give rise to lexical scoping of procedure parameters; however, they require the use of ports in procedure call. This complication, slight though it may be, involves the introduction of a concept unacceptably difficult for the purposes of the Boxer computation model, coming as it does so early in the user’s Boxer career.

Figure 4.2: The procedures to be stepped:

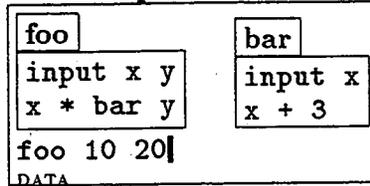
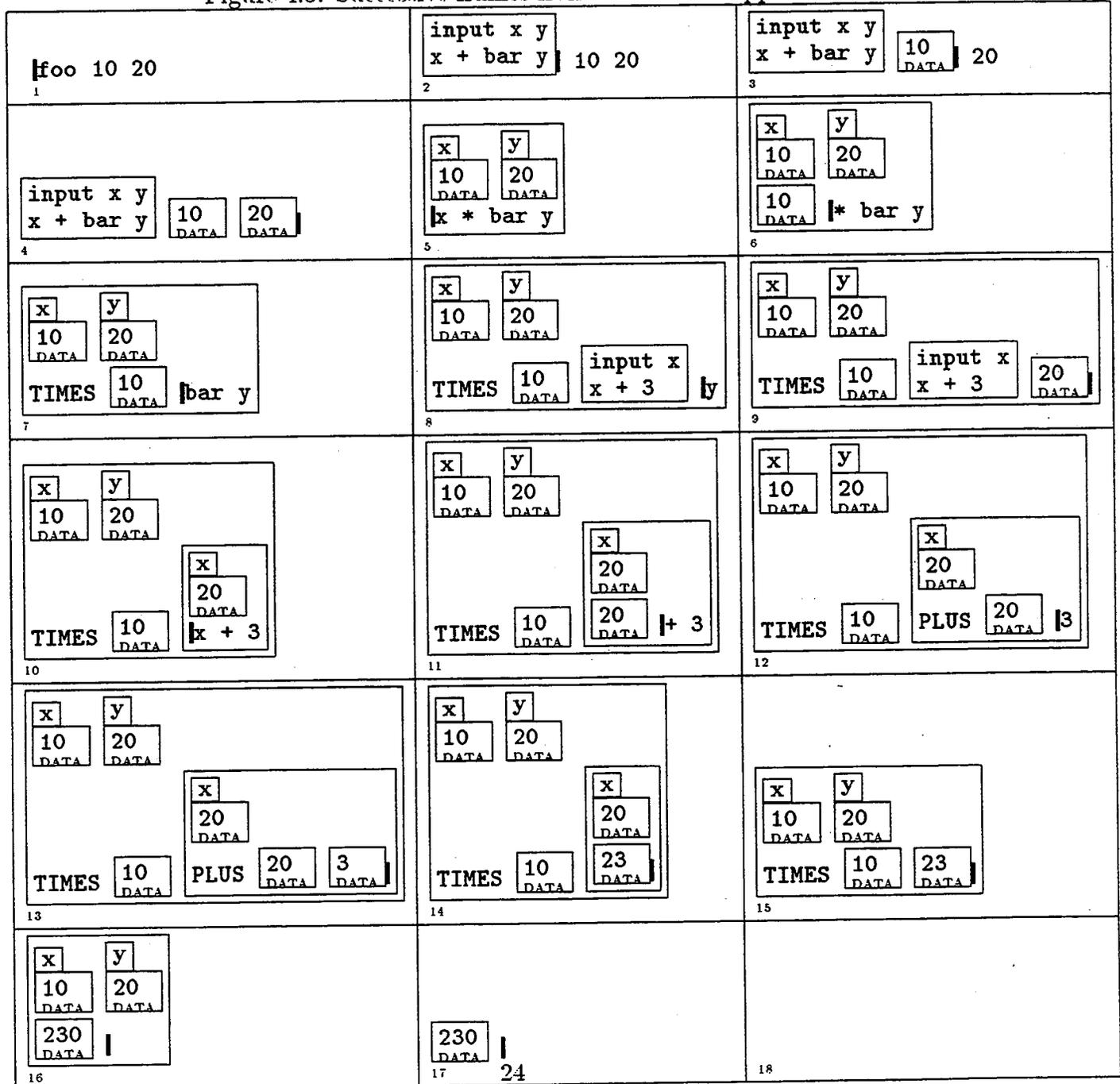


Figure 4.3: Successive frames from the movie stepper:



In contrast with languages like Logo, Scheme, and earlier languages such as FORTRAN, Boxer is true to the model used for explanatory purposes. [10] and [9] offer discussions of the importance of programming models, and in particular the importance of a simple, correct model. Although lexical scoping avoids programming pitfalls produced by name conflicts in dynamic scoping, it also complicates the user model. The presence of dynamic scoping has, however, made it difficult to write a compiler for Boxer. This price is one we have paid for adherence to a simple model.

4.4 Bugs in the Model

The above model describes quite accurately how Boxer runs programs. The implementation described in the succeeding chapters generally follows this model. The details of box copying, of data mutation, and of variable scoping are all self-consistent and implementable.

There is, however, one slight internal inconsistency, independent of any implementation considerations. By the definition above, pressing the DOIT key on an expression should cause the expression to be *replaced* by its value; expressions which return no value should simply disappear. Since this behavior would be inconvenient in the extreme, Boxer makes a special allowance for the top-level expression, and places the returned value after the expression.

One explanation of this behavior is to say that Boxer copies the initial expression to that place, and then applies the above model to the copy. However, that copying operation does not preserve the identity of DATA boxes present on the line. This modified model, however, cannot explain how the following use of the CHANGE command alters the first DATA box:

```
CHANGE 

|      |
|------|
| 1    |
| DATA |



|      |
|------|
| 2    |
| DATA |


```

We have chosen to ignore the problems which the returned-value convenience feature engenders, as they appear not to cause confusion in Boxer users.

Chapter 5

Language Implementation Issues

Chapters One through Four presented language design issues, explained the Boxer system and language, and chronicled the evolution of the language model, all independently of any implementation considerations.

Figure 5.1 is a “road map” to the design and implementation issues treated in this paper. Note that it is not a block diagram of the Boxer system, or even of the Boxer evaluator. Not all components of the evaluator system appear in the diagram; some of the parts in the diagram represent system-wide concepts or choices rather than code modules.

The map is loosely organized into three regions: language design issues, language implementation issues, and implementation efficiency issues. The arrows in the map show the influence of the constraints and decisions of one item on another. Although by no means an exhaustive listing, this map is a useful tool in understanding the choices we have made in implementing Boxer.

This chapter begins the discussion of language implementation issues, which are dominated by the choice of an *explicit-control* evaluator. The following section is a history of the previous implementations of the Boxer language, and explores some of the reasons for this choice. Section 5.2 is a more technical explanation of the modules of the evaluator system and of its interaction with the rest of Boxer.

Language implementation issues not covered in this chapter are described in Chapter 7.

5.1 Implementing Boxer: An Apologia

My work with Boxer has been concerned primarily with the programming language, and its interpreters, compilers, debuggers, and error handlers. The first language implementation of Boxer was done in 1981 on an MIT CADR Lisp machine, with the editor and compiler both developed by David Neves. The compiler was a Boxer-to-Scheme translator. Eric Tenenbaum, developed a Boxer-to-Lisp compiler that produced code running natively on the Lisp Machine. Gregor Kiczales was responsible for the Boxer variable mechanism, the editor redisplay algorithm, and Boxer internal data structures for both the editor and the programming system. Edward Lay took over the reins from Kiczales, and has thoughtfully

re-designed and re-implemented the system.

One of my tasks in 1983, when I became involved with Boxer, was to take charge of the programming language. The existing language was only a place-holder, as the evolutionary design process was expected to take many years. Execution speed and convenience of implementation were never the primary forces in the design of the Boxer specification, but were of course important in the actual implementation.

