Palo Alto Research Center

06-80 80-10

Requirements for an Experimental Programming Environment

by L. Peter Deutsch and Edward A. Taft

XEROX

Requirements for an Experimental Programming Environment

edited by L. Peter Deutsch and Edward A. Taft

CSL-80-10 June 1980

Abstract: We define *experimental programming* to mean the production of moderate-size software systems that are usable by moderate numbers of people in order to test ideas about such systems. An *experimental programming environment* enables a small number of programmers to construct such experimental systems efficiently and cheaply—an important goal in view of the rising cost of software.

In this report we present a catalog of programming environment capabilities and an evaluation of their cost, value, and relative priority. Following this we discuss these capabilities in the context of three existing programming environments: Lisp, Mesa, and Smalltalk. We consider the importance of specific capabilities in environments that already have them and the possibility of including them in environments that do not.

CR Categories: 4.20, 4.33, 4.34, 4.4

Key words and phrases: programming environment, experimental programming

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Preface

Computer software costs rise steadily, as we expand our ambitions to include ever more complex systems using ever cheaper hardware. The software that we can produce, and the rate at which we can produce it, are too often limiting factors in our research within the Xerox Palo Alto Research Center's Computer Science Laboratory (CSL). We believe that it is increasingly desirable, feasible, and economic to use computers to directly assist the process of experimental programming.

This report was produced two years ago as the report of an ad hoc "Programming Environment Working Group" that I chaired. It provided much of the initial impetus for Cedar, a major project now underway in CSL. Cedar is developing an advanced programming environment for the Mesa language as the basis for most of our programming during the next several years. We plan to report in due course both on various novel aspects of the design of Cedar and on our experiences in constructing and using it.

Meanwhile, interest in the important properties of programming environments has been growing. For example, the U. S. Department of Defense recently published "Requirements for Ada Programming Support Environments." [Stoneman, 1980] Our situation has much in common with that of other groups needing programming environments; hence we believe that others may be interested in the requirements that we originally set for Cedar.

This report was originally edited by Peter Deutsch for consideration within CSL. Ed Taft has edited it slightly to make it more comprehensible outside its original context, but has *not* attempted to either update or generalize it. Considering its age, and the speed with which it was prepared, I believe that it stands up very well. However, the reader should bear in mind that it was addressed to the problems of CSL as we saw them in mid-1978.

J. J. Horning
June 1980

1. Introduction

Charter and history

The purpose of this report is to collect and set down our experience and intentions in the area of programming environments. This material was prepared by a working group consisting of the following members of the PARC Computer Science Laboratory:

L. Peter Deutsch
James J. Horning
Butler W. Lampson
James H. Morris
Edwin H. Satterthwaite (Xerox SDD)
Warren Teitelman

with the occasional participation of Alan Perlis (Yale University).

We quickly decided that the right way to proceed was to address what we felt was the real question requiring resolution, namely to produce a catalog of programming environment capabilities which included justified value, cost, and priority evaluations.

The working group was given a one-month deadline and held eight two-hour meetings. Needless to say, there were many areas we had to treat superficially because of the time constraint, and some areas in which we realized we simply could not reach agreement. We also realize that we have undoubtedly overlooked some significant issues and viewpoints. However, we expected much more in the way of intractable disagreement than we actually experienced. This suggests that we were successful at avoiding religious debates, and instead we concentrated on the technical issues.

How should we compare programming environments?

Before considering particular features that we feel contribute to "good" programming environments, it is important to consider how we can tell that one programming environment is "better" than another.

Any evaluation must be in some context. There is no reason to believe that some one programming environment could be optimal for all kinds of programming in all places at all times. In our discussions, we have focussed our attention on the foreseeable needs within CSL in the next few years, with particular attention to *experimental programming.* We have taken experimental programming to mean the production of moderate-sized systems that are usable by moderate numbers of people in order to test ideas about such systems. We believe that it will be important to conduct future experiments more quickly and at lower cost than is possible at present.

It is difficult to quantitatively compare programming environments, even in fixed contexts. A large number of qualitative comparisons are possible, but the more convincing ones all seem to fall into two categories, both based on the premise that the purpose of a programming environment is, above all, to facilitate programming.

First, a good programming environment will reduce the *cost* of solving a problem by software. The cost will include the time of programmers and others in design, coding, testing, debugging, system integration, documentation, etc., as well as of any mechanical support (computer time, etc.) Since our human costs continue to be dominant, one of the most convincing arguments in favor of a particular programming environment feature is that it speeds up some time-consuming task or reduces the need for that task (e.g., it might either speed up debugging or reduce the amount of debugging needed). Bottleneck removal is the implicit argument in much that follows.

Second, a good programming environment will also improve the *quality* of solutions to problems. Measures of quality include the time and space efficiency of the programs, as well as their usability, reliability, maintainability, and generality. Arguments relevant to this category tend to be of the form "this feature reduces the severity of a known source of problems," or "this feature makes it easier to improve an important aspect of programs." Thus a feature might be good because it reduces the frequency of crashes in programs or because it makes it convenient to optimize programs' performance.

(These two categories could be reduced to one by noting that there is a tradeoff between cost and quality. Thus by fixing cost, we could compare only quality; by fixing quality, we could compare only cost. This seems to complicate, rather than simplify, a qualitative evaluation of features, so we will not seek to unify these two kinds of argument.)

In the discussion that follows, we have attempted to relate our catalog of "important features" to these more general concepts of what makes a "good environment" whenever the connection is not obvious. In some cases, we have not been entirely successful, because our experience *tells* us that something is essential, but we haven't been able to analyze that experience to find out *why*. In all cases, our arguments are more intuitive than logically rigorous. Strengthening these arguments would be an interesting research topic.

