BOCHSER: AN INTEGRATED SCHEME PROGRAMMING SYSTEM

by

Michael Allen Eisenberg

B.A., Columbia College, 1978

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements of the Degree of

Master of Science

at the

Massachusetts Institute of Technology

AUGUST 1985

Signature of Author .. Signature redacted ....

Department of Electrical Engineering and Computer Science

Signature redacted 4 August 1985

Certified by ....

Thesis Supervisor

Accepted by Signature redacted ....

Chairman, Departmental Committee

BOCHSER: AN INTEGRATED SCHEME PROGRAMMING SYSTEM

by

Michael Allen Eisenberg

Submitted to the Department of Electrical Engineering and Computer
Science on August 9, 1985 in partial fulfillment of the requirements for
the Degree of Master of Science

Abstract

BOCHSER is a new programming system for the Scheme dialect of LISP which incorporates many of the editor features used by the Boxer language in development in the Educational Computing Laboratory at M.I.T. BOCHSER exploits the expressiveness of a high-resolution bitmapped video display to provide a Scheme interface that is at once comprehensible for students and useful for experienced programmers. The BOCHSER system is motivated by several principles: that a language interface should support, as much as possible, an appropriate abstract model for the language; that it should exhibit integration of normally distinct interface modes; and that it should provide an expressive and powerful programming medium. This report argues that BOCHSER does, in fact, adhere to these principles. By way of illustration, a sample session with BOCHSER is outlined, as well as several representative programming projects.

Thesis Supervisor: Professor Harold Abelson
Title: Professor of Electrical Engineering and Computer Science

ACKNOWLEDGMENTS

It is hard to convey the sort of debt that I owe to Harold Abelson and Andrea di Sessa. "Acknowledgment" doesn't really seem a sufficient word. I do think the happiest decision of my life was to pick up a book named "Turtle Geometry" in the library of the Rockefeller University; it was reading that book that convinced me to apply to M.I.T. Since coming here, Drs. Abelson and di Sessa have provided guidance, help, and most importantly two wonderful role models both as scholars and people. Thank you.

Jim Miller has been a friend and teacher. Conversation with him has constituted a significant portion of my education over the last three years. He has given generously to me of his time, his breadth and depth of knowledge, and his good company. I would also like to gratefully acknowledge his help in programming my first Scheme interpreter.

Thanks to Jim Fulton for spending months wrestling with BOCHSER and providing thoughtful feedback; to Roland Ouellette, whose knowledge of operating systems is without apparent bound, for his valuable advice; and to my officemates Linda Morecroft and Henry Lieberman for their good humor.

Besides Drs. Abelson and di Sessa, the Boxer system is largely the work of three amazing men: Gregor Kiczales, Leigh Klotz, and Ed Lay. Each of them is so smart, it's a little scary. I want to thank them for making Boxer (and BOCHSER) work. Similarly astounding are Chris Hanson and Bill Rozas, who taught me about M.I.T.'s Scheme system, and who supplied encouragement and fascinating conversation.

Bell Laboratories has provided generous financial support throughout the past two years of my graduate study; my warmest thanks to them.

Thanks also to Tim McNerney, Franklyn Turbak, Laura Bagnall, and Don Allen for feedback along the way.

Finally, my gratitude and love to my parents and sister.

Table of Contents

Table of Figures

Chapter One

Introduction

This report is a description of a new programming system for the Scheme dialect of LISP. The system -- called BOCHSER -- incorporates many of the editor features used by the Boxer language which is presently in development in the Educational Computing Laboratory at M.I.T. BOCHSER exploits the expressiveness of a high-resolution bitmapped video display to provide a Scheme interface that is at once comprehensible for students and useful for experienced programmers. Although the present implementation of BOCHSER is still slow in operation, and not without its flaws, it should represent a valuable prototype for future Scheme systems. The current system has been implemented by the author (using the already-existing Boxer editor) on a Symbolics 3600 Lisp Machine with one megaword of primary memory.

Why develop a system like BOCHSER? The most straightforward answer is that present Scheme systems are too difficult to work with, and fail to suggest the richness of the programming language that they represent. BOCHSER is an attempt to incorporate worthwhile ideas of interface design into the development of Scheme systems. More generally, experimentation with the interface should ideally be an ongoing enterprise in the life history of every programming language; and it is in this spirit that BOCHSER has been implemented.

It should also be stressed at the outset that the point of creating a system like BOCHSER is not to invent a new programming language, but rather to design an informative and powerful interface for an existing language -- namely, Scheme. Nevertheless, as this report will argue in its later chapters, a new interface can suggest new paradigms of programming even within an established language; and it can suggest extensions and future directions for that language as well. The development of BOCHSER, then, may hopefully stimulate continued evolution of (and creativity within) the Scheme language itself.

1.1 The Rest of This Report: an Outline

The remainder of this report will be devoted to a thorough discussion of the BOCHSER system, including the design principles that motivate it, its strengths and current weaknesses, and a few of the programming projects which the system suggests. Since BOCHSER is after all a Scheme programming system, this first chapter will conclude with a brief description of the Scheme language and the current Scheme system at M.I.T. The following chapter, by way of an introductory overview, focuses on the major tenets of interface design that underlie the BOCHSER system. In the third chapter, a sample session with BOCHSER is presented both to introduce the system and to give the reader an idea of how one actually works with it. The fourth chapter is an in-depth discussion of BOCHSER, relating its features to some of the design principles discussed earlier. Here, the system is also compared to the present Scheme interface. The next (fifth) chapter outlines three representative programming projects in BOCHSER: the intent is to convey the power and utility of the new interface. In the sixth chapter, the BOCHSER system is analyzed within the tradition of similarly "model-explicit" interfaces, including the current Boxer language system. Finally, the current status and potential future evolution of BOCHSER is discussed, and the report concludes with some general observations about the construction of man-machine interfaces and its place in programming language design.

1.2 The Scheme Programming Language

This section is intended to supply a few prefatory words about the Scheme programming language. Scheme, invented by Guy Lewis Steele Jr. and Gerald Jay Sussman, is a lexically scoped dialect of LISP. An early manifestation of the language is described in Steele and Sussman [Steele 78], and a more up-to-date description can be found in the report of a 1984 workshop meeting at Brandeis University [Clinger 85]. By and large, the BOCHSER system follows the presentation of Scheme used in Abelson and Sussman's textbook Structure and Interpretation of Computer Programs, which is the standard text in the introductory programming course (6.001) at M.I.T.

1.2.1 Scheme's Major Features

Although it is well beyond the scope of this report to provide a complete introduction to Scheme, some of the salient features of the language are worth mentioning here. Scheme is best understood as a dialect of LISP; it shares many of its most visible features -- e.g., prefix notation for procedure calls, a heavy reliance on spaces and parentheses as syntactic markers, and the primacy of lists as data structures -- with other LISP dialects. As a variant of LISP, Scheme is notable in several respects:

1. Its simple syntax,

2. The minimal number of primitive procedures and special forms which it employs,

3. Its blending of the notions of block structure and lexical scoping with LISP's traditional ideas of an interactive interpreter-based programming environment, and

4. Its notion of procedures and environments as *first-class objects* -- that is, objects that can be named, used as arguments to procedures, returned as the results of procedure calls, and stored within data structures such as lists and arrays. (The term *environment* here refers to a collection of name-value associations, or *bindings*, and the representation of Scheme environments will be described later in this chapter.)

The first two of these points reflect Scheme's minimalist style; they are graphically illustrated by the refreshingly small size of the language manual. Scheme's syntax is basically the same as LISP's, and derives its simplicity from these origins; but even beyond this, Scheme employs very little in the way of "syntactic sugar" -- special-purpose constructions that employ their own unique syntax. For instance, Scheme does not contain the PROG and DO forms found in most LISP dialects; the only common examples of such constructions in Scheme are the LET special form (analogous to LISP's) and an alternative syntax of DEFINE used for defining procedures. Moreover, throughout the language's history, the number of special forms in Scheme has traditionally been held to a very small number of extremely useful examples.[1]

The third point mentioned above relates to the fact that Scheme procedures may contain internally defined subprocedures; this idea is illustrated in the procedure below, which computes the exponent of a given base argument to some power *n*:

```
(define (expt base n)
   (define (expt-iter count result)
       (cond ((= count n) result)
            (else (expt-iter (1+ count) (* base result)))))
   (expt-iter 0 1))
```

As in the case of other block-structured languages, such as ALGOL 60 or PASCAL, the names of

[1] Briefly, a *special form* in Scheme is a language operation which does not obey the rules of procedure evaluation -- that is, an operation which does not necessarily evaluate all expressions in the "argument positions" which follow it. For instance, the IF special form is followed by three expressions: the first is evaluated, and depending on whether or not the result of that evaluation is the Scheme value FALSE, only one of the two remaining expressions will be evaluated. This distinction between procedures and special forms will be elaborated upon in the following section.

Scheme subprocedures are accessible only within the scope of the surrounding procedure; for instance, the name EXPT-ITER can only be used within the body of EXPT above. This use of block structure in an interpreter-based programming system enables Scheme to accommodate both those users who in classical LISP fashion like to create programs one procedure at a time, trying out new ideas while at the terminal, and those who like to fashion programs in a "top-down", structured manner. In fact, as with most apparent dichotomies in programming, the split between bottom-up and top-down design strategies is hardly a pure either-or decision: most programmers employ at least a little of both strategies in their work. Scheme programmers, then, have the best of both worlds -- they can add one or two new procedures directly, test them out, and then edit their code to impose a block structure on the new procedures that they have created. This is no doubt one of the major reasons that programmers familiar with Scheme often refer to it as their favorite language.

The final point above is the most interesting one from the standpoint of programming language design. In Scheme, procedures and environments are objects with much the same status as other, more conventional language objects -- like numbers. This is of course not to say that all these objects are semantically equivalent: one cannot, for instance, add procedures together like numbers. However, insofar as the language operations listed above -- naming, and so on -- are concerned, these objects are essentially interchangeable. In the 6.001 course, the inherent power of this idea is demonstrated by the use of first-class procedures and environments to illustrate a variety of important programming techniques, such as object-oriented programming, logic programming, delayed evaluation, and the development of package systems. The BOCHSER examples to be shown later will also emphasize the utility of this notion.

1.2.2 Scheme Procedure Objects and Environments

Since understanding the precise definitions of "procedures" and "environments" is an important element of expertise in Scheme, and since this understanding will be helpful to the reader in comprehending the description of BOCHSER to follow, this section will be devoted to elaborating on these definitions. Readers familiar with Scheme may choose to skip the following paragraphs; those interested in more detail are recommended to the Abelson and Sussman text.

In Scheme, an environment is a collection of name-value associations, or *bindings*. Environments are represented as a linked chain of *frames*, each of which contains zero or more bindings. Figure 1-1 shows an example; each box is a frame, and the arrows represent links between

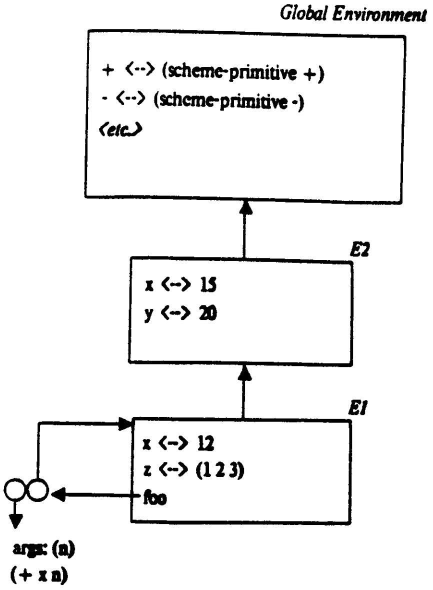


Figure 1-1:A Scheme environment diagram

frames. Within any frame, all of the bindings in that frame and those frames above it are accessible; for example, in Figure 1-1, the names X, Y, Z, and + (among others) are accessible from the frame labeled E1. Another way of phrasing this is to say that those names are *bound* in the environment represented by the frame labeled E1: X is bound to the value 12, Y is bound to 20, and so on. Name conflicts, when they exist, are resolved by taking the binding from the lowest frame possible in the chain; for instance, the binding for X in E1 is derived from the frame labeled E1, not that labeled E2. Note that within any particular frame, only one binding for a given name is allowed; it would be impossible to have a second binding for X in the frame labeled E1.

All Scheme computations proceed by the evaluation of Scheme expressions within environments of this kind. Evaluating the name X within environment E1, for example, would result in a value of 12. Evaluating the expression

```
(+ x y)
```

would result in a value of 32. Outside of some special rules relating to the application of Scheme procedures, which will be discussed below, the rules for Scheme evaluation are entirely analogous to those of LISP and will not be treated any further here.

There are still, however, two other points to note about Figure 1-1. First, the uppermost frame is labeled "Global Environment". This corresponds to the environment in which the names of Scheme's primitive procedures and pre-defined variables (like NIL) are bound. The global environment is the uppermost frame in every Scheme environment; thus, the bindings of primitive procedures are accessible in every environment. A second point is that each frame in Figure 1-1 represents one unique environment -- namely, that consisting of the frame itself and those frames in the chain above it. There is thus a one-to-one correspondence between Scheme environments and frames.

Scheme procedure objects are what other LISP dialects call closures -- that is, combinations of the procedure code and the environment in which that procedure was created. For example, FOO in Figure 1-1 is the name of a procedure associated with environment E1. The code of FOO indicates that when called with a numeric argument, the procedure binds this argument to the value N and returns the result of adding this value to the value of X. The way in which Scheme procedure objects are applied to arguments may be summarized as follows:

1. The argument expressions are evaluated.

2. A new frame is created in which the results of step 1) above -- that is, the values of the arguments -- are bound to the formal parameters of the procedure being applied.

3. The newly-created frame is linked to the frame associated with the procedure object. This linkage creates a new environment.

4. Within this newly-created environment, the body of the procedure is evaluated; the result of the last expression to be evaluated is the result of the procedure application.

Note that there is a certain recursive quality to the summary above. For instance, step 1, the evaluation of argument expressions, may itself require the application of a procedure to arguments, as in the expression

```
(foo (foo 1))
```

Similarly, step 4 often involves the evaluation of sub-procedure calls within the body of the procedure being applied.

A few additional details about this model of procedure application should also be mentioned. First, primitive procedures (like Scheme's addition primitive) do not have bodies to be evaluated; thus, the application of a primitive effectively proceeds invisibly after step 1 above. That is, the arguments to the primitive are evaluated, and then the primitive is applied directly to those arguments without the creation of any new frames. Another point is that Scheme contains a number of special forms, like DEFINE, LAMBDA, LET, QUOTE, IF, COND, and SET! -- all analogous to their LISP counterparts (SET! corresponds to SETQ) -- which obey their own rules of evaluation. The rules for each special form must be learned on an ad hoc basis. For example, SET! is followed by a symbol and an expression; the expression is evaluated, and the symbol (which should already be bound) has its most local binding changed to the result of the expression evaluation. As noted earlier, the number of Scheme special forms has been kept to a minimum. Finally, the third step listed above is what in essence embodies the lexical scoping of Scheme procedures: the result of this step is that free variables in the procedure body will obtain their values from the environment in which the procedure was created, not from the environment in which the application was performed.

If the reader new to Scheme finds this model of evaluation difficult to understand at first, he or she is not alone: this model (commonly referred to as the "environment model" of Scheme) comprises one of the most confusing and frustrating elements of the 6.001 course. It is in fact one of the purposes of BOCHSER to make this model more understandable and exploitable for Scheme novices. As we elaborate further upon the BOCHSER system, we will return to the environment model of evaluation in Scheme, and its uses in programming; still, although this report will strive for clarity in its examples even for those without Scheme programming experience, it would undeniably be helpful to the reader to have at least a partial familiarity with some LISP dialect, such as Common LISP, as background for the following chapters.[2]

1.2.3 The Current Scheme System at M.I.T.

At present, the Scheme system used in the intoductory course at M.I.T. has been implemented on Hewlett-Packard 9836 machines, and requires approximately 1.5 megabytes of primary memory. Past versions of the course have used a Scheme system implemented in MACLISP which runs under the TOPS-20 operating system on DECSystem 10 and 20 computers. Both of these Schemes are modeled on the MACLISP interface, in that both maintain a separate EMACS editor in which programs are developed. The Scheme interpreter itself (again in both instances) also takes its basic

[2] It should also be pointed out that the preceding description of Scheme semantics is based on the version of the language implemented at M.I.T. and described in the Abelson-Sussman text, since it is this version upon which BOCHSER is based. Other implementations may differ in some respects. For instance, some Schemes do not include first-class environments; others might include the DO special form mentioned earlier. The "Revised Revised Report" on Scheme [Clinger 85] provides a more formal description of the language, intended to take varying implementations or "sub-dialects" of the language into account.

interface ideas from MACLISP: expressions are typed in and evaluated once complete. In the TOPS-20 Scheme, evaluation takes place automatically upon completion of the expression; in 9836 Scheme, the user presses an EXECUTE key to evaluate a previously typed expression. In this report, references to the "standard Scheme interface" are meant to indicate features common to both systems, unless otherwise noted.

Chapter Two

Motivation: An Integrated, Powerful, Model-Explicit System for Scheme

There are several major motivating themes behind the implementation of BOCHSER as it has evolved. These themes -- "model-explicitness", integration, and programming power -- form the basis of the system's interface-design philosophy. Since these three notions will surface repeatedly in the ensuing chapters, this chapter will be devoted to elaborating briefly upon each of them.

2.1 Support for a Programming Language Model

One of the primary purposes of the BOCHSER system is to supply its users with an interface that provides explicit and useful information about the Scheme "language model". Among cognitive scientists, there is a growing consensus that people who learn a programming language acquire in the process a mental model of the computer -- that is to say, an internal semantic representation of the primitive objects and operations of the computer as employed by the language. The model that people acquire need not be (and typically isn't) at the level of actual hardware; often it is abstract, incomplete, consciously metaphorical, or even inaccurate. For example, people might view storage locations as "containers" with special shapes that accommodate different types of data; a computer's primary memory might be viewed as an "erasable scoreboard"; procedure invocation might be represented as the assignment of a certain task to a "little man" who may call other little men (subprocedures) in the course of his work.[3] Beyond this consensus -- i.e., that mental models of computers as derived from programming languages do exist -- there are vast numbers of still-open questions: How does one characterize mental models in general, and models of computers (and computer languages) in particular? How do programmers' models change with age and experience? How do they vary from one programming language to another? What kind of computer models do people come up with spontaneously, in the absence of specific instruction? Are explicitly presented models useful for people learning a programming language? If models are presented to a language

[3] The first two examples are taken from [Findlay 81] and [Mayer 75]; the last is due to Seymour Papert for use in teaching LOGO, and is described in [Weir 85].

learner, what are the criteria for a "good" (pedagogically speaking) model?

As of now, all these questions are the subject of active research, and the state of knowledge regarding these "language-appropriate" computer models can be fairly described as embryonic. Young [83], for example, writes

> "Although it is widely accepted that people's ability to use an interactive device depends in part on their having access to some sort of a mental model... the notion of the 'user's conceptual model'... remains a hazy one, and there are probably as many different ideas about what it might be as there are people writing about it."

Even so, there is broad agreement -- supported by some corroborative experimental work -- that 1) explicit presentation of a concrete computer model is of benefit to someone learning a new computer language, and 2) creative or expert use of a programming language (as opposed to stereotyped "rote usage" of standard patterns) makes essential use of a powerful, language-appropriate computer model. Du Boulay, O'Shea, and Monk [81] write, for example,

> "Novices should be introduced to programming through languages that embody simple notional machines with the facilities for making certain of the actions of the notional machine open to view."

A "notional machine", in their description, is

> "an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed. That is, the properties of the notional machine are language, rather than hardware, dependent."

In a similar vein, Mayer [79] suggests that instruction in BASIC should make essential use of what he calls "transactions":

> "A transaction is a unit of programming knowledge in which a general operation is applied to an object at a general location... Transactions provide a means, for the novice, of 'explaining' what is going on inside the computer when a particular statement is executed and of relating the new technical language that he or she must learn to the general operations, locations, and objects that he or she is already familiar with."

He concludes,

> "To enhance learning in novices, a concrete or familiar model of the computer should be introduced early in learning and used throughout learning."

Similar ideas crop up consistently in studies of programmer behavior. Mawby et.al. [84], in a series of interviews with children to determine their understanding of LOGO, attribute much of the children's weakness in programming to an imperfect model of the language. Adelson and Soloway [84] present a picture of expert programming that can be described as a sort of "stepwise refinement of models" -- that is, that construction of a program proceeds from a top-level model of the working program to successively less abstract models, culminating at the point where procedures may be written using the primitive elements of the language model itself. Halasz and Moran [83], in studying people's use of a stack calculator, compared two groups of subjects: one trained solely via the presentation of examples, and another given in addition to the examples an explicit model of the calculator's operation. They found "the model provides no advantage on problem solving performance for the routine and combination problems... However, for the invention (i.e., novel) problems, the model users performed considerably better than the no-model users." Finally, Mayer [81] again reports on a series of experiments in which language-appropriate computer models were tested for their value as "advance organizers" -- that is, frameworks into which later programming instruction could be assimilated.[4] He concludes,

> "If your goal is to produce learners who will not need to use the language creatively, then no model is needed. If your goal is to produce learners who will be able to come up with creative solutions to novel (for them) problems, then a concrete model early in learning is quite useful."

Despite this widespread acknowledgment of the importance of language models in creative programming, however, most language interfaces are remarkably sparse in their support of these models. Presumably, when a language is implemented, the language designer (or interface designer, if the language is not a new one) has in mind some sort of ideal "designer's model" of the language that ought to be conveyed to users. The implementation, one would hope, would be effective in conveying this model; the interface would be designed with this communication as its primary aim. This sort of view is espoused by Norman [83]:

> "In the ideal world, when a system is constructed, the design will be based around a conceptual model. This conceptual model should govern the entire human interface with the system, so that the image of that system seen by the user is consistent, cohesive, and intelligible. I call this image the *system image* to distinguish it from the conceptual model upon which it is based, and the mental model one hopes the user will form of the system.... the instructors of the system would teach the underlying conceptual model to the user and, if the system image is consistent with that model, the user's mental model will also be consistent."

All of this is very reasonable advice; the surprising fact is that it is so rarely adhered to. There is nothing in most standard BASIC interfaces that represents a "location" -- that is, nothing on the screen which changes when one performs a LET statement interactively. There is nothing in the standard LOGO interface that reflects the "little man" model of control (short of, say, TRACE facilities, which are rather cryptic in their own right); nor is the distinction between "editor world" and "interpreter world" ever made clear in the LOGO model that one would want to convey.

[4] Additional discussion about language models can be found in [Carroll 85], [di Sessa 85a], and [di Sessa 85b].

Nothing in the standard LISP interface represents a list -- there is no place on the screen to look for the result of a RPLACA operation. In all of these cases, the style of the user's interaction with the system is essentially one of "evaluation of expressions" -- one types in an expression, or series of expressions, and watches the result -- but always the evaluating mechanism, and the world of abstract objects in which this mechanism works, remain invisible.

There is little reason for this "expression-oriented" interface style to remain the norm; it is essentially an artifact of the limited teletype-based programming environments of early computers. Users of machines with high-resolution bitmapped screen displays quite simply can do better. Over the years a number of (usually experimental) programming systems have attempted to provide more "transparent" interfaces (in the sense that the "system image", in Norman's phrase, is more openly representative of the designer's system model); and it's fair to say that this value of transparency represents an increasingly important trend in computer interface design.

One purpose of the BOCHSER system, then, is to support a rich and appropriate model of the Scheme language for the beginning Scheme programmer. That is, the designer's model, "system image", and (eventual) user's model of the language are intended to be consistent: the interface openly reflects the designer's model of Scheme and encourages the development and exploitation of that model by the user.

2.2 Integration in Language Interfaces

A second major motivating idea behind BOCHSER is that of integration. Rather than have a variety of subsystems -- editor, interpreter, debugger, and so on -- each with its own particular style of interface, the BOCHSER system unifies these functionalities within one internally consistent, simple programming environment. Much of BOCHSER's quality of integration, as this report will show, is a result of its use of the Boxer editor -- indeed, BOCHSER derives from Boxer a principle design concern that di Sessa [85b] refers to as *detuning*:

> *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.

A companion notion to detuning, as described by di Sessa, is that of *diffusing functionality* -- the idea that one construct may serve several functions -- and here too the BOCHSER system shares in the advantages of Boxer. It is in large part through the Boxer editor's exemplification of "detuning" and "diffusing functionality" that BOCHSER is able to achieve some measure of integration. We will return to these design principles in greater detail in the fourth chapter of this report, after presenting some necessary background about the BOCHSER system itself.

It should be noted, of course, that the concept of integration is hardly a new or unpopular one. Within the realm of applications programming, "integration" has by now become something of a buzzword; in this context, it usually implies that a number of separate applications programs (e.g., text editors, chart-creating programs, spreadsheets) will maintain consistent interface conventions and will all be able to work with similarly-formatted data. As for programming languages, the SMALLTALK system was founded in part on the idea of integration, and Tesler [81] has a spirited defense of "modeless programming" in his description of SMALLTALK-80; the INTERLISP system as well is often invoked as an example [Teitelman 81]. More recently, Heering and Klint [85] have written a comprehensive argument on behalf of what they call "monolingual programming environments". Their introduction represents a good case study in current support for integration:

> "This hodgepodge of languages [within a typical computer system] makes fast and efficient interaction with the system difficult. There are several reasons for this. The first and most obvious one is that the user has to remember so many different details. This would be acceptable if the domains of discourse corresponding to the various interactive modes were sufficiently distinct. But, at least for some modes, the opposite is true. There are profound analogies between command mode, programming mode, and symbolic debugging mode, but in most existing systems a substantial intellectual effort is required to see them, because they tend to be obscured by the differences between the various languages."

Despite the acknowledged trends in commercial applications software, and despite the long-standing atmosphere of (at least academic) support implied by the examples of Smalltalk, Interlisp, and Boxer, truly integrated programming environments still remain few and far between. BOCHSER is an attempt to provide such an environment for Scheme programmers.

2.3 Expressive Power in a Programming Environment

Historically, many of the systems that could be described as "model-explicit" have been designed primarily as pedagogical devices -- that is to say, they were seen as useful for students learning the language but not for advanced programmers, for whom the model was well-understood and efficiency was of greater concern. This report will argue that the BOCHSER system, by virtue of its close adherence to the Scheme model, is of interest to experienced Scheme programmers as well as novices. The reason for this is that the BOCHSER interface suggests programming strategies that are compatible with the Scheme model, but which the standard Scheme interface fails to highlight (or actively discourages). In this context, the reader may recall the final quote from Mayer above: he indicates that creative programming -- not merely the assimilation of a language -- depends on the mastery of a language-appropriate model. It should hardly be surprising, then, that an interface which encourages the use of hitherto unexploited elements of a language model can spark inventiveness among its users.

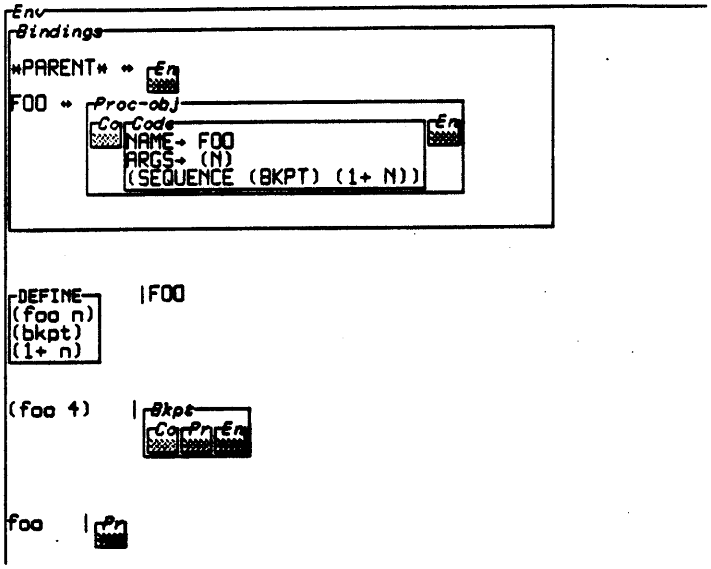
As was noted earlier, these three notions -- model-explicitness, integration, and expressive power -- form the major design principles behind the BOCHSER system. The next step is to examine the system itself; and the following two chapters are devoted to that purpose.

Chapter Three

BOCHSER: An Overview and Sample Session

This chapter will provide a general introduction to the BOCHSER system; our strategy will be to proceed through a sample session with BOCHSER, introducing salient features of the system as we go. (Another "tour through BOCHSER", though with some differences in emphasis, may be found in Jim Fulton's review of the system.[Fulton 85])

The text-editing commands of the BOCHSER editor are based on Emacs [Stallman 81], and are largely derived from the existing command set of the Boxer system (cf. also [Boxer 84]). A list of these editor commands is included as an appendix to this report; outside of those elements used to deal with boxes and objects specific to BOCHSER -- which elements will be introduced in the following discussion -- there is no need to expend any more time on the text-editing features of the system here.