I first implemented a Boxer-to-Lisp compiler based on a Pratt parser.[14] I chose this approach because it seemed to provide a flexible substrate for implementation, and because the possibility of compiling the resulting Lisp code was hoped to make up for other inefficiencies in Boxer.

Nevertheless, from my experience with Logo, I knew that implementing Boxer as a translator would be fraught with difficulties. The translator approach would put pressure on Boxer to conform to the semantics of the host language. The translator approach also made it attractive, and sometimes necessary, to write heavily system-dependent code (see paragraph *Previous Implementations* on page 34). Writing a Boxer error handler would also have required a complicated set of system-dependent routines for stack manipulation.

Furthermore, in a language with Logo-like syntax, it is not possible to parse a line of code without knowing the number of arguments to each procedure.¹ Redefinition of a procedure may invalidate the parsed code of many procedures. The Logo-to-Maclisp compiler LLOGO skirted this problem by the use of backpointers from procedures to their uses.[15]

The MIT PDP-11 version of Logo[28] and the later microcomputer implementations of Logo[29] instead used a one-token lookahead evaluator, combining parsing and evaluation.

Later, Gary Drescher at LCSI used a global counter, timestamping each procedure and incrementing the counter whenever a procedure was redefined with a different number of arguments (or a new procedure was defined with other than the number of arguments assumed for undefined procedures). After a new definition, all existing procedures (affected or not) are re-parsed as they are invoked.² The overhead of re-parsing potentially the entire workspace after each procedure definition was justified by the reduction in storage requirements over the LLOGO scheme. In practice, the overall LCSI implementation was slower than the MIT version by 10

Despite the shortcomings of the translator approach, I felt that it was the best approach for Boxer, as we required rapid prototyping for defining and re-defining the precise semantics of Boxer, and that approach offered extreme flexibility. I chose to ignore the redefinition problem in the compiler, requiring our users to handle the redefinition themselves, and I resolved to write an interpreter when the language became more firmly defined.

In 1984, during a hiatus in my association with the project, the group decided that

¹Boxer has the added problem that a given variable might refer data in one invocation and a procedure in another.

²Gary Drescher, personal communication. Drescher notes that "the counter only increments for redefinition of a procedure which is called in some other, already-parsed procedure (each procedure has a bit that says if this has happened); otherwise, no reparsing is necessary."

issues of Boxer syntax and semantics were stable enough to move to an interpreter. Indeed, an interpreter seemed necessary to experiment with flavored inputs which were then one focus of discussion in Boxer. Lay implemented an evaluator that was composed of recursive Lisp procedures. The emphasis was on experimenting with flavored inputs, the TELL primitive, and other related issues, while solving some of the known problems with the existing parser. As the evaluator was written as a series of inter-calling Lisp procedures, the Boxer stack was still represented as internal Lisp state, and was not accessible to a debugger.

In 1985, work on the Boxer language shifted to design at a finer detail, with an eye toward tuning the existing semantics, and writing a stepper and debugger. I returned to M.I.T. to work on an explicit-control evaluator, which would keep its state in Lisp variables accessible to a stepper, debugger, or error handler written in standard Lisp. We chose the emerging Common Lisp as an implementation language because of its stability and closeness to the Lisp Machine Lisp in which the existing Boxer code was written.

Beginning in 1984, I had been developing a portable implementation of Logo with Sobalvarro. Taking his explicit-control Logo evaluator written in C as a base, I began work on the Boxer interpreter.

5.2 Boxer and the Interpreter

Figure 5.2 shows the organization of the Boxer system. This chart was derived from an analysis of the source code. Modules which serve only as interfaces to other modules have been omitted for clarity.

Boxer is organized around the editor module, which defines and maintains the box hierarchy. By the Boxer principle of concreteness, every Boxer object must be represented in the hierarchy; due to the pervasive nature of boxes, the subsystem that maintains boxes themselves occupies a central position in the diagram.

Shown above the editor module is the evaluator module. The direct line connecting them is a narrow interface, confined mostly to the operation of the *doit* key and other Boxer keys. The majority of interaction between these two modules takes place through the virtual copy module, which carries out most of the copy operations that occur during program execution.

The evaluator itself is divided into six parts, as shown in the figure. The definitions package is nothing more than support for the other systems, and a detailed examination of the stepper and error handler systems is outside of the scope of this paper.

The variables module is explained in Section 5.2.2 and Section 5.2.3.

The following sections will describe the state machine and stacks.

5.2.1 State Machine

The heart of the evaluator system is a single Lisp function, `BOXER-EVAL`. This function is an *explicit-control* evaluator written in Common Lisp. Explicit-control means that it

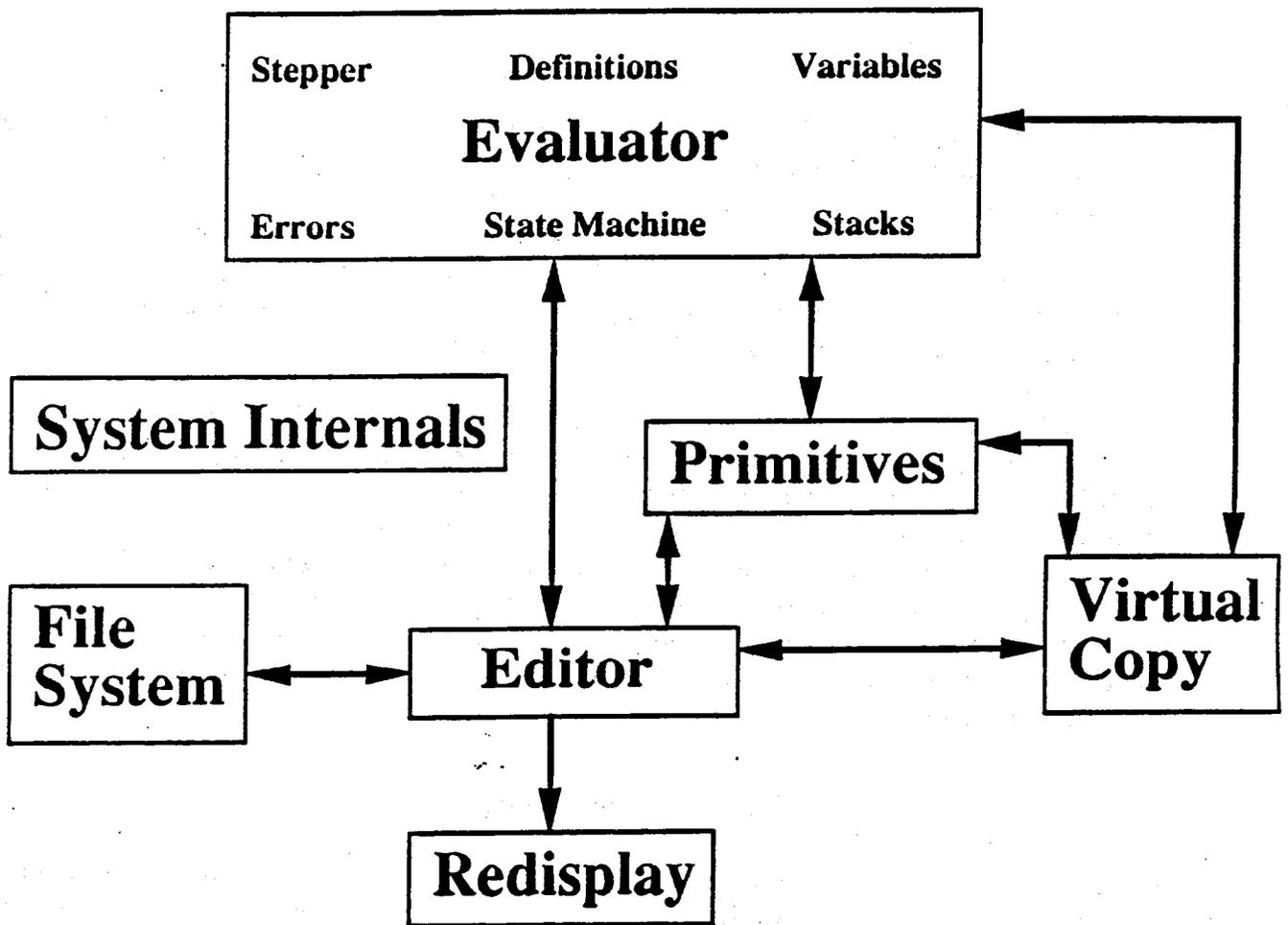


Figure 5.2: Overall Structure of the Boxer System

does not rely on the underlying language for support of procedure calling and argument passing.