We have been largely guided by experience with three current programming environments available to the PARC community: Interlisp, Mesa, and Smalltalk [Teitelman, 1978; Mitchell *et al.*, 1979; Ingalls, 1978]. Both what we know about the strengths and what we know about the limitations of these environments have been taken into consideration. It is of course dangerous to generalize too boldly from the intuitions and preferences of users: it is virtually impossible to be certain that the useful features have been clearly distinguished from those that are merely addictive.

2. Catalog of programming environment capabilities

We have divided the capabilities of a programming environment into four categories. *Virtual machine / programming language* refers to those capabilities that are primitive concepts in the programming language or in the virtual machine on which the programming language runs. *Tools* refers to capabilities for operating *on* programs. *Packages* refers to readily available programs that implement particular clearly defined higher-level concepts. *Other* includes documentation and non-technical considerations.

It is important to note that the division between tools and packages is a somewhat arbitrary one: in a good environment, there is little distinction between the two, in that all the capability provided by the tools is available to the programmer in the form of packages, and all the capability of the language (including the packages, of course) is available to the human user to extend the functions of the tools.

Additionally, it should be understood that we are discussing a programming environment for a computationally rich environment: many of these capabilities are feasible only when each programmer has substantial computing power available to him at all times. More specifically, we expect our computing facilities to be dominated by high-performance personal computers such as the Dorado [Lampson & Pier, 1980], which is a successor to the now inadequate Alto computer [Thacker, *et al.*, 1979]. Our orientation toward single-user machines in a distributed environment requires us to consider a number of "operating system" capabilities that might be taken for granted in a time-sharing environment.

2.1. Summary

Here we enumerate the programming environment capabilities we have considered and place each in one of the four categories. In subsequent sections we shall discuss each capability in varying amounts of detail.

Virtual machine / programming language

(L1) Large virtual address space ($\geq$ 24 bits)
(L2) Direct addressing for files
  (L2a) Segmenting
  (L2b) An enormous virtual address space ($\geq$ 48 bits)
(L3) Well-integrated access to large, robust data bases
(L4) Memory management—object/page swapping
(L5) Object management—garbage collection, reference counting

(L6) Some support for interrupts

(L7) Adequate exceptional condition handling

(L8) User access to the machine's capability for packed data

(L9) Program-manipulable representation of programs

(L10) Run-time availability of all information derivable from source program (e.g., names, types, scopes)

(L11) Statically checked type system

(L12) Self-typing data (*a la* Lisp and Smalltalk), run-time type system

(L13) Encapsulation/protection mechanisms (scopes, classes, import/export rules)

(L14) Abstraction mechanisms; explicit notion of "interface"

(L15) Non-hierarchical control (coroutines, backtracking)

(L16) Adequate run-time efficiency

(L17) Inter-language communication

(L18) Uniform screen management

(L19) Inheritance/defaulting (e.g., Smalltalk subclassing; difficulty depends a lot on how much it has to do)

(L20) Ability to extend language (e.g., operator overloading)

(L21) Ability to create fully integrated local sublanguages

(L22) User access to the machine's capability for multi-precision arithmetic

(L23) Good facilities for processes, monitors, interrupts

(L24) Simple, unambiguous syntax (including infix notation)

(L25) Control over importation of names

(L26) User packages as "first-class citizens"

(L27) Closures

(L28) Full-scale inter-language communication

(L29) User microprogramming

(L30) Clean data and control trapping mechanisms

(L31) "Good" exceptional condition handling

Tools

(T1) Fast turnaround for minor program changes (less than 5 seconds)

(T2) Compiler/interpreter available with low overhead at run time

(T3) Cross-reference/annotation capability

(T4) Prettyprinter

(T5) Consistent compilation

(T6) Version control

(T7) Librarian, program-oriented filing system (including Browser)
(T8) Source-language debugger
(T9) Dynamic measurement facilities
(T10) Checkpoint, establishing a protected environment
(T11) History and undoing
(T12) Editor integrated with language system
(T13) More optimizing compiler if user willing to bind more tightly—with full compatibility
(T14) Aids for incremental development (stubs, outstanding task list)
(T15) Regression testing system
(T16) Random testing aids
(T17) (high capability) Masterscope
(T18) Access to on-line documentation (Helpsys)
(T19) Static analyzers: verifier, performance predictor

Packages

(P1) Text objects and images
(P2) Line objects and images
(P3) Scanned (bitmap) objects and images
(P4) Formatted document files
(P5) More elaborate screen management
(P6) Remote file storage
(P7) Small data base manager
(P8) Message transmission system
(P9) Remote procedure call
(P10) Event logging
(P11) Background processing
(P12) Generalized cache
(P13) Document editing
(P14) Forms
(P15) Menus and other standard user interfaces
(P16) History lists
(P17) User access to full bandwidth of disk
(P18) (English) dictionary service
(P19) Teleconferencing
(P20) Audio

(P21) User access to full bandwidth of networks

Other

(X1) Adequate reference documentation

(X2) "Efficient" interface for experts

(X3) Uniformity in command interface

(X4) "Self-teaching" interface for beginners

(X5) Good introductory documentation

2.2. Virtual machine / programming language

(L1) *Large virtual address space ($\geq$ 24 bits)*

(L2) *Direct addressing for files*

- (L2a) *Segmenting*
- (L2b) *An enormous virtual address space ($\geq$ 48 bits)*

(L3) *Well-integrated access to large, robust data bases*

The issue here is that things should scale smoothly as programs grow to encompass more functions or larger data bases. As matters stand now, the *time* required tends to grow in predictable ways, but when the *space* (addressing or memory) requirements grow beyond a certain point, radical redesign of the program is usually required. A secondary issue is the ability to combine programs without running into the same kind of hard constraint on the space taken up by the code.

An address space of, say, 2^{24} items would be adequate to hold all the code actually being used, even in a large system, but not adequate for all of its data (e.g., the American Heritage dictionary); whereas an address space of, say, 2^{48} items *would* be adequate for all the code and data of a very large, even multi-machine system. We tend to favor the former, more conservative definition, since we do not understand how to provide an efficient, robust implementation of the latter (more on the robustness question below). However, we feel that it is essential that a transition across the 2^{24}-object boundary not require the kind of wholesale reorganization of programs that such a transition requires in current language systems.