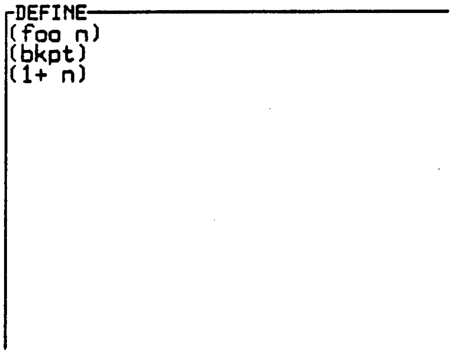


```
*PARENT* ⇒
FOO ⇒
NAME→ FOO
ARGS→ (N)
(SEQUENCE (BKPT) (1+ N))

(foo n)
(bkpt)
(1+ n)          |FOO

(foo 4)         |

foo             |
```

Figure 3-1:A BOCHSER screen

3.1 Boxes

The most striking and pervasive feature of the BOCHSER (and Boxer) editor is its use of the *box* as an editor object. Boxes come in a variety of types in BOCHSER: Figure 3-1 shows a screen configuration in which we can see boxes of type environment (the box that surrounds all the others), bindings, procedure-object, code, breakpoint, and syntax (this is the box with the label DEFINE in its upper left-hand corner). All of these types will be fully explained shortly, but a few words about box-editor commands in general are in order here. Boxes may be shown on the screen in three sizes: shrunken, normal, and full-screen. A shrunken box (like the one at the bottom of Figure 3-1) appears as a little gray rectangle on the screen; shrunken boxes may be expanded to normal size (like the DEFINE syntax box), in which state the box is made large enough so that all of the text inside it may be shown. Normal-sized boxes may be further expanded to full-screen size. The user can change the sizes of boxes on the screen -- shrinking and expanding them as desired -- either by using mouse commands (employing the 3600's mouse) or via keyboard commands. For example, to expand the DEFINE box to full-screen size, one might use the cursor associated with the mouse ("mouse arrow") to point to that box and expand it. Alternatively, one might use keyboard commands to move the cursor inside the DEFINE box (there are editor commands which move the editor cursor in and out of boxes) and then expand the box. (See Figure 3-2.)



```
DEFINE
(foo n)
(bkpt)
(1+ n)
```

Figure 3-2: The DEFINE box expanded to full-screen size

In general, a box is treated like a screen character. If the editor cursor has been placed on a line which contains a box as a subsequent character, and one now inserts text at this point, then the box on the line moves to the right exactly like all the other subsequent characters on the line. Similarly, if one uses editor commands to advance the cursor one character at a time, then boxes will be "advanced over" just as other characters are.

Most box types in BOCHSER are "read-only" -- that is to say, one cannot insert text into the interior of most boxes. The only species of boxes that allow the user to type input inside them are environment and syntax boxes. This point will be elaborated upon later; the reason it is mentioned now is simply to note that when one inserts text into (or deletes text from) a normal-sized box, the box borders adjust to accommodate the new text. With the exception of the restriction on "input-accepting box types", and the box types themselves, virtually all of these features are derived from the original Boxer system; further description of general box-editing commands may be found in [di Sessa 85b], [Boxer 84], and in documentary videos prepared by the Educational Computing Group at M.I.T. [Boxer 83, 85]

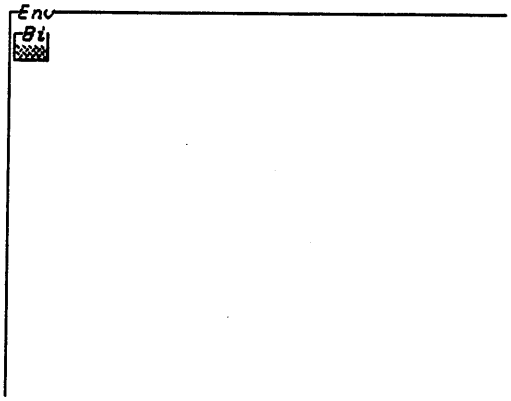
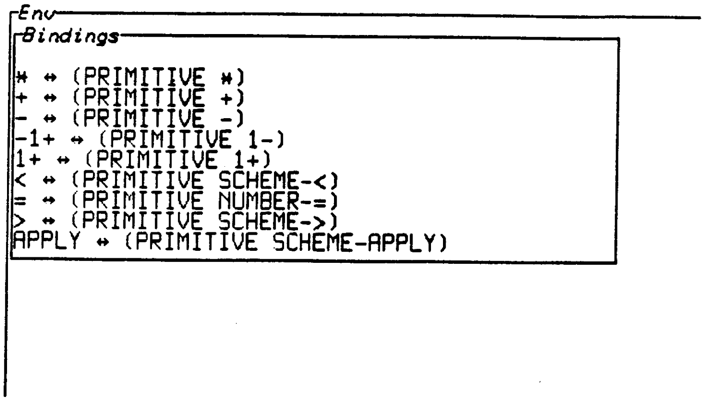


Figure 3-3:A user's first view of the BOCHSER screen

3.2 Sample Session, part 1: Environment and Bindings boxes

We can begin our sample session with the first view of BOCHSER that the user encounters (Figure 3-3). We are now inside a full-screen box with an "Env" header -- an environment box. Every environment box, including this one, contains a bindings box on its first row; when environment boxes are first seen, their bindings boxes are shown in "shrunken" size, by default.

To examine the contents of the bindings box of this environment, we can expand it to normal size. The definition of "normal size" for bindings boxes is actually a little different than that for all others; a "normal-sized" bindings box is kept at one fixed size. Thus, if the box contains a large

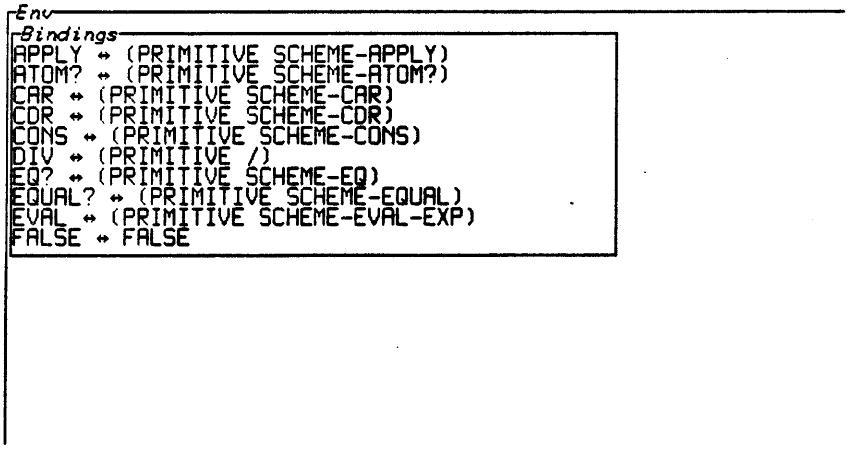


```
* ↔ (PRIMITIVE *)
+ ↔ (PRIMITIVE +)
- ↔ (PRIMITIVE -)
-1+ ↔ (PRIMITIVE 1-)
1+ ↔ (PRIMITIVE 1+)
< ↔ (PRIMITIVE SCHEME-<)
= ↔ (PRIMITIVE NUMBER-=)
> ↔ (PRIMITIVE SCHEME->)
APPLY ↔ (PRIMITIVE SCHEME-APPLY)
```

Figure 3-4:The bindings box expanded to "normal" size

number of rows, not all of them may be visible on the screen. In Figure 3-4, we see that the bindings box has been expanded to its fixed "normal" size; not all its contents are yet visible.

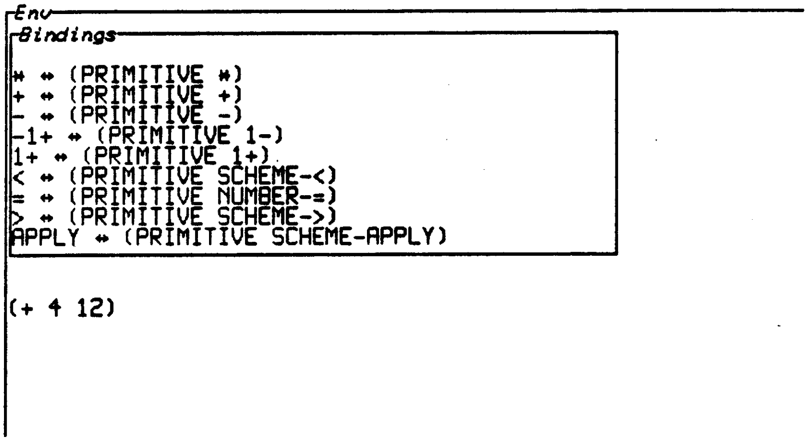
An examination of the bindings box in Figure 3-4 shows that we are in BOCHSER's global environment. Each row in the bindings box represents one binding in this environment. The bindings are arranged in alphabetical order (by variable name); and we can look at more bindings by scrolling the view in the bindings box (as in Figure 3-5). Examining the bindings box at this stage tells us, for example, that ATOM? and APPLY are already-defined symbols, bound to primitive Scheme procedures.



```
APPLY ↔ (PRIMITIVE SCHEME-APPLY)
ATOM? ↔ (PRIMITIVE SCHEME-ATOM?)
CAR ↔ (PRIMITIVE SCHEME-CAR)
CDR ↔ (PRIMITIVE SCHEME-CDR)
CONS ↔ (PRIMITIVE SCHEME-CONS)
DIV ↔ (PRIMITIVE /)
EQ? ↔ (PRIMITIVE SCHEME-EQ)
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
```

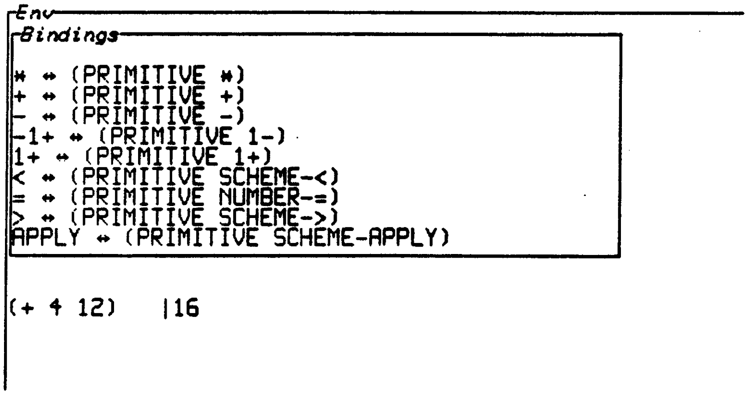
Figure 3-5:The bindings box after scrolling

The reader may recall from the discussion in the first chapter that in Scheme, an environment is represented as a succession, or chain, of linked frames. Consequently, there is a one-to-one correspondence between environments and frames; each environment may be uniquely associated with the lowest frame in its representative succession-of-frames. In the BOCHSER interface, the bindings box of each environment box shows the bindings present in the frame which corresponds to the represented environment. (The reader might thus be equally comfortable thinking of environment boxes as "frame boxes".) In our present example in Figure 3-5, we are in an environment box corresponding to BOCHSER's global environment; the bindings shown in the bindings box, then, are the "global bindings".



```
Env
Bindings
* ↔ (PRIMITIVE *)
+ ↔ (PRIMITIVE +)
- ↔ (PRIMITIVE -)
-1+ ↔ (PRIMITIVE 1-)
1+ ↔ (PRIMITIVE 1+)
< ↔ (PRIMITIVE SCHEME-<)
= ↔ (PRIMITIVE NUMBER-=)
> ↔ (PRIMITIVE SCHEME->)
APPLY ↔ (PRIMITIVE SCHEME-APPLY)

(+ 4 12)
```

Figure 3-6:Typing an expression to evaluate



```
Env
Bindings
* ↔ (PRIMITIVE *)
+ ↔ (PRIMITIVE +)
- ↔ (PRIMITIVE -)
-1+ ↔ (PRIMITIVE 1-)
1+ ↔ (PRIMITIVE 1+)
< ↔ (PRIMITIVE SCHEME-<)
= ↔ (PRIMITIVE NUMBER-=)
> ↔ (PRIMITIVE SCHEME->)
APPLY ↔ (PRIMITIVE SCHEME-APPLY)

(+ 4 12)    |16
```

Figure 3-7:Evaluating an expression

The area underneath the bindings box -- that is, the rest of the environment box's interior -- is a "Scheme interpreter area". In this area, the user may type Scheme expressions and have them evaluated. For example, Figure 3-6 shows a Scheme expression typed into the global environment box; Figure 3-7 shows the configuration of the screen after evaluating this expression. Expressions are evaluated by placing the cursor at the closing parenthesis of the expression and pressing an "execute" key (on the 3600, the LINE key serves this function); atomic expressions may be evaluated by pressing the execute key on a row containing only the atom.

It is worth summarizing the aspects of the BOCHSER interface that we have seen so far. Environment boxes represent environments; each environment box contains a bindings box; the rows of the bindings box represent the bindings in the frame corresponding to the relevant environment. Bindings, as the preceding figures have shown, are simply name-value pairs. Finally, each environment box contains its own local Scheme interpreter area.

3.3 Sample Session, part 2: Defining Variables

To continue our session with BOCHSER, we can observe what happens when we execute a DEFINE expression. In Figure 3-8, the user types in the expression

```
(define foo 5)
```

and executes it. The returned value of this expression is the symbol FOO; this is in keeping with the standard Scheme interface. More interestingly, however, a look at the bindings box shows that a binding for the symbol FOO has now been added to the global environment. The row

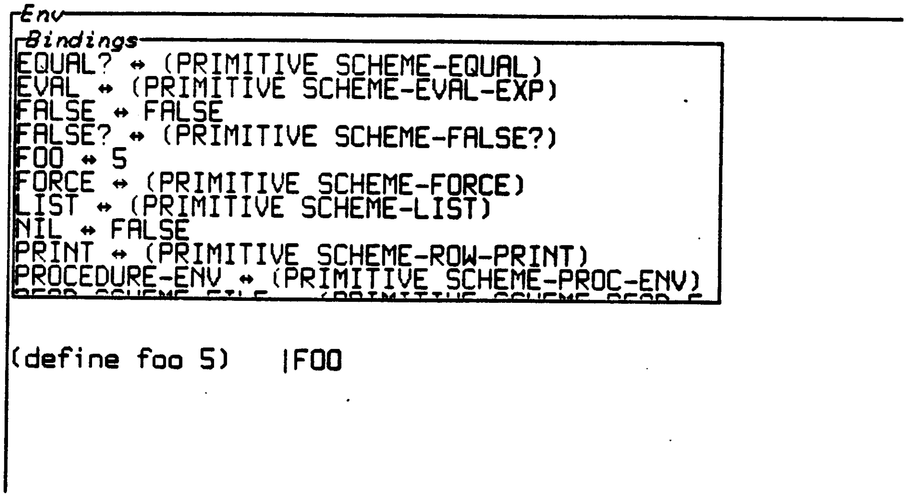
```
FOO <--> 5
```

in the bindings box indicates that FOO is now bound to the value 5. In Figure 3-9, the user types in the name FOO and evaluates it; as expected, the returned result is 5.

The bindings box, then, tells us the present <u>state</u> of the BOCHSER world at any particular time. This information is updated after every BOCHSER evaluation. For instance, in Figure 3-10, the user types in the expression

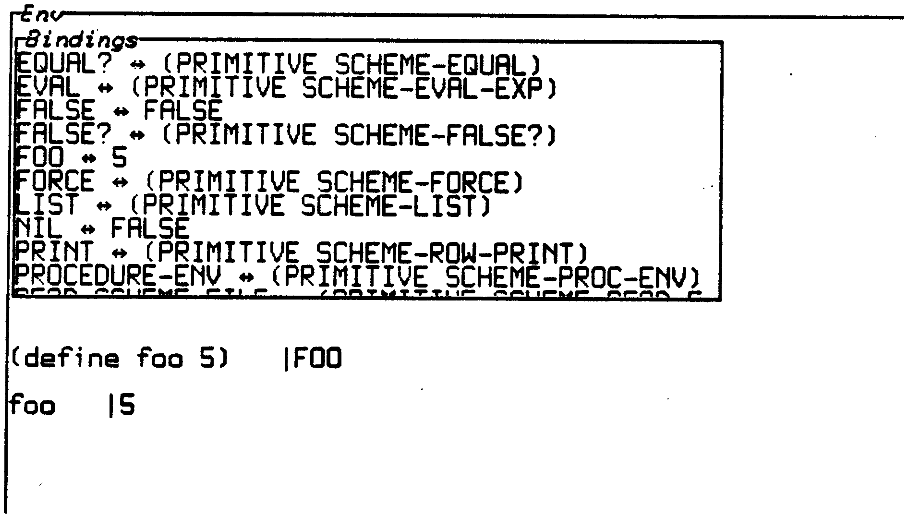
```
(set! foo 6)
```

Evaluating this expression causes BOCHSER to return the old value of FOO, namely 5, and to update the bindings box in accordance with the new binding for FOO.



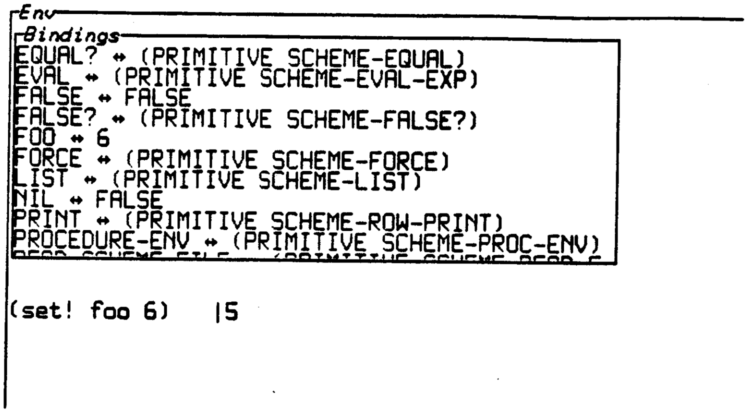
```
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
FALSE? ↔ (PRIMITIVE SCHEME-FALSE?)
FOO ↔ 5
FORCE ↔ (PRIMITIVE SCHEME-FORCE)
LIST ↔ (PRIMITIVE SCHEME-LIST)
NIL ↔ FALSE
PRINT ↔ (PRIMITIVE SCHEME-ROW-PRINT)
PROCEDURE-ENV ↔ (PRIMITIVE SCHEME-PROC-ENV)

(define foo 5)    |FOO
```

Figure 3-8:Defining FOO



```
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
FALSE? ↔ (PRIMITIVE SCHEME-FALSE?)
FOO ↔ 5
FORCE ↔ (PRIMITIVE SCHEME-FORCE)
LIST ↔ (PRIMITIVE SCHEME-LIST)
NIL ↔ FALSE
PRINT ↔ (PRIMITIVE SCHEME-ROW-PRINT)
PROCEDURE-ENV ↔ (PRIMITIVE SCHEME-PROC-ENV)

(define foo 5)    |FOO

foo    |5
```

Figure 3-9:Evaluating FOO



```
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
FALSE? ↔ (PRIMITIVE SCHEME-FALSE?)
FOO ↔ 6
FORCE ↔ (PRIMITIVE SCHEME-FORCE)
LIST ↔ (PRIMITIVE SCHEME-LIST)
NIL ↔ FALSE
PRINT ↔ (PRIMITIVE SCHEME-ROW-PRINT)
PROCEDURE-ENV ↔ (PRIMITIVE SCHEME-PROC-ENV)

(set! foo 6)    |5
```

Figure 3-10:Resetting FOO

To carry the example one step further, in Figure 3-11, the user creates a new environment by typing the expression:

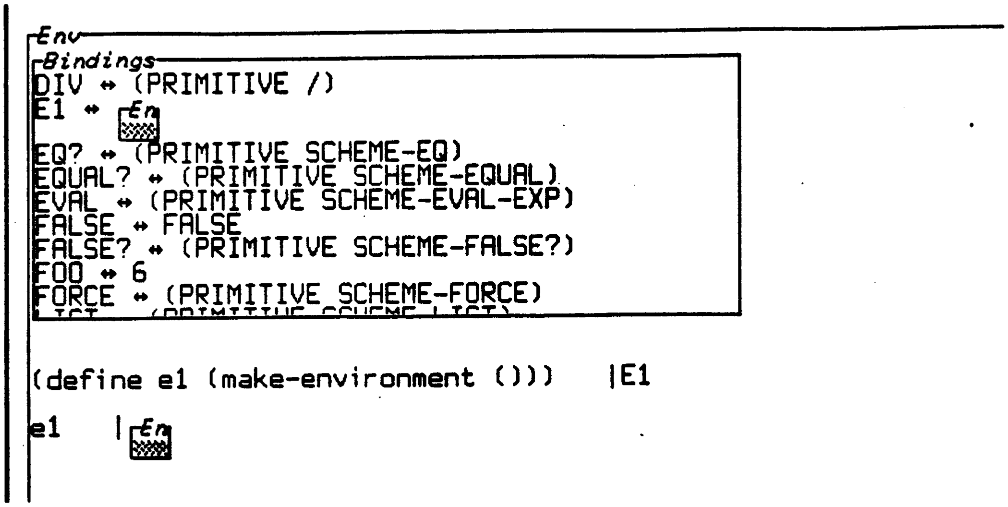
```
(define e1 (make-environment ()))
```

We can see a new binding created for the symbol E1, and a returned value of E1 on the screen. Now, by evaluating the name E1 on the next line of Figure 3-11, the user can access the newly created environment.

In Figure 3-12, the environment box returned earlier is expanded and the user has opened up its bindings box. Initially, the only binding present in the new environment is for the special symbol *PARENT*. This symbol is bound to the parent environment; thus, by opening up the environment box corresponding to *PARENT*, we can view the bindings of the parent frame -- here, the bindings of the global environment. Notice that there is no binding for *PARENT* in the global environment, since there is no "parent frame" for this environment.

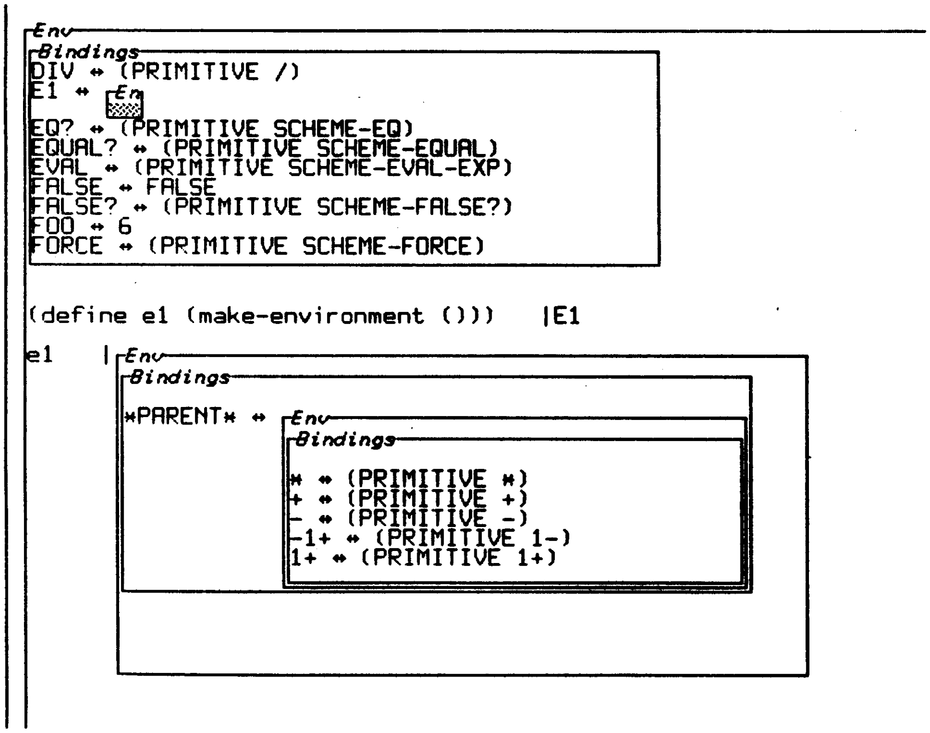
Figure 3-13 depicts what happens when the user types in a definition for FOO in the newly opened environment. Here, FOO is bound to 10 in the new environment; but it is still bound to 6 in the global environment. As Figure 3-13 shows, if we evaluate the symbol FOO in the interpreter area of the global environment, the returned result is 6, whereas in the new environment, the result is 10.

An interesting aspect of the BOCHSER system is illustrated in Figure 3-14. Here, we have defined E2 to have the same value as the environment bound to E1. Figure 3-14 shows the definition



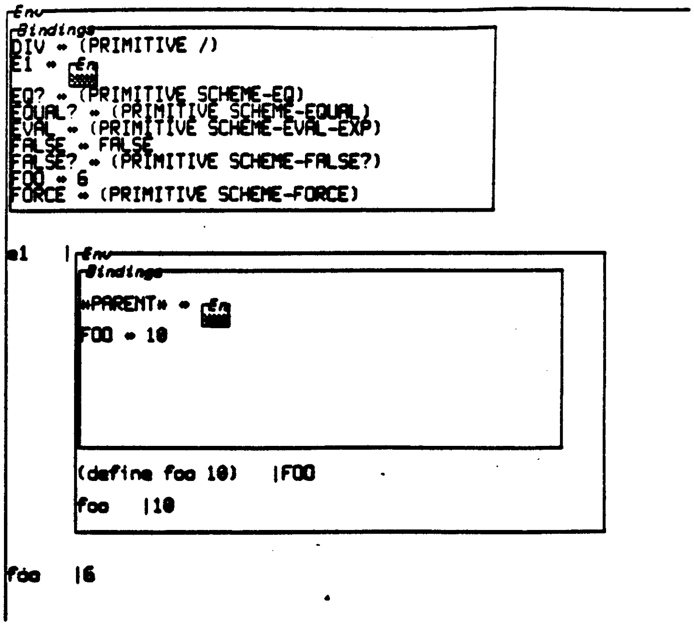
```
DIV ↔ (PRIMITIVE /)
E1 ↔ Env
EQ? ↔ (PRIMITIVE SCHEME-EQ)
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
FALSE? ↔ (PRIMITIVE SCHEME-FALSE?)
FOO ↔ 6
FORCE ↔ (PRIMITIVE SCHEME-FORCE)
LIST ↔ (PRIMITIVE SCHEME-LIST)

(define e1 (make-environment ()))    |E1

e1    |Env
```

Figure 3-11:Making a new environment



```
DIV ↔ (PRIMITIVE /)
E1 ↔ Env
EQ? ↔ (PRIMITIVE SCHEME-EQ)
EQUAL? ↔ (PRIMITIVE SCHEME-EQUAL)
EVAL ↔ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ↔ FALSE
FALSE? ↔ (PRIMITIVE SCHEME-FALSE?)
FOO ↔ 6
FORCE ↔ (PRIMITIVE SCHEME-FORCE)

(define e1 (make-environment ()))    |E1

e1    |Env
        Bindings
        *PARENT* ↔ Env
                     Bindings
                     * ↔ (PRIMITIVE *)
                     + ↔ (PRIMITIVE +)
                     - ↔ (PRIMITIVE -)
                     -1+ ↔ (PRIMITIVE 1-)
                     1+ ↔ (PRIMITIVE 1+)
```

Figure 3-12:Opening the new environment



```
DIV ∙ (PRIMITIVE /)
E1 ∙ Env
EQ? ∙ (PRIMITIVE SCHEME-EQ)
EQUAL? ∙ (PRIMITIVE SCHEME-EQUAL)
EVAL ∙ (PRIMITIVE SCHEME-EVAL-EXP)
FALSE ∙ FALSE
FALSE? ∙ (PRIMITIVE SCHEME-FALSE?)
FOO ∙ 6
FORCE ∙ (PRIMITIVE SCHEME-FORCE)

e1    |
        *PARENT* ∙ Env
        FOO ∙ 10

        (define foo 10)    |FOO
        foo    |10

foo    |6
```

Figure 3-13:Redefining FOO in the new environment

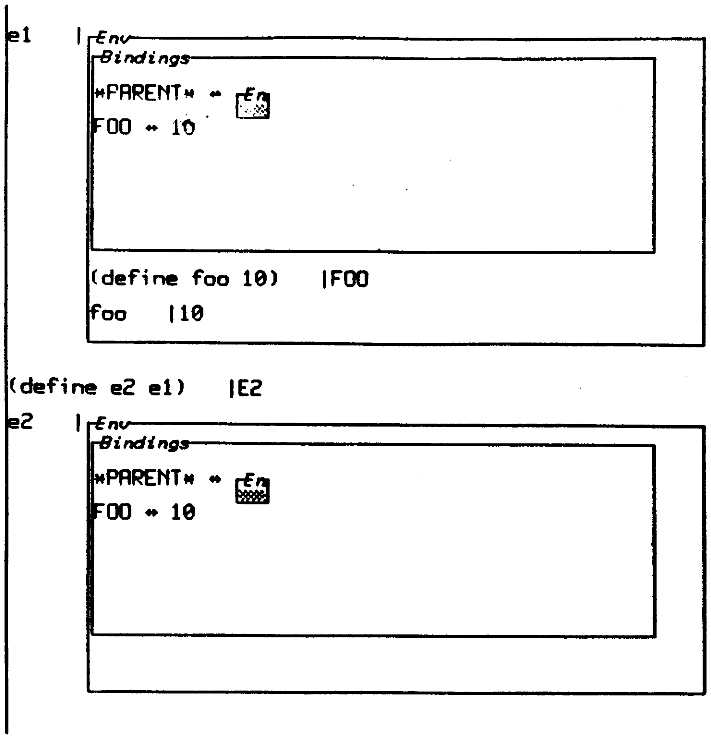
of E2, and, on the subsequent line, the result of evaluating the name E2. We have expanded the bindings box of the E2 environment box; as expected, it contains the same bindings as the El environment box, since these symbols are bound to the same environment object. Another way of phrasing this is to say that the values of the symbols El and E2 are EQ. In BOCHSER, two environment boxes that correspond to the same object will always have identical bindings boxes; however, their interpreter areas are independent. Thus, we can type into the "El environment box" without seeing any change in the "E2 environment box". The equivalence of the represented environments can be seen in Figure 3-15; here, the user types the expression

```
(define bar 12)
```

into the "El box". When this expression is evaluated, a binding for BAR is added to both the "El bindings box" and the "E2 bindings box". Similarly, if the user were now to perform a SET! operation on FOO or BAR within either environment box, the bindings boxes of both would change.