The evaluator is coded as a single function to prevent the growth of Lisp stack, which would make writing a Boxer debugger system difficult, and would make tail recursion in Boxer impossible. Implementation of the explicit-control evaluator as a single function posed no additional design difficulties and resulted in a gain in execution efficiency, since the overhead associated with needless Lisp procedure calls is avoided.

The state machine maintains its state in variables defined in the evaluator definitions module. These variables hold the currently executing function and pointers to the current line and token, the function for which arguments are being gathered and its argument list and flavors, the most recently generated value, and the three stack pointers described in Section 5.2.2.

Recursive Evaluation

One constraint imposed by the explicit control evaluator is that it not be invoked recursively. That is, primitive operations which require the evaluation of expressions may not simply call the evaluator; all control information must be kept on the Boxer control stack.

The implementation provides a mechanism for defining primitives, such as `RUN` and `REPEAT`, that perform recursive evaluation. These primitives are automatically broken up into four segments: a Lisp function to be executed before the recursive invocation, a list of Boxer code to be evaluated, a Lisp function to be executed after the recursive invocation, and finally a (possibly empty) clean-up function to be executed in the event of a error or other non-local exit. The before and after Lisp functions may communicate via state variables, which are preserved in PDL³ stack frames should calls to the primitive become nested.

5.2.2 Stacks

Boxer has three different stacks. The *VPDL*, the *PDL*, and the *Dynamic Variables Stack*.

VPDL

The *VPDL*, or “value push-down list,” is the intermediate resting place for values which will become the arguments to functions, either primitive functions defined in Lisp or Boxer (DOIT box) functions. It is implemented as an array with a global index pointer.

When a form generates a value, the evaluator checks whether arguments are being gathered for a function, and if so, pushes the the value on the VPDL.

The evaluator clause that invokes primitive operations simply leaves the values on the VPDL. All primitive operations are defined as Lisp functions of no arguments which pop their real arguments from the VPDL into local variables. This mechanism avoids the

³See Section 5.2.2.

inefficiency associated with creating an argument list for the Lisp `APPLY` primitive, while still allowing fast reference to the values within compiled primitive functions.

PDL

The PDL (from the archaic “push-down list”) contains the control flow information for the evaluator. It is organized into typed stack frames, which are implemented as lists. Each stack frame points to the previous stack frame and the next stack frame. To reduce garbage collection overhead, Boxer pre-allocates the various kinds of PDL stack frames and maintains them on free lists. The use of typed stack frames has made the `CATCH` and `THROW` operations (implemented in Boxer by procedure names and the `STOP` command, respectively) particularly easy. Additionally, it has helped in debugging the evaluator at a very small overhead cost.

There are three main kinds of stack frames. Argument-evaluation frames preserve the function and (remaining) argument/flavor list of the function for which values are being gathered. Function-call frames preserve the pointers to the currently executing function, line, and token, and a few other state variables. `DOIT` port frames preserve the static and dynamic variables contexts across lexical function calls.

The recursive evaluation primitives also define PDL stack frame types. In addition to a minimum of state variables associated with the recursive evaluation mechanism, these frames preserve the state variables of the recursive evaluation primitives. For example, the `REPEAT` primitive frame preserves the list of Boxer commands to execute and the remaining repeat count.

Each stack frame type is defined with a name, an initial number of frames to be pre-allocated, a growth rate, a list of state variables to be preserved, and an unwind-protect function. The unwind-protect function pops the frame in case of error or non-local exit, and performs any necessary clean-up operations. The unwind handler for function call frames is responsible for removing dynamic variable binding information from the dynamic variables stack. Similarly, the handler for argument evaluation frames removes values that have been gathered but not yet used from the `VPDL`. Frames associated with recursive-evaluation functions use the handler specified in the primitive definition. The default handler merely pops the frame and restores the values of the variables it protects.

Dynamic Variables Stack

As will be described in Section 5.2.3, formal parameters and local variables of procedures are bound to their values by deep binding, using a list of association lists. The evaluator state machine is responsible for binding and unbinding these variables. The evaluator clause which invokes user functions calls a function that performs dynamic binding. This function builds an association list of the the argument list from the user function and the values that it pops from the `VPDL`. Additionally, the dynamic binding function makes (virtual) copies of the local variables of the `DOIT` box and includes them and their names

in the association list. This association list is then pushed onto the dynamic variables stack.

5.2.3 Variable Reference

Dynamic Variables

Although variable lookup is defined in the Boxer model as an outward scan of boxes, the implementation takes advantage of the fact that dynamic scoping is obtained under the copy-and-execute model and implements it directly, dividing variable lookup into two segments: dynamic lookup and static lookup. Dynamic variables are the arguments and local variables of the current DOIT box, and as well as those of all its callers.

In an earlier implementation of Boxer, the use of shallow binding for dynamic variables resulted in an unacceptable performance in context switches done with the TELL primitive and with port execution. The current dynamic variable binding mechanism is implemented as set of simple association lists, one per DOIT box.

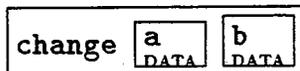
On procedure invocation, the formal parameters, already copied or treated in accordance with the input flavor of the procedure, are associated directly with the variables. Local DATA variables of the DOIT box, however, are (virtual) copied before being pushed in the binding list.

Problems with the Implementation The dynamic variable binding function allocates almost all of the temporary storage that the evaluator uses, especially in compute-bound functions. The presence of *ephemeral garbage collection*⁴ in the Lucid implementation of Common Lisp makes this storage allocation — and its attendant garbage collection — tolerable. A future implementation might use two simple vectors for this association list, and eliminate temporary storage allocation except in deep recursion.

Tail recursion, though not presently implemented, is expected to give a speedup in what is currently the slow case of dynamic lookup, by eliminating stack growth in tail-recursive loops.

In violation of the strict definition in the copy-and-execute model, DOIT boxes remain uncopied on procedure invocation. This expedient is not without problems: self-modifying code, rather than being “safe” and changing on the executing copy of a DOIT box, changes the original box. Calls to the CHANGE function which perform mutation on named data boxes are not affected.

We have chosen to ignore this problem, because it is difficult to make use of self-modifying code which does not effect its modification without the use of variables names. Such code is either difficult to write, or difficult to use. For example, a DOIT box which contains the expression



⁴See [30] for an explanation of this feature of Lucid Common Lisp.

is an instance of self-modifying code; yet in effect such expressions are almost entirely useless, as there is no way for a procedure to access the a DATA box without a name or a port.

Using accessor and the UNBOX or @ functions to reach the contents of a DOIT box which is inside a named DATA box is not feasible either, since neither of the de-reference functions can unbox DOIT boxes. UNBOX issues an error, and @ causes immediate execution of the box.

Only by creating a port directly to a DATA box which is part of a procedure can a Boxer user create self-modifying code. Since DATA boxes have a list of back-pointers to any ports, the dynamic variable binding mechanism could be changed to use some cached information regarding which DATA boxes in procedures are actually ported to and must therefore be copied on procedure invocation. Implementing such a feature would be possible, but perhaps time-consuming, and is not contemplated for the immediate future.