System facilities for accessing large, external data bases are required for several reasons:

> Many questions of organization and efficient implementation can be solved once, rather than over and over again by applications.
>
> The system itself needs data base facilities for tools like the program librarian.
>
> Access to externally stored data objects needs to be smooth for the debugger and other system facilities, not just the application programs.

Programs normally refer to internal objects with individual references, but to external data bases with both individual references and mass queries. There are three basic techniques for speeding up references to external data:

Caches (good for individual references);

Using sequentiality properties (good for searching);

Inversion (good for searching).

All of these techniques should be available in package form.

Integrity. A programming environment in which the file system is viewed as an extension of the address space must be extremely robust—much more so than any programming system we now have. Our current practice is to make the "truth" for a data base be a text file, and not to expend a lot of effort on armoring the binary version against all imaginable errors. Notable exceptions are the basic file facilities; we believe that a system that provides robustness facilities usable at higher levels would have a lot of advantages.

Long-term integrity seems to require some kind of *history* or redundancy information. To protect against "cosmic rays," it is sufficient to record this information in a form that only the implementing program understands. To provide Undo capability, the history must be in a form that makes sense to clients.

Another kind of integrity has to do with not losing information. The following kinds of ideas are important in this connection:

A computed object (e.g., a Mesa configuration) should keep track of how it got made, in enough detail to make it again. The description of the putting-together process should be program-manipulable.

Files should be self-identifying in a way that is not altered by their transfer between storage media or locations.

It should not be possible to destroy all traces of a file containing input (as opposed to purely computed) information. The space requirements can be kept under control by storing descriptions of files (such as changes from other files) rather than all the bits.

Even the dangling reference problem and various garbage collection approaches can be viewed as integrity questions to some extent.

(L4) *Memory management—object/page swapping*

(L5) *Object management—garbage collection, reference counting*

These facilities are important for essentially the same reasons as (L1) through (L3), namely, to free programmers from excessive concern for the size and location of their code and data, for both explicitly named and dynamically constructed objects.

(L6) *Some support for interrupts*

The ability to interrupt program execution is essential, at least to gain control of runaway programs, to allow interaction with long computations, and to provide local multiprocessing; "good" facilities are less important. We have no strong religious feelings about the form of a process mechanism.

(L7) *Adequate exceptional condition handling*

An integrated mechanism for handling exceptional conditions is required for clean debugging and for the construction of robust programs; it helps clarify program structure by separating *normal* and *exceptional* algorithms. Certain mechanisms are required in this area: some way to unwind the stack, some way of catching an unwind, and program control of error processing. What should be provided beyond this is not agreed—the controversies raging around the Mesa "signal" facilities are a reflection of our poor understanding of the problem.

(L8) *User access to the machine's capability for packed data*

The ability to pack data is essential to obtain acceptable storage efficiency for large data structures. However, in a system without static (pre-runtime) type checking, the desire to access packed structures efficiently (without checking or indirection through descriptors) conflicts with the desire to prevent corruption of the storage management system.

There was a long discussion of why it is hard to add packed data to. Dorado Lisp, which centered around this protection question in the following guise: how carefully should the system prevent the user from smashing its underlying data structures. There wasn't much agreement on this point, but it does seem to have considerable practical importance, since a highly restrictive attitude makes it difficult to code low-level parts of the system in itself.

(L9) *Program-manipulable representation of programs*

(L10) *Run-time availability of all information derivable from source program (e.g., names, types, scopes)*

These two issues are closely related: underlying them is our desire to make it easy to extend the set of tools, to make communication between sublanguages easy, and to break down as many as possible of the artificial distinctions between programs and their compilers, on one hand, and data and their programs, on the other.

A long discussion led to the conclusion that all this information is currently available in Mesa, modulo the question of how stable the compiler's internal representation of the program as a tree should be expected to be. Straightforward methods (like the ones used in CLisp) will allow compiled code to be attached to source in a user-created data structure, although it is certainly easier to do this in Lisp, where interpretation is simple. Not explored was the usefulness of an interpreter for a subset of the language, such as currently exists in the Mesa debugger.

(L11) *Statically checked type system*

(L12) *Self-typing data (*a la *Lisp and Smalltalk), run-time type system*

The primary value of type systems is descriptive and structural—specifying the intended properties of one's data, and providing a mechanism for ensuring consistency between the suppliers and clients of an interface. Enabling a compiler to generate better code is secondary. There are at least two dimensions of variability in type systems:

—whether type information is bound early, as in Mesa, or late, as in Lisp and Smalltalk. There is a need to provide the programmer a selection from a spectrum of alternatives—Mesa provides variant records, which are a limited form of run-time binding, and Lisp has DeclTran, which provides some compile-time binding.

—whether types form a strict partition of the data values, a coercion hierarchy, or some even richer structure. We believe that a richer structure is desirable. It might include provision for generic operators, for type-parameterized programs or *schemes* [Mitchell & Wegbreit, 1977], for a more general notion of type as simply a partial specification of the behavior of an object (perhaps like the Alphard "needs" list [Shaw, *et al.*, 1977]), and for automatic pointwise extension of operators over collections.

It appears feasible to deduce much of the type information for a program automatically, starting from the assumption that each variable only takes on values of one type [Milner, 1978; Cousot & Cousot, 1977]. This would alleviate some of the nuisance of having to write type declarations. Name conventions (e.g., capitalization or standard prefixes), if interpreted by the compiler, would also eliminate most of the need for separate specification of type information.

This is an area ripe for research. We believe that, at a minimum, both the early-bound Mesa type system and the late-bound Lisp/Smalltalk type system must be supported (as alternatives) by an EPE.