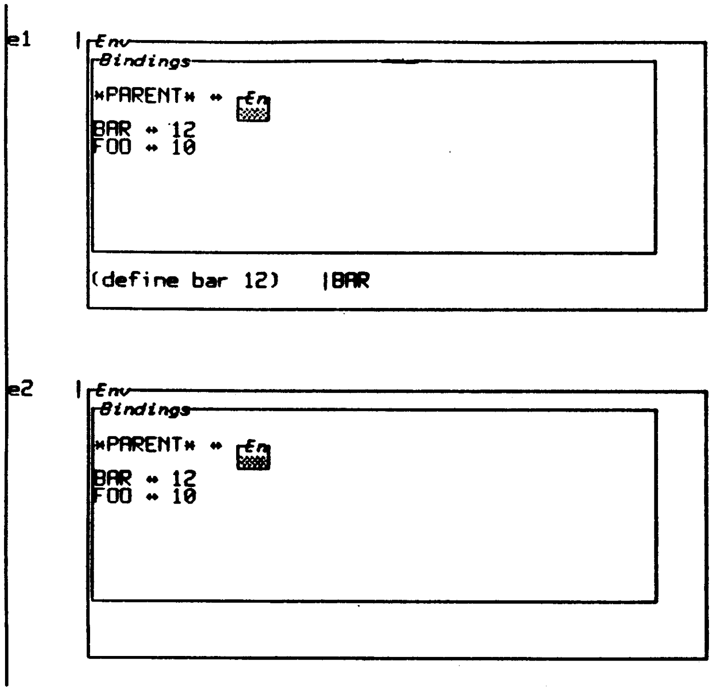
3.4 Sample Session, part 3: Procedure Objects

In Figure 3-16, we have expanded one of the new environment boxes from the previous section to full-screen size, so that we can work with a relatively "uncluttered" bindings box. The figure shows that the user has typed in the expression



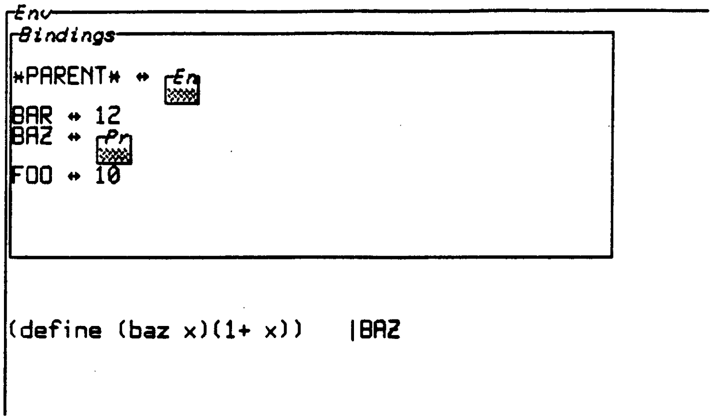
```
e1  | Env
      Bindings
      *PARENT* ↔ En
      FOO ↔ 10

      (define foo 10)   |FOO
      foo   |10
(define e2 e1)   |E2
e2  | Env
      Bindings
      *PARENT* ↔ En
      FOO ↔ 10
```

Figure 3-14:Two environment boxes representing one environment object



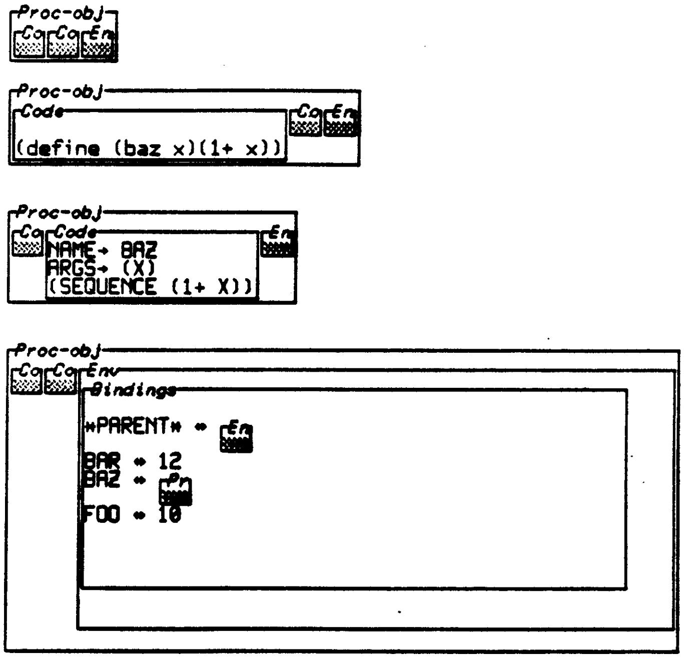
```
e1  | Env
      Bindings
      *PARENT* ↔ En
      BAR ↔ 12
      FOO ↔ 10

      (define bar 12)   |BAR

e2  | Env
      Bindings
      *PARENT* ↔ En
      BAR ↔ 12
      FOO ↔ 10
```

Figure 3-15:Defining BAR in the upper box causes both to change



```
*PARENT* ↔
BAR ↔ 12
BAZ ↔
FOO ↔ 10

(define (baz x)(1+ x))    |BAZ
```

Figurc 3-16:Dcfining a procedure



```
(define (baz x)(1+ x))
```

```
NAME→ BAZ
ARGS→ (X)
(SEQUENCE (1+ X))
```

```
*PARENT* ↔
BAR ↔ 12
BAZ ↔
FOO ↔ 10
```

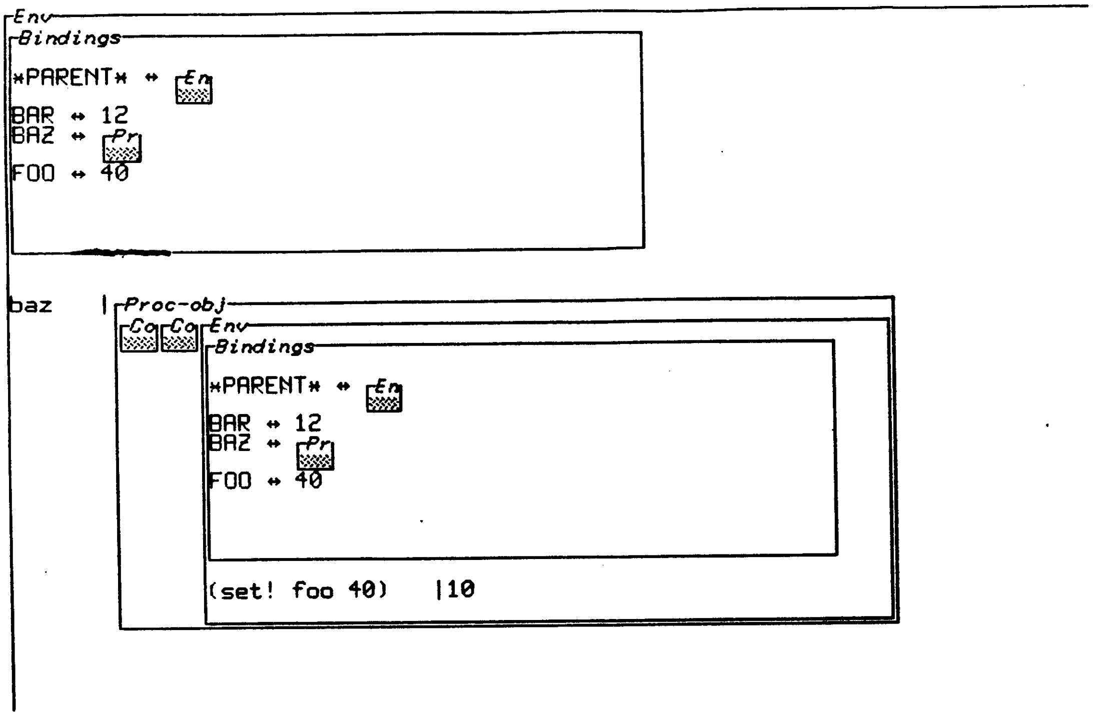
Figure 3-17:Four views of a procedure object

```
(define (baz x)(1+ x))
```

and evaluated it. The symbol BAZ is now bound to what in Scheme is known as a procedure object. As noted earlier, procedure objects in Scheme (and in the BOCHSER system) are essentially closures -- in the Abelson and Sussman text, they are represented as combinations of the procedure code and the lexically enclosing environment of the procedure definition. The BOCHSER system represents procedure objects as a new type of box -- suitably enough, a "procedure-object" box. Thus, in Figure 3-16, we see that the value of the symbol BAZ is a procedure object, which is represented as a shrunken (by default) box.

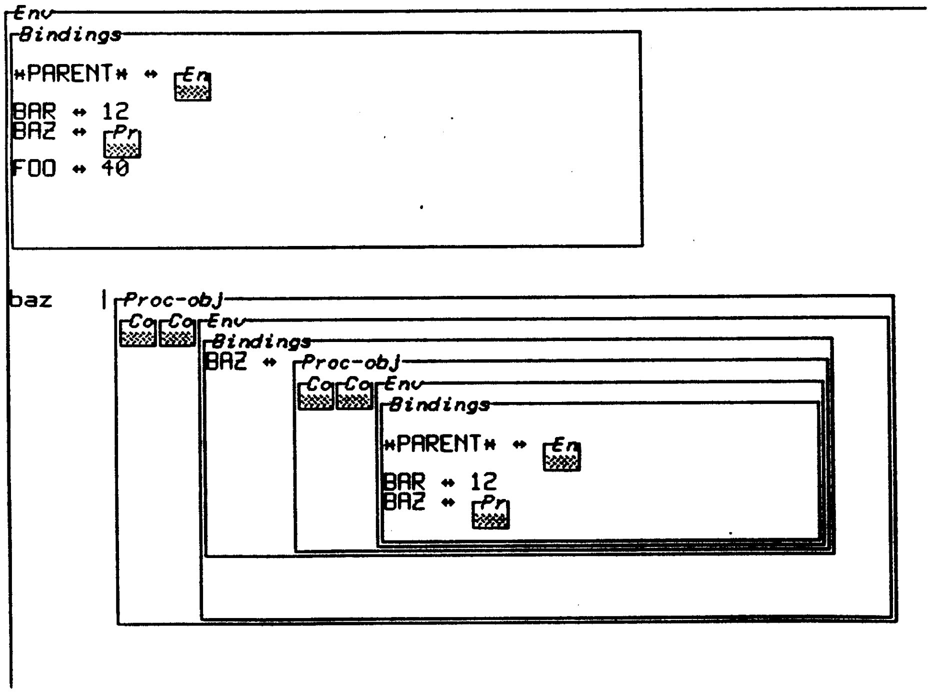
In Figure 3-17, we see a number of views of the procedure object to which BAZ is now bound. Procedure-object boxes contain three sub-boxes: two of type code (to be explained shortly), and one of type environment. The first "code box" contains the top-level expression that was evaluated to produce this procedure object. This box (which perhaps could be better labeled as a "history box") has no analogue in the standard Scheme procedure object representation, but -- as we will see -- it provides useful information. The second code box contains the actual procedure code which BOCHSER uses in applying the procedure; this corresponds to the "code" portion of the standard representation. In the case of our particular procedure, BAZ, the two code boxes contain very similar information. We can see that the representation of BAZ's code in the second box is slightly different than that in the first; this corresponds to the automatic syntaxing step which is performed when a BOCHSER procedure is created. Future examples will show that the first two sub-boxes in a procedure-object box need not always contain similar information.

The final sub-box in a procedure-object box is an environment box corresponding to the lexically enclosing environment of this procedure's definition. Here, the environment associated with BAZ is simply the same environment that we have been working in; expanding the bindings box of BAZ's environment shows the same bindings as the bindings box at the top of our screen. Note that, as with any other environment box, we can type expressions into the environment portion of a procedure-object box. In Figure 3-18, we have entered the environment box of the BAZ procedure-object box, and set the value of FOO to 40; the change is reflected in both the BAZ environment box and the bindings box at the top of the screen. The reader may have noted by this time that the BOCHSER screen allows for circularity in the representation of environments: for instance, in Figure 3-19, we have expanded the representation of BAZ inside the bindings box within the environment portion of a BAZ procedure-object box; one could continue the process indefinitely to produce an



```
Env
Bindings
*PARENT* ↔ En

BAR ↔ 12
BAZ ↔ Pr

FOO ↔ 40

baz     | Proc-obj
          Co Co Env
                Bindings
                *PARENT* ↔ En

                BAR ↔ 12
                BAZ ↔ Pr

                FOO ↔ 40

                (set! foo 40)    |10
```

Figure 3-18:The procedure object's environment is EQ to the "outer one"



```
Env
Bindings
*PARENT* ↔ En

BAR ↔ 12
BAZ ↔ Pr

FOO ↔ 40

baz     | Proc-obj
          Co Co Env
                Bindings
                BAZ ↔ Proc-obj
                      Co Co Env
                            Bindings
                            *PARENT* ↔ En

                            BAR ↔ 12
                            BAZ ↔ Pr
```

Figure 3-19:Infinite regress

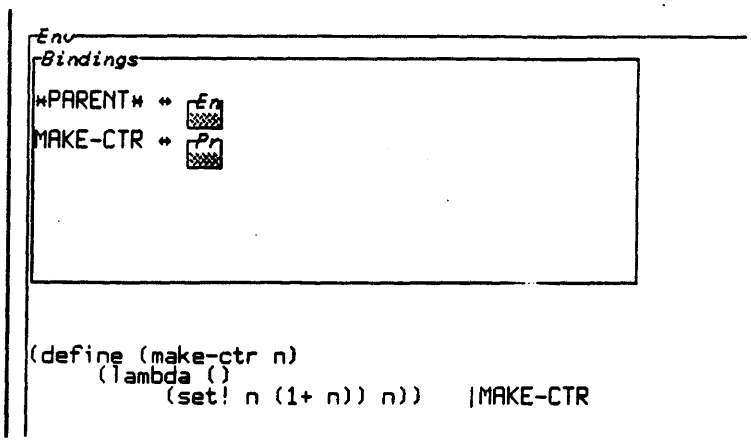
arbitrarily nested view of the BAZ procedure-object box.

The BOCHSER representation of procedure objects really comes into its own when the environment associated with a given procedure object is not EQ to an already-existing environment. An example of this situation can be seen in Figure 3-20. Here, the procedure MAKE-CTR is created; MAKE-CTR, when applied to a numeric argument, creates an "application environment" in which the local variable N is bound to the argument value. The returned result will be a procedure object whose associated environment is this newly-created "application environment". Thus, in Figure 3-21, when we define C0 to be the result of the expression

```
(make-ctr 0)
```

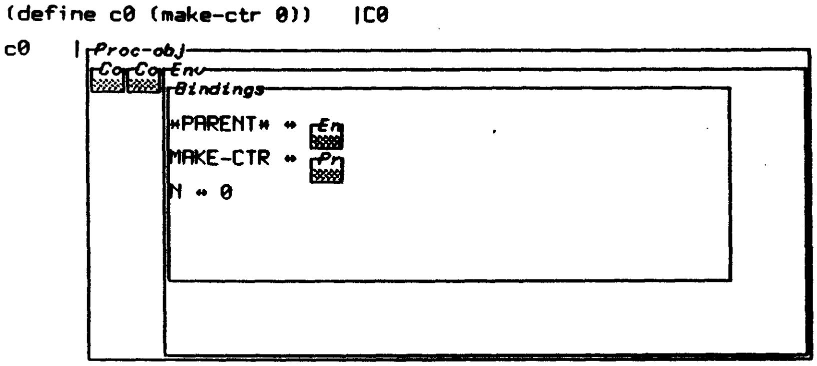
we find that C0 is now bound to a procedure object. The environment associated with the C0 procedure object is one that we have not seen elsewhere -- in this frame, N is bound to 0. The situation is represented graphically in Figure 3-22.

Now, in Figure 3-23, we can see the result of applying the C0 procedure object to no arguments. As we would expect from the code of C0, the value of N is set to 1; this new binding is reflected in C0's bindings box. One of the advantages of the BOCHSER interface is the ability to monitor bindings in this way -- in the standard Scheme interface, there would be no straightforward way to examine the environments of procedure objects like C0. Moreover, the BOCHSER representation suggests an even more powerful notion: evaluating arbitrary expressions within the environments of procedure objects. In the case of C0, we can see (Figure 3-24) the effect of defining a new procedure within C0's environment; a new procedure, named RESET, is created in that environment. Now, if we apply the RESET procedure to no arguments within the C0 environment, the value of N is set back to 0. The MAKE-CTR procedure, in the terminology of the Abelson and Sussman text, is a simple example of an "object-creating" procedure; C0 itself can be viewed as a tiny message-accepting object. The point, then, of defining RESET within C0 is to demonstrate that in BOCHSER it is a simple matter to "customize" objects like C0 by adding new procedures to the objects' local environments. Thus, we could have one counter object with its own private RESET procedure, another with an INCREMENT-BY-TWO procedure, and so on. In the fifth chapter, we will return to the subject of object-oriented programming in BOCHSER.



```
Env
Bindings
*PARENT* ↔ En
MAKE-CTR ↔ Pr

(define (make-ctr n)
     (lambda ()
          (set! n (1+ n)) n))     |MAKE-CTR
```

Figure 3-20:Defining MAKE-CTR



```
(define c0 (make-ctr 0))    |C0

c0    |Proc-obj
       Co Co Env
             Bindings
             *PARENT* ↔ En
             MAKE-CTR ↔ Pr
             N ↔ 0
```

Figure 3-21:Creating a counter object

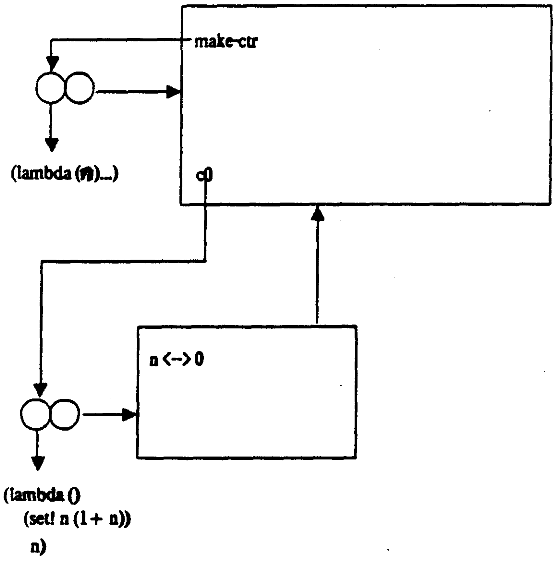
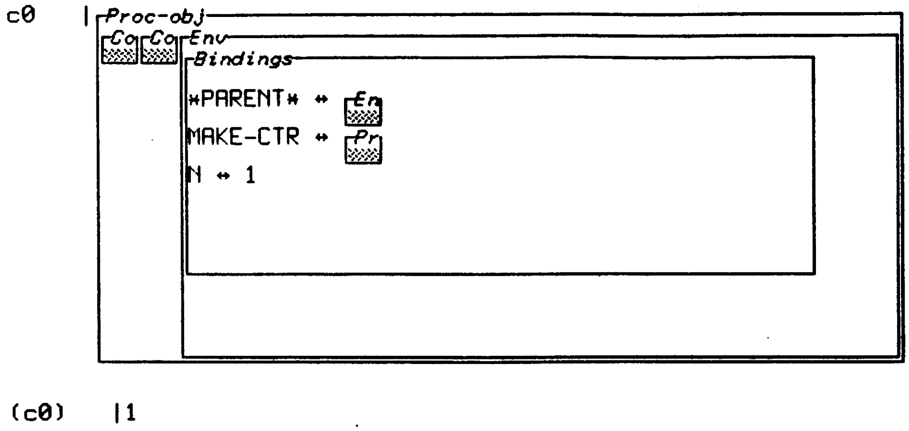
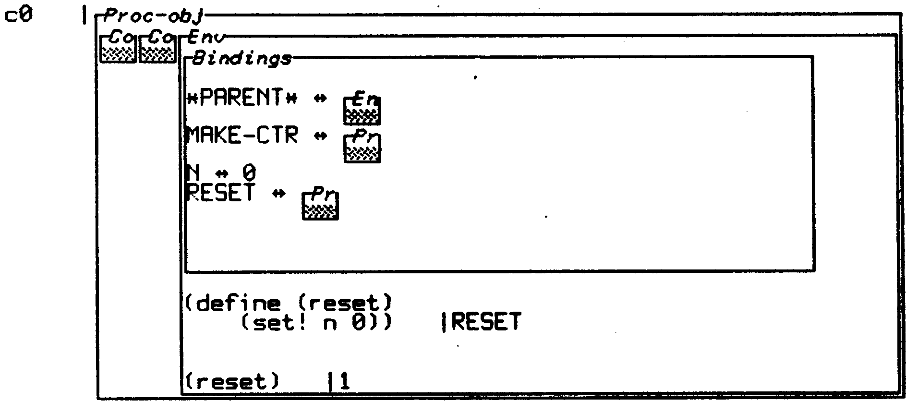


Figure 3-22:An environment diagram for MAKE-CTR and C0



```
c0      |
*PARENT* ↔
MAKE-CTR ↔
N ↔ 1

(c0)    |1
```

Figure 3-23:Applying C0 to zero arguments

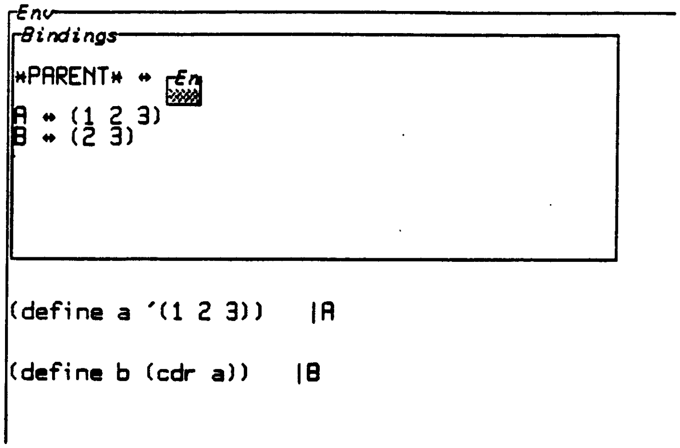


```
c0      |
*PARENT* ↔
MAKE-CTR ↔
N ↔ 0
RESET ↔

(define (reset)
    (set! n 0))      |RESET

(reset)    |1
```

Figure 3-24:Defining a "customized" procedure for C0

3.5 Sample Session, part 4: Shared Structures

As we have seen, the bindings boxes in BOCHSER environments are intended to reflect the state of the programmer's "world" at any particular time. This can be particularly useful when two BOCHSER variables exhibit sharing -- that is, when the two variables are bound to objects some of whose structure is held in common. An example of this situation can be seen in Figure 3-25. Here, the user has defined the symbol A to denote the list (1 2 3), and B to denote the CDR of that list, or (2 3). The situation is represented graphically in Figure 3-26; it is clear that the list object bound to A shares some of its structure with that bound to B.



```
*PARENT* ↔
A ↔ (1 2 3)
B ↔ (2 3)

(define a '(1 2 3))    |A

(define b (cdr a))    |B
```

Figure 3-25:Defining two lists that share cons cells

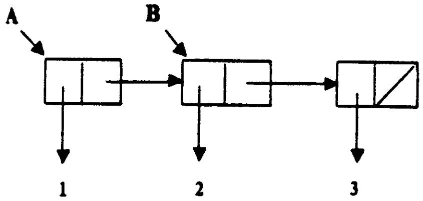
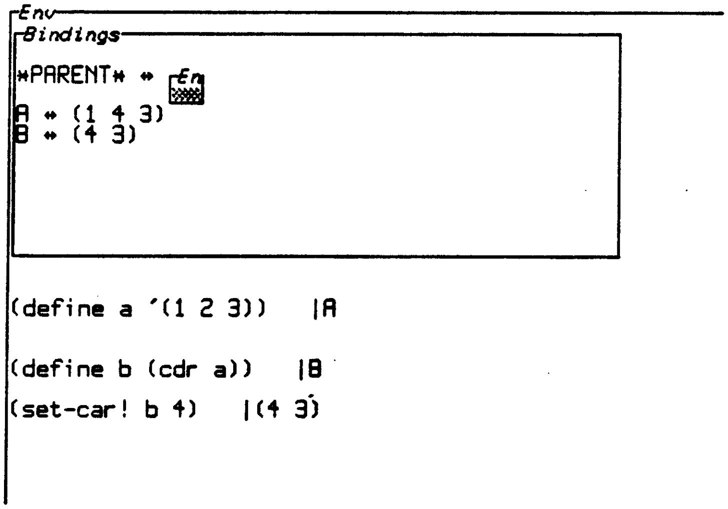


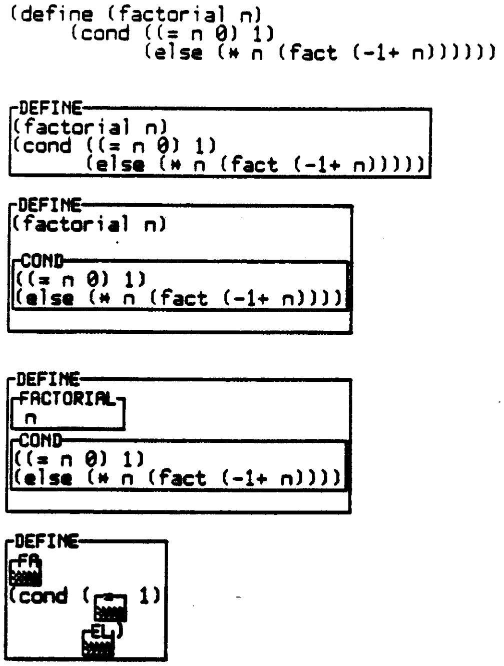
Figure 3-26:Sharing between A and B



```
*PARENT* ↔
A ↔ (1 4 3)
B ↔ (4 3)

(define a '(1 2 3))    |A

(define b (cdr a))    |B
(set-car! b 4)    |(4 3)
```

Figure 3-27:Changing B causes both to change

In Figure 3-27, we can see the result of performing a SET-CAR! operation on B; both the bindings of A and B change on the screen. Thus, the BOCHSER system is able to keep track of sharing between list objects and reflects their commonality in its representation of bindings.



```
(define (factorial n)
    (cond ((= n 0) 1)
          (else (* n (fact (-1+ n))))))
```

```
DEFINE
(factorial n)
(cond ((= n 0) 1)
      (else (* n (fact (-1+ n)))))
```

```
DEFINE
(factorial n)
 COND
 ((= n 0) 1)
 (else (* n (fact (-1+ n))))
```

```
DEFINE
 FACTORIAL
 n
 COND
 ((= n 0) 1)
 (else (* n (fact (-1+ n))))
```

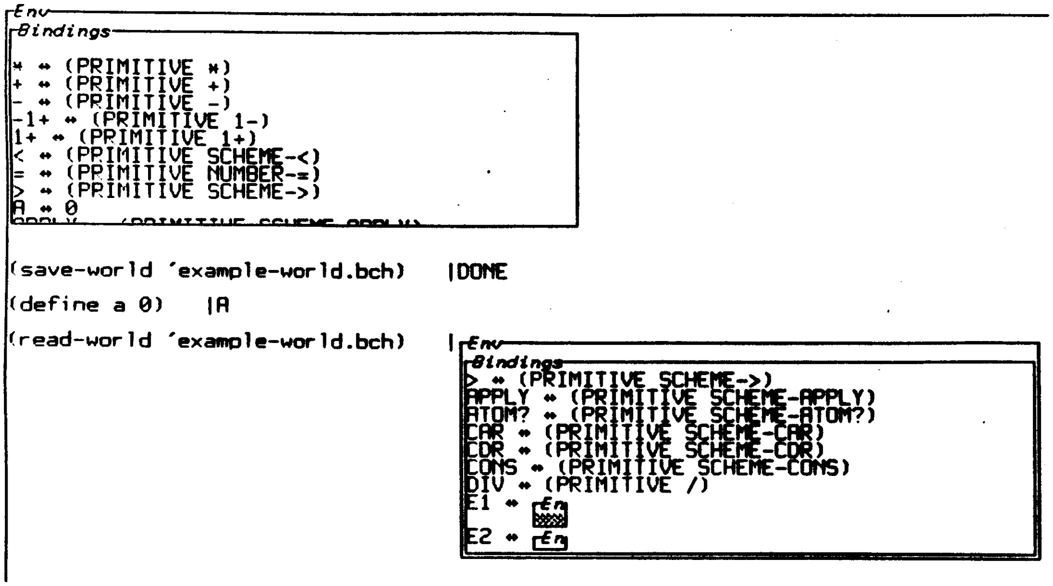
```
DEFINE
 FA
(cond (      1)
        EL )
```

Figure 3-28:A demonstration of syntax boxes

3.6 Sample Session, part 5: Syntax boxes

Thus far, all of the expressions that have been typed into the various interpreter areas during our sample session have been standard Scheme expressions -- they could equally well have been typed into the usual Scheme interface. BOCHSER allows for more interesting and elaborate formatting of expressions on the screen, however, via the construction of syntax boxes. A syntax box is simply an alternative representation of a list whose CAR is an atom; any such list may be represented as a box with the starting atom as the box-header in the left-hand corner. Figure 3-28 shows a variety of ways of representing the same expression by the use of syntax boxes. The user may transform any appropriate list into a syntax box by moving the cursor to the closing parenthesis of the list and pressing a special key (META-LINE on the 3600).