Previous Implementations Dynamic scoping in Boxer has had a number of flip-flops in implementation. In the first implementation it was difficult, as the Boxer evaluator was a Scheme interpreter written in Lisp Machine Lisp. In the second implementation, since procedure (DOIT) boxes were translated directly into interpreted Lisp procedures, variable scoping was automatically dynamic. Gregor Kiczales implemented the static Boxer variables which come from box nesting through an ingenious mechanism: associating with each box a Lisp Machine closure⁵ over variables which were contained in the box it represented. Moving the cursor from one box to another caused the box functions to throw to the lowest common superior point and then invoke one another through a path leading to the new box. Taking advantage of the host system's shallow binding made variable reference fast, but also made context switch particularly slow. Adding new variables was difficult, because the lexical environment was at that point dynamic, and represented by the internal Lisp stack. On the CADR[26], Kiczales was able to use a "stack-blt" routine to manipulate the Lisp stack directly in order to add variables to the running closure functions. Increased emphasis on context switch speed eventually made this approach less attractive. The Symbolics 3600 implementation used deep binding both for portability and for context switch speed.

Static Variables

Static lookup continues where dynamic lookup leaves off. It begins in the box where the top-level procedure was invoked; that is, it begins in the box where the user pressed the *doit* key, and continues with that box's superiors in the editor hierarchy.⁶

In the editor data structure, static variables are stored in association lists variable/value pairs, one per box. Static variable lookup begins with an association-list lookup of the

⁵Lisp Machine closures did not close over an environment, but instead over a specified list of variables.

⁶Static lookup actually begins from the `static-variables-root`, which is set by the *doit* key, by the TELL primitive, and by DOIT port execution.

STATIC-VARIABLES-ALIST of the static variables root box, and proceeds with the superior box, via the value of the **SUPERIOR-BOX** editor box instance variable. Primitive function bindings, being outside the outermost box, are stored in the Lisp value cells of the symbols which name them.

Section 6.2.4 describes a caching algorithm which speeds up static variable reference.

Transparent Boxes

In Boxer, boxes define environment contours. In general, variables (named boxes) inside a box are not accessible outside the scope of the box. The **TELL** primitive is provided for access to the static scope of a box. Also, executing a port to a **DOIT** box performs the execution in the static scope of the target box.

In addition to these mechanisms, Boxer provides a third way of breaking the box borders environment boundaries: the *transparent* box. Graphics boxes, for example, are transparent. Although they have variables inside them, the scope of the variables is the same as if they existed in the surrounding box. The transparency allows Boxer users to use **TELL** to access the sprite boxes inside graphics boxes without first having to use **TELL** on the graphics box itself.⁷

Transparent boxes are also sometimes used as *toolboxes*.^[17] A toolbox is an aggregation of sub-procedures which implement operations which should be considered primitive operations in a particular box, or microworld. The toolbox groups the procedures together and places them *inside* the box where they are to be used. Normal Boxer usage would put all of the procedures inside the box directly, or would place them outside the box to be used.

Although the user is free to open the toolbox and examine or change the procedures, the nature of toolboxes discourages experimentation. Since toolboxes do not follow the normal scoping rules of Boxer, their use can be confusing to Boxer users. Toolboxes are generally used in conjunction with *closets*.

Each box has a hidden place called the closet.⁸ The closet is used mostly in microworlds and interactive workbooks. It is a place for hiding internal procedures and data which are visually distracting and not germane to the microworld. However, the boxes in the closet remain available for inspection and modification.

Toolboxes and closets are still considered experimental features in Boxer. Beginning users find them confusing, and users exposed to transparent boxes too early in their Boxer careers have trouble understanding Boxer scoping rules.

The implementation of neither transparent boxes nor closets required any additions to the evaluator. Editor routines handle the insertion and deletion of named boxes in transparent boxes by acting on the **static-variables-list** of the superior box. Closets are implemented by the editor and the **CHANGE** function. Transparent **DOIT** boxes, though

⁷The notion of an "exporting" box, which would be transparent to only certain variables, is another way that graphics boxes have been implemented in the past.

⁸Closets are a descendent of diSessa's local libraries. I suggested the name, and Lay implemented them.

contemplated at one point, have not been implemented.

5.3 The Interpreter and the Editor

This section talks about the relationship between the editor and the evaluator in Boxer.

Since Boxer is an integrated system, the editor and evaluator are closely related. As a consequence of this integration, the implementor is able to choose between an editor system and an evaluator system implementation of many of Boxer's component features. The following sections discuss the various points of interface between the two systems, and mention the space/time or amortization⁹ tradeoffs in certain functions.

5.3.1 Editor Top Level

The top-level Lisp function of Boxer is a loop that reads events from the keyboard and pointing device, creates Boxer function names from the key or click names, and calls the evaluator on these functions. For example, pressing the F key causes Boxer to invoke the CAPITAL-F-KEY function. Likewise, pressing META-B calls META-B-KEY. Mouse buttons or keypad presses are similarly named.

Since all editing commands are defined as Boxer functions, the user is free to redefine them in the scope of any particular box. This decision is part of the influence of diffused functionality in Boxer: The popular Logo program INSTANT, which reads single characters and moves the turtle accordingly, can be implemented in Boxer with almost no programming.

Boxer editing commands are also available as primitive operations in Boxer: several of them may be grouped together in a DOIT box to form an instant "keyboard macro." Values returned from key functions are inserted at the cursor position. Thus, assigning a DATA box to a key function causes that key to insert copies of the box.¹⁰

5.3.2 Polling

The editor system is responsible for interrupting the evaluator should the user request it. The evaluator calls an internal polling function after examining a certain number of tokens (currently fifty). The polling function should perform the input stream handling for the editor. If input is not available from the operating system, it should return as quickly as possible. Otherwise, it must get all available input, check it for the presence of the

⁹Spreading out of expensive run-time operations. I am indebted to Ed Lay for pointing out this consolidating factor in the interface.

¹⁰The Boxer editor and evaluator conspire to create menus with no programming effort: To make a menu, the user need only place the commands in a box. To use the menu, s/he simply uses the mouse or keyboard to select and evaluate the desired line. In the present system, a double middle click on the mouse moves the cursor to the indicated spot and evaluates the line; this is merely a convenience, as using the mouse to move the line and pressing the *doit* key is hardly any more effort, and a natural operation in Boxer. Making a menu DATA box into a key function results in "pop-up" menus, again with almost no programming effort.

interrupt character, and queue other events internally. The polling function sets a flag if an abort was signalled.

In the stepper, the number of tokens to examine between calls to the polling function is set to one.

5.3.3 Naming

Since variables are simply boxes with names attached, the editor is responsible for creating Boxer variables. To create a variable or procedure, the user uses the left parenthesis or bracket key to create a box (DOIT and DATA, respectively), and then presses the *name* key or moves the cursor to the name tab of the box to type the name. Exiting the name area causes the name to be assigned to the box. This exception prevents boxes from receiving successively longer names as the user types. Early implementations of Boxer had separated the box from its name, and required using the *doit* key to make the assignment. Although the idea of "assignment" was conflicted with the Boxer concrete idea of a variable, we found it difficult to discern changes to the box name until we placed it in a special part of the box.

The editor is responsible for informing the evaluator of the addition and deletion of named boxes, and is furthermore responsible for ensuring that there is but one box with a given name in each binding contour (box or transparent box). Presently, this is done by renaming any existing box to add a version number.

5.3.4 Transparent Boxes

Transparent boxes are strange kinds of boxes that do not represent binding contours. Their presence is presently a topic of debate. Early implementations of transparent boxes handled them in the variable lookup, but the modern system places this onus on the editor. The editor is responsible for maintaining the static variables associated with transparent boxes just as it is responsible for maintaining those variables associated with normal boxes, with the exception that the box in which the binding occurs is different. This change, though complicating the editor slightly, made variable lookup faster. Section 5.2.3 describes transparent boxes in detail.

5.3.5 Prompting

Boxer provides a key, the *prompt* key, to insert after a function (that is, after a DOIT box, the name of a DOIT box, or a primitive name) the names of its inputs. Each input name is followed by a colon. The Boxer user then fills in the input objects after the names. The prompts can remain in the code to serve as documentation. Currently, the names are not significant, and do not correspond to Common Lisp keywords, which allow specification of arguments in any order.