(L13) *Encapsulation/protection mechanisms (scopes, classes, import/export rules)*

(L14) *Abstraction mechanisms; explicit notion of "interface"*

Abstraction mechanisms are important because they make explicit the logical dependencies of one part of a program on another, while concealing implementation choices irrelevant to the communication between such parts. Thus, these mechanisms enable the ability to factor the development, debugging, testing, documentation, understanding, and maintenance of programs into manageable pieces, while leaving individual programmers the appropriate freedom to design those pieces.

The ability to specify interfaces in the abstract, and to conceal their implementation, is important, but a difficult research area. It is possible to derive this information about *implicit* interfaces after the fact using tools like Masterscope. The code produced by an optimizing compiler need not reflect source-level modularization, if tighter binding improves efficiency and the user is willing to pay the price of more compilations and possibly decreased debugging information.

We believe one facility in this area is absolutely essential: it must be possible for a programmer to control which names get exported from a package. In addition, it is important for the system to conceal the distinction between user-defined packages and system-provided primitives (types, operations, etc.) at least as well as Mesa does.

(L15) *Non-hierarchical control (coroutines, backtracking)*

Coroutines and generators are essential: they provide a natural way to write transducers (programs that consume a data stream and produce another), which in turn are often the best way to modularize a data transformation algorithm. Lisp has backtracking, a control discipline sometimes used to explore alternatives in goal-directed searching, but there is considerable disagreement over the mechanism used to provide it, and its uses can probably be covered by more restricted coroutine and Undo mechanisms. Closures are a method of providing a wide variety of interesting control and binding environments, but we are not sure whether they can be implemented efficiently enough, or are structured enough, to replace the more specialized coroutine constructs.

(L16) *Adequate run-time efficiency*

The ultimate efficiency criterion is whether a system can meet its external specifications and constraints (responsiveness to human users or program clients). Computational efficiency equivalent to Mesa as implemented on the Alto, coupled with the larger real memory and faster disk of the Dorado, is adequate for many projects; for Lisp and Smalltalk, we think we can attain this through internal re-engineering. For other, more computation-intensive systems, at least another factor of 5 is attainable (based on raw hardware speed).

(L17) *Inter-language communication*

We believe the best way to attain the benefits of uniform methods for accessing the machine's facilities at the lowest level is for all language systems to run under a single operating system that is carefully constructed not to impose unnecessary space or time penalties on its clients. For the Dorado, we believe that the Pilot operating system satisfies this criterion [Redell, *et al.*, 1979].

Inter-language communication at a higher level helps reduce duplication of effort and also provides one way of attaining extra efficiency for particular functions. Calling Mesa subroutines from Lisp or Smalltalk will probably be adequate. Communication through the network or file system interface requires very little work, but is too inefficient. The general problem seems difficult and less important.

(L18) *Uniform screen management*

Use of the display is pervasive in our interactive systems. Lack of uniformity leads to duplicated effort, often of low quality since an individual builder cannot easily draw on all past experience or devote the time to taking advantage of it. On the other hand, too much central control over screen management may frustrate the desire to experiment with new paradigms for interaction.

We believe that it is possible to "virtualize" the screen and the user input devices—that is, require people to write programs on the assumption that they will only have access to a subpart of the screen and to a slightly filtered stream of input events—in a way that will not markedly impede our ability to experiment, and that will have a large payoff in terms of the user's ability to construct a screen environment containing multiple windows on different programs. At a minimum, the user must have direct access to all the capabilities of RasterOp [Newman & Sproull, 1979], appropriately mapped or confined to work on a virtual screen.

(L19) *Inheritance/defaulting (e.g., Smalltalk subclassing)*

Languages that provide for programmer-controlled defaulting or inheritance reduce the time and chance for error in the programming process by making it unnecessary to write the same code or parameter values over and over again. The basic idea is that one should be able to write programs in a way that only specifies how they differ from some previously written program. Examples include default standard values for procedure arguments (how does this call differ from a "standard" call?), variant records (how does this particular record distinguish itself from the invariant part?), and the Smalltalk subclass concept (how does this class of objects differ from some more general class?)

We did not discuss this area beyond observing that it is somewhat related to the schemes question discussed under (L12), and that Smalltalk seems to derive considerable benefit from it.

(L20) *Ability to extend language (e.g., operator overloading)*

Languages may be extended by users in a variety of ways. Data structure extension, through user-defined data types and associated operations, makes it possible to write programs in terms of concept-oriented rather than implementation-oriented data objects. Syntax extension, through user definition of new language constructs, allows the user to define specialized notations that may be valuable for particular tasks: this is discussed in detail in (L21). Operator extension, the ability to define meanings for basic language constructs such as arithmetic or iteration when applied to user-defined objects, brings some of the benefits of notational extension with less drastic consequences in program readability.

Data structure extension is accepted as an important part of all modern programming languages. Syntax extension has fallen into disfavor because of a lot of bad experience; we believe this happened partly because the tools did not support extensions as well as they did the base language. Operator extension is already present in a number of languages such as Algol 68: we did not discuss its merits. It is interesting to note that Smalltalk is founded on the notions of data structure and operator extension, but that the syntax extension facilities present in the 1972 version of the language have been removed.

(L21) *Ability to create fully integrated local sublanguages*

All language systems actually have many small sublanguages for special purposes. For example, Mesa has not only the Mesa language, but the C/Mesa configuration language, the debugger command language, the programming language subset acceptable to the debugger's interpreter, and the very small languages used to control the compiler and binder from the Executive command line. "Fully integrated," as an ideal, means that control and data should be able to pass freely between sublanguages, and that the facilities (editor, prettyprinter, I/O system, etc.) applicable to the primary programming language should also be applicable to the other sublanguages.

Lisp is unique in that its dozen or so sublanguages all provide the ability to embed arbitrary Lisp computations in them. For example, in the middle of the editor one can compute (by calling an arbitrary program) data to be inserted, possibly as a function of the thing being edited, or even a sequence of edit commands to execute. The following features of Lisp seem to have made the creation of integrated sublanguages easier:

S-expressions are a simple, standard internal representation of parse trees for all sublanguages.