The advantages of this flexible formatting technique will become more apparent in later chapters. For now, suffice it to say that the user can gain a great deal of power by employing the existing box-editing facilities -- shrinking, expanding, and labeling (to be introduced shortly) on arbitrary list expressions.



```
Env
Bindings
* ↔ (PRIMITIVE *)
+ ↔ (PRIMITIVE +)
- ↔ (PRIMITIVE -)
-1+ ↔ (PRIMITIVE 1-)
1+ ↔ (PRIMITIVE 1+)
< ↔ (PRIMITIVE SCHEME-<)
= ↔ (PRIMITIVE NUMBER-=)
> ↔ (PRIMITIVE SCHEME->)
A ↔ 0

(save-world 'example-world.bch)    |DONE

(define a 0)    |A

(read-world 'example-world.bch)    |Env
                                    Bindings
                                    > ↔ (PRIMITIVE SCHEME->)
                                    APPLY ↔ (PRIMITIVE SCHEME-APPLY)
                                    ATOM? ↔ (PRIMITIVE SCHEME-ATOM?)
                                    CAR ↔ (PRIMITIVE SCHEME-CAR)
                                    CDR ↔ (PRIMITIVE SCHEME-CDR)
                                    CONS ↔ (PRIMITIVE SCHEME-CONS)
                                    DIV ↔ (PRIMITIVE /)
                                    E1 ↔ Env
                                    E2 ↔ Env
```

Figure 3-29:Saving and reading a BOCHSER world

3.7 Sample Session, part 6: Saving and Reading the BOCHSER World

As of now, the file-management capabilities of BOCHSER are fairly skeletal. However, the user can save and load the BOCHSER global environment via the SAVE-WORLD and READ-WORLD primitive procedures. SAVE-WORLD takes a file-name argument and saves all the bindings of the BOCHSER global environment into a file with that name; READ-WORLD takes a file-name argument corresponding to a saved BOCHSER world and returns an environment identical to the one saved. Note that READ-WORLD does not change any bindings in the environment in which it is applied; it simply returns a new environment as its result.

An example of these procedures in action can be seen in Figure 3-29. Here, the present BOCHSER world is saved via the expression

```
(save-world 'example-world.bch)
```

After this expression has been evaluated, the user defines a new variable A. Now, the old version of the world is read back in via the expression

```
(read-world 'example-world.bch)
```

Expanding the environment returned, we see a copy of the old global environment, before the binding of A was created.

A "destructive read-in" procedure named REPLACE-WORLD is also present in BOCHSER. When invoked from the global environment with a file-name as argument, this procedure replaces the current BOCHSER world with the one saved in the given file. Since this procedure effectively cancels any work done in the original BOCHSER environment, it is rather risky to use indiscriminately.

Chapter Four

BOCHSER in Depth

The previous chapter introduced most of the important features of the BOCHSER system. In this chapter several aspects of BOCHSER are explored in a little greater depth; the system is compared to the existing Scheme interface, and we begin to relate its features to the design principles discussed in the second chapter. The overall intent of this chapter is to give the the reader a better feel for the power and utility of the BOCHSER interface; the chapter following this one will address the topic of creating actual programs in the BOCHSER system.

Our discussion of BOCHSER in this chapter will focus on three areas: the BOCHSER editor and its utility in creating and working with Scheme programs; the advantages of BOCHSER's maintenance of a visible Scheme namespace; and the power of interacting with arbitrary Scheme environments.

4.1 Using the BOCHSER Editor to Create and Work with Scheme Programs

4.1.1 Integration of Editor and Intepreter

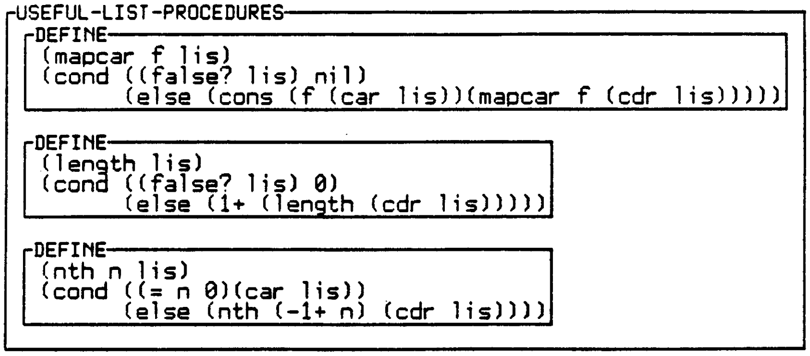
Our discussion of the BOCHSER editor may as well begin by noting its integration with the BOCHSER interpreter, since the most immediately apparent difference between BOCHSER and the standard Scheme interface is that in the former there is no separate editor buffer in which programs are created. BOCHSER programs are developed using exactly the same interface in which they will be run. This integration of editor and interpreter exemplifies di Sessa's concept of "detuning", mentioned earlier; the editor functions that one learns in order to manipulate program text in the "creation stage" of work may be used to examine the state of the system in the "running stage" of work.

Although the Boxer system takes this detuning notion a little farther than does BOCHSER at present (for instance, in Boxer, the user can redefine the action of a keystroke), the absence of a separate "editor mode" is a considerable step toward simplifying a programming environment. One of the more confusing features of the present 9836 Scheme system is that some editor features make sense in the interpreter -- for example, one can use the control-A keystroke to go to the beginning of a line -- while others do not. For instance, one cannot, in the standard Scheme interface, use control-P (which in EMACS moves the cursor to the previous line on the screen) to go to a previously typed-in expression in the interpreter. Even more irritating, the control-B key, which in EMACS moves the cursor back one character position, when used in the interpreter invokes a breakpoint!

BOCHSER also presents some good examples of the "diffusing functionality" notion -- that of one construct serving multiple functions. For example, syntax boxes may be used to collect well-understood groups of procedures into shrunken "black boxes" on the screen (this functionality will be elaborated upon later), or they may be used to store a group of related procedures within one screen region, as in Figure 4-1. Soon, we will see how syntax boxes can be used to effectively create menus of expressions as well. The integration of the editor and interpreter modes thus allows syntax boxes to be used as formatting devices both in the representation of code (an "editor-type" usage) and in the arrangement and grouping of expressions to be invoked from the interpreter.

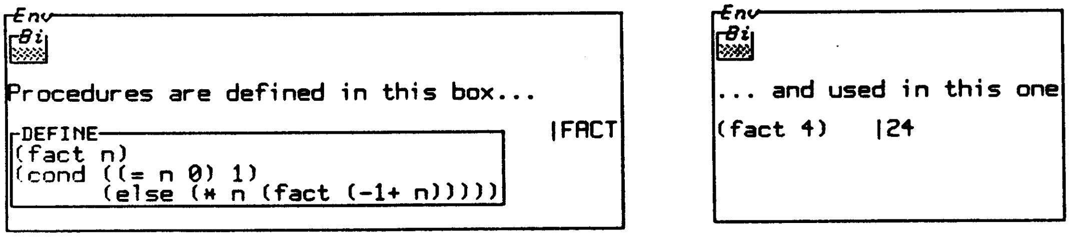
It should also be mentioned that the BOCHSER interface can accommodate those users who actively prefer to maintain a separate "editor buffer". In the book Interactive Programming Environments, Richard Stallman and Erik Sandewall engage in a small debate on this subject, with the former supporting the notion of a powerful separate text-based (as opposed to list-structure-based) editor [Sandewall 78]. Many of Stallman's arguments -- indeed, many of the points on both sides -- are now approaching obsolescence. For example, Stallman argues that an advantage of text editing is that the user can edit expressions with unbalanced parentheses; yet even the limited editing facilities in the present Scheme interpreter allow editing to take place on unbalanced expressions of this kind. Certainly in BOCHSER there is no language-based restriction on the text being edited, unless and until one tries to evaluate that text as a Scheme expression. Nevertheless, one might still argue that the additional functionality that comes with a separate editor is worth the difficulties of "programming with modes"; an editor might include a variety of souped-up special purpose commands or separate language-specific packages of macros that would be at odds with the needs of an interpreter.

Anyone sympathetic to these arguments could in fact work with BOCHSER in such a way as to effectively maintain a separate "editor buffer" by creating two environment boxes corresponding to the same environment object; in one box, the user would write procedures, and in the other evaluate



```
USEFUL-LIST-PROCEDURES
 DEFINE
  (mapcar f lis)
  (cond ((false? lis) nil)
        (else (cons (f (car lis))(mapcar f (cdr lis)))))

 DEFINE
  (length lis)
  (cond ((false? lis) 0)
        (else (1+ (length (cdr lis)))))

 DEFINE
  (nth n lis)
  (cond ((= n 0)(car lis))
        (else (nth (-1+ n) (cdr lis)))))
```

Figure 4-1: Grouping related procedures together within an interpreter area



```
Procedures are defined in this box...
DEFINE                                  |FACT
(fact n)
(cond ((= n 0) 1)
      (else (* n (fact (-1+ n)))))
```

```
... and used in this one
(fact 4)    |24
```

Figure 4-2: Using "EQ environment boxes" as editor and interpreter

expressions (see Figure 4-2). To make this strategy truly workable, it would be desirable to augment the BOCHSER editor with more powerful user-defined parameters; for instance, in the "interpreter box" on the right of Figure 4-2, one might set the default indentation parameters to be consistent with Scheme expressions, whereas in the "editor box" one might use text-oriented indentation. Additions of this kind, which will be discussed later on in the context of future directions for BOCHSER, are basically a matter of fine-tuning the system in accordance with observations of a community of programmers.

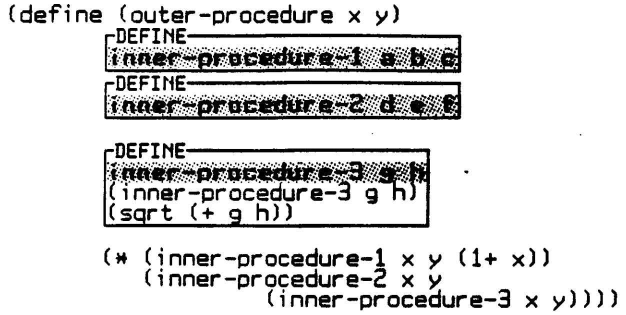
4.1.2 Hiding Subprocedures

Perhaps the most telling complaint about Scheme's syntax in comparison to that of most earlier LISP dialects is a corollary of the former's lexical scoping discipline. In particular, in Scheme, the code of many procedures begins with a series of definitions of internal procedures. That is, the pattern that a procedure definition follows often looks something like this:

```
(define (outer-procedure x y)
        (define (inner-procedure-1 a b c)
                ...
                ...)
                (define (inner-procedure-2 d e f)
                ...
                ...)
        ...
        ...
        <body of outer-procedure after DEFINEs>)
```

The problem with this syntax is that in some sense the heart of OUTER-PROCEDURE -- what we really think of as the procedure body -- is textually separated from the top of the definition. On an editor screen, it may even be impossible to view this essential portion of the procedure body and the definition line at the same time. We would like to think of the inner procedure definitions as subsidiary, and yet they are taking up the bulk of our attention in reading the code for OUTER-PROCEDURE.

This problem -- that Scheme syntax does not accurately reflect the programmer's perception of his or her work -- is one reason that BOCHSER's syntax boxes are a desirable editor feature. Figure 4-3 shows a procedure with several subsidiary procedure definitions inside it; the subsidiary procedure definitions have been placed inside their own local syntax boxes and the first two of these syntax boxes have been shrunken. Thus, any sub-expression which the user wishes to conceive of as a



```
(define (outer-procedure x y)
        DEFINE
        inner-procedure-1 a b c
        DEFINE
        inner-procedure-2 d e f

        DEFINE
        inner-procedure-3 g h
        (inner-procedure-3 g h)
        (sqrt (+ g h))

        (* (inner-procedure-1 x y (1+ x))
           (inner-procedure-2 x y
                  (inner-procedure-3 x y))))
```

Figure 4-3:Using syntax boxes to format a Scheme procedure meaningfully

"black box" may be represented -- virtually literally! -- as a shrunken box on the screen. Alternatively, the user can format procedures so that syntax boxes correspond to lexical scoping boundaries, as in the contour model often used to explain Algol 60's block structure; thus, one might elect to use syntax boxes exclusively on DEFINE, LAMBDA, and LET expressions, each of which essentially delimits a scoping boundary within a Scheme program. The point is not that syntax boxes must be used in any particular way, but that they tremendously expand the programmer's ability to format code meaningfully.

Figure 4-3 also illustrates another feature of BOCHSER (taken from an earlier version of the Boxer editor) -- namely, the ability to label a box. The grey lines at the top of the syntax boxes in Figure 4-3 represent labels for these boxes. Labels have no semantic content; they have the status of one-line comments. Their utility is shown by the fact that when a box is shrunken, the label remains visible, so that the user can get a brief indication of the box's contents. Any BOCHSER box -- environment boxes, procedure-object boxes, and so on -- may be given a label; the label line is created by pressing a special key while the cursor is inside the box, at which point the user can type in the desired label contents.

4.1.3 Non-scrolling

One of the simplest properties of the standard Scheme interpreter -- the fact that it employs a scrolling screen -- is also one of its most problematic. Often, the programmer wishes to examine the result of an earlier evaluation step, but is unable to because the expression and its result have scrolled off the top of the screen. A similar problem can be seen when the programmer wishes to re-evaluate an expression (or one of a few standard expressions); even though the Scheme system does retain the last several expressions evaluated in memory and can recall them onto the screen via the use of a RECALL key, this is a rather clumsy and in the final analysis insufficient technique for re-evaluating expressions. It is after all not improbable that the user might wish to evaluate one of a set of many "standard" expressions in some arbitrary order. This sort of situation arises often in the context of object-oriented programming. For example, the programmer might have a "turtle" object that accepts a number of messages: "forward", "right", "back", and "left". If the programmer wished to have the turtle execute an arbitrary series of steps, a sequence of expressions of the following sort would have to be typed:

```
((turtle 'forward) 100)
((turtle 'right) 90)
((turtle 'forward) 100)
((turtle 'left) 90)
((turtle 'back) 100)
...
...
```

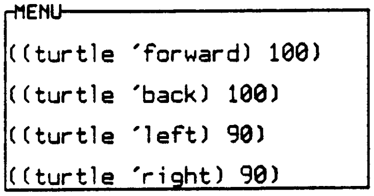
The point of this scenario is that the programmer must repeatedly retype expressions which are better thought of as part of a "menu" of standard expressions. What is desired is an easily accessible region of the general appearance:

```
((turtle 'forward) 100)
((turtle 'back) 100)
((turtle 'left) 90)
((turtle 'right) 90)
```

which can be placed on the screen and whose expressions can be used arbitrarily many times and whenever desired. An example of such a menu, created using a BOCHSER syntax box, is shown in Figure 4-4 Note that the syntax box here was made from a list whose car is the symbol MENU; as such, an attempt to evaluate the expression corresponding to the box itself (i.e., a list starting with the atom MENU) would produce an error if there were no MENU procedure. However, any complete expression in the interpreter area of a BOCHSER environment, whether or not it is a subexpression of a larger list, may be evaluated. Here, the user could evaluate the expression

```
((turtle 'forward) 100)
```

simply by moving the cursor to the closing parenthesis (this can be achieved with the mouse pointer), and evaluating the expression.



```
MENU
((turtle 'forward) 100)
((turtle 'back) 100)
((turtle 'left) 90)
((turtle 'right) 90)
```

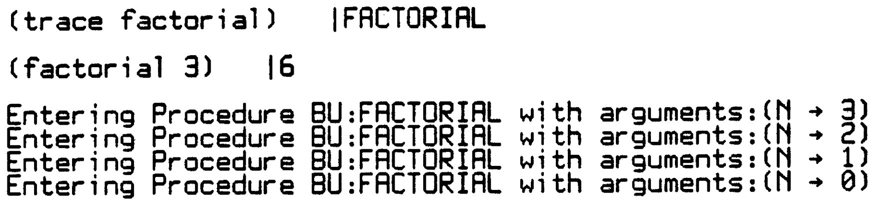
Figure 4-4:A BOCHSER "turtle menu"

There is one major disadvantage to BOCHSER's non-scrolling screen; it is rather easy for the interpreter area of an environment box to become cluttered with previously evaluated, and now unimportant, expressions. In a way, this is the complementary problem to that of the standard Scheme interface's scrolling screen. Possible solutions to this problem will be discussed later in this report.

4.1.4 Working with the History of a BOCHSER Session

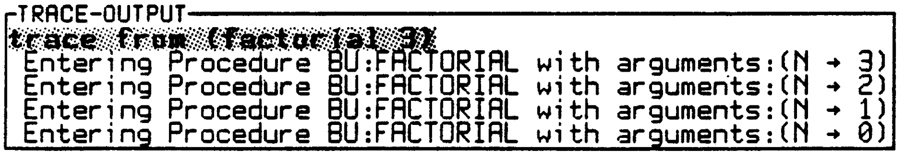
One of the strengths of the BOCHSER editor in working with Scheme programs is that it allows the user to maintain and store some record of a BOCHSER session. Earlier, in the context of discussing the non-scrolling screen, it was mentioned that often a programmer wishes to look back at the results of an interaction done earlier in the work session. In BOCHSER, this kind of history-maintenance is easy to do.

Figure 4-5 shows an illustration of this idea. Here, the programmer has put a TRACE on the FACTORIAL procedure; the results of the call to FACTORIAL have then been printed out on the screen. In the usual Scheme interface, these TRACE output lines would print out on the screen, eventually scroll off, and subsequently be lost from the programmer's view. A common experience for Scheme programmers is to apply TRACEd procedures many times in order to get an additional look at the same TRACE output that they saw before. Figure 4-6 shows how this problem can be handled in BOCHSER. The TRACE output has been placed inside a syntax box and labeled; in Figure 4-7, the shrunken syntax box is placed at the top of the screen for later use. One could of course save other useful portions of the session history: for example, evaluations that result in an error message could be placed into a special box for later reference.



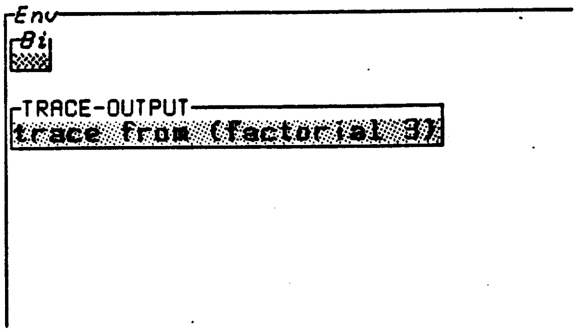
```
(trace factorial)    |FACTORIAL

(factorial 3)    |6

Entering Procedure BU:FACTORIAL with arguments:(N → 3)
Entering Procedure BU:FACTORIAL with arguments:(N → 2)
Entering Procedure BU:FACTORIAL with arguments:(N → 1)
Entering Procedure BU:FACTORIAL with arguments:(N → 0)
```

Figure 4-5:The result of a trace on the FACTORIAL procedure



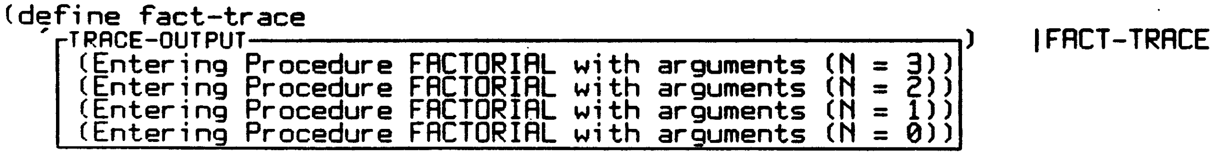
```
TRACE-OUTPUT
trace from (factorial 3)
 Entering Procedure BU:FACTORIAL with arguments:(N → 3)
 Entering Procedure BU:FACTORIAL with arguments:(N → 2)
 Entering Procedure BU:FACTORIAL with arguments:(N → 1)
 Entering Procedure BU:FACTORIAL with arguments:(N → 0)
```

Figure 4-6:Placing the result inside a syntax box



```
trace from (factorial 3)
```

Figure 4-7: Leaving the trace result on the screen for future examination



```
(define fact-trace
 'TRACE-OUTPUT                                                     )     |FACT-TRACE
  (Entering Procedure FACTORIAL with arguments (N = 3))
  (Entering Procedure FACTORIAL with arguments (N = 2))
  (Entering Procedure FACTORIAL with arguments (N = 1))
  (Entering Procedure FACTORIAL with arguments (N = 0))
```

Figure 4-8: Editing the trace result and saving it as the value of FACT-TRACE

It is even the case that much of what the user sees on the screen can be used as the value of a BOCHSER object. As of now, the BOCHSER reader does not accept, among other things, environment boxes and procedure object boxes as evaluable elements of Scheme expressions; that is, one cannot evaluate, say, an environment box. However, much of the screen content is understood by the BOCHSER reader, and at least this portion can be treated as a BOCHSER object. For instance, in Figure 4-8, portions of the FACTORIAL trace output have been placed inside a list and stored as the value of the variable FACT-TRACE; this value can now, of course, be saved on secondary storage as part of a BOCHSER world.[5]

4.2 Visible Namespace

The previous section dealt with the BOCHSER editor, and most of the scenarios demonstrated the utility of BOCHSER's syntax boxes. In this and the following section, we turn our focus to BOCHSER's environment and bindings boxes. One advantage of including these elements in the BOCHSER interface is the usefulness of being able to evaluate expressions within any environment box; this will be the topic of the next section. Here we explore another issue: the value to a programmer of having a visible namespace.

A significant proportion of a Scheme programmer's time is spent examining the state of the interpreter "world": evaluating variable names, pretty-printing procedures, going back to the editor to look at program code, and so on. In the standard Scheme interpreter, the usual way to examine a binding is to evaluate the variable name in question, or, in the case of a procedure, calling the PRETTY-PRINT primitive on the procedure. Should the original code of the procedure be of interest, the user must go back to the editor buffer to look for that code. If a number of separate bindings are of interest, the user can invoke the WHERE primitive procedure, which places Scheme in its "debugging" mode, from which state a special command will allow the user to view all the bindings in a given environment.

In every one of these techniques, the standard Scheme interface unwittingly forces the user into viewing "world-examination" as a sort of digression. For instance, suppose the user has created a Scheme procedure that sets a global variable in the course of execution:

[5] A similar sort of functionality is seen in Semantic Microsystems' newly-released MacScheme version of Scheme for the Apple Macintosh computer (cf. [MacScheme 85]).

```
(define (proc x y)
        ...
        ...
        (set! *global-variable* (+ x y))
        ...
        ...)
```

In order to see whether the global variable is being set correctly, the user would have to type a series of expressions more or less as follows (the arrow "--->" indicates the Scheme prompt):

```
---> *global-variable*
0

---> (proc 1 2)
<result>

---> *global-variable*
3
```

Here, the user has to type in the expression *GLOBAL-VARIABLE* every time he or she wishes to examine the state of that binding. If there are a number of mutable variables of this sort, the amount of overhead spent examining their bindings becomes onerous. Even worse, if the variables of interest are not global (for example, "state variables" like N in our earlier counter example), there is no straightforward way -- short of invoking breakpoints -- to examine their bindings. Even assuming that the variable is accessible, there are other problems: since the screen scrolls, the programmer might examine the variable binding of interest and then perform several other unrelated evaluation steps, at which point the variable value examined earlier is irretrievably lost from view and possibly forgotten.

Similar comments can be made about the pretty-printing of procedures, and invoking the DEBUG facility. In every case, there is a picture of the programmer's activity as linear -- first examine bindings, then evaluate, then examine bindings again -- rather than as a combination of several ongoing activities. The BOCHSER strategy of providing a consistent place on the screen where the current namespace can always be examined supports the notion of "world examination" as a constant, ongoing background activity. In the earlier example, the binding of *GLOBAL-VARIABLE* could be under observation when the procedure PROC is applied; in fact, any binding in any accessible environment (such as that for N in the counter example) is visible in exactly the same way -- no special system facilities need to be invoked.

Perhaps more importantly, BOCHSER's visible namespace is a major element in its support of the Scheme "language model" as described in the Abelson and Sussman text. This topic of BOCHSER and its representation of the Scheme model will be discussed at the outset of the next chapter; but it is worth noting here that the concept of the visible namespace has been an important one in a number of "model-transparent" programming systems (cf. [Shapiro 74], [Leap 84]).

4.3 Interacting with BOCHSER Environments

In this section, we explore several ways in which a BOCHSER programmer can exploit the ability to interact with arbitrary BOCHSER environments. First, we demonstrate how BOCHSER environment boxes can be used to implement customizable package systems; then, the inclusion of environment boxes within procedure-object boxes is shown to be useful for redefining buggy procedures. The final example in this section introduces BOCHSER's breakpoint facility, and demonstrates the advantages of including environment boxes within the system's representation of breakpoints.

4.3.1 Environment Hierarchies

In Abelson and Sussman's text, the authors explain how the concept of a Scheme environment can subsume that of "packages". A package, as defined in most LISP systems, is a group of (usually functionally related) procedure and variable definitions; the point of grouping definitions in this way is to segment and access the programming environment namespace in a natural way. Whenever a procedure in one package wishes to access a name in another package, it can do so by prefixing the name with that of the second package. For instance, one might create a MATH package in which all procedures specifically geared toward mathematical programming are placed. A procedure in the GRAPHICS package might access a matrix-inversion procedure by calling MATH:INVERT. One of the advantages of this strategy is that names can be re-used in separate packages: there may be a new INVERT procedure, for example, defined in the GRAPHICS package. Packages may also contain sub-packages: the MATH package, say, might contain a sub-package named MATRIX for handling matrix procedures.

Powerful as this package notion is, it is really just a special case of the broader environment notion. An environment, after all, is -- like a package -- just a collection of bindings. Essentially, one could access the MATH package in the example above by using Scheme's EVAL primitive as follows:

```
((eval 'invert math-environment) graphics-matrix)
```

This is a useful idea, but the BOCHSER interface makes it even a little more powerful by virtue of its representation of environments. In BOCHSER, as we have noted, a user can interact with any environment in exactly the same way; thus, if one wishes to access the MATH environment procedures by default, one need only go to an environment box which represents the MATH environment object and expand that box to full-screen size. From that point on, the system interface is virtually the same as the global environment interface; no calls to EVAL are needed (although they might be used if one wished to access procedures in an environment other than the MATH package). Thus, if one wished to define a new procedure in the MATH package, one would simply evaluate the definition inside the MATH environment box; and the notion of sub-packages would similarly be represented by including a sub-environment (e.g., MATRIX) inside the MATH environment. Inheritance of names is achieved automatically by virtue of Scheme's environment model; for instance, if the MATH environment contains a TRUNCATE procedure and the name TRUNCATE is not bound in the MATRIX environment, then calls to TRUNCATE inside MATRIX will use the definition inside MATH.

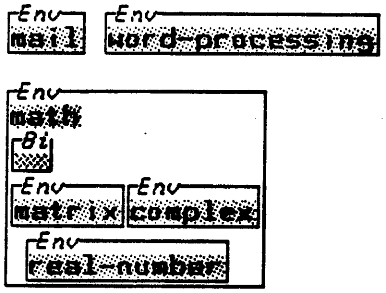


Figure 4-9:A hierarchy of environments used as a "package system"

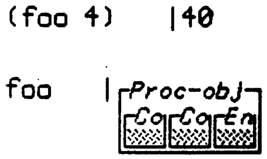
Figure 4-9 shows an extension of this idea, illustrating the notion of an environment (or package) hierarchy. The global environment contains several sub-environments labeled MAIL, WORD-PROCESSING, and MATH; the MATH environment itself has been opened up to reveal sub-environments that act as real-number, matrix, and complex-number packages. Any particular sub-environment could be loaded in as a separate modular unit[6]; thus, the user can easily configure a package hierarchy for any working session. For instance, a hierarchy like that in Figure 4-9 could be created by reading in MAIL, WORD-PROCESSING, and MATH environments from secondary storage while working in the global environment, and then, if desired, reading in MATRIX, COMPLEX-NUMBER, and REAL-NUMBER while working inside the MATH environment. In fact, one could, by reading in a new copy of the WORD-PROCESSING environment while inside MATH, achieve the effect of having two separate WORD-PROCESSING packages -- one which inherits procedures from the MATH package and one which does not. Accessing any of these worlds is simply a matter of maneuvering about the screen, shrinking and expanding the appropriate boxes. The kind of functionality achieved here is similar to that described in the SCREEN system described by Sandewall et. al. [81].