5.3.6 Stepping

The Boxer movie stepper, described in Section 4.2, is a debugging aid and pedagogical tool that shows the process by which Boxer runs programs. To step through the evaluation of an expression, the Boxer user simply presses the *step* key instead of the *doit* key. Boxer then presents a moving picture of the evaluation process, starting with the expression, showing its transformations according to the Boxer model, and ending with the value of the expression.

The stepper is a slight altered copy of the state machine evaluator Lisp program. During evaluation, this version of the evaluator uses calls to the editor system to copy, move, change, and delete boxes in order to make the screen state reflect the current evaluator state.

Additionally, the altered state machine does not use the mechanism of Section 5.2.2 to bind the local variables and formal parameters of DOIT boxes; the box nesting that comes about as a consequence of the execution of the model handles variable binding automatically. It is the strong integration between the editor and the evaluator that makes this implementation possible.

5.3.7 Triggers

The Boxer *trigger* mechanism described in Section 2.2 is another point of interface between the evaluator and editor systems. Its design was influenced by the Boxer concreteness principle, and its implementation was influenced by the explicit-control evaluation.

The present implementation of triggers works much like the recursive evaluation feature. Triggers are invoked either in the editor, by keyboard commands, or in programs, by mutation (or editing) primitives. In either case, low-level functions in the box hierarchy maintenance functions set flags and variables, which upper-level routines of the editor and the evaluator examine.

Triggers tripped in editor commands are handled by the editor command loop, as the evaluator is not actually invoked by most key and mouse operations.

Since triggers are signalled only during the execution of primitive operations, the primitive function return handling clause is the only place in the evaluator that examines the trigger variables. In the present implementation, the trigger function is simply inserted into the token stream. The evaluator will then execute that function as if the trigger code appeared immediately after the primitive which caused the mutation (or entry or exit) operation which invoked the trigger. Special care is taken to prevent the loss of returned values from previous expressions.

Trigger functions are cached inside boxes. Cache maintenance is done by the editor.

Triggers and Sprites Modification triggers are used to implement one direction of the concreteness feature of sprite instance variables. For example, changing the value of the X-POSITION sprite instance variable moves the sprite along the *x*-axis. This behavior is

Chapter 6

Implementation Efficiency Issues

This chapter treats the material in the “Implementation Efficiency Issues” dashed-line area of Figure 5.1.

Implementation efficiency issues in Boxer have generally been concerned with execution speed; memory usage has been a secondary consideration. An analysis of Boxer program execution under the Lay Lisp evaluator and under the explicit-control evaluator identified four major areas where speed improvements were necessary: box copying, variable lookup, arithmetic and number processing, and flavored input handling.

As Figure 5.1 shows, three mechanisms make these processes efficient: a detailed data-type case analysis system in the explicit-control evaluator, a particular choice of the data types used to represent objects in the evaluator, and a caching algorithm for static variable reference. Section 6.1 describes the first two mechanisms, and Section 6.2 talks about how all three mechanisms accounted for speed improvements in the various processes.

6.1 Mechanisms for Efficiency

6.1.1 Explicit Control Evaluator Data-type Case Analysis System

The central choice the language implementation was that of the explicit-control evaluator. In the evaluator state machine (described in Section 5.2.1), each state which produces values is composed of a series of handlers which dispatch on the data type of the produced object. The flexible analysis this organization makes possible allows efficient flavored input handling, reduces unnecessary copying, and makes arithmetic faster.

6.1.2 Restriction of Evaluator Data-Types

The exact nature of the data objects used to represent boxes and their constituents were found to have a great effect on execution time of Boxer programs. Boxes in the editor hierarchy are presently represented as PCL/CLOS (Portable Common Loops/Common

accomplished by placing a call to a sprite updating function in the modification trigger of each sprite instance variable.

Bugs There are presently bugs in this implementation of triggers: If the trigger gives an error, the error is reported as having occurred inside the primitive operation which invoked the error.

A correct implementation would make use of a proposed general pause/interrupt mechanism, rather than altering the token stream.

The current mechanism does not preserve the static environment of the trigger function. Using **CHANGE** to alter a port to the instance variable of a sprite box not accessible by name does not work because the sprite update function is invoked in the environment of the **CHANGE** operation. A proper implementation would execute the trigger code in the environment of the box with the trigger.

Lisp Object System, described in [18]) objects. PCL, PCL/CLOS and the earlier Flavor system provided important flexibility for the Boxer development platform. For the editor, this flexibility remains important; however, in this implementation of the Boxer evaluator, the speed expense of operations on CLOS objects outweighed the advantages.

In particular, type predicates on CLOS objects, and indeed type predicates on almost any Common Lisp object, are unacceptably slow. Lay and I decided to unify a new evaluator representation of objects with his emerging virtual copy system, and to base the new objects on Common Lisp structures. Even so, predicates on structures were unacceptably slow. As this slowness is inherent in the design of the Common Lisp type system, we decided to move away from it.

Accordingly, the central evaluator function almost completely limits the objects it deals with to symbols, numbers, and vectors whose first element is a type symbol. With few exceptions, `SYMBOLP` and `NUMBERP` are the only Common Lisp predicates used in the evaluator. Case analysis is constructed such that these two predicates occur first, with the implication that any object not one of these two types must be a vector with its type symbol in slot zero.

The exceptional cases are in the routine that first examines the values of variables, which may operate on any type of Boxer editor or evaluator object. In this case, a `VECTORP` test precedes any calls to predicates of CLOS objects.¹ Furthermore, the evaluator uses implementation-specific “fast CLOS” predicate macros. These macros may assume that their arguments are CLOS objects and need not perform sub-type checking. Most CLOS and Common Lisp implementations provide some facility for this kind of type checking.

6.2 Use of the Efficiency Mechanisms

6.2.1 Copying

The single factor which slows down Boxer programs the most is the copy phase of the copy-and-execute model. As the Boxer execution model requires frequent copying of possibly large data structures, any savings in the speed and frequency of use of the box copy routine will have a large effect on overall execution speed.

Abelson and Lay have developed a “virtual copy” algorithm that reduces most fully-recursive copy operations to a top-level copy performed once and cached until the first side-effect operation.[4] Even so, box copying remains a potentially expensive operation. Some form of block compilation with side-effect analysis could presumably eliminate almost all unnecessary copying in a particular set of procedures. Boxer syntax, the lack of a function/variable reference distinction, and the presence of dynamic scoping, however, make it difficult to perform semantic analysis, as no assumptions can be made about the bindings of free variables, including most primitive names.²

¹Compilation macros detect systems in which `VECTORP` returns true on CLOS objects.

²For expediency, infix primitive functions are not re-definable.

The design of the explicit control evaluator took into account the necessity of reducing copying as much as possible. Previous Boxer compilers and evaluators had structural problems which caused them to make multiple copies of procedure arguments and return values.

The organization of the evaluator state machine makes it possible to handle each different kind of value-producing expression separately. The distribution of data-type case analysis throughout the evaluator has made it possible to avoid successive, needless copies of data, while preserving information necessary for creating ports.

For example, a DATA box returned by a procedure or primitive need not be copied, unless it is being gathered as a PORT-TO flavored input.³ By contrast, a DATA box present immediately in a program must be copied when encountered, unless there is a PORT-TO flavor in effect, in which case a port should be made instead.

The implementation-internal DONT-COPY flavor inhibits copying arguments to Boxer primitive functions which are guaranteed neither to mutate nor to return any portion of their input, doing at least part of the job of a side-effect and copying analysis system. Predicate And counting primitives, for example, make use of this flavor.

6.2.2 Flavored Inputs

Flavored inputs are implemented in a remarkably straightforward way, mirroring the description in Section 4.1. The explicit-control evaluator has a series of case analyses in each section of code in which values (from primitives, procedures, or self-evaluating objects) are produced. The cases are based on the current input flavor, and control the creation of copies of, and ports to, the value. The input flavor of each argument is maintained in a list with the argument in the ARGLIST evaluator state variable. Each case handler is constructed to look at the minimum number of cases; unhandled cases default to copying. As the COPY flavor (the default) is represented in the ARGLIST variable as the parameter name symbol itself, this common case can be handled without any dispatch operation.