Lisp provides a standard method of sharing names and passing environments, namely a single, very simple name environment (atoms) that all sublanguages share. (This has both advantages and drawbacks: it leads to the "FLG" phenomenon, for example.)

Many internal system "hooks" are available to the sublanguage implementor. (This too has its drawbacks: it tends to make sublanguages more fragile.)

The standard system contains packages (prettyprinter, table-driven lexical scanner and parenthesis parser) which make I/O of program-like structures easy.

The sublanguages that don't take advantage of these characteristics, such as CLisp, QLisp, and KRL, find their lives a lot more difficult. If we were willing to limit the complexity of sublanguages to that of S-expressions, i.e., procedures and conditionals, then we could devise an S-expression-like representation for Mesa also. (Extending this to, say, arithmetic expressions not only involves a complex parser and prettyprinter, but in a statically typed language like Mesa also requires taking type declarations into account to decide what the operators in the source text actually mean. Admittedly, the S-expression-like approach doesn't allow embedding of a reasonable subset of Mesa itself in a sublanguage, and it doesn't address the point that some of the highest payoff comes from the integration of languages, like KRL, that *don't* look like S-expressions.)

We also noted that no matter what features the language and environment provided, proper proceduralization of facilities was essential: even Lisp has sublanguages—in particular, the compiler control sublanguage—that are implemented so as to interact with the user directly, and that therefore cannot be considered integrated.

To sharpen our ideas about what integration means, we considered a "straw man": a system in which all languages (editor, interpreter, etc.) shared a screen interface (window manager) but were otherwise entirely separate. This led us to the following observations:

> This model was proposed as one that imposed minimal requirements on the individual subsystems, at least if they only dealt with the screen as a sequential character I/O device. This may not be a proper assumption, however: despite numerous attempts, no such package has ever been developed for the Alto, and this may be because nobody has been able to develop a satisfactory model for the interface. We agreed that things become more complex as the subsystem's view of the display becomes more sophisticated (e.g., an editable document, a bitmap).
>
> While this model allowed for considerable communication (by human transfer of characters from output in one window to input in another), it had two serious deficiencies: it made no provisions for communication under program, rather than manual, control, and it required that all transmitted information be in text form. (Note that even transmission of file names requires integration in the sense of sharing a common file system.)

While we did not reach any clear conclusions, we were able to agree on the following:

> This is an important area for discussion, but it needs more time than we had available to us.
>
> Integration really means a common model for communication.
>
> The one catalog entry we have now should be broken down into multiple features of differing priorities.
>
> If all the other Priority A features in the catalog were provided, Mesa could readily support sublanguages of the complexity of S-expressions.
>
> Adequate proceduralization is necessary for a package implementing a sublanguage to be usable.
>
> A major source of difficulty is the sharing or passing of environment information between sublanguages. One part of this difficulty is simply addressing or naming objects to be shared. Another part is making sure that shared objects are interpreted the same way (in Mesa, making sure the communicants share the same declarations for the objects). One way around this is to have a limited number of globally agreed-upon structures, such as strings or S-expressions, and then encoding more specialized languages within them and interpreting them by convention or agreement (as Lisp does).

We also do not agree on the importance of fully integrated sublanguages: a number of Lisp users feel this item should have very high priority.

(L22) *User access to the machine's capability for multi-precision arithmetic*

Many language systems, though implemented on machines in which multi-precision arithmetic in assembly language is relatively straightforward, make it impossible for the user to get at these facilities (such as the carry from single-precision addition, or the double-by-single division that most machines provide in hardware). There is no excuse for this, especially on machines with relatively short (16-bit) words.

(L23) *Good facilities for processes, monitors, interrupts*

Synchronization between logically asynchronous processes is necessary in many programs, either for functional reasons (a system should be able to listen for incoming mail, send a file to be printed, and carry out interactive editing simultaneously) or for efficiency (overlapping computation with disk transfers). The language system, through an underlying operating system if necessary, should provide mechanisms that help the programmer write programs that involve multiple processes. There are a number of adequate, though conflicting, models of these mechanisms available, such as the Mesa and Pilot facilities [Lampson & Redell, 1980] and the Smalltalk scheduler. We did not discuss this area at all.

(L24) *Simple, unambiguous syntax (including infix notation)*

While CLisp leaves a very large gap between what can be precisely defined and what the user can reasonably do, and while Mesa syntax is regrettably complex (5 closely-spaced pages), we are willing to tolerate problems of this sort under the assumption that the primary users of the system will be experts, and that novices will be able to learn a useful subset easily.

(L25) *Control over importation of names*

As discussed under (L13) and (L14), import control seems less important than export control. The reason is that the implicit use of an interface can be deduced locally by noting what names are used; for export, the issue is global and not under control of the provider of the package without some explicit specification.

(L26) *User packages as "first-class citizens"*

We would like user-defined packages to function as "first-class citizens" on a par with built-in primitives (types, operations, etc.) However, the Euclid experience seems to indicate that this is very difficult [Popek, *et al.*, 1977]. In Smalltalk, this goal has been achieved except for some I/O issues, at the expense of not having static type structure in the language at all.

(L27) *Closures*

See (L15) for discussion.

(L28) *Full-scale inter-language communication*

(L29) *User microprogramming*

The ability to share data and pass control freely between programs written in any of the major languages depends on carefully coordinated use of certain basic resources such as the peripheral devices and the machine's address space. We think this is less urgent than the more restricted facilities of (L17): the primary motivation for changing languages in mid-program is increased efficiency, and dropping into Mesa from Lisp or Smalltalk provides this, although it does not address the secondary motivation, which is the ability to take advantage of work already done (or more conveniently done) in another language.

In cases requiring extreme speed, it may be necessary to give the programmer a way to write application-dependent microcode and link it to the system in a way that provides at least some checking that it will not destroy the rest of the system. We did not discuss this.