4.3.2 Redefining procedures

As another brief example of how one might exploit the ability to interact with BOCHSER environments, consider the following situation: one is working in a "deeply nested" environment box (that is, an environment box corresponding to a frame many levels down from the global environment), and discovers a bug in a procedure named FOO. Now, in order to fix the bug in FOO, one must find the procedure to examine its code; but FOO may have been created in any one of the parent environments to this one. As a last resort, of course, the programmer could examine the parent frames to this environment, one at a time, until finding FOO inside a bindings box. This would, in fact, not be terribly difficult to do -- but there is a better way to approach the problem.

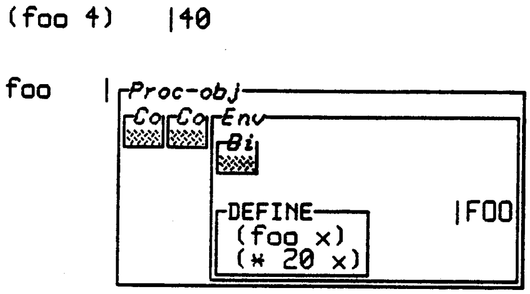
In Figure 4-10, the situation just described is illustrated. Here, the programmer uses a FOO procedure which is intended to multiply its argument by 20, but instead appears to be multiplying by 10. The programmer, to find the FOO procedure, simply evaluates the name FOO: the result of this evaluation is a procedure-object box corresponding to the desired procedure object. Now, in Figure 4-11, the programmer enters the environment box associated with the FOO procedure object, retrieves the code from FOO's code-box (there is a special editor function that allows the user to retrieve text from code-boxes), and redefines FOO in the environment in which it was originally

[6] Admittedly, creating the appropriate sub-environments is at present a somewhat arduous task in BOCHSER; to save BOCHSER worlds which do not include the usual global bindings, one would have to UNDEFINE all these bindings before saving. Future BOCHSERs should allow for selective saving of individual bindings, which would make the suggested sort of sub-environment creation a good deal simpler.



```
(foo 4)    |40

foo        |
```

Figure 4-10:In a lower-level environment, a bug in FOO is spotted



```
(foo 4)    |40

foo        |
```

Figure 4-11:FOO is redefined within its associated environment

created. Once this is done, the programmer can go back to working with the newly-debugged procedure in the original "nested" environment. The idea illustrated here is that in order to find a BOCHSER procedure, one need only evaluate its name; and because procedure objects come with the environment in which they were created, they may be redefined in that environment if the need should arise.

4.3.3 Breakpoints

BOCHSER does not at this time contain an extensive debugging system. A TRACE procedure has been implemented (and in fact, its use was demonstrated during the earlier discussion of working with the history of a BOCHSER session); but much more interesting, and powerful, is BOCHSER's breakpoint facility. The breakpoint facility provides a particularly good illustration of useful interaction with BOCHSER environments; before launching into a discussion of it, however, the problematic nature of breakpoints in the current Scheme system should be outlined.

In Scheme, breakpoints constitute a powerful debugging aid. A breakpoint may be inserted into Scheme code by the inclusion of a BKPT expression:

```
(define (proc-with-breakpoint x y)
        ...
        ...
        (bkpt "Stop here" (list x y))
        ...)
```

When the BKPT expression is evaluated, Scheme prints out the string and value arguments and stops evaluating the procedure body. In the standard interface, the user is now presented with a "breakpoint" prompt, at which point he or she may evaluate expressions in the environment current at the time of the breakpoint. Using the DEBUG procedure can provide further options: this places the user in a special Scheme subsystem from which special commands may be invoked to perform a variety of tasks (examine the bindings within the environment created by the procedure application, evaluate expressions within that environment, examine the calling procedure, and so on).

Despite the power and usefulness of the breakpoint concept, observation of Scheme students reveals that very few of them avail themselves of breakpoints in the course of debugging. There are several probable reasons for this. One is that the present interface's handling of breakpoints, although wonderfully thorough and powerful, suffers from the same assumption of "linearized programming activity" that we saw before in the discussion of namespace visibility. The programmer, by assumption, works at "top level" (i.e., with the normal interpreter), then in the breakpoint system, and, once the debugging activity is complete, back at top level again. Another problem is related to the earlier notion of "detuning"; although typing at the breakpoint prompt is similar functionally to typing at the top-level Scheme prompt, and one wants to do similar things -- evaluate expressions, for example -- the properties of the interface are subtly different. Typing control-G, for example, abandons the breakpoint and returns back to the top-level; therefore, if the user evaluates an expression at the breakpoint prompt and this evaluation results in an error, the usual technique for "leaving the error state" -- namely, typing control-G -- has the unwanted effect of leaving the breakpoint state altogether. As for the DEBUG system, it too exemplifies the "linearity problem"; and in order to use it the programmer has to work with commands that have no analogues anywhere else in Scheme. Most students find its complexity daunting.

The overall difficulty is again one of too many "modes". The earlier quote from Heering and

Klint pointed out that debugging and programming share a good deal of conceptual overlap; why, then, should the user be forced into different modes of interaction during these activities?

In BOCHSER, breakpoints have been implemented through the use of *breakpoint boxes*, which represent an attempt to "detune" some of the activity of debugging and make it consistent with what the user already knows of the system. When a breakpoint expression is encountered in a procedure written in the BOCHSER system, the value returned is a breakpoint box. Figure 4-12 shows an example. Here, the FACTORIAL procedure has been written to include a breakpoint expression, and the user has called this procedure on argument 4; the system has returned a breakpoint box.

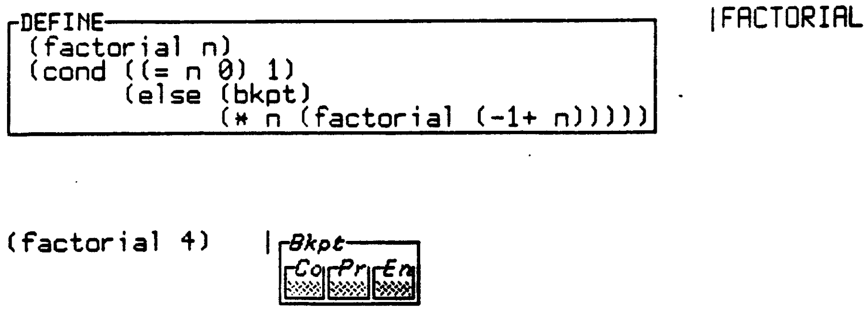
In Figure 4-13, several views of the previous figure's breakpoint box are provided; three sub-boxes are revealed. The first, a code box, contains the code of the smallest expression of which this breakpoint is a sub-expression; here, the code corresponds to the consequent portion of the COND clause being executed. The third box (we will return to the second in a moment) is the environment of the breakpoint; examining the bindings here shows that N, the local parameter of the FACTORIAL procedure, is bound to 4. The second sub-box is a procedure-object box indicating the procedure whose invocation created the breakpoint environment -- in this case, the procedure object bound to FACTORIAL.

Now, in order to evaluate expressions or examine bindings, the user need only enter the environment of the breakpoint box and do the things he or she has learned to do in every other environment box: namely, look inside the bindings box or type expressions at the interpreter area. Although there are a few editor commands (to be explained shortly) that apply specifically to breakpoint boxes, the environment box within the breakpoint is in no way special -- it is just like every other environment box. This fact accounts for a great deal of the simplicity of the BOCHSER debugging system. There is no special "debugging mode" here -- only an environment box that corresponds to the environment of a not-yet-completed procedure application.

To continue with our example, in Figure 4-14, the user types in the expression

```
(set! n 3)
```

to change the value of N in the breakpoint environment. Then, the user places the cursor to the right of the breakpoint box and uses an editor key to continue on from the breakpoint. The result of this step (Figure 4-15) is yet another breakpoint box, since the recursive call to FACTORIAL has encountered another breakpoint. In the environment of the new breakpoint, we can see that the value

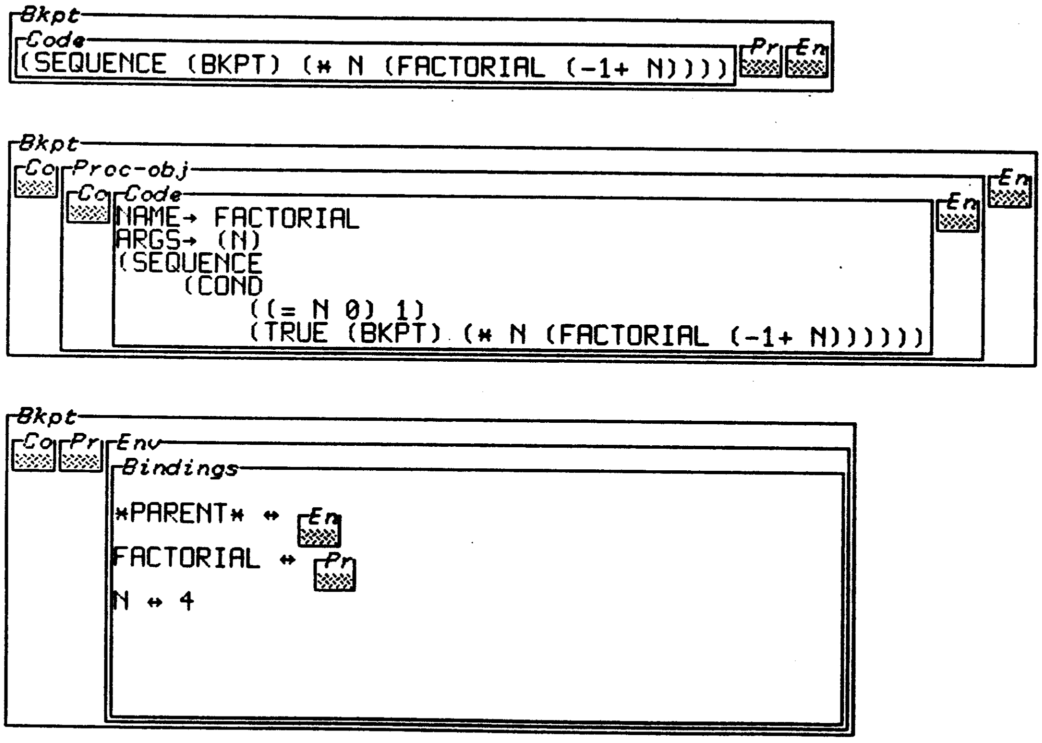


```
DEFINE
 (factorial n)
 (cond ((= n 0) 1)
       (else (bkpt)
             (* n (factorial (-1+ n)))))
```

```
|FACTORIAL
```

```
(factorial 4)     |
```

Figure 4-12:A procedure with a breakpoint; a breakpoint box

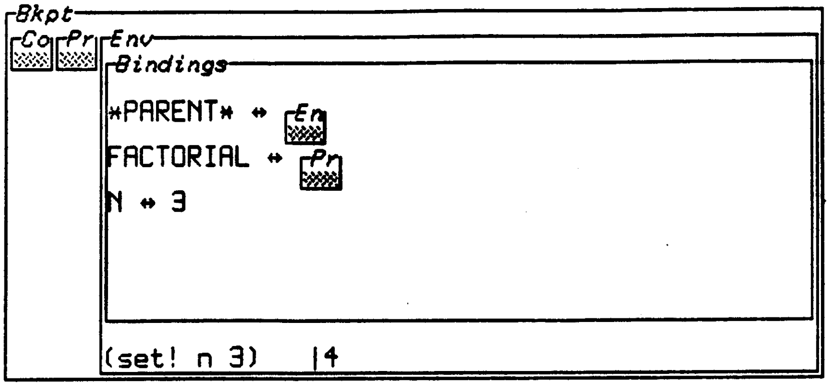


```
(SEQUENCE (BKPT) (* N (FACTORIAL (-1+ N))))
```

```
NAME→ FACTORIAL
ARGS→ (N)
(SEQUENCE
     (COND
          ((= N 0) 1)
          (TRUE (BKPT) (* N (FACTORIAL (-1+ N))))))
```

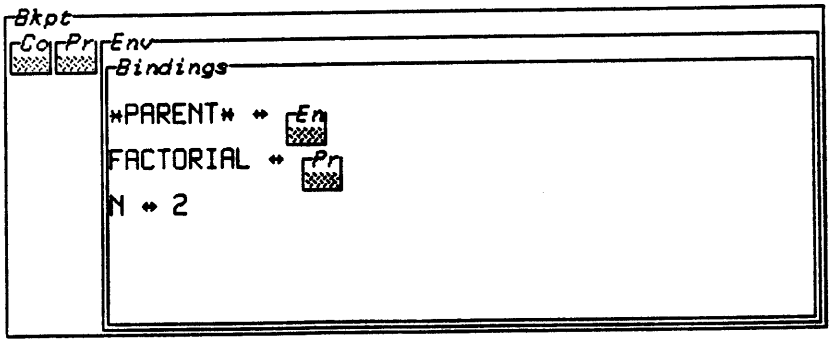
```
*PARENT* ↔
FACTORIAL ↔
N ↔ 4
```

Figure 4-13:The three sub-boxes within a breakpoint box



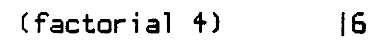
```
*PARENT* ↔ En
FACTORIAL ↔ Pr
N ↔ 3

(set! n 3)    |4
```

Figure 4-14:Evaluating an expression in the breakpoint environment



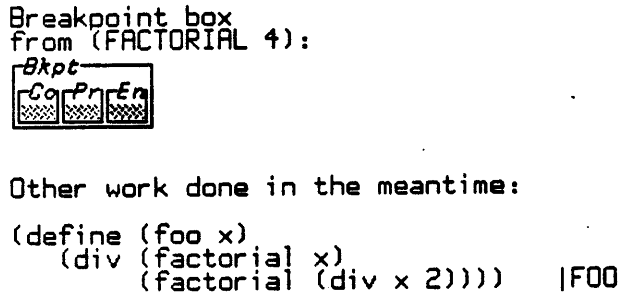
```
*PARENT* ↔ En
FACTORIAL ↔ Pr
N ↔ 2
```

Figure 4-15:The next breakpoint after continuing the evaluation



```
(factorial 4)        |6
```

Figure 4-16:The result of the evaluation, ignoring remaining breakpoints



```
Breakpoint box
from (FACTORIAL 4):
```

```
Other work done in the meantime:

(define (foo x)
   (div (factorial x)
        (factorial (div x 2))))    |FOO
```

Figure 4-17:Temporarily ignoring a breakpoint box to do other work

of N is now 2, as expected. Using yet another editor key outside this breakpoint box tells BOCHSER to continue on ignoring any future breakpoints (Figure 4-16); the system finally returns 6 (which is not the factorial of 4, but is consistent with the reset value of N earlier).

A useful fact to note about breakpoint boxes is that they need not be continued from immediately; in fact, they need not be used at all. Thus, one might evaluate an expression that returns a breakpoint box and move this box to the top of the screen to be used later (as in Figure 4-17). In the meantime, the programmer might work in the outer environment until ready to continue from the breakpoint. If there are free variables in the code of the procedure to be continued, of course, altering bindings in the parent environment of the breakpoint may change the result one gets by continuing it; in our earlier example, redefining, say, the "*" procedure to the value of the "+" procedure in the global environment before continuing the FACTORIAL breakpoint would have altered the eventual returned result. Again, the point is that BOCHSER "de-linearizes" the programmer's activity. Rather than do all debugging at once, the programmer can do a little work in the breakpoint environment, then a little work back at top level, and finally, when ready, continue on from the breakpoint created earlier in the session.

Throughout this chapter, we have seen a variety of scenarios which demonstrate the positive features of the BOCHSER interface -- the ability to format code meaningfully, to monitor current bindings in a given environment, and to debug procedures using breakpoints, among others. The following chapter takes up the question of what kinds of programming projects might be facilitated by the BOCHSER interface.

Chapter Five

BOCHSER Programming

5.1 BOCHSER and the Novice Scheme Programmer

Having provided an overview of the BOCHSER system and some of the particular programming techniques suggested by it, we are now prepared to re-examine the issue of "computer models" raised earlier in this report. The BOCHSER system is designed around the most important and powerful model of Scheme presented in M.I.T's introductory course -- namely, the environment model described in Chapter 3 of the Abelson and Sussman text, and outlined in the first chapter of this report. The objects seen by the BOCHSER user -- procedure objects, environment objects, and bindings -- correspond precisely to their counterparts in the abstract model presented in the course. Procedure objects in BOCHSER, just as one would expect, are combinations of procedure code and an associated environment; environments are represented by frames linked upward to other frames, with a global environment as a final step in this sequence; bindings are name-value associations that are created and altered according to the actions of the programmer.

My own experience as teaching assistant and recitation instructor for the 6.001 course suggests that the standard environment model, as presented in the classroom, is very difficult for students to understand. This observation is corroborated by Steven Strassmann [84], a former teaching assistant for 6.001:

> "...environment diagrams remain in students' minds a rather elusive part of the curriculum; and very few students feel confident of their knowledge of what they are or what they're good for."

The environment model, besides being difficult, also constitutes from the instructor's standpoint a kind of watershed. Students who master it not only have a better understanding of Scheme throughout the remainder of the course, but also seem to acquire a firmer grasp of the material presented earlier (the environment model is presented in full about five weeks into the course). Students who don't master it struggle through the rest of the term.

In essence, the environment model in Scheme acts as the central instance of a "notional machine" in DuBoulay, O'Shea, and Monk's sense, or a framework for working with "transactions", to use Mayer's terminology. This being the case, and since the model seems difficult to master, it would seem wise to support the model as explicitly as possible at the level of the Scheme interface itself. Most of the research done in the pedagogical value of language-specific computer models has appeared to make the tacit assumption that the model will be presented by a human teacher, away from the machine. The opacity or sparseness of the language interface itself, on this view, is a given. However, there is obviously a great deal of value in letting the language interface carry some of the pedagogical burden of presenting a computer model; and the BOCHSER system attempts to perform this sort of service for the Scheme language.

5.2 BOCHSER and the Experienced Scheme Programmer

Beyond pedagogical questions, the contention of this report is that the BOCHSER interface holds some advantages for experienced programmers as well. This is primarily because of a rather special property of Scheme. Unlike most languages, which can be adequately (if not very informatively) conveyed by the classical "teletype-style" interface, Scheme is actually a much more powerful language than its standard interface suggests. In the standard interface, the user is led into the habit of working entirely in Scheme's global environment; all interaction with Scheme -- typed input and printed output -- takes place within that one context. And yet, as our earlier discussions of objects, breakpoints, and package hierarchies showed, there is tremendous utility in being able to interact directly with arbitrary Scheme environments. Of equal importance, the centralization of interactive channels in the standard interface is not in keeping with the semantics of the language. Scheme environments, as mentioned in the first chapter of this report, are first-class objects: they may be passed as arguments, returned as the results of applying procedures, and stored within compound data structures. In all these aspects, there is no special privilege associated with the global environment. Why, then, should there be any special privilege associated with the interactive capabilities of that environment? BOCHSER is thus simply reifying at the interface level the power inherent in the Scheme language itself.

In the following sections of this chapter, we will examine several programs that have been implemented in BOCHSER. The programs, though not large, illustrate some of the ways in which the BOCHSER system might be used. The first example, adapted from the 6.001 curriculum, is intended to suggest BOCHSER's utility as a pedagogical aid; the subsequent examples make use of programming strategies that may not be suggested by the standard Scheme interface. The intent here is to show that the BOCHSER interface, by virtue of its faithful realization of the full power of

Scheme, is a potentially exciting medium for future Scheme programmers.

5.3 Object-Oriented Programming: an Example from 6.001

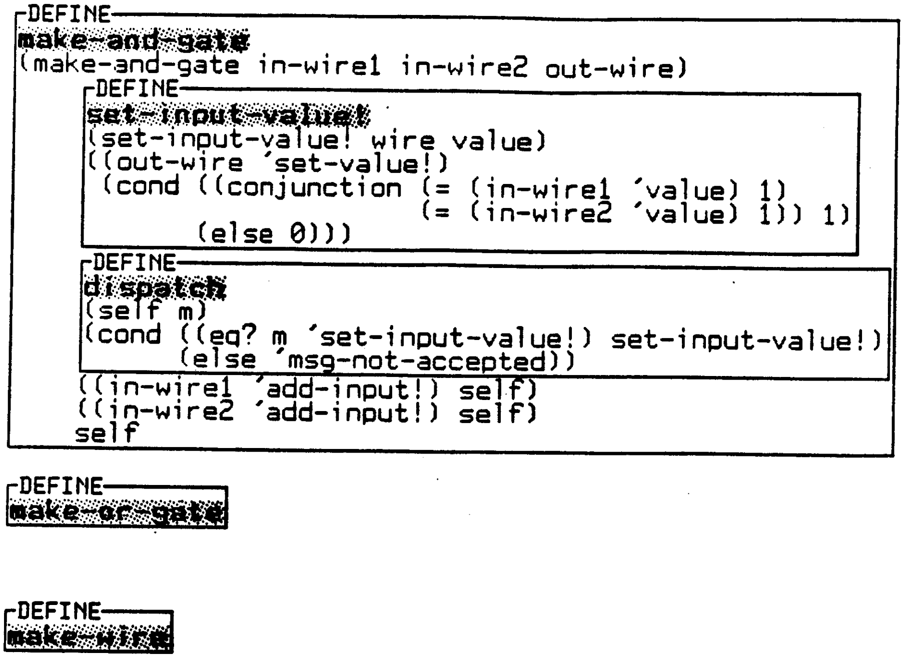
The first example shown here is an extension of the object-oriented programming idea mentioned earlier in the discussion of the MAKE-CTR procedure. In this example, similar to one used in the 6.001 course, our program is a simple simulator for digital logic elements. The objects we will create are of type *wire*, *and-gate* and *or-gate*. Figures 5-1 and 5-2 show the text of the program on the BOCHSER screen. Wire objects can accept messages which inform them of new gates to which they will act as input, and which tell them to set their state to a high (1) or low (0) value. Gates can accept messages telling them whether their input wires have changed state.

In Figure 5-3, a simple combinational circuit has been created in which two input wires are connected to an and-gate, which sends its output, along with that of a third input wire, to an or-gate. The output of the or-gate constitutes the output of the circuit, which thus computes the logical function

```
(or (and input-1 input-2) input-3)
```

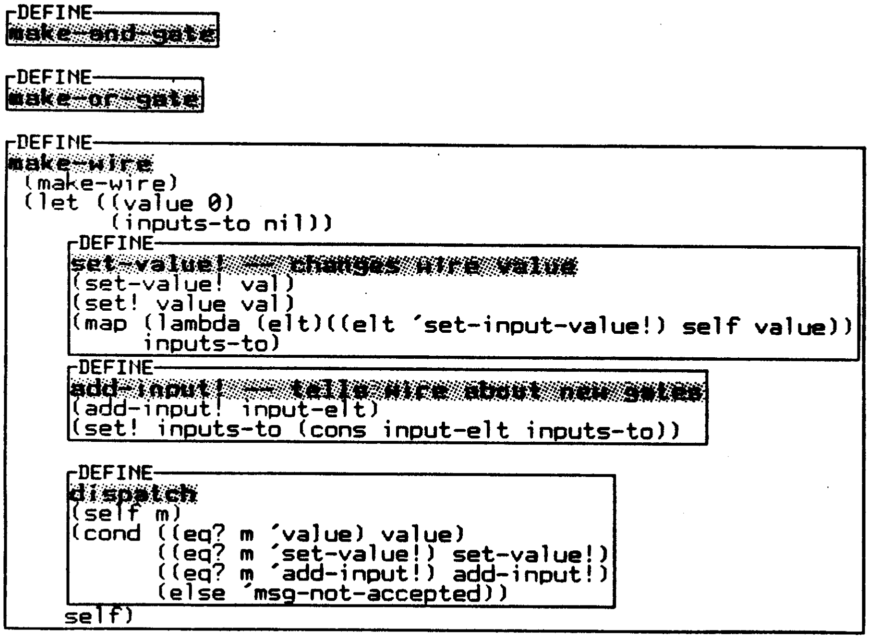
We see in Figure 5-3 that the environments of wire-4 (the output wire of the and-gate) and wire-5 (the output of the or-gate) have their associated environments open to view. In this case, the state of both wires is low -- that is, 0 -- since all three input wires are themselves in low states. In Figure 5-4, the two input wires to the and-gate have been set to high values, and thus outputs of both gates have changed to 1.

The advantages of doing such a simulation in the BOCHSER system as opposed to a more "classical" interface should be readily apparent. Here, the student can select which logic elements to view, and can observe the (possibly manifold) effects of the expressions he or she evaluates. In a standard interface, such observation would have to take place by continually retyping expressions "asking" the various objects to print out their state. A related point is that the code for this BOCHSER example is less complicated than an equivalent simulation for the standard Scheme system would be, since there is no need to include a variety of purely "interface-driven" object methods. For instance, the wire objects shown do not contain a "PRINT-STATE" method, since one can observe the state of a wire by looking at its environment; in the standard Scheme interface, the inclusion of this and similar methods for other objects would be a necessity.