6.2.3 Arithmetic

A number in Boxer is represented as a box containing a single, numeric word. Unboxed numerals may be thought of as objects that evaluate to Boxer numbers.⁴ A series of numbers already inside a box may be accessed individually with Boxer data-selector primitives. Since these operations do not perform de-reference, the conversion from numeric word to Boxer number occurs automatically.⁵

The original implementation of numbers in Boxer followed this implementation closely. That is, arithmetic primitives constructed and returned boxes containing numerals. A

³In this pathological case, a helpful message is also printed, telling the user that the port is being made to a copy.

⁴In fact, CHANGE 3 4 acts as if this were the case. See footnote 4 on page 15.

⁵Of course, UNBOXing a Boxer number is an error, but applying @ results in another Boxer number.

concession to efficiency was made in that numerals produced Lisp numbers when evaluated, so arithmetic primitives were required to accept either boxed or unboxed Lisp numbers.

This recognition, boxing, and unboxing of numbers made any operation involving numbers quite slow. The new evaluator system removed all boxing and unboxing from the interpreter and placed it in the primitives, and made the Lisp number be the preferred internal form. Rather than requiring arithmetic primitives to box numbers, the new system requires primitives that require boxes as input to create them internally, if given numeric inputs. As a result, arithmetic operations became quite fast, and other data manipulation was slowed down only by a Lisp `NUMBERP` check, which is usually only two machine instructions in compiled Lisp code.

The case-analysis mechanism of Section 6.1.1 makes possible the optimization of number boxes into Lisp numbers. Specifically, formal parameters bound to bare Lisp numbers on the dynamic variables association list do not have boxes associated with them. If the inputs are of type `PORT-TO`, then the evaluator creates a new, virtual box, and binds the parameter to a port to that box. Since this analysis happens when the value is created, the evaluator can take action appropriate to the particular case. For example, typing `CHANGE 3 4` causes a `CHANGE` operation on a port to a copy of a number box to take place; however, it is impossible for any part of `Boxer` (outside the stepper) to determine that this transaction has taken place.

A more insidious case involving numbers and ports occurs when a procedure invokes `CHANGE` on a formal parameter (not a local variable, since they already have virtual-copy boxes associated with them) of a procedure. The `PORT-TO` case of the `SYMBOL` handler in the evaluator notices this precise operation, and replaces the bare number on the dynamic variables list with a virtual copy representing a boxed number. Inductively, it is guaranteed that there is no prior reference to this variable still extant, since any such reference would have come from a `PORT-TO` flavor (and we have shown that the only other such case results in the value being discarded.)

6.2.4 Variable Lookup and Cache

With a straightforward static-variable lookup scheme, execution time of `Boxer` programs increases dramatically with nesting level.

Initially, static lookup was implemented as one association-list lookup per box, combined with recursive Lisp flavor⁶ message calls to the superior box. A first attempt at making static lookup faster involved adding flavor methods to maintain a series of association lists beginning in each box and reaching upwards to the outermost box. However, even with the expedient of eliminating flavor calls, static lookup time was unacceptably slow, and still varied with box nesting level.

Accordingly, Abelson and I decided to implement a caching scheme for static-variable

⁶The Lisp Machine *Flavor System* is a predecessor to the current Common Lisp Object System. Message invocations are more time-consuming than function calls. See [24].

lookup. Since the static context of an evaluation remains constant (except during brief operations such as TELL), we were able to develop an algorithm that reduces static-variable lookup to a single association-list lookup. The algorithm is independent of the design of the explicit-control evaluator, and was initially tested with the Lay evaluator.

The Cache Algorithm The Lisp objects associated with Boxer variables normally remain constant from evaluation to evaluation. For example, the Boxer CHANGE mutation operator does not change the identity of the underlying box implementation objects, only their contents. Furthermore, the shadowing of a previously cached variable by dynamic scoping would not affect the cached value of any static variable caching scheme, as dynamic lookup happens before static lookup.

For these reasons, we decided to create a static variable cache in each box which is used as a static-variables root. That is, each box in which the *doit* key is pressed or a TELL or port-execution operation is done contains a cache which maps variable names to boxes for all static variables referred to in procedures invoked during the evaluation. Variables not found in the cache are searched for in the regular box hierarchy, and cached if appropriate.

Implementing the cache required the addition of two new internal data structures, the `static-variable` object and the `static-variable-cache-entry` object.

A `static-variable` is a structure which is a list of a variable name, the editor box object which is the value, and a unique ID. Static variables are represented internally in boxer by these `static-variable` objects, whether they be in the `static-variables-alist` of a box or in the global Lisp value cells entries of primitives.

Each static-variable cache is implemented as an association list: the `static-variables-cache`. The cache is a list of `static-variable-cache-entry` objects. A `static-variable-cache-entry` object is a structure of type list. It contains the variable name, a unique ID, and a pointer to the `static-variable` object which it caches.

Given a variable name, the cache lookup function performs an association list lookup on the `static-variable-cache` list. (The variable name is defined to be the first element of the `static-variable-cache-entry` object to permit the use of a fast association list lookup function.) The cache lookup function then compares the unique ID of the resulting `static-variable-cache-entry` object with the unique ID of the `static-variable` object pointed to by that cache entry. If the two do not match, then the cached entry is invalid. As some special unique ID values are used for particular conditions, the algorithm examines the errant value to determine what course of action to take.

Deleting a named box (i.e., a variable) from a box by an editor operation or a mutation primitive causes the corresponding `static-variable` object's unique ID to be set to `deleted-variable-uid`. It is the responsibility of the editor to call an evaluator-supplied function on each named box which is about to be removed from the box hierarchy; that function handles any cache deletion marking. When the cache lookup algorithm encounters a reference to a variable which has been deleted from the Boxer hierarchy but which is still present in the cache, it removes the entry from the cache and allows regular (non-cache) static variable lookup to proceed.

The case of adding a new variable of the same name as an already cached variable to a level of box nesting which should cause the cache entry to become invalid is also easily handled, once the necessity for handling this case is recognized. The Scheme lexical variable reference compilation algorithm⁷ handles this case. In the Boxer algorithm, adding a variable to a box causes the next outermost variable with the same name to be given a new unique ID. The entry for the name in the static variables cache, however, still points to both the old variable and its former unique id. On the next reference to that variable, the ID mismatch will be discovered, and the variable re-cached. As with deletion of a variable, addition is handled by having the editor call evaluator-supplied functions.

Changing a variable name is not considered a separate action; the editor is required to notify the evaluator of the deletion of one variable and the addition of another.

Copying a box does not copy the static variables cache.

Deleting a box empties the `static-variables-cache` list of that box and the cache of all its inferior boxes, since that box might be inserted directly into the Boxer hierarchy in some other place.⁸

Some variables need to be exempt from caching. The values of the functions associated with key- and mouse events, for instance, are numerous, frequently called, and not time dependent. Were they to be cached, the cache would quickly become full with the functions key- and mouse-event functions, and cached-variable lookup would no longer be faster than normal variable reference. If the unique ID of a `static-variable` is `non-caching-variable-uid`, then the algorithm never caches the variable.

Future Cache Efficiency Issues In the current implementation, outside of wholesale invalidation, cache entries for subsequently deleted variables are removed from the cache only when the variables are referred to. This restriction is an unfortunate one, since deleted variables are generally not used except in errant code. In the future, a global sweep operation which removes these dead cache entries might be useful. This deficiency is not as much of a problem as it might be, however, as these caches are not saved across Boxer sessions. Furthermore, the cache of a box is emptied if it or its superior is removed from the hierarchy, even temporarily.

An adaptive tradeoff algorithm which places an upper limit on the number of cache entries might be useful. The present algorithm simply caches every static reference on the assumption that the overhead of manipulating the cache any further would overwhelm any advantages.