(L30) *Clean data and control trapping mechanisms*

There was a long and inconclusive discussion. Apparently we don't really know what this point is about. At one extreme of "data trapping" there is a simple address trap, as on early machines. At the other extreme is KRL. No one was willing to espouse either extreme. We noted that:

Programming with data abstractions can help with this problem, since there is then a better handle on when data is being changed.

Checking some predicate at periodic intervals (e.g., on every control transfer) may be quite adequate when data trapping is being used to catch "core smashing" types of bugs. Mesa already has this facility.

Many interesting cases can probably be handled by using the primitive trapping facilities of the mapping hardware.

(L31) *"Good" exceptional condition handling*

As discussed above, we really don't know what this would mean. Roy Levin's thesis, among other published papers, may be relevant [Levin, 1977].

2.3. Tools

(T1) *Fast turnaround for minor program changes (less than 5 seconds)*

Our concern with fast turnaround comes from the observation that programming should be *think bound*, not *compute bound*. There are several "knees" (points of substantial non-linearity) in one's perception of response delays. One such knee is in the vicinity of 3 to 5 seconds. We believe that it is essential to reduce the system time for minor program changes to below this point.

(T2) *Compiler/interpreter available with low overhead at run time*

The issues here are similar to those in (L9-10), namely, to reduce the mental and execution "gear-shifting" overhead caused by artificial divisions between compilation and execution environments. A sufficiently fast compiler is just as good as an interpreter for executing typed-in programs or programs constructed on-the-fly by other programs. However, it is essential that one be able to save some form of compiled code, for applications that embed procedures in data structures.

(T3) *Cross-reference/annotation capability*

(T4) *Prettyprinter*

These capabilities contribute substantially to the readability of programs, which in turn has a large effect on the ease of maintenance. A simple "batch" cross-reference facility is essential; more sophisticated facilities, such as those of Masterscope, are less urgent.

(T5) *Consistent compilation*

(T6) *Version control*

Consistent compilation is an efficiency issue: to get the right thing to happen without blindly recompiling and reloading everything. Version control is more fundamental. It has two major aspects: history and parameterization.

Under history, we want to be able to tell exactly how a particular system or component was constructed, and what it depends on (e.g., microcode version, interface definition or whatever). Furthermore, we want to be able to reconstruct a component automatically. This requires that every computation involved in its original construction must record *all* its inputs, and be prepared to repeat itself from this record. Since the inputs may be (references to) files, it is also necessary to have a naming scheme for files that is unique over the whole universe, and a guarantee that no file will ever be destroyed (unless the rule for reconstructing it is saved, together with all the required inputs).

Under parameterization, we want a systematic way of specifying the construction of a system that never existed before (e.g., it is for a new microcode version, or different implementations of the same interfaces are combined in a new way). We agreed that we don't aspire to solve this problem in the full generality required by IBM.

Replacing code in an existing system is in principle a special case of consistent compilation—the general question is when a complete but expensive procedure (recompilation, reloading, etc.) can be bypassed. We note that replacing code is in practice *not* just an efficiency issue, since getting the system back to the exact current state is not possible in general. The reason is that the current state depends on user program execution, and the user program cannot be counted on to follow the rules mentioned under history, which we impose on the system programs that make up the environment.

(T7) *Librarian, program-oriented filing system (including Browser)*

Coordinating access by multiple maintainers to a program made up of many packages requires some automation to avoid loss of consistency or even valuable information. For example, a *librarian* allows programmers to "check out" (in the sense of a library book) a module for modification; other tools can assist in re-integrating versions of modules that have been modified separately. A system richly endowed with packages also needs some automation to catalog them and their documentation in a way that actively aids users in finding what they need.

(T8) *Source-language debugger*

It is essential that the programmer be able to debug using the same language constructs and concepts used in writing the original program. This is facilitated by a minimum of distinctions between *compile-time* and *run-time* environments—see also (L10) and (T2).

(T9) *Dynamic measurement facilities*

These facilities are necessary to understand the behavior of complex programs under conditions of actual use. Smalltalk and Mesa have a "Spy," which works by sampling the program counter, and Lisp has Breakdown. The Smalltalk and Mesa facilities are relatively little used: the Mesa facilities are poorly documented and supported, and the nature of Smalltalk is such that there is often little meaningful tuning one can do. Available for Mesa programs are

a facility for counting frequency and time between any pair of breakpoints;

a facility for writing an event in a log, either by procedure calls, or by action to be taken at a breakpoint; and

a "transfer trap" mechanism that logs data at every control transfer, together with some standard ways of reducing this data to produce the same kind of information that a Spy produces.

We agree that something as good as Breakdown is good enough.

(T10) *Checkpoint, establishing a protected environment*

Checkpointing is needed to be able to protect oneself against unexpected machine or system failures. The weakness of the facilities in all three current systems was noted: file state is not saved, nor is there any check that it hasn't changed on a restart. *Protected environment* means the ability to install a new version of something in a system and still be able to revert to the old version very cheaply; cheap checkpoints can provide this. Cheap checkpoints can be done in a paged system with copy-on-write techniques.

(T11) *History and undoing*

History refers to the system keeping a typescript file, the ability to feed information from the typescript back to the system, and the ability to have a handle on the values returned as well. This is an attribute of the interactive interface: its value comes from the observation that the operations one performs often are similar to, or use the results of, the operations one has recently performed.

An *undoing* mechanism should cover both system-implemented actions such as edits, and a way for users to supply a procedure that will undo the effects of another specified procedure. This can be a very inexpensive alternative to checkpointing as a way to give the user the ability to experiment with alternatives without imposing the burden of manually saving and restoring the relevant state.