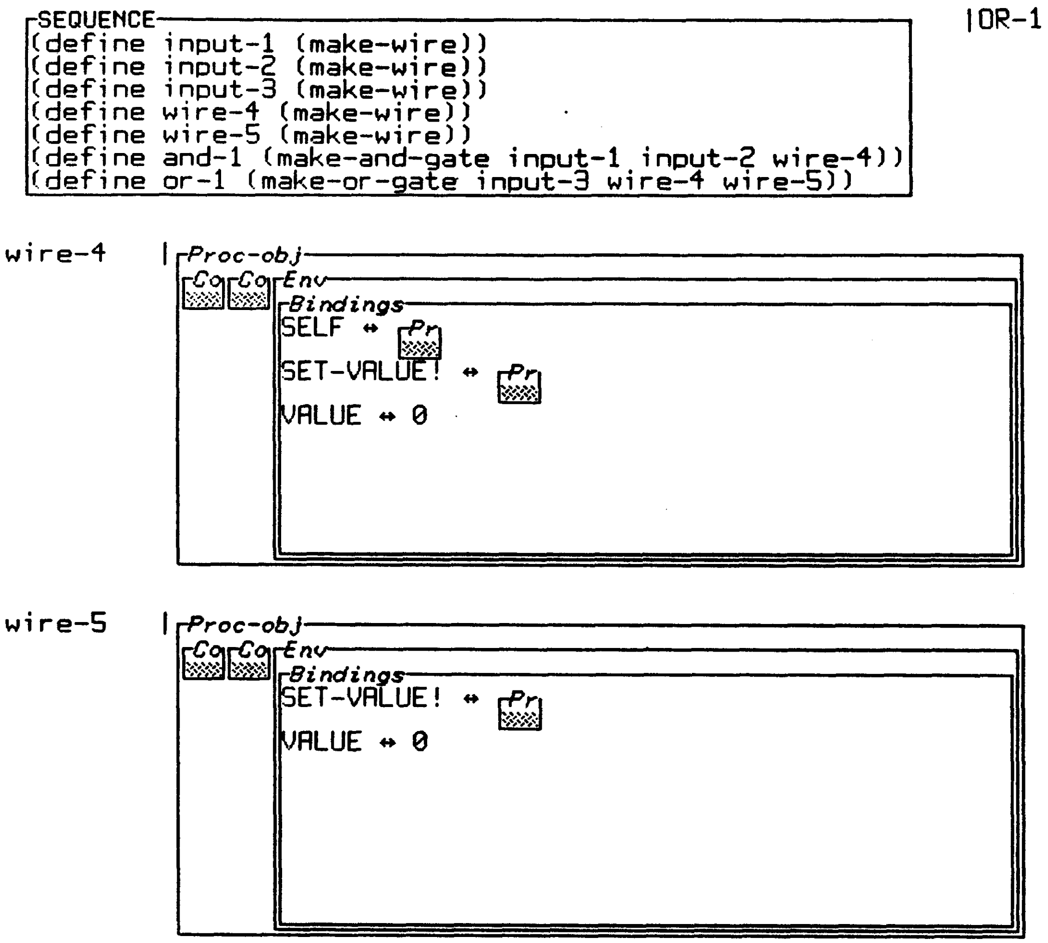
```
DEFINE
make-and-gate
(make-and-gate in-wire1 in-wire2 out-wire)
    DEFINE
    set-input-value!
    (set-input-value! wire value)
    ((out-wire 'set-value!)
     (cond ((conjunction (= (in-wire1 'value) 1)
                         (= (in-wire2 'value) 1)) 1)
           (else 0)))
    DEFINE
    dispatch
    (self m)
    (cond ((eq? m 'set-input-value!) set-input-value!)
          (else 'msg-not-accepted))
    ((in-wire1 'add-input!) self)
    ((in-wire2 'add-input!) self)
    self

DEFINE
make-or-gate

DEFINE
make-wire
```

Figure 5-1:The code for MAKE-AND-GATE. MAKE-OR-GATE is analogous.



```
DEFINE
make-and-gate

DEFINE
make-or-gate

DEFINE
make-wire
 (make-wire)
 (let ((value 0)
       (inputs-to nil))
    DEFINE
    set-value! -- changes wire value
    (set-value! val)
    (set! value val)
    (map (lambda (elt)((elt 'set-input-value!) self value))
         inputs-to)
    DEFINE
    add-input! -- tells wire about new gates
    (add-input! input-elt)
    (set! inputs-to (cons input-elt inputs-to))

    DEFINE
    dispatch
    (self m)
    (cond ((eq? m 'value) value)
          ((eq? m 'set-value!) set-value!)
          ((eq? m 'add-input!) add-input!)
          (else 'msg-not-accepted))
    self)
```

Figure 5-2:The code for making a WIRE object



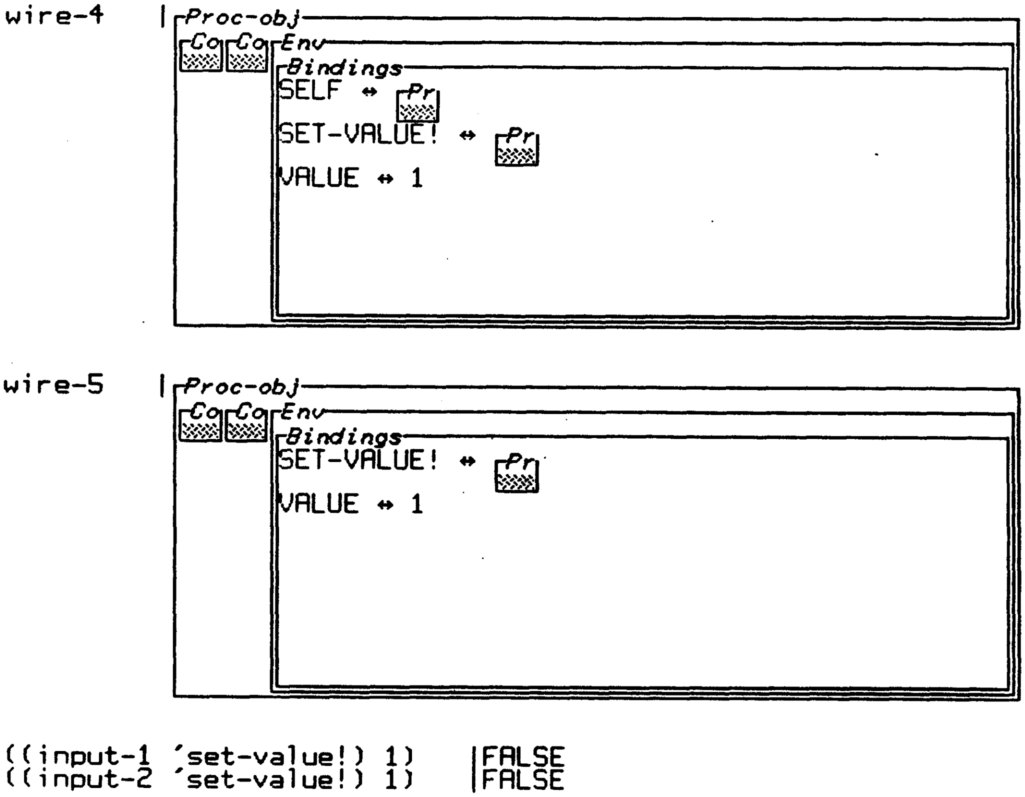
```
SEQUENCE
(define input-1 (make-wire))
(define input-2 (make-wire))
(define input-3 (make-wire))
(define wire-4 (make-wire))
(define wire-5 (make-wire))
(define and-1 (make-and-gate input-1 input-2 wire-4))
(define or-1 (make-or-gate input-3 wire-4 wire-5))
```

```
|OR-1
```

```
wire-4    |Proc-obj
           Co Co Env
                 Bindings
                 SELF ↔ Pr
                 SET-VALUE! ↔ Pr
                 VALUE ↔ 0
```

```
wire-5    |Proc-obj
           Co Co Env
                 Bindings
                 SET-VALUE! ↔ Pr
                 VALUE ↔ 0
```

Figure 5-3:The simulation is initialized, and two wires are examined



```
wire-4     |Proc-obj
            Co Co Env
                  Bindings
                  SELF ↔ Pr
                  SET-VALUE! ↔ Pr
                  VALUE ↔ 1

wire-5     |Proc-obj
            Co Co Env
                  Bindings
                  SET-VALUE! ↔ Pr
                  VALUE ↔ 1

((input-1 'set-value!) 1)     |FALSE
((input-2 'set-value!) 1)     |FALSE
```

Figure 5-4:The two wires change state after the input wires are set to 1

Other accounts of object-oriented programming in BOCHSER, using examples of mutable queues and movable "monster" objects -- both adapted from the 6.001 curriculum -- may be found in Fulton [85]. These examples also describe the value of visible objects to students, both for purposes of comprehension and debugging.

5.4 Object-Oriented Programming: Objects containing Sub-Objects

The second program to be discussed here, outlined in Figure 5-5, is perhaps more interesting in that it illustrates the notion of objects which contain "sub-objects". In this example, we have created a "perceptron" object which examines two-dimensional bit-arrays and seeks out particular features of the bit-patterns within the arrays. For instance, a bit array whose contents are:

```
0   0   0   0   0
0   1   1   1   0
0   1   1   1   0
0   1   1   1   0
0   0   0   0   0
```

might be said to represent a square if we interpret the 1 elements as "set pixels" and the 0 elements as "cleared pixels".

The perceptron that we will discuss is designed to cope with solid shapes (i.e., shapes like the square above, with no "holes" in them) whose borders are completely contained within the presented bit-array. Our perceptron looks for "corners" in the bit-array, whether convex or concave. For example, the perceptron should be able to tell of the square above that it has four convex and no concave corners. The following shape, on the other hand, has five convex and one concave corner:

```
0   0   0   0   0
0   1   1   1   0
0   1   1   1   0
0   1   1   0   0
0   0   0   0   0
```

In its implementation, our perceptron will contain sub-objects which are smaller, lower-level perceptrons -- elements which look specifically either for convex or concave corners in the given bit-array. A "convex corner" perceptron, for example, looks at a particular two-by-two chunk of the bit-array and increments a "convex-corner" counter if it finds that only one of the four observed bits is set to 1. (The four bits in the upper left hand corner of the "square" array have this property, and

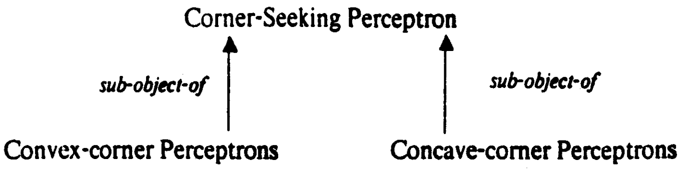
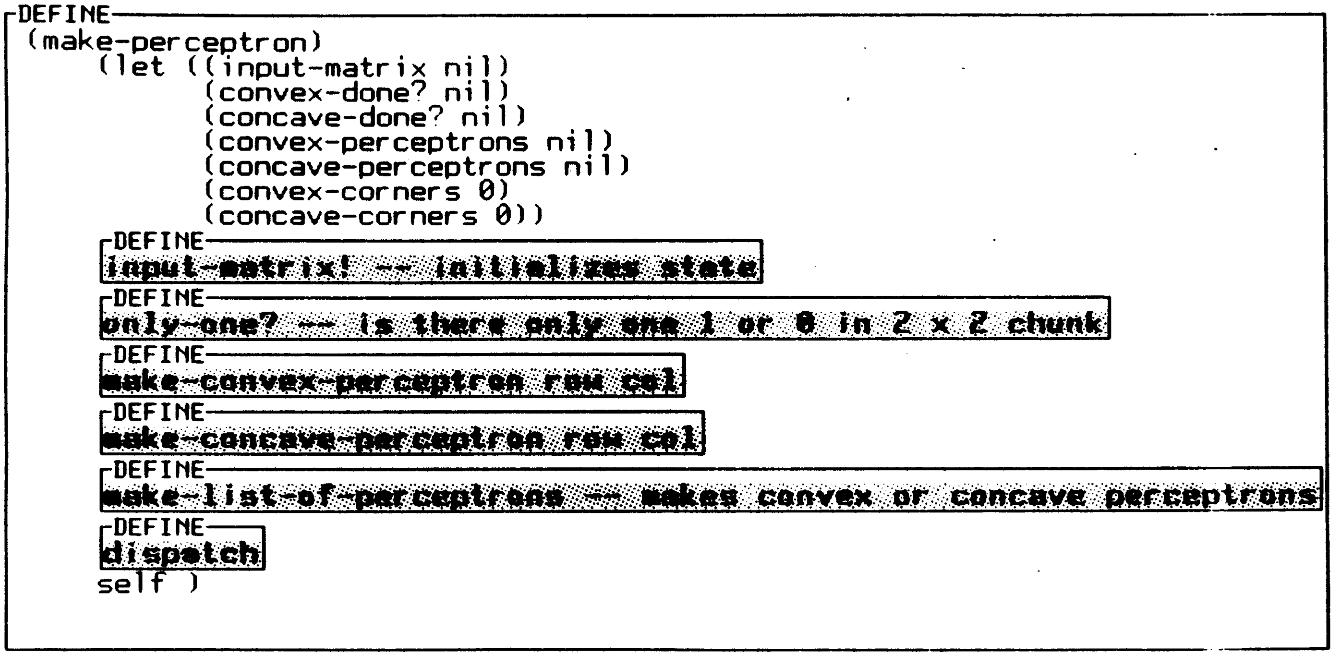

Corner-Seeking Perceptron
sub-object-of
sub-object-of
Convex-corner Perceptrons
Concave-corner Perceptrons

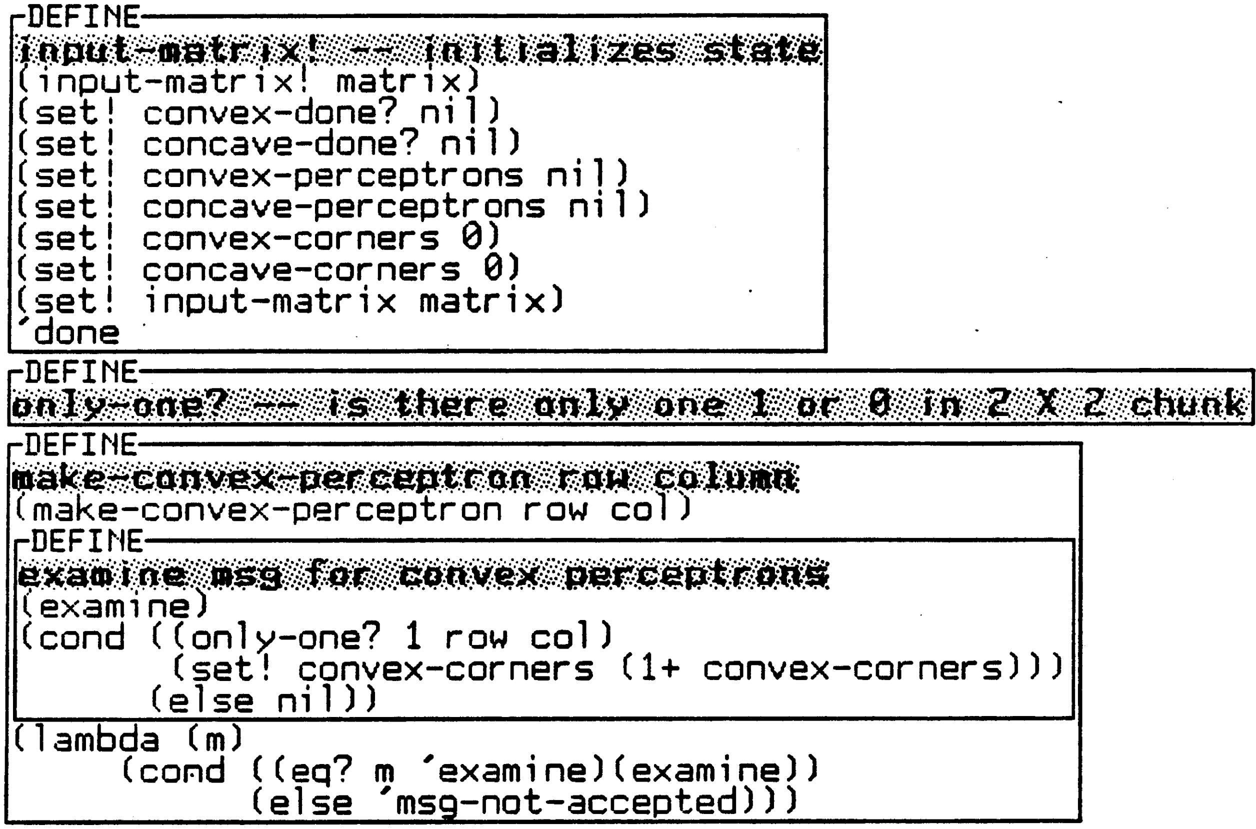
Figure 5-5:An outline of the arrangement of perceptrons to be modeled



```
DEFINE
 (make-perceptron)
      (let ((input-matrix nil)
            (convex-done? nil)
            (concave-done? nil)
            (convex-perceptrons nil)
            (concave-perceptrons nil)
            (convex-corners 0)
            (concave-corners 0))
      DEFINE
      input-matrix! -- initializes state
      DEFINE
      only-one? -- is there only one 1 or 0 in 2 x 2 chunk
      DEFINE
      make-convex-perceptron row col
      DEFINE
      make-concave-perceptron row col
      DEFINE
      make-list-of-perceptrons -- makes convex or concave perceptrons
      DEFINE
      dispatch
      self )
```

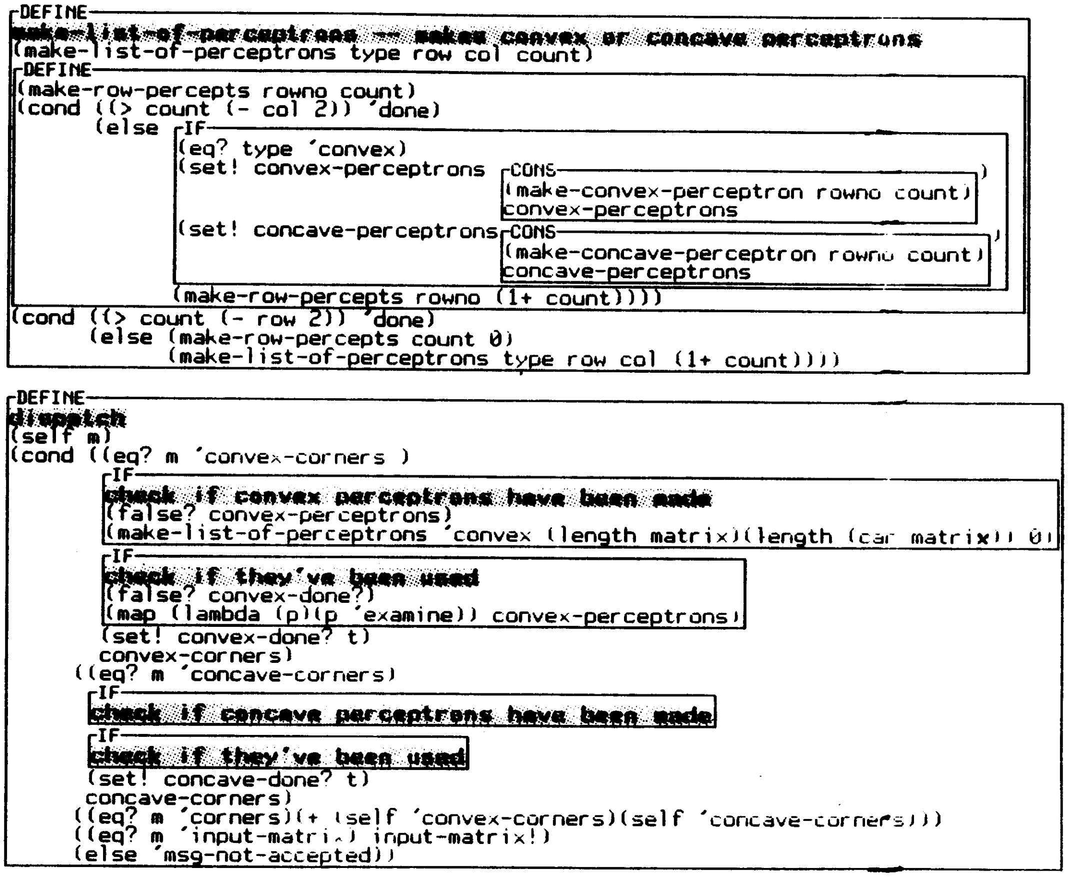
Figure 5-6:An overview of the code for MAKE-PERCEPTRON

Figure 5-7:
The code for initializing the higher-level perceptron with a new matrix as input, and for making a lower-level convex perceptron object. The code for ONLY-ONE?, not shown, simply checks whether the 2 x 2 chunk of the matrix whose upper left corner is given by the ROW and COLUMN values contains one "1" (or "0").



```
DEFINE
input-matrix! -- initializes state
(input-matrix! matrix)
(set! convex-done? nil)
(set! concave-done? nil)
(set! convex-perceptrons nil)
(set! concave-perceptrons nil)
(set! convex-corners 0)
(set! concave-corners 0)
(set! input-matrix matrix)
'done
DEFINE
only-one? -- is there only one 1 or 0 in 2 X 2 chunk
DEFINE
make-convex-perceptron row column
(make-convex-perceptron row col)
DEFINE
examine msg for convex perceptrons
(examine)
(cond ((only-one? 1 row col)
        (set! convex-corners (1+ convex-corners)))
       (else nil))
(lambda (m)
     (cond ((eq? m 'examine)(examine))
           (else 'msg-not-accepted)))
```

Figure 5-8:
The code for MAKE-LIST-OF-PERCEPTRONS creates a list of lower-level perceptrons (of type convex or concave), one for each appropriate matrix point, and assigns the list to the state variable CONVEX- (or CONCAVE-) PERCEPTRONS. The message dispatch code for the higher-level perceptron is shown underneath.



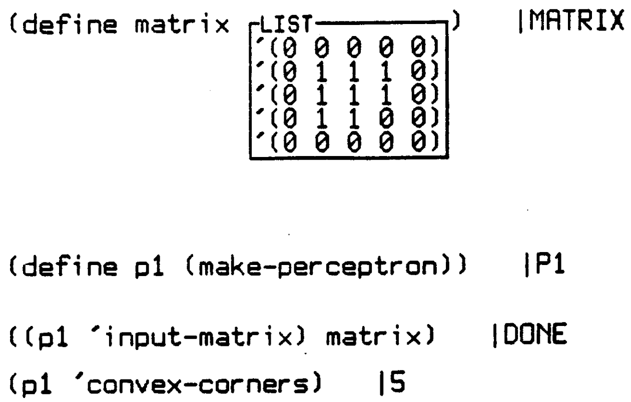
```
DEFINE
make-list-of-perceptrons -- makes convex or concave perceptrons
(make-list-of-perceptrons type row col count)
 DEFINE
 (make-row-percepts rowno count)
 (cond ((> count (- col 2)) 'done)
       (else IF
             (eq? type 'convex)
             (set! convex-perceptrons CONS                                  )
                                      (make-convex-perceptron rowno count)
                                      convex-perceptrons
             (set! concave-perceptrons CONS                                  )
                                      (make-concave-perceptron rowno count)
                                      concave-perceptrons
             (make-row-percepts rowno (1+ count))))
(cond ((> count (- row 2)) 'done)
      (else (make-row-percepts count 0)
            (make-list-of-perceptrons type row col (1+ count))))

DEFINE
dispatch
(self m)
(cond ((eq? m 'convex-corners )
        IF
        check if convex perceptrons have been made
        (false? convex-perceptrons)
        (make-list-of-perceptrons 'convex (length matrix)(length (car matrix)) 0)
        IF
        check if they've been used
        (false? convex-done?)
        (map (lambda (p)(p 'examine)) convex-perceptrons)
        (set! convex-done? t)
        convex-corners)
      ((eq? m 'concave-corners)
       IF
       check if concave perceptrons have been made
       IF
       check if they've been used
       (set! concave-done? t)
       concave-corners)
      ((eq? m 'corners)(+ (self 'convex-corners)(self 'concave-corners)))
      ((eq? m 'input-matrix) input-matrix!)
      (else 'msg-not-accepted))
```

thus indicate the presence of a convex corner.) Similarly, a "concave corner" perceptron will observe a two-by-two chunk of the bit-array and fire if it sees that only one bit has been set to 0. Both of these low-level perceptron types only exist within the environments of larger "corner-seeking" perceptrons; there is no global definition of these types of objects.

Our sample perceptron object accepts a number of messages (see Figures 5-6, 5-7, and 5-8). The "input-matrix" message tells the perceptron to initialize its various state variables and input a new bit-array to observe. The "convex-corners" message tells the perceptron to count convex corners in the bit-array; if necessary, the perceptron will create a set of lower-level "convex-corner" perceptrons, one for each two-by-two chunk of the bit-array, and send each a message to examine its own particular chunk of the array. The "concave-corners" message works similarly, again creating lower-level perceptrons if needed. Finally, the "corners" message tells the perceptron to sum the existing convex and concave corners and output the result. A demonstration of the working program can be seen in Figure 5-9. Figure 5-10 shows the environment of a perceptron object, within which we may examine the environment of a lower-level perceptron.

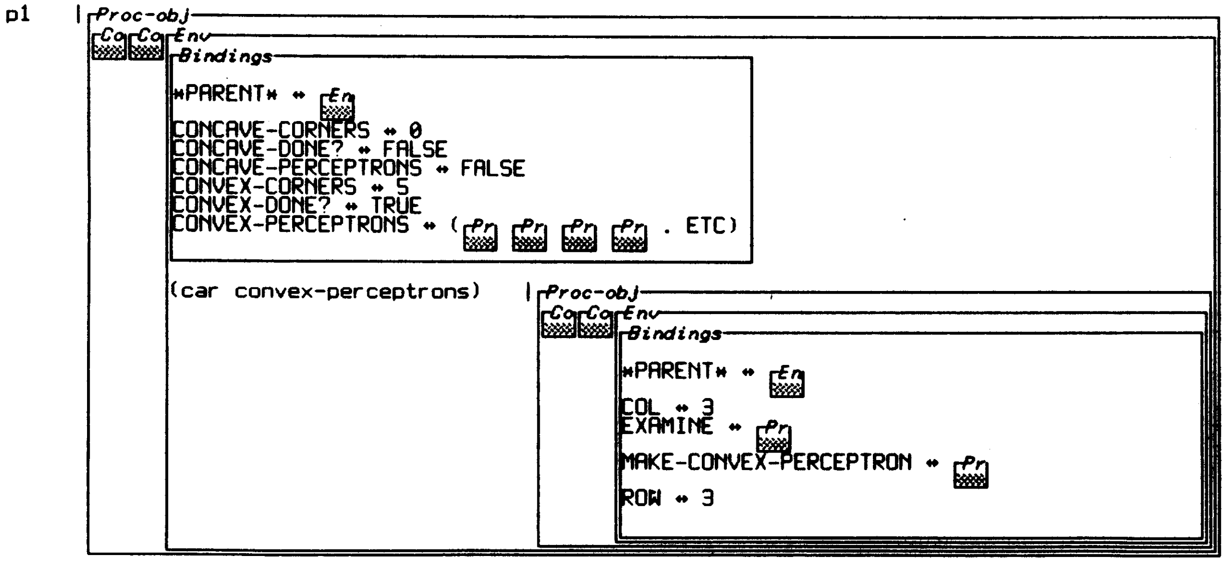
Several aspects of this code are worthy of mention. First, since the lower-level perceptrons are sub-objects of the larger perceptron, they are both able to access bindings in the environment of that larger object. For instance, both the "convex" and "concave" perceptrons make use of the "only-one?" helping procedure defined within the larger perceptron's environment; similarly, both are able to access the value of the "input-matrix" state variable. If these low-level perceptrons had instead been defined within the global environment, they would have had to contain their own local (and redundant) bindings for these purposes, or would have had to send messages to other objects to access this information. In more elaborate versions of the same idea, low-level perceptrons might effectively communicate information to each other via setting and resetting state variables in the outer environment (that is, the outer environment acts as a sort of "bulletin board" for the low-level component objects), rather than by sending explicit messages to one another.

More importantly, this code is a closer reflection of the way in which the programmer would like to think of the perceptron object -- namely, as a compound object containing loosely coupled sub-objects. In the standard Scheme interface, it would be much easier to implement this program with the low-level perceptron objects defined globally, since one could not access the higher-level object's environment for debugging purposes; and besides complicating the code itself, this decision would lead to a program which is inappropriate to its domain. Programming with sub-objects in



```
(define matrix ┌LIST─────────┐)    |MATRIX
               │'(0 0 0 0 0) │
               │'(0 1 1 1 0) │
               │'(0 1 1 1 0) │
               │'(0 1 1 0 0) │
               │'(0 0 0 0 0) │
               └─────────────┘

(define p1 (make-perceptron))    |P1

((p1 'input-matrix) matrix)    |DONE

(p1 'convex-corners)    |5
```

Figure 5-9:A demonstration of the working perceptron program



```
p1  |┌Proc-obj
     ┌Co┌Co┌Env
            ┌Bindings
            *PARENT* ↔ ┌En
            CONCAVE-CORNERS ↔ 0
            CONCAVE-DONE? ↔ FALSE
            CONCAVE-PERCEPTRONS ↔ FALSE
            CONVEX-CORNERS ↔ 5
            CONVEX-DONE? ↔ TRUE
            CONVEX-PERCEPTRONS ↔ (┌Pr ┌Pr ┌Pr ┌Pr . ETC)

            (car convex-perceptrons)    |┌Proc-obj
                                         ┌Co┌Co┌Env
                                                ┌Bindings
                                                *PARENT* ↔ ┌En
                                                COL ↔ 3
                                                EXAMINE ↔ ┌Pr
                                                MAKE-CONVEX-PERCEPTRON ↔ ┌Pr
                                                ROW ↔ 3
```

Figure 5-10:A view of a perceptron sub-object

Scheme enables the user to maintain an abstraction barrier with respect to the problem: in this particular instance, one need only work with higher-level perceptrons, whose internal sub-objects may, in a variety of situations, be safely ignored.

5.5 Production Systems

Our final programming example is a simple version of a system that works with productions -- that is, rules of the form "If a certain situation applies, then take the following action". The use of productions is a common (perhaps the most common) technique in the development of expert systems; typical examples of expert systems which make essential use of such rules are the MYCIN system developed at Stanford and the R1 system in use at Digital Electronics Corporation (cf. [Hayes-Roth 83]). The program to be shown here is a scaled-down version of a system described in Winston [77].

Our production program is one whose aim is to identify an animal based on information presented about that animal. The input to the program will be a list of facts about the unknown animal -- e.g., "(hairy (eats meat)(color tawny))" -- and based on this information the program will use various productions to draw conclusions about the creature's species. For instance, one rule might be of the form:

```
"If: The animal is hairy,
Then: It is a mammal."
```

Another might read:

```
"If: The animal is a mammal, and it eats meat,
Then: It is a carnivore."
```

One of the difficulties involved in creating a production system lies in organizing the collection of productions into meaningful units. For instance, in small production systems, one might simply have a list of all existing productions, and in order to solve a particular problem one would go through the list applying the productions rules, one by one. (In the situation above, it would be important to have the two sample rules "fire" in the order presented, since the conclusion of the first rule can be used as a premise for another.) The problem with this idea is that it can rapidly become impossible to understand the mutual relationships between productions -- which must fire before which, which can be ignored based on the results of others, and so on. Moreover, there is no simple way of looking at the collection of productions and getting an idea of the domain structure which they are intended to represent. For example, in Winston's version of the "animal" program, all productions are assumed to be equally "active" at any one time; the image is of many competing processes all waiting for their premises to be met by some external stimulus. However, a better notion would be to have the applicability of the rules which are only meaningful for mammals contingent upon the identification that the animal is indeed a mammal. It is difficult to gain any feeling for animal taxonomy from examining Winston's rules; with a larger production system it would be impossible.

Winston is well aware of this situation, and writes of such systems:

> "The *advantage* of not needing to worry about the interactions among the productions can become the *disadvantage* of not being able to influence the interactions among the larger number of productions.... One possible solution, of course, is to partition the facts and the productions into subsystems such that at any time only a manageable number are under consideration."