⁷Personal communication, H. Abelson

⁸Presently, Boxer uses an EMACS-like *kill ring*, where objects recently deleted by editor commands are stored for possible retrieval. These objects are, however, outside the Boxer hierarchy. Future changes in the editor may make the kill ring be a part of the hierarchy, in which case this operation will be handled perforce as a deletion and addition.

Chapter 7

Data-Manipulation Primitives

This chapter discusses the design and efficient implementation of certain data manipulation functions in Boxer. Figure 5.1 shows that the Boxer principle of naïve realism and the use of the spatial metaphor had a strong influence on the definition of data manipulation primitives.

The choice of an explicit-control evaluator has complicated the definitions of primitives which perform recursive evaluations. The primitives described in this chapter make extensive use of the recursive-evaluation mechanism for the efficient solution of problems posed by design constraints of Boxer.

7.1 BUILD

This section describes the efficiency issues in the implementation of the BUILD primitive of Section 2.2.

The BUILD primitive is a spatially-oriented data constructor based on the concept of a *template*. BUILD is a descendent of a series of similar operators designed by the Boxer group. Michael Eisenberg implemented the first modern BUILD function, and is responsible, with diSessa, for many of its basic features. Schweiker and Muthig[16] showed that programming novices were able to use BUILD three times faster than the Logo data constructor primitives. As an excellent discussion of BUILD is available elsewhere[11], this section discuss only the implementation and unusual cases.

The present BUILD primitive takes a single input, which is a DATA box, and outputs another DATA box which is on the surface very similar to the input box. The output box differs from the input box in that all expressions preceded by the ! (*doit*) character are evaluated, and all preceded by the @ (*unbox*) character are evaluated and unboxed. All other boxes and words remain the same.

Early implementations of BUILD were slow. Analysis showed that template parsing took most of the time. As part of the new evaluator system, I developed a BUILD compiler to speed up the BUILD operation.

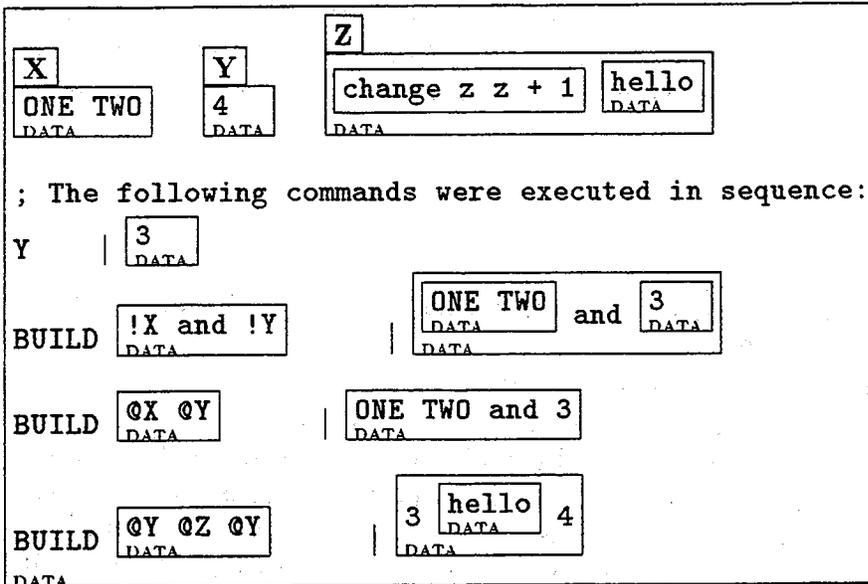


Figure 7.1: Examples of BUILD.

The implementation of BUILD is divided into two parts: a template walker and a compiler. The walker, due to Lay, recursively examines the template argument and constructs a Lisp function which takes a series of arguments and produces the Boxer object which BUILD returns. The arguments are the objects which are the result of the ! and @ evaluations. The walker also returns a list of the Boxer expressions to be evaluated, that is, the expressions which followed the special characters in the BUILD template.

The BUILD compiler, part of the evaluator system, creates an internal Boxer primitive which takes the evaluated arguments and passes them to the Lisp function returned by the BUILD walker. The BUILD primitive itself is merely a macro (recursive eval primitive) that expands into a call to this internal primitive with its arguments.

Nesting and combination of BUILD special characters ! and @ is allowed. Since these operations may result in multiple, successive evaluations, a straightforward implementation of BUILD would interleave this evaluation with the construction of the BUILD result.

The @ character in BUILD operates by "unboxing" the box it is given. If that operation results in multiple rows, the rows after the first are combined with the succeeding rows of the original template structure. If any of the resulting elements are DOIT boxes, they are evaluated, and their results inserted into the result. In the case of multiple @ characters, all but the last level of @ are handled by direct insertion of @ primitive into the expressions to be evaluated.

Figure 7.1 shows an example of a simple use of BUILD and a pathological case involving automatic @ execution of a procedure, side effects of external variables, and preservation of order of evaluation.

Bugs Named boxes inside the template appear correctly in the result of BUILD, but are not available for reference during the execution of any BUILD expressions. Note that this constraint runs counter to the Boxer intuitive notion of variable scoping, and is primarily an implementation-dependent artifact.

7.2 @

The @ operator of the BUILD primitive is also available as a general programming construct in Boxer. It is described in Section 2.2.

The @ primitive is more than a de-reference operator. It is designed to act like the BUILD primitive @ special character, only inside Boxer code. That is, an @ followed by a box or variable name first causes that item to be evaluated according to the normal evaluation rules. The *contents* of the resulting box are then used in place of the @ and its following item. (If there is more than one row, the rows are concatenated end-to-end.¹)

In this sense, @ is like the Common Lisp #. reader macro, which substitutes the result of the following expression into the code. In Boxer, however, the substitution happens at run time. Therefore, it more nearly resembles the MIT PDP-10 TECO² C-] (control right bracket) operator, which performs the same run-time substitution. As Boxer differs in syntax and semantics from TECO,³ the precise meaning of the command is different.⁴

If @ is used on something that evaluates to a DATA box containing only one element, itself a data box, then the behavior is the same as UNBOX of the same argument. This is true because DATA boxes are self-evaluating, so the extra evaluation which @ introduces has no effect; however, if the argument contains more than one DATA box, the result is the *last* DATA box. If such an @ combination occurs in a place where multiple inputs are expected, each DATA box will provide one input. Furthermore, a box that contains DOIT boxes or names will cause the execution of the DOIT boxes or named programs!

A straightforward application of the Boxer programming model obtains all these results immediately, and is much more instructive than the case-by-case analysis given here; however, this analysis is presented because even if the underlying implementation does not correspond exactly to the specification in the model, it must exhibit the appropriate behavior. As it happens, @ is implemented as a primitive operation that takes one input, with no special input flavor. It then copies the contents of its input, concatenating rows, and gives the evaluator a list of commands to be executed.

¹This particular behavior is still open to debate, but no better meaning has been determined.

²PDP-10 TECO, the Text Editing and Correction system, is the language in which the popular EMACS editor was first written. MIT PDP-10 TECO is not documented; PDP-6 TECO, its predecessor, is described in [27].

³Though perhaps not in spirit; Boxer is a language for operating on boxes, and its programs are boxes. It integrates the programming language and the editor. TECO is a language for operating on characters; its primitive operators are single characters, and its programs are strings of characters. PDP-10 EMACS, which is essentially a huge TECO program is even closer to the Boxer idea of an integrated system.

⁴In EMACS, the precise meaning of the command is affected by whether overwrite mode is in effect.

One behavior of @ that has proven difficult to implement is the interaction between @ and boxes containing named boxes. By the Boxer evaluation model, using @ to unbox a box containing named variables should result in the introduction of those named boxes into the currently-executing DOIT box. At top level, the editor performs this operation correctly; however, the evaluator does not presently implement it.

7.3 TELL

TELL is the primary means for accessing and changing under program control the variables in a box other than the current one. Similarly, TELL is used for object-oriented programming in Boxer.

Since the name TELL sounds declarative, Boxer provides the more interrogatory ASK synonym for the evaluation of variables and expressions that execute without side effect.