(T12) *Editor integrated with language system*

Editing is just one function of a language system, carried out using a particular sublanguage. (In fact, we believe it may be the appropriate model for the *major* sublanguage presented to the user at the terminal.) As such, it should be integrated with the rest of the language system in that

> the user doing editing can call on arbitrary programs to compute commands or data needed for the editing process, including the ability to pass selections from the thing being edited to the computation as arguments;
>
> any program can call on the editor as a package.

The latter seems very useful and relatively easy to achieve. We agree the former is also valuable, but there is disagreement over whether it is merely valuable or extremely important.

(T13) *More optimizing compiler if user willing to bind more tightly—with full compatibility*

A recurring source of difficulty in programming is the generality/efficiency tradeoff: when it is time to tune a system for better performance, it is traditionally necessary to make major logic changes as well. An alternative is to keep the logical structure of the program the same, but have a compiler that can do the necessary rearrangements as part of the compilation process: the programmer instructs it as to what kinds of flexibility or modularity should be sacrificed, and the critical choices to be made in the representation of data structures. The Lisp "block compiler" is one example of this idea; the Mesa inline procedure is another. We did not discuss this area except in passing.

(T14) *Aids for incremental development (stubs, outstanding task list)*

Top-down programming, or independent development of modules in a system, often benefits from the ability to replace as yet unavailable modules with *stubs* which have the same functional behavior but simpler (and presumably less efficient) implementation. This and other aids for keeping track of the status of parts of a large project have been used successfully in many system development efforts.

(T15) *Regression testing system*

(T16) *Random testing aids*

Both regression testing—keeping a record of standard tests and results with a program module, and automatically checking them after a change to the module—and testing with random data have proven to be worthwhile methods for checking programs too large or complex to verify or describe analytically.

(T17) *(high capability) Masterscope*

Any facility like Masterscope, which maintains an up-to-date data base of relations among parts of a program, must be integrated into the system at a fundamental level. Our discussion revealed that the fundamental aspect is the need for a *single* funnel for changes to the system (Mesa pretty much has this now, but Lisp does not). Relation to the file system was discussed, and it was agreed that manual use of the file system should be outlawed. A consequence is that the programming environment must do recovery at least as well as the file system does. Of course, having a reliable file system underneath makes this much easier. A variety of techniques are possible, which we did not explore in detail.

(T18) *Access to on-line documentation (Helpsys)*

Good on-line documentation, both for reference and for learning, can greatly reduce the need for time spent studying an enormous manual, can provide instant cross-linking of related subjects in a way that hardcopy cannot, and can use one's current context to implicitly locate relevant material—Lisp's Helpsys facility is unique in these respects. However, creating and maintaining such documentation is a tremendous amount of work, even if the process is partly automated.

(T19) *Static analyzers: verifier, performance predictor*

We agree that, especially for programs used by many people in low-tolerance environments, an ounce of prevention is worth a pound of cure: effort expended on eliminating bugs or bottlenecks beforehand can save a lot of time and trouble locating them afterward. Unfortunately, verification technology is still unable to accommodate programs of significant size written in languages of realistic complexity, and very little has been done on deriving performance information from the program text (in contrast to analytic models of systems at a gross level, of which there are many).

2.4. Packages

Beyond the virtual machine and programming language, which are forced on all users, and the tools, which should be applicable to all users, the quality of a programming environment is largely determined by the presence of packages that provide functional capabilities useful to many applications. We cannot stress too strongly that, from our experience, the only way to ensure the necessary high quality for such packages is to have a very small group (one to four people) with the final authority and responsibility for deciding which packages are to be incorporated in the system in a way that makes them readily available to all.

(P1) *Text objects and images*
(P2) *Line objects and images*
(P3) *Scanned (bitmap) objects and images*
(P4) *Formatted document files*

The manipulation of images is of primary concern to us in our experimental systems. We can divide these manipulations into two categories:

> Manipulation of abstract objects such as formatted documents, forms, line drawings, and continuous-tone images. The operations on these objects are defined by the semantics of the objects, not by their representation on a medium.
>
> Manipulation of the images of these objects on displays or printers. These operations must take the nature of the medium into account.

In the first category we might find editing operations on document files such as insert, replace, search; on drawings and pictures such as scale, rotate, reflect, clip, shade, connect points with a spline curve. In the second we find operations for mapping objects onto media in a variety of ways, some of which must be reversible (e.g., when a user makes a selection in the displayed image of a document, that selection really refers to the data in the document itself).

We believe that enough experience has been gained in these areas that it is possible to construct packages that will be useful in a wide range of programs, and that will markedly decrease the effort required to write programs that use them.

An interesting area that we have not discussed *per se* is the general notion of *annotation* of documents (or data structures): formatting information can be considered an annotation to the text, comments to a program, meta-descriptions in the KRL sense to a slot or another description. Pilot, for example, provides a notion of *subsequences* of a byte stream, which can easily be used to represent formatting information in a way that uninterested programs can ignore.

(P5) *More elaborate screen management*

In addition to the basic screen management capabilities mentioned under (L18), there are some additional facilities (scrollable windows, for example) that many programs will want to share. Again, we believe there will be a payoff from the presence of some carefully designed packages in the environment.

(P6) *Remote file storage*

The manual transfer of files between machines is a significant source of errors and wasted time. Such transfers are necessary either because of space problems or because one machine has a capability (such as a printer or high-performance display) not possessed by all.

(P7) *Small data base manager*

As a goal, we believe that the well-integrated access to large data bases mentioned under (L3) has a potentially enormous payoff, since many tools as well as experimental programs will benefit from it. However, if it turns out that we can't figure out how to provide this, then we will need a well-designed data base package for managing locally stored data.

(P8) *Message transmission system*

Message transmission is a useful paradigm for many kinds of inter-machine communication.

(P9) *Remote procedure call*

The ability to call a procedure on another machine as though it were on one's local machine is a different, less well understood communication paradigm.

(P10) *Event logging*

Event logging is a useful technique for redundancy and crash protection, for gathering statistics, and for reducing the cost of updating a data base in response to events affecting it.

(P11) *Background processing*

In an interactive system with enough real memory, both external communication (sending and receiving mail, printing) and computation (recompilation, Masterscope data base maintenance) can make effective use of time when the user is thinking.

(P12) *Generalized cache*

Many applications can benefit from a cache mechanism that provides local copies of more remote data, e.g., copies in memory of data from the disk, or copies on a local machine of files stored remotely. A package could keep track of which items had been used least recently, schedule rewriting of changed items, and deal with locks and timeouts.

(P13) *Document editing*

Underlying document editing and manipulation facilities have been re-implemented time after time, because insufficient thought was given to organizing them as a general-purpose package. There is no good technical reason for this.

(P14) *Forms*
(P15) *Menus and other standard user interfaces*

Packages that provide standard user interface tools such as forms, menus, selection, etc. are desirable both in the interests of uniformity and simply to save work.

(P16) *History lists*

Programs should be able to take advantage of the same mechanisms used by the system to provide the history and undoing capabilities discussed in (T11).

(P17) *User access to full bandwidth of disk*

Data base manipulations and code overlaying require brief bursts of high-bandwidth disk activity. The system should not prevent the programmer from using the disk's full bandwidth, and a package should make it easy.

(P18) *(English) dictionary service*

Office applications involving documents can benefit from easy access to an English dictionary (for spelling correction, hyphenation, and thesaurus applications, for example).

(P19) *Teleconferencing*

Inter-person communcation should play more of a role in our future experiments; we need a package to handle the mechanics of keeping several users' views of the screen, cursor, etc., consistent.

(P20) *Audio*

We have hardware support for capturing and playing back audio information, but hardly any software support. Something like the current audio message system ought to be a very small project.

(P21) *User access to full bandwidth of networks*

As in (P17), the system should not obstruct the programmer's access to the machine's full I/O bandwidth in experimental situations.

2.5. Other

(X1) *Adequate reference documentation*

Reference documentation must be complete and reasonably well organized and indexed. The Interlisp manual is a shining example of how well an entire environment can be documented. It also demonstrates that keeping this documentation up to date is a lot of work.

(X2) *"Efficient" interface for experts*

For experts, the desire for common operations to require a minimum of human effort often rightly takes precedence over the desire for the greatest possible uniformity or simplicity in the human interface. However, such interfaces are too often constructed without paying any attention to the few principles of interface design that we do know. We believe it is important to consider consciously the design of certain key command interfaces (editing, debugging, screen management).

(X3) *Uniformity in command interface*

In the process of using interactive programs, such as the tools listed in the previous major section, the user will inevitably accumulate perceptions, opinions, and models of the programs, and conjectures as to their workings. The net effect of these perceptions is referred to as the *user illusion*. The intent is to allow the user to see the programs only in relation to his own needs and purposes, and not have to concern himself with the internal workings of the programs. What is important about a standard user interface package is that the user be able to confidently predict the general manner of interaction with a program that uses the package, even though he hasn't experienced it yet; and that, by and large, the user will be right. This has been called the Law of Least Astonishment.

The concept of a consistent user interface also simplifies the design and coding of any program that interacts with the user. By adopting the conventions and making use of the facilities, it is a relatively simple matter to create useful interactive programs, because the programmer can concern himself with the algorithm rather than with creating his own user interface.

We believe that, in addition to consistency, another important goal to pursue in the design of the user interface might be called the Principle of Non-Preemption. Individual interactive programs should operate in a non-intrusive manner with respect to the user's activities. The system does not usurp the attention and prerogatives of the user. A program responds to the user's stimuli, but then quietly retains its context and logical state until the user elects to interact with the program again, not (for example) monopolizing the resources of the computer.

(X4) *"Self-teaching" interface for beginners*

(X5) *Good introductory documentation*

Since we are concerned with a programming environment primarily for CSL, and secondarily for the rest of the local research community, we feel that concern for novices should have low priority, since the rate at which new people join the community is low and most of them are already sophisticated.

3. Priority ranking and interrelation of capabilities

We arrived at a priority ranking for capabilities by giving each member of the working group 100 votes to be divided among the capabilities on the list. The vote total below is simply the sum of the votes of the 5 members who actually voted, identified by their initials: Deutsch, Horning, Lampson, Morris, and Satterthwaite. (As a check, we also ranked the capabilities according to the median rather than the total, and the results were essentially the same.) We found that a natural division into 5 priority groups emerged from the ranking. The reader should be aware that we were ranking capabilities on their utility if present in some reasonable form, not on the value of doing research into how to improve what we now know about providing them.

We were able to reach a consensus about how fundamental each capability was, in the sense of how difficult it would be to add that capability if it were not allowed for in the initial system design. In doing this we drew heavily on the experience gained from the three existing language systems. This consensus is expressed below according to the following code:

F—Fundamental, much harder if not allowed for originally

I—Intermediate, somewhat harder if not allowed for

A—Add-on, difficulty does not depend significantly on pre-planning (although it may be intrinsically hard anyway)

We identified an *enabling* relation between certain pairs of capabilities, in the sense that capability *x* is almost certainly required to provide capability *y*. So many things depend on item (L1) that we have omitted mention of this.

Finally, we estimated the difficulty of providing each capability in each of the three presently existing environments (Lisp, Mesa, and Smalltalk). Each capability has a 3-digit difficulty code referring to the effort required to provide the capability in Lisp, Mesa, and Smalltalk in that order. The difficulty codes have the following meanings:

0—available
1—easy
2—straightforward but takes time
3—hard
4—impossible (we found we didn't need to use this)

3.1. Priority ranking

In the tabulation of votes, "x" means zero votes, while "*" means that the person gave no votes to this item because he assumed it would be provided anyway. Votes of the form "3/4" mean 3 votes if we are primarily interested in E(PE), i.e., investigation of programming environments *per se*, but 4 votes if we are concentrating on (EP)E, i.e., production of experimental programs.

Priority A