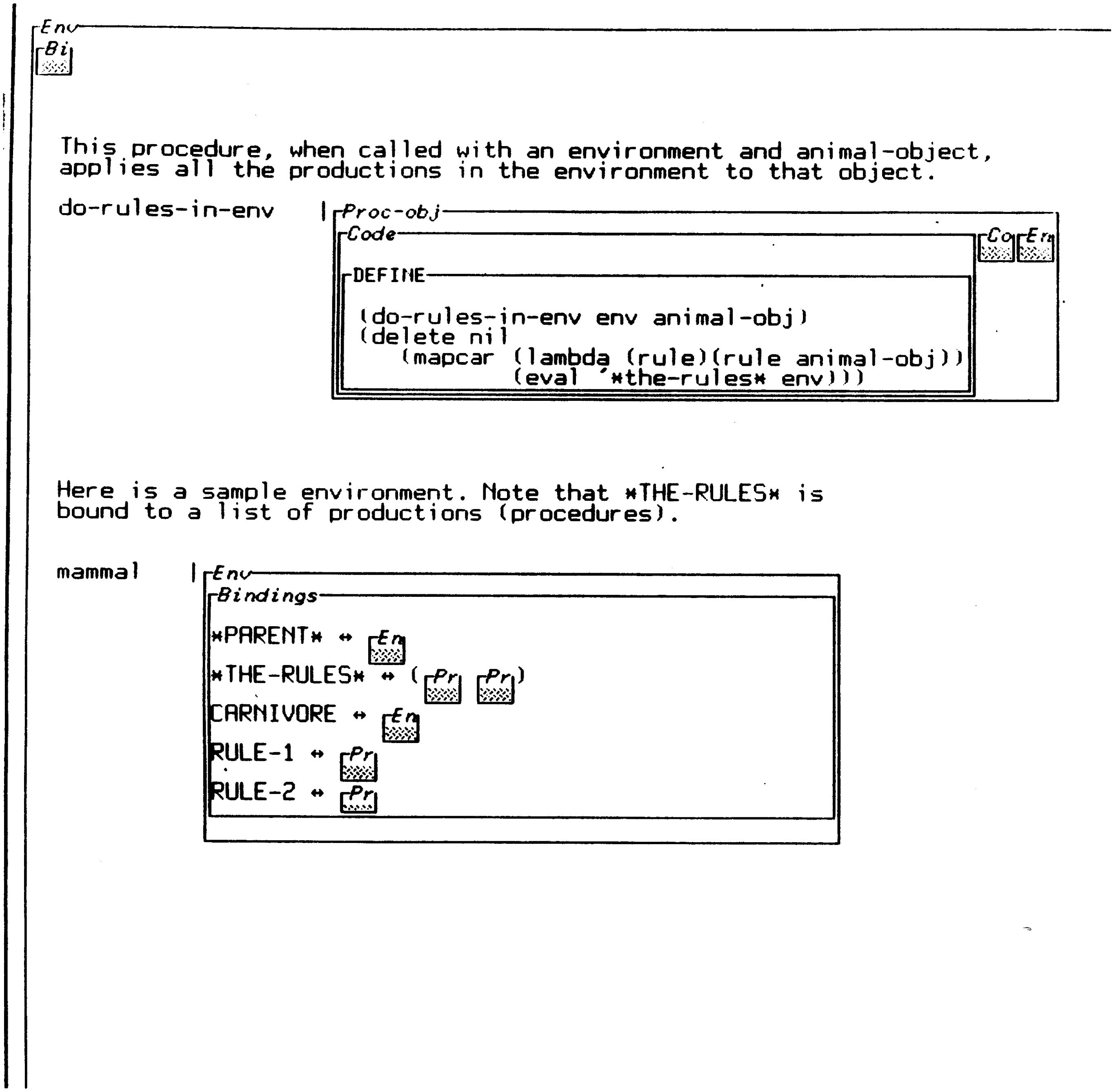
Our BOCHSER program uses exactly this suggested strategy. The way in which productions will be partitioned is by using environments: rules applicable only to, say, mammals will be represented as procedures in a particular "mammal-examining environment". The basic idea is that each environment contains a small set of rules applicable to one piece of the problem domain; within each such environment, the variable name *THE-RULES* will be bound to a list of all the local rules. In order to find out which rules in a particular environment apply to a particular animal, we invoke the procedure DO-RULES-IN-ENV, as follows:

```
(DO-RULES-IN-ENV env animal-object)
```

A look at the code in Figures 5-11 and 5-12 will help to explain this idea. Essentially, DO-RULES-IN-ENV applies each of the rules in the particular environment (as found within the list *THE-RULES*), one at a time, to the given animal-object. Each rule will return either the atom FALSE (if the rule does not apply), or a list identifying that this rule has fired and which subsequent productions in other environments may have applied as a result. Thus, rules may themselves make recursive calls to the DO-RULES-IN-ENV procedure. It should also be noted that the environments are arranged hierarchically according to the taxonomy being modeled: the ALL-ANIMALS environment has as sub-environments MAMMAL and BIRD, the MAMMAL environment has CARNIVORE and UNGULATE sub-environments, and so on.

A demonstration of the program is shown in Figure 5-13. Here, we type in the expression:

Figure 5-11:
Inside the ALL-ANIMALS environment. The code for the DO-RULES-IN-ENV procedure, and a view of the MAMMAL sub-environment. Note that inside the MAMMAL environment there are CARNIVORE and UNGULATE environments (the latter cannot be seen just now).



This procedure, when called with an environment and animal-object, applies all the productions in the environment to that object.

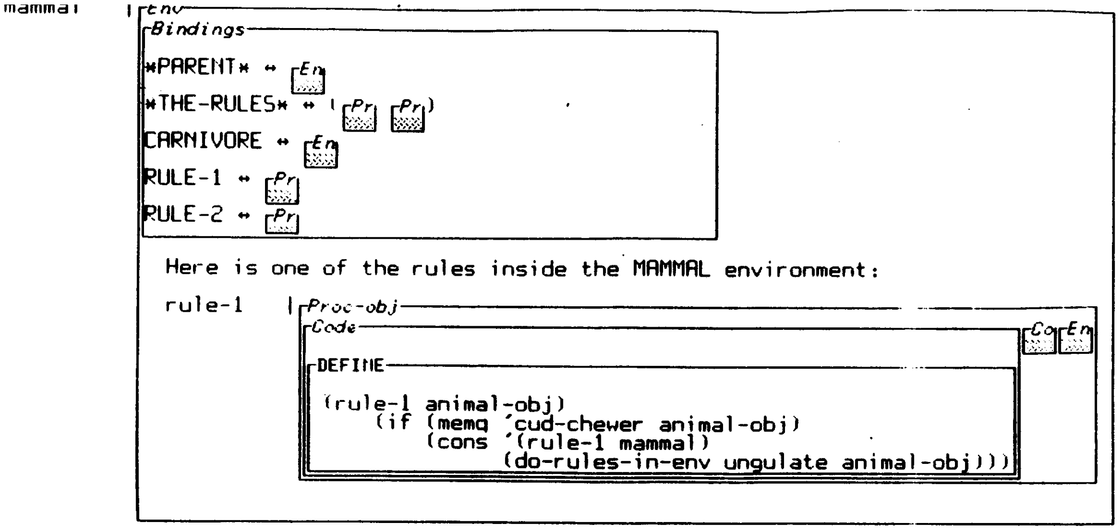
do-rules-in-env

```
DEFINE
(do-rules-in-env env animal-obj)
(delete nil
   (mapcar (lambda (rule)(rule animal-obj))
                (eval '*the-rules* env)))
```

Here is a sample environment. Note that *THE-RULES* is bound to a list of productions (procedures).

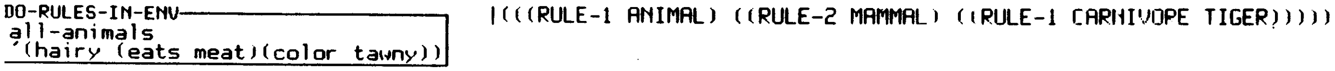
mammal

```
*PARENT* ↔ En
*THE-RULES* ↔ (Pr Pr)
CARNIVORE ↔ En
RULE-1 ↔ Pr
RULE-2 ↔ Pr
```



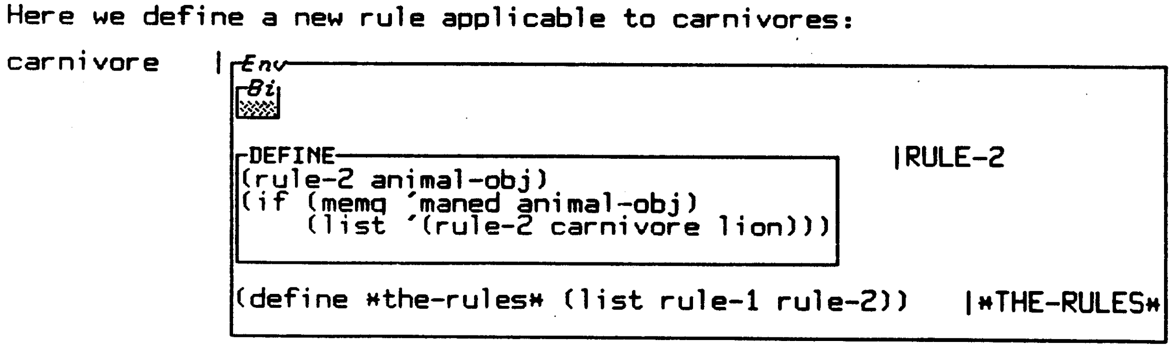
```
mammal   |Env
          Bindings
          *PARENT* ↔ En
          *THE-RULES* ↔ (Pr Pr)
          CARNIVORE ↔ En
          RULE-1 ↔ Pr
          RULE-2 ↔ Pr

            Here is one of the rules inside the MAMMAL environment:

            rule-1    |Proc-obj
                        Code                                        Co En
                        DEFINE
                         (rule-1 animal-obj)
                             (if (memq 'cud-chewer animal-obj)
                                 (cons '(rule-1 mammal)
                                       (do-rules-in-env ungulate animal-obj))))
```

Figure 5-12:One of the rules applicable to mammals only



```
DO-RULES-IN-ENV                         |(((RULE-1 ANIMAL) ((RULE-2 MAMMAL) ((RULE-1 CARNIVOPE TIGER)))))
 all-animals
 '(hairy (eats meat)(color tawny))
```

Figure 5-13:Examining an animal-object in the ALL-ANIMALS environment



```
Here we define a new rule applicable to carnivores:

carnivore   |Env
             Bi

             DEFINE                                       |RULE-2
             (rule-2 animal-obj)
             (if (memq 'maned animal-obj)
                 (list '(rule-2 carnivore lion)))

             (define *the-rules* (list rule-1 rule-2))    |*THE-RULES*
```

Figure 5-14:Defining a new rule for carnivores

```
(do-rules-in-env all-animals '(hairy (eats meat)(color tawny)))
```

The result of the procedure call is a list that indicates which rules fired in which environments; the nesting of the list indicates the dependencies of certain environments upon others. The list tells us that RULE-1 in the ALL-ANIMALS environment fired, which in turn caused the rules in the MAMMAL environment to be tested, at which point RULE-2 fired, and so on. Thus, the result of our expression not only provides an identification of the animal but gives us a map of its "reasoning" process.

Working with the production system that results from this kind of structuring is actually extremely straightforward. For instance, Figure 5-14 shows the process of adding a new rule. Here, we wish to note that if the animal is a carnivore with a mane, then it is a lion. To add the rule, we go into the CARNIVORE environment and define the new procedure, adding it into the *RULES* list as well. Now the rule will be applied any time the "carnivore productions" are applied.

There are other advantages to this sort of structuring. An immediate observation is that the rules are kept relatively simple by virtue of the information implicit in their surrounding environment. For example, the rule in our system analogous to the second "if-then" one above has as its premises merely that the animal eats meat -- the knowledge that the animal is a mammal is assumed, since the rule is defined within the MAMMAL environment. Moreover, the placement of rules within local environments, and the hierarchical relationships of these environments to each other, tells us a great deal about the domain being represented: we know, simply from examining the system, that carnivores and ungulates are mutually exclusive types of mammals.

An even bigger advantage is seen when the need arises to debug the system. In this situation, the user is able to test out pieces of the system individually by evaluating rules only in specified environments. For example, by entering the MAMMAL environment and trying out calls to DO-RULES-IN-ENV within that environment, only the productions applying within MAMMAL and its sub-environments, like CARNIVORE, will be run. The same technique can be used to employ the system in a top-down fashion -- that is, to use the strategy of hypothesizing that the consequences of a particular rule apply and then seeing whether there are premises that match that rule. To work in this way, one simply treats the environment argument to DO-RULES-IN-ENV as an indicator of the "hypothesized conclusion". For instance, if we wish to assume that the animal is a carnivore, we could evaluate:

```
(do-rules-in-env (eval '*parent* carnivore)
                 '(hairy (eats meat)(color tawny)))
```

By examining the result of this expression, we can see whether the assumption that the animal is in the superclass of carnivores (that is, the assumption that the animal is a mammal) justifies our hypothesis that the animal is a carnivore. Recursive calls to DO-RULES-IN-ENV could now determine whether our assumption that the animal is a mammal is itself justified.

Chapter Six

BOCHSER and Other Language Interfaces

There is very little in the BOCHSER interface that can fairly be called new -- virtually every individual element of the system has been included or at least presaged in one form or another by earlier language interfaces. Features of the BOCHSER system may be found in various traditions in interface design -- particularly those of graphical languages, "structure-editing" languages, and educational "visible computer model" systems. Most fundamentally, BOCHSER reflects its origins in the Boxer system; many characteristics of the interface are of course derived from its use of the Boxer editor, and the "interface philosophy" of BOCHSER, involving the aforementioned notions of "detuning" and "diffusing functionality", is firmly in the Boxer tradition.

This chapter will attempt to provide a general overview of the BOCHSER system's relation to other language interfaces. The intent is not only to place BOCHSER in perspective historically, but also to illuminate some of the strengths and weaknesses in the system to be discussed later. One of the additional themes that emerges from this exercise is the power and interest of the Scheme language; Scheme forms a fascinating framework in which to experiment with the interface ideas that motivate the BOCHSER system.

6.1 BOCHSER and Boxer

Without question, BOCHSER owes most of its fundamental ideas to the Boxer system (cf. [di Sessa 85b], [Boxer 84]). As already noted, such BOCHSER features as the use of boxes as editor objects, and the integration of interpreter and editor, are also crucial elements of Boxer. Nevertheless, there do exist differences between the two systems. Probably the best way to summarize these is to say that the systems have adapted one editor to different underlying semantics; the Boxer language is roughly based on LOGO (though with important changes), while BOCHSER is an implementation of Scheme. It is beyond the scope of this paper to provide anything like a complete description of Boxer; the following paragraphs, then, will focus only on the salient points of comparison between Boxer and BOCHSER.

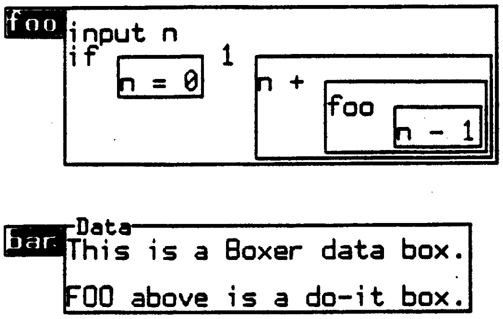
One of the concepts supported by the Boxer interface, but notably absent in BOCHSER, is that of "naive realism": there is an identification in Boxer between screen objects and the "language objects" that they are intended to represent. For example, a procedure in Boxer may be represented as a labeled "doit-box", as in Figure 6-1. In order to change the code of this procedure, the user would move the editor cursor into the box and simply edit the procedure text. Thus, if one wished to change the illustrated FOO procedure to have the effect of computing factorials, one would enter the box with the cursor and change the "+" symbol to "*". Similarly, boxes of the "data-box" type, also shown if Figure 6-1, may be edited directly (data-boxes are Boxer's general data-structuring element, rather analogous to lists in LISP and LOGO).

The BOCHSER system, based as it is on Scheme, does not enforce the idea of naive realism; there is not a one-to-one identification between a screen object and a language object. Many screen objects may represent the identical Scheme object. We have already seen this in the case of environments, where several environment boxes represent one environment. Indeed, any situation where two variable names are bound to one object will also exhibit this discrepancy with the Boxer model; in BOCHSER, we would observe two separate bindings represented on the screen, such as:

```
A <---> (1 2 3)
B <---> (1 2 3)
```

where each binding refers to the linking of a name and, as it happens, one identical object. Moreover, there is no equivalent in BOCHSER to Boxer's technique of "direct modification". of objects. In order to change the code of a procedure, or to change the value of any binding, one has to evaluate expressions within a Scheme interpreter area. One cannot, as Boxer semantics would imply, type changes directly into a bindings box on the screen; similarly, one cannot change the code of a procedure-object by editing the code box for that object, but only by redefining the procedure through a DEFINE expression. In fact, as mentioned earlier, only environment boxes and syntax boxes in BOCHSER are directly editable; all other species of boxes are "read-only".

There are limitations inherent in both approaches. In Boxer, an additional mechanism -- that of "ports" -- is introduced in order to achieve the desirable effects of sharing data structures. For instance, if box B is defined as a port to box A, as in Figure 6-2, then any changes typed into box B will be reflected in box A. Box A is referred to, in this situation, as the "target" of port B. The point of including ports in Boxer is that many boxes on the screen may come to represent, in some sense, the same object, even though there is still one unique screen object that is identified as the target of all

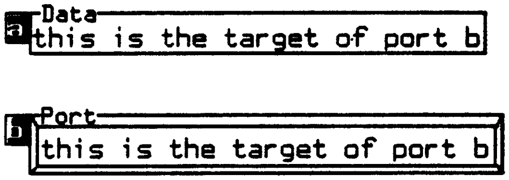


```
input n
if n = 0  1  n + foo n - 1
```

This is a Boxer data box.

FOO above is a do-it box.

Figure 6-1:A do-it box and data box in Boxer



this is the target of port b

this is the target of port b

Figure 6-2:Box B is a Boxer port. Typing inside B will change A as well

the others; and indeed, if this target element is deleted from the screen, then all the ports which formerly pointed to it essentially become undefined. Besides having to introduce an additional mechanism to handle sharing, Boxer runs into occasional difficulty with objects that don't fit easily into identification with a screen region; for example, there is at present no object in Boxer analogous to a circular list.

The problems that BOCHSER experiences in this regard are complementary to those of Boxer. One would like to have a direct handle on Scheme objects; and even though one can always see an accurate representation of those objects within a bindings box, the "real object" remains to some degree an abstraction. For example, to understand the sharing inherent in the situation where list B is bound to the CDR of list A, one has to maintain a mental model of list sharing which BOCHSER can support, but not fully present. A related issue is that it would often be simple and desirable to edit bindings or procedure code directly, as one sees them on the screen. The editor function that allows the user to copy code from a procedure-object's code box into an interpreter area is intended to simplify the process of redefining procedures, but it would often be easier to redefine a procedure simply by altering the contents of its code box. There are some avenues along which BOCHSER's constraints vis-a-vis "editable boxes" could be loosened, but one must be cautious in making changes of this sort. To take an example, suppose again that we have a situation in which B is bound to the CDR of A. Included in our environment's bindings box, then, we might see two lines like the following:

```
A <--> (1 2 3)
B <--> (2 3)
```

Now, if we move our cursor into the bindings box and change the first element of the B list to 4, should A change as well? That is, our editing change may be analogous to

```
(SET-CAR! B 4)
```

or to

```
(SET! B '(4 3))
```

and there is no way of distinguishing these intentions. In other situations, however -- such as editing code within a procedure-object box -- it might be worthwhile to experiment with the naive realism idea in BOCHSER.

A second major point of difference between the Boxer and BOCHSER systems involves the

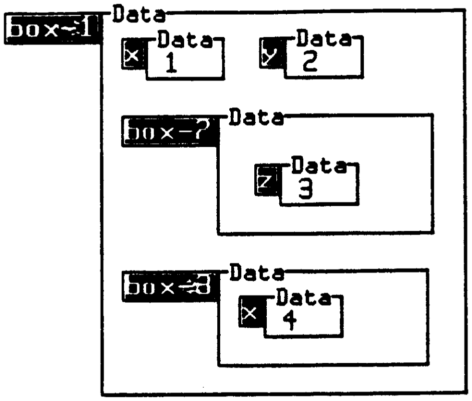
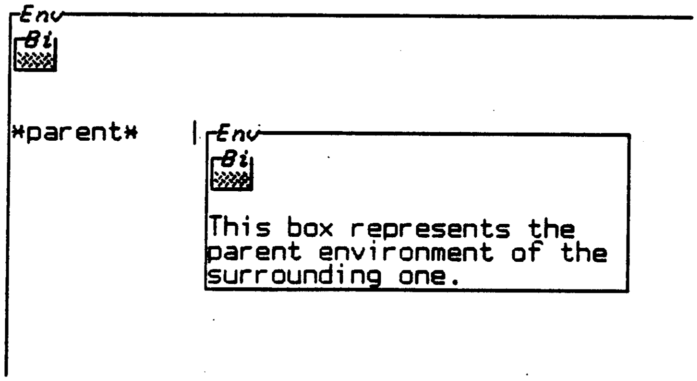


Figure 6-3:In Boxer, environment structure corresponds to screen structure



This box represents the parent environment of the surrounding one.

Figure 6-4:In BOCHSER, an environment box may, on the screen, surround its logical parent

meaning of box containment. In Boxer, all box boundaries implicitly specify environment boundaries as well; that is to say, every box represents its own particular environment. The bindings which are accessible within a box are those resulting from definitions performed within the box itself and all its parent boxes. Figure 6-3 illustrates this idea. Here, BOX-1 has bindings for variable names X and Y; BOX-2 has bindings for X, Y, and Z; and BOX-3 has bindings for X and Y, where the former binding results from the "interior" definition of X. This identification between box boundaries and environment boundaries makes for a straightforward mapping between the information presented on the screen, and the logical structure of Boxer environments.

In BOCHSER, the situation is somewhat more problematic. It is trivial to arrange the screen such that box containment fails to correspond with environment containment. Figure 6-4 shows one such arrangement: the user has evaluated the name *PARENT* within an environment box, and the result is an environment box which corresponds to the parent environment of the one in which we are typing. Thus, the user is still forced to maintain some internal model of an environment hierarchy; and again, although BOCHSER can help in this regard, it will not in general present a complete and unambiguous representation of that hierarchy to the user.

A final point of comparison between the two systems really stems from a difference between LOGO (Boxer's closest predecessor in terms of language semantics) and Scheme. The BOCHSER system employs a lexical scoping discipline, while Boxer uses dynamic scoping. The choice in Boxer is motivated by adherence to a two-stage model of procedure invocation:

1. Copy the text of the procedure into the calling environment on the screen.

2. Execute the text of the called procedure.

This model is extremely simple to grasp in the context of working with the Boxer system: the environment of procedure execution is associated with the place from which it is called. It is, however possible to run into difficulty when a procedure is invoked from an environment in which some of its free variables have unknowingly been re-bound. Essentially, the Boxer system has opted for understandability and representational consistency at the possible expense of what is generally acknowledged to be, in large systems at least, a less problematic lexical scoping discipline.

The BOCHSER system, as we have seen, makes essential use of Scheme's lexical scoping. BOCHSER's model of procedure invocation is thus somewhat more difficult to understand than Boxer's (and, indeed, LOGO's). Still, the graphical representation of procedures as "objects associated with environments" is intended to alleviate the understandability problem as much as possible while preserving the tractability of lexical scoping in the construction of large programs.

6.2 Other Interfaces

As noted above, virtually every feature of the BOCHSER system can be found in one form or another in other language interfaces. The notion of "shrinking and expanding structures" to designate language objects can be seen in the Cornell Program Synthesizer for PL/CS (an instructional dialect of PL/1) described by Teitelbaum and Reps [81], in which entire syntactic units like IF-THEN statements can be inserted and deleted like one character, and in which "comment templates" act rather like labeled boxes in BOCHSER. The Xerox Star interface (cf. [Smith 83]), in its notion of "opening up" icons such as file folders to display their contents, also involves this notion. In fact, the use of icons in general seems strongly associated with "shrunken" and "expanded" representations: multiply-sized icons can be found in interfaces like that of the PICT/D graphical language developed at the university of Washington [Glinert 84] and the "Robot Odyssey" adventure game [Dewdney 85]. In PICT/D, program icons can be expanded to reveal their pictorial contents; in the Robot Odyssey game, the user enters a "shrunken" robot in order to see its interior expanded to full-screen size. Additional examples of the "expandable icon" idea are not hard to find.[7]

The concept of integrating usually distinct programming language "subsystems" (e.g., editors, interpreters, debuggers, and so on) is widely associated with the SMALLTALK system [Goldberg 83] [Tesler 81]. SMALLTALK supports an object-based programming language with various menu-operated subsystems: a "browser" for editing objects and doing file-management, an "inspector" for examining objects, and "notifier" and "debugger" views for error notification and program debugging, respectively. A less well known but nonetheless interesting example of integration can be found in Wilander's description of the PATHCAL system, an interpretive system for Pascal [Wilander 80]. In the PATHCAL system, as in BOCHSER, program text is edited in the same "mode" in which it is debugged and run. Moreover, PATHCAL checks the syntax of expressions as they are being typed in by the programmer; thus, some of the features of a compiler are also present within the system.

[7] Cf. the representation of procedure activations in Turbak's GRASP system [Turbak 86], Ciccarelli's example of an icon-style interface [Ciccarelli 84], and the syntax-directed editor of the Pecan system [Reiss 84], among many others.

Numerous programming language systems have been implemented which attempt to provide a visible language-appropriate model of computer operation. Typically, the systems are designed to teach the particular language being illustrated, rather than as true production programming environments. Languages which have been represented in this way include BASIC [Barr 76], assembly code [Leap 84], FORTRAN [Shapiro 74], microprogramming code [Parker 84], and a simple language intended to demonstrate the use of concurrent processes [Colville 83]. More ambitious systems, which demonstrate language operation graphically for the purposes of debugging and program development, include Lieberman's TINKER system for LISP [Lieberman 84], and the PECAN system in use at Brown University [Reiss 84]. Both of these systems present information about the control structures of programs as they run, and can be used to obtain multiple perspectives (i.e., both textual and graphical) of the objects being manipulated by the developing program. The PECAN system also includes a syntax-directed editor like that of the Cornell Program Synthesizer. Franklyn Turbak's GRASP system (presently in development) is, like BOCHSER, designed around the Scheme language; in GRASP, procedure invocations are constructed and represented graphically, and the changing state of program control is visible as a little animated man tracing his way through the program as it runs [Turbak 86].

Turbak's system represents, in part, yet another tradition of language interface design -- the construction of graphical languages. The general motivation of these systems is to replace at least part of the procedural representation of programs with pictorial elements instead of text. One of the best known early efforts in this direction is Sutherland's graphical programming system [Sutherland 66]; subsequent graphical languages include AMBIT/G, a SNOBOL-like language implemented at M.I.T. [Rovner 69], Smith's PYGMALION system [Smith 75], and the aforementioned PICT/D language.

There are many dimensions along which BOCHSER may be compared to these systems. The "box as editor object" seems to be a particularly useful construct both for the purposes of managing screen space (as in the Cornell Program Synthesizer's editor) and as a uniform general-purpose iconic device with which to delimit procedures, environments, regions devoted to graphics (Boxer has "graphics boxes" for this purpose) and other standard language constructs. The fact that a box is treated as one character by the editor, although simple in description, has extremely positive consequences for users: for instance, when BOCHSER prints out a list of procedures, the screen representation of this list is one of boxes surrounded by parentheses, and this text may be traversed with the cursor just like any other text on the screen. This incorporation of "multiply-sized elements" within the context of a text editor is not seen in most systems employing expandable icons, while the multi-purpose nature of the box object makes it more versatile than a language-specific syntactic "template".

BOCHSER's level of integration as a system is enhanced both by its use of the Boxer editor and by the Scheme language's notion of first-class environment objects. The former enables an easy diffusion of the interpreter and editor functionalities; the latter, by virtue of its inclusion in breakpoint boxes, significantly enriches BOCHSER's debugging capabilities. In this respect, BOCHSER may be easier to work with than SMALLTALK, since the notion of environment is much more fundamental within the Scheme language than are SMALLTALK's "inspector" debugging views within that language. Beyond that, however, it must be admitted that the level of BOCHSER's integration is still embryonic when compared with that of SMALLTALK; the runtime error system and file-management capabilities of the former are at present rather spare. This topic will be addressed once more in the next chapter.

BOCHSER, as has been noted, is intended to provide a useful "model-explicit" programming environment for students of Scheme. The last two sample programs shown in the previous chapter, however, were meant to illustrate that the BOCHSER interface can potentially transcend classification as a purely pedagogical device. The power that a user may derive by easy interaction with Scheme environments is applicable to the design of large and interesting programs. Many of the educational systems mentioned above are geared only toward the creation of small programs. To take two examples, the system described by Leap for teaching assembly code is limited to programs of 100 words, of which only 22 are visible on the display screen; while programs written in the PICT/D graphical language cannot employ procedures with more than four input parameters (since each parameter is represented by one easily distinguishable color), and the parameter values must be numeric.

An area in which BOCHSER is clearly weak is in its depiction of the control paths of Scheme programs. In this respect, BOCHSER is no more informative than the standard Scheme interface (less, in fact, since the 9836 interface includes a stepper). Many of the systems described above pay particular attention to this element of program understanding: the PECAN system, to take the most spectacular instance, shows the control path of a program not only through highlighting portions of program code, but also through dynamic depictions of abstract syntax trees and Nassi-Schneiderman flowcharts. At the very least, some kind of stepping mechanism should be added to BOCHSER to enhance the comprehensibility of the Scheme programs written within the system.

Finally, although BOCHSER does contain some graphical component in that it is not a purely textually-based Scheme system, that component is minimal. This is a deliberate choice, inasmuch as the intent was to enhance the representation of Scheme objects, but not the straightforward syntax of Scheme expressions themselves. Besides, as di Sessa points out in his discussion of Boxer, there is some benefit in allowing a computational environment to take advantage of the user's familiarity with natural language; in largely graphical systems, even the act of speaking or writing about programs requires the use of a textual "metalanguage" -- a language for describing the graphical elements of programs. It would nevertheless be desirable to incorporate graphical objects into BOCHSER, not as elements of Scheme syntax or new representations of Scheme primitives, but rather as data objects analogous to numbers or procedure objects. This idea will be taken up later within the context of adding new object types to BOCHSER.

Chapter Seven

BOCHSER - Current Status, Problems, and Future Directions

The present BOCHSER system is more in the nature of a prototype than a true production-quality system. Nevertheless, it has been used as a demonstration tool for Scheme on numerous occasions (to 6.001 students among others), and to unanimously positive response. Besides the author, the only extensive programming projects in the system have been done by Jim Fulton, an M.I.T. senior [Fulton 85]. His opinion, too, was highly positive (despite noting problems with the system):

> "Perhaps the best thing that I liked about BOCHSER was that it made working with Scheme *a lot* of fun. Many of the things that I didn't understand when I first learned the language fell quickly into place after just a few minutes of playing with environments. A number of my friends who are learning the language now responded with envy when I described BOCHSER. In spite of its implementation problems, BOCHSER presents a very good model [of] how a language can be melded with its environment to produce a better tool for programming."

That said, however, there are still many existing problems with the system, and a variety of additional positive features that could be incorporated into the BOCHSER interface. The remainder of this chapter will be devoted to a description of BOCHSER's flaws, as well as potentially exciting future directions in which the system might be taken.

7.1 Speed

BOCHSER is too slow. This is probably its most glaring weakness at the moment, and is the first problem mentioned by Fulton in his review. In particular, those operations that require heavy use of the display functions -- for example, transforming a large area of the screen, with many sub-boxes, into a syntax box -- take a demoralizingly long time. Most problematic of all, the system evinces the cardinal sin of programming environments; it gets slower as the number of defined procedures gets larger.

BOCHSER will have to run much faster in order to be a serious alternative to existing systems.

It is likely that a thorough but unimaginative rewrite of the code could increase its speed by a factor of at least five -- but even greater improvement will be necessary (some operations, such as opening up new environment boxes with a large number of bindings, take more than five minutes). Eventually, then, a working BOCHSER system would require some radical improvements either in the design of many of its key algorithms, the hardware on which it is implemented, or the level at which its software runs (e.g., some of the display algorithms might eventually be programmed at the microcode level). An incremental compiler for BOCHSER procedures would also be an important addition in making the user's programs run faster. Actually, improvement in all of these directions is eminently possible; an increased speed of operation by about two orders of magnitude appears feasible, and would make BOCHSER a viable production-quality Scheme system.

7.2 Customizability of the BOCHSER Interface

At present, outside of the formatting capabilities inherent in the Boxer editor, there is very little customizability to the present BOCHSER interface. For example, it was mentioned earlier that a BOCHSER programmer might achieve the effect of having an "editor buffer" by maintaining an environment box for this purpose; and that it might be desirable in this context to set various editor parameters independently within the two boxes. At present, there is nothing in the BOCHSER interface that would allow this kind of customization. Fulton, too, mentions the value of being able to do such things as specify properties of box icons (e.g., default box size, whether bindings boxes will automatically appear within environment boxes, and so on).

Essentially what is being proposed is that BOCHSER be augmented by some kind of "presentation system" (to use Ciccarelli's [84] terminology) for the properties of the BOCHSER interface itself. It would be extremely desirable to experiment in this direction, although there is an inevitable tension between the range of user options and simplicity of a given system: beyond a certain point, the complexity of the presentation system itself outweighs the power of the options it offers. Nevertheless, greater user control over the BOCHSER interface itself would be, in the system's present state, an unambiguously positive step.

7.3 Scrolling Boxes

One possible area for customizing boxes might be to create scrolling portions of an environment box, or perhaps environment boxes in which the interpreter area has a default scrolling property. The discussion in the fourth chapter was mainly devoted to the advantages of non-scrolling boxes -- and indeed, given an either/or situation, the non-scrolling box is strongly preferable. Even so, there are common situations in which the user wishes to evaluate an expression only once; and to make the user responsible for erasing all such one-time expressions on the screen is bothersome. Often in the course of programming in BOCHSER, the screen becomes cluttered with no-longer-wanted expressions, and extra time must be taken to eliminate them from the screen.

This problem could perhaps be alleviated by allowing the user to specify whether a particular environment or syntax box has a scrolling property, or by instituting scrolling regions (analogous to "panes" in many existing window systems) of boxes.

7.4 Hardcopy

One of the thornier issues in making a production-quality BOCHSER interface involves generating hardcopy of Scheme programs created in the system. It is an inherent dilemma in interface design that the more one exploits the power of the display screen, the more problems are created for representing programs on paper, and BOCHSER has not escaped this problem. We have already seen a number of BOCHSER programs that, when represented on the screen, contain too many nested boxes to be translated directly to the printed page.

Although difficult, this problem is not unapproachable. An experimental box-printout program was developed for an earlier version of the Boxer system; this program used multiple pages, with flowchart-style pointers as off-page references, to represent boxes that were too large or complex to depict on one page. More elaborate programs might in fact be developed for BOCHSER, allowing the user to format hardcopy on the screen so as to print out or highlight only meaningful portions of boxes. In any case, the non-trivial relationship between the BOCHSER screen and the printed page should be seen as an impetus to develop better methods of representing complex Scheme programs, rather than as an excuse not to exploit the power of display screens.

7.5 New Object Types

One of the more exciting prospects generated by an interface like BOCHSER is that it suggests various extensions to the range of object types usually supported by Scheme. For example, Scheme systems tend to be weak in the area of text manipulation; although there are string objects in the language, there are only a minimal number of primitive procedures for dealing with them. More important, one often wishes to deal with text in a two-dimensional representation; that is, viewing the text not as a long string but rather as a series of rows (or rows divided into columns). An interesting project for BOCHSER, then, would be to include a two-dimensional "text box" object for which row and column selectors, as well as a powerful set of text-editing procedures, could be supplied.

Along the same lines, it would be desirable to experiment with "graphics boxes", and Scheme primitives for manipulating them, in BOCHSER. This would be an ideal use of the BOCHSER screen, as well as suggesting a variety of new programming projects. Graphics boxes could be seen as Scheme objects in the same vein as numbers or procedures, with their own associated primitives and semantics. The Boxer system has historically made excellent use of graphics, while at present the 9836 Scheme system is limited to rather simple line-drawing primitives and a single "display object" -- namely, the entire terminal screen employed in "graphics mode".

Another interesting type of object to'include in BOCHSER -- one not new to Scheme -- would be continuations. At present, these are not included in the system, although breakpoint boxes, inasmuch as they represent processes which can be completed at a later time, have some similarities to what might eventually be thought of as "continuation boxes". One challenge in including this type of object would be to develop a representation for continuations as straightforward and useful as are BOCHSER's environment boxes.

BOCHSER's file-management system could also be augmented with "directory boxes"; similarly, communication with other systems might be accomplished with boxes that can be shared in real time with other terminals. For instance, one environment might be represented by environment boxes on two different screens; and any changes made to the bindings of the environment would be reflected on both screens simultaneously. This kind of addition would be useful for extending BOCHSER's applications to those of local-area networks (another of Fulton's desiderata). On a less elaborate scale, "mail box" objects might also be included in the system so that BOCHSER users could communicate via electronic mail. All of these species of boxes -- for file-management, networking and mail -- if made into sensible Scheme objects, would greatly enhance the system's level of integration.

7.6 Stepper

BOCHSER's power of illustrating the Scheme "language model" would be served well by including a stepper in the system -- that is, a facility in which the various sub-expressions encountered in the course of applying a procedure could be evaluated one at a time. Although one can use breakpoints within procedures to generate some of the effects of this type of stepper, an explicit set of primitives for this purpose would be useful for students and within the debugging process. As noted earlier, this inclusion would help to remedy the weakness of BOCHSER in presenting Scheme's control structure in comparison to previous "model-explicit" systems for other languages.

7.7 Multiprocessing Schemes

Finally, it is interesting to consider how the BOCHSER interface could be extended to more complex and elaborate future versions of Scheme which include notions of parallelism as part of the language semantics. At this point, such an extension to BOCHSER is purely a matter of speculation, but one might imagine, say, boxes representing individual processes or subtasks. These "process boxes" could each maintain individual user interfaces, just as environment boxes do now; thus, programs that involve the generation of individual processes might be debugged via inserting breakpoints into the program and then interacting with individual processes to examine their states. In any case, however notions of parallelism are dealt with in a BOCHSER-like system, it is certain that the more complex and non-intuitive the underlying Scheme "language model" becomes, the more necessary an interface like that of BOCHSER will be for Scheme programmers at all levels of sophistication.

Chapter Eight

Conclusion: Toward a Pragmatics of Programming Languages

The design of a programming language interface is often seen as an effort distinct from the design of the language itself. Typically, the computer science community, when it "defines" a language, thinks in terms of various levels of formal description of the language itself, such as a Backus-Naur grammar representation for syntax, a description in terms of denotational semantics for the "meaning" of language constructs, or a quasi-formal English definition of the language for programmers. Rarely if ever are there standards or even suggestions for what the programmer in this language will see. A typical example of this phenomenon is the report of the Algol 60 committee, whose work is regarded as among the most influential in the history of language design: there is not a word in their report about how this language will be presented to the programmer [Algol 63].

This omission within the Algol 60 report is actually quite reasonable; at the time the report was drafted there were few options in the design of language interfaces, and little concern for the subject in general. But the neglect of these considerations has not changed all that much in the past two and a half decades. Although Alan Kay [84] writes,

> "The user interface was once the last part of a system to be designed. Now it is the first. It is recognized as being primary because, to novices and professionals alike, what is presented to one's senses *is* one's computer."

... nevertheless, the fact is that the prevalent practice in the design of new computer language concepts and new computer architectures is still to think of the user interface as an afterthought. One case in point is illustrated by the description of the "Cosmic Cube" multiprocessing project at the California Institute of Technology; after a fascinating discussion of the hardware itself, and the programming projects to which the new machine is applicable, the author acknowledges that a good interface to the machine has yet to be developed [Seitz 85].

The BOCHSER system which is the subject of this report is unfortunately still an instance of the "interface following the language" phenomenon -- the point of the system is to take an existing language and supply the most informative and powerful possible interface to it. It is interesting that one result of this process is that new types of Scheme objects and programming applications are suggested, as the previous chapters have demonstrated. A similar effort for other languages, like PROLOG or ADA, would no doubt be highly desirable; but future language design projects need to include considerations of user interface in the very earliest stages. To date, the SMALLTALK, INTERLISP, and Boxer projects have been among the very few to really follow Kay's suggestion.

Part of the problem in this regard is that the computer science community, on the whole, has paid too little attention to how programming languages are defined by use rather than by abstract definition. The academic definition of computer languages regards languages as entities that exist on paper, and whose "power" and "expressiveness" are somehow divorced from what people might understand of them. A language like FP is described by Backus [78] as powerful because it is mathematically tractable, even though to this day (to the author's knowledge) no one has ever written a large program in it; the language is simply too abstruse with regard to normal patterns of human communication (although it might indeed be a good language for computers themselves to work with). One might similarly imagine another language with a GO-TO statement that in practice is rarely used; computer scientists would likely publish diatribes about the dangerous semantics of the language, while programmers would quietly go about the business of writing perfectly understandable code.

Semiologists, in studying systems of symbolic communication, refer to a dialectic between "language" and "speech": the former is seen as the abstract system of communication, considered apart from its actual use, while the latter is seen in the actual communication which takes place (cf. [Barthes 64]). The important point to make about this distinction is that the two entities, language and speech, are seen as in some sense dependent upon one another: speech can only take place given the framework of an existing language, but the language is learned and altered over time via its realization in speech. Similar considerations apply to programming languages as well; in order to understand them, we will have to regard languages not merely as a set of syntactic rules to express an abstractly-defined semantics, but rather as what they are in practice -- a means of human expression and communication. It will therefore be important to study what programmers actually do with languages: which concepts they use, and which they ignore; what kinds of programming projects the language suggests; and how programmers generate their ideas and convey their knowledge to one another. Ultimately, the point of interface design must be to provide its users not merely with an automated realization of a language manual, but with a tool for expanding their very human powers of communication.

Appendix A

BOCHSER Editor Commands

What follows is a description of the editor commands available in BOCHSER. The prefix "Ctrl-" before a key name indicates that the given key is to be pressed while the "Control" key of the 3600 is held down; the prefix "Meta-" indicates the analogous situation for the 3600's "Meta" key. The prefix "Ctrl-Meta-" indicates that both the Control and Meta keys are to be held down when using the given key. The reader familiar with EMACS will note that most of the cursor movement commands are identical or similar to those of that editor (cf. [Stallman 81]). Note that for those commands which refer to "characters" and "words", a box is treated as both a character and a word (i.e., a one-character word). Thus, for example, if the cursor is to the left of a box and the user types Ctrl-F, the cursor will move to the right of the box -- one character position forward.

Commands followed by a star (*) are commands special to BOCHSER -- that is, they do not exist in the original Boxer editor.

CURSOR MOVEMENT; TEXT EDITING

Ctrl-A	Move cursor to beginning of line
Ctrl-B	Move cursor back one character
Ctrl-D	Delete character following the cursor
Ctrl-E	Move cursor to end of line
Ctrl-F	Move cursor forward one character
Ctrl-K	Delete row following cursor
Ctrl-L	Redisplay the screen
Ctrl-N	Move cursor to the next line
Ctrl-O	Open a line at the cursor
Ctrl-P	Move the cursor to the previous line

Ctrl-V	Vertically scroll the box surrounding the cursor
Ctrl-Y	Retrieve the last deleted item and insert it at the cursor position
Ctrl-(	Move the cursor into the box beside it
Ctrl-)	Move the cursor out of the box surrounding it
Rubout	Delete the character preceding the cursor
Meta-B	Move the cursor back by one word
Meta-D	Delete the word following the cursor
Meta-F	Move the cursor forward one word
Meta-V	Vertically scroll backward the box surrounding the cursor
Meta-Rubout	Delete the word preceding the cursor
Mouse-Middle	Move the cursor to the spot pointed to by the mouse arrow

BOCHSER BOXES

Ctrl->	Expand the box surrounding the cursor
Ctrl-<	Shrink the box surrounding the cursor
Symbol-K	Create a label line for a box (or move the cursor to an already-existing label line)
Mouse-Left	Shrink the box surrounding the mouse arrow; double-left shrinks the box to its shrunken state
Mouse-Right	Expand the box surrounding the mouse arrow; double-right expands the box to full-screen size

BOCHSER EXPRESSIONS

Line(*)	Evaluate the expression preceding the cursor (if the cursor is to the right of a closing parenthesis; to evaluate an atomic expression, the LINE key is used with the cursor to the right of the expression on an otherwise empty row).
Ctrl-Space(*)	Save an expression to be restored with the Meta-Space key.
Meta-Space(*)	Restore an expression saved with Ctrl-Space. These two keys, Ctrl-Space and Meta-Space, are typically used to retrieve expressions from Code boxes so that they can be edited and evaluated inside an environment box.

Meta-Line(*) When executed with the cursor to the right of a closing parenthesis of a list whose CAR is atomic, this key creates a syntax box with that CAR as header, replacing the original expression.

BREAKPOINTS

Ctrl-Meta-C(*) When executed with the cursor to the right of a breakpoint box, the computation is continued from the point of the breakpoint; the result of the computation replaces the breakpoint box on the screen.

Ctrl-Meta-I(*) When executed with the cursor to the right of a breakpoint box, the computation is continued from the point of the breakpoint, ignoring if necessary any future breakpoints encountered; the result of the computation replaces the breakpoint box on the screen.

Appendix B

BOCHSER Primitive Procedures and Special Forms

This appendix contains a brief description of the primitive procedures, pre-defined variable names, and special forms in BOCHSER. The first two categories are lumped together in the following tables, since they simply indicate those bindings that are found in the BOCHSER global environment; there is of course no deep difference between bindings with procedure values (like CAR) and those with non-procedure values (like FALSE) in the sense that in either case the binding might be altered by a DEFINE or SET! expression.

In most cases, the BOCHSER primitives and special forms are identical or analogous to their Scheme counterparts, and the description of those terms here is minimal. Readers interested in a thorough explanation are referred to the current Scheme manual [Scheme 84]. In those cases where the BOCHSER term is not found in Scheme or is significantly different from its counterpart, a longer description is provided.

Primitive Procedures and "Pre-defined" Variable Names

*	Multiply
+	Add
-	Subtract
-1+	Decrement
1+	Increment
<	Less-than predicate
=	Numeric equality predicate
>	Greater-than predicate
apply	Scheme APPLY
atom?	Atom predicate

car Scheme CAR

cdr Scheme CDR

cons Scheme CONS

div Divide

eq? Scheme EQ? predicate

equal? Scheme EQUAL? predicate

eval Scheme EVAL -- takes an expression and an environment as arguments, and evaluates the expression in the given environment

false This name is bound to the Scheme FALSE boolean object.

false? Predicate which returns TRUE if its argument evaluates to FALSE, and FALSE otherwise.

force Scheme FORCE. "Forces" the evaluation of a delayed object.

list Scheme LIST

nil Bound to FALSE. In future versions, the empty list object and the boolean false object may be separated.

print Analogous to Scheme PRINT. Prints out its argument expression on the line following the cursor's present position, moving (if necessary) subsequent already-existing lines downward on the screen.

procedure-env Takes as argument a procedure object and returns the environment object associated with that procedure.

read-scheme-file Takes a file-name as argument. The file-name should contain a specially-formatted list saved from an EMACS editor buffer; this primitive will return a large quoted expression containing the sequence of list expressions in the file.

read-world Takes a file-name as argument. The file should be a BOCHSER world saved with SAVE-WORLD;the result of the READ-WORLD expression is the saved environment.

replace-world Takes a file-name as argument. When invoked from within the global environment, the present BOCHSER world will be replaced by that in the saved file (which should have been created using SAVE-WORLD).

save-world Takes a file-name as argument. Saves the current BOCHSER world in an appropriately-named file. Note that only bindings are saved; thus, objects which are not accessible via the bindings of the global environment will not be saved.

set-car! Scheme SET-CAR!

set-cdr! Scheme SET-CDR!

set-print-depth Takes a numeric argument. Tells the BOCHSER editor how many top-level elements of a list to print out when a list value is returned. Elements beyond the print-depth value are represented by ". ETC" at the end of the list.

sqrt Scheme SQRT

t Bound to TRUE

trace Analogous to Scheme TRACE-ENTRY

true Bound to TRUE

untrace Analogous to Scheme UNTRACE

Special Forms

bkpt Takes an optional symbol argument to be printed out when the breakpoint is encountered. When this special form is evaluated, a breakpoint box is returned. See the description of BOCHSER's breakpoint facility in the fourth chapter of this report.

comment The remainder of the expression is unevaluated.

cond Scheme COND

conjunction Scheme CONJUNCTION

cons-stream Takes two arguments; evaluates the first argument and returns a cons pair of the result of that evaluation and a DELAYed version of the second argument. That is,

```
    (CONS-STREAM <exp1><exp2>)
```

can be viewed as an alternate way of writing

```
    (CONS <exp1> (DELAY <exp2>))
```

define Scheme DEFINE

delay Scheme DELAY. Takes an expression and returns an object of type "delayed-object". If this result is used as the argument to FORCE, the original argument expression to DELAY is evaluated in the environment in which the DELAY expression was evaluated.

disjunction Scheme DISJUNCTION

if Scheme IF

lambda Scheme LAMBDA

let Scheme LET

make-environment

Analogous to Scheme MAKE-ENVIRONMENT. Takes as argument a list of Scheme expressions to be evaluated sequentially within the newly created environment. This is slightly different from the syntax of the Scheme version, in which MAKE-ENVIRONMENT may take a series of expressions as arguments, rather than a list of expressions.

named-lambda Scheme NAMED-LAMBDA

named-let Scheme NAMED-LET

quote Scheme QUOTE

sequence Scheme SEQUENCE

set! Scheme SET!

undefine Takes a name as argument, like DEFINE. When evaluated, the most local binding for that name is eliminated. Actually, this special form might better be named UNSET!, since it is closer to an inverse of SET! than DEFINE, in that it may undo bindings outside the most local frame of the environment in which it is evaluated.

References

[Abelson 85]
Abelson, Harold and Sussman, Gerald Jay with Sussman, Julie
Structure and Interpretation of Computer Programs
M.I.T. Press, Cambridge, Massachusetts, 1985

[Adelson 84]
Adelson, Beth and Soloway, Elliot
A Model of Software Design
Yale University Dept. of Computer Science Research
Report #342 October 1984

[Algol 63]
Naur, P. et. al.
Revised Report on the Algorithmic Language ALGOL 60
Communications of the ACM 6, 1963, pp. 1-20

[Backus 78]
Backus, John
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs
Communications of the ACM 21:8, 1978, pp. 613-641

[Barr 76]
Barr, Avron; Beard, Marian; and Atkinson, Richard C.
The computer as a tutorial laboratory: the Stanford BIP project
International Journal of Man-Machine Studies 8:5 1976 567-596

[Barthes 64]
Barthes, Roland
Elements of Semiology
Hill and Wang, New York, 1968

[Boxer 83]
M.I.T. Educational Computing Group
Boxer I (documentary video)
Produced by Umbrella Films, Brookline, Massachusetts

[Boxer 84]
Lay, Ed; Klotz, Leigh; and Neves, David
Boxer User Manual
M.I.T. Educational Computing Group, 1984

[Boxer 85]
M.I.T. Educational Computing Group
Boxer II: Applications of an Integrated Computing Environment
(documentary video)

Produced by Umbrella Films, Brookline, Massachusetts

[Carroll 85]
Carroll, John M. and Mack, Robert L.
Metaphor, computing systems, and active learning
International Journal of Man-Machine Studies (1985) 22 39-57

[Ciccarelli 84]
Ciccarelli, Eugene C. IV
Presentation Based User Interfaces
Ph.D. thesis, Massachusetts Institute of Technology, August 1984

[Clinger 85]
Clinger, Willam (editor), et. al.
The Revised Revised Report on Scheme, or, An UnCommon Lisp
AI Memo No. 848, Massachusetts Institute of Technology, August 1985

[Colville 83]
Colville, John
A Pictorial Demonstration of Concurrent Processes
ACM SIGCSE Bulletin 15:4 1983

[Dewdney 85]
Dewdney, A. K.
Computer Recreations column in *Scientific American*, July 1985

[di Sessa 85a]
di Sessa, Andrea
Models of Computation
To appear in D. A. Norman and S. W. Draper, eds.,
User Centered System Design: New Perspectives on Human-Computer Interaction Lawrence Erlbaum, Hillsdale, New Jersey 1985

[di Sessa 85b]
di Sessa, Andrea
A Principled Design for an Integrated Computational Environment
Human-Computer Interaction 1:1 1985, pp. 1-47

[du Boulay 81]
Du Boulay, Benedict; O'Shea, Tim; Monk, John
The black box inside the glass box: presenting computer concepts to novices
International Journal of Man-Machine Studies 14, 1981, pp. 237-249

[Findlay 81]
Findlay, William and Watt, David A.
Pascal: an Introduction to Methodical Programming
Computer Science Press, Rockville, Maryland 1981

[Fulton 85]
Fulton, James L. IV
A Tour Through the BOCHSER Programming Environment

Bachelor's thesis, Massachusetts Institute of Technology, 1985

[Glinert 84]
Glinert, Ephraim P. and Tanimoto, Steven L.
Pict: An Interactive Graphical Programming Environment
IEEE Computer 17:11 1984

[Goldberg 83]
Goldberg, Adele
The Influence of an Object-Oriented Language on the Programming Environment
In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Halasz 83]
Halasz, Frank G. and Moran, Thomas P.
Mental Models and Problem Solving in Using a Calculator
In *Human Factors in Computing Systems: CHI '83 Proceedings*, 1983

[Hayes-Roth 83]
Hayes-Roth, Frederick; Waterman, Donald A.; and Lenat, Douglas B.
An Overview of Expert Systems
In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat (eds.),
Building Expert Systems, Addison-Wesley, Reading, Massachusetts, 1983

[Heering 85]
Heering, Jan and Klint, Paul
Towards Monolingual Programming Environments
ACM Transactions on Programming Languages and Systems, 7:2, 1985, pp. 183-213

[Kay 84]
Kay, Alan
Computer Software
Scientific American, September 1984

[Leap 84]
Leap, Thomas R.
Animations of Computers as Teaching Aids
ACM SIGCSE Bulletin, 16:1 1984

[Lieberman 84]
Lieberman, Henry
Seeing what your programs are doing
International Journal of Man-Machine Studies 21, 1984 pp.311-331

[MacScheme 85]
MacScheme Reference Manual
Semantic Microsystems, Sausalito, California
(in preparation)

[Mawby 84]
Mawby, Ronald; Clement, Catherine A.; Pea, Roy D.; and Hawkins, Jan

Structured Interviews on Children's Conceptions of Computers
Bank Street College of Education, Technical Report no. 19,
February 1984

[Mayer 75]
Mayer, Richard E.
Different proglem-solving competencies established in learning computer programming with and without meaningful models
Journal of Educational Psychology 67, 1975 pp. 725-734

[Mayer 76]
Mayer, Richard E.
Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order
Journal of Educational Psychology 68, 1976 pp. 143-150

[Mayer 79]
Mayer, Richard E.
A psychology of learning BASIC
Communications of the ACM 22:11, 1979 pp. 589-593

[Mayer 81]
Mayer, Richard E.
The Psychology of How Novices Learn Computer Programming
ACM Computing Surveys 13:1 1981

[Norman 83]
Norman, Donald A.
Some Observations on Mental Models
In Dedre Gentner and Albert L. Stevens, eds. *Mental Models*
Lawrence Erlbaum, Hillsdale, New Jersey 1983

[Parker 84]
Parker, J. R. and Becker, K.
A Microprogramming Simulator for Instructional Use
ACM SIGCSE Bulletin, 16:1 1984

[Reiss 84]
Reiss, Steven P.
Graphical Program Development with PECAN Program Development System
Brown University Technical Report CS-84-04

[Rovner 69]
Rovner, P.D. and Henderson, D. A. Jr.
On the Implementation of Ambit/G: A Graphical Programming Language
From *Proceedings of the Int. Joint Conference on Artificial Intelligence*
May 7-9, 1969 Washington, D.C. (Donald Walker, Lewis M. Norton, eds.)

[Sandewall 78]
Sandewall, Erik
Programming in an Interactive Environment: The Lisp Experience

In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Sandewall 81]
Sandewall, Erik; Stromberg, Claes; and Sorensen, Henrik
Software Architecture Based on Communicating Residential Environments
In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Scheme 84]
Scheme Manual, Seventh Edition
Massachusetts Institute of Technology Dept. of Electrical Engineering and Computer Science, September 1984

[Seitz 85]
Seitz, Charles L.
The Cosmic Cube
Communications of the ACM, 28:1, 1985 pp. 22-33

[Shapiro 74]
Shapiro, Stuart C. and Witmer, Douglas P.
Interactive Visual Simulators for Beginning Programming Students
ACM SIGCSE Bulletin, 6:1 11-14

[Smith 75]
Smith, David Canfield
PYGMALION: A Creative Programming Environment
Stanford Artificial Intelligence Laboratory Memo AIM-260, June 1975
Computer Science Department Report No. STAN-CS-75-499

[Smith 83]
Smith, David Canfield; Irby, Charles; Kimball, Ralph; Verplank, Bill; and Harslem, Eric
Designing the Star User Interface
In P. Degano and E. Sandewall (eds.),
Integrated Interactive Computing Systems
North-Holland, 1983

[Stallman 81]
Stallman, Richard M.
EMACS Manual for TWENEX Users
AI Memo 555, Massachusetts Institute of Technology, October 1981

[Steele 78]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay
The Revised Report on SCHEME: a Dialect of LISP
AI Memo No. 452, Massachusetts Institute of Technology, January 1978

[Strassmann 84]
Strassmann, Steven
Learning Lisp: The Barriers to Novice Programmers at MIT

Bachelor's thesis, M.I.T. May, 1984

[Sutherland 66]
Sutherland, William Robert
The On-Line Graphical Specification of Computer Programs
Ph.D. thesis, Massachusetts Institute of Technology, 1966

[Teitelbaum 81]
Teitelbaum, Tim and Reps, Thomas
The Cornell Program Synthesizer: A Syntax-Directed
Programming Environment
In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Teitelman 81]
Teitelman, Warren and Masinter, Larry
The Interlisp Programming Environment
In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Tesler 81]
Tesler, Larry
The Smalltalk Environment
Byte, August 1981

[Turbak 86]
Turbak, Franklyn
GRASP: A Visible Abstract Machine for a Procedural Programming Language
Master's thesis, Massachusetts Institute of Technology
(in preparation)

[Weir 85]
Weir, Sylvia
Cultivating Minds: a Logo Casebook
Harper and Row, 1985
(in preparation)

[Wilander 80]
Wilander, Jerker
An Interactive Programming System for Pascal
In D. R. Barstow, H. Shrobe, and E. Sandewall (eds.),
Interactive Programming Environments, McGraw-Hill, 1984

[Winston 77]
Winston, Patrick Henry
Artificial Intelligence
Addison-Wesley, Reading, Massachusetts 1979

[Young 83]
Young, Richard M.
Surrogates and Mappings: Two Kinds of Conceptual Models
for Interactive Devices

In Dedre Gentner and Albert L. Stevens, eds. *Mental Models*
Lawrence Erlbaum, Hillsdale, New Jersey 1983

Biographical Note

Michael Eisenberg is a recipient of an AT&T Bell Laboratories Ph.D. Scholarship.