TELL requires two inputs: the first is a box or port, and the second is a series of Boxer commands, ending at the end of the row. TELL then executes the commands *as if* they had been typed in the argument box at top level.⁵ The value of the tell is the value, if any, returned by the last command.

TELL and BUILD-Flavored Inputs Since the local variables of the DOIT box which invokes TELL are not accessible to the commands which TELL is executing, some mechanism must be provided for passing in data to those commands. The present implementation of Boxer maintains the fiction that the second input to TELL is "BUILD-flavored." This means that the regular BUILD characters @ and ! are available, and that a BUILD operation is done in the environment of the caller. Thus, it is possible to pass in values of variables in the callers environment by preceding them with the ! character.

Lately, this interpretation has fallen into disfavor. The implementation section below shows that TELL maintains a pair of *previous environment* variables, which are queried to obtain the value of any expression which is preceded by an ! character, whether on a TELL line or not. Unfortunately, this means that lines inside procedures invoked indirectly by TELL also have access to the variables in the scope of the TELL command itself.

A new proposal⁶ offers the primitives MY and YOUR, which make legitimate the above behavior, and allow explicit specification of which variable is asked for in the TELL line. Although the implementation is straightforward, no integration with the Boxer model has yet been developed. Also, it is not clear which primitive should refer to which environment. At first glance, MY appears to mean the TELL environment, and YOUR the distant box; however, by the time the commands actually get executed, they will be executing in the distant environment. Texas Instruments Logo 99/4 faced similar problems: the

⁵Note that this is top level as defined in the Boxer model: see the implementation section for a fuller discussion of this subtle point.

⁶A. diSessa, personal communication, January 9, 1989.

MYNUMBER primitive, which returned the number of the current sprite (turtle), was changed to YOURNUMBER by TI staff (and further shortened to the cryptic YN).

Implementation TELL and its synonym ASK are implemented as recursive evaluation primitives. The first input is PORT-TO flavored, and the second input is commands forming the remainder of the line. TELL executes the commands *as if* they had been typed in the argument box at top level. That is, it binds the DYNAMIC-VARIABLES-ROOT to NIL and the STATIC-VARIABLES-ROOT to the new box, causes evaluation of the commands, and restores the state variables.

Multiple ! characters in TELL arguments are handled by an entry in the PDL stack frame for TELL which contains a pointer to the previous TELL frame. Each successive ! character causes the variable lookup mechanism (or MY primitive, in future implementations) to back up one level of TELL function calling.

Chapter 8

Conclusion

Boxer is an integrated computing environment for naïve computer users. The integration has made some parts of the implementation difficult, but has also made other parts easier. That Boxer is intended for naïve users called for a re-thinking of all aspects of traditional programming languages. This re-thinking has been a continuing process, involving a cycle of theorizing, discussion, testing, and implementation involving many people and taking over five years.

The development of the Boxer model of computation, proposed by Profs. diSessa and Abelson, has directly guided the implementation of the language. Yet many parts of it were, and some still remain, open to interpretation. In those cases, we have tried to appeal to the higher level Boxer principles, and where that failed, we adopted an experimental attitude.

The difficulties of creating the programming language Boxer lie to some degree in this interpretation, but in recent years have been more in the implementation of a language which is true to its model and at the same time which provides the functionality that users need. Half of the difficulty is the proper fitting of the language model and the model for the auxillary concepts; that is, the concrete model of variables, the semi-magical aspects of flavored inputs, and so on.

The other half of the difficulty is the actual implementation of an interpreter true to the model of execution. A process of give and take with definition of model, and with the definition of and interaction with the rest of the integrated system has been the rule, rather than the exception.

Bibliography

- [1] Seymour Papert. *Mindstorms*. Basic Books, New York, 1980.
- [2] M. L. Minsky. *A LISP Garbage Collector Algorithm Using Serial Secondary Storage*. Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 58, Cambridge, Massachusetts, 1963.
- [3] Edward H. Lay. *An Interactive Optics Workbook*. S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. 1981.
- [4] H. Abelson and E. H. Lay. "Virtual Copies of Visible Objects," unpublished paper.
- [5] D. Ploger and A. diSessa. "Rolling Dice: Exploring Probability in the Boxer Computer Environment," Experimental Report E1, U. C. Berkeley School of Education, 1987.
- [6] D. Ploger & A. A. diSessa, "Learning science in Boxer: Perspectives of researchers and teachers," Symposium to be presented at the Annual Meeting of the American Educational Research Association, March 1989.
- [7] Jeff Conklin. "Hypertext: An Introduction and Survey" in *IEEE Computer*, September 1987.
- [8] Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts, 1985.
- [9] Michael A. Eisenberg. *Boxer: An Integrated Scheme Programming System*. S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.
- [10] Andrea A. diSessa. "A Principled Design for an Interactive Computational Environment," in *Human-Computer Interaction*. 1(1), Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1985.
- [11] Andrea A. diSessa. "Reference and Data Construction in Boxer," in *Visual Aids in Programming*, P. Gorny and M. J. Tauber (eds.), Springer Verlag, Heidelberg, 1987.

- [12] E. A. Taft and T. N. Standish. *PPL User's Manual*, TR-21-74, Harvard University Center for Research in Computer Technology, 1974.
- [13] Richard M. Young. "Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices," in *Mental Models*, Dedre Gentner and Albert L. Stevens, (eds.). Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.
- [14] Vaughan R. Pratt. "Top Down Operator Precedence," in ACM Symposium on Principles of Programming Languages, Boston, MA, October, 1973.
- [15] Ira Goldstein, Henry Lieberman, Harry Boehner, and Mark Miller. "LLOGO: An Implementation of LOGO In LISP." Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 307A, Cambridge, Massachusetts, 1975.
- [16] H. Schweiker and K. Muthig. "Solving Interpolation problems with Logo and Boxer," in *Visual Aids in Programming*, P. Gorny and M. J. Tauber (eds.), Springer Verlag, Heidelberg, 1987.
- [17] D. Van Couvering, L. Klotz, and D. Ploger. "Toolboxes: A Method for Managing Computational Tools," School of Education, Division of Education in Math, Science, and Technology, Boxer Group, Technical Report T2. U. C. Berkeley, Berkeley, California, 1987.
- [18] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification XJ313, Document 88-002R, published as *Sigplan Notices*, Volume 23, Special Issue, ACM Press, New York, September 1988.
- [19] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor." Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo AIM-516, Cambridge, Massachusetts, 1981.
- [20] Jeremy Rochelle, "A Graphics System for BOXER" S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1984.
- [21] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. "Alto: A Personal Computer," Memo CSL-79-11, Xerox Palo Alto Research Center, Palo Alto, California, 1979.
- [22] Alan Kay, "Personal Computers," in *Scientific American*, September, 1979.
- [23] A. diSessa and H. Abelson, "Boxer: A Reconstructable Computational Medium," in *Communications of the ACM*. Volume 29, Number 9, p. 859, September 1986.
- [24] Daniel Weinreb and David Moon, "Flavors: Message Passing in the Lisp Machine." Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo AIM-602. Cambridge, Massachusetts, 1980.

- [25] M.I.T. Educational Computing Group, *Boxer II: Applications of an Integrated Computing Environment* (documentary video).
- [26] Thomas F. Knight, David A. Moon, Jack Holloway, Guy L. Steele, Jr., *CADR*, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo AIM-528, Cambridge, Massachusetts, 1979.
- [27] Peter Samson "PDP-6 TECO" Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo AIM-81, Cambridge, Massachusetts, 1965.
- [28] Hal Abelson, Nat Goodman, Lee Rudolph, *Logo manual*, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo AIM-313, Cambridge, Massachusetts, 1974
- [29] Hal Abelson, *Logo for the Apple II*, Byte/McGraw Hill, Peterborough, New Hampshire, 1981.
- [30] Patrick G. Sobalvarro, *A Lifetime-based Garbage Collector for Lisp Systems on General-Purpose Computers*, S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1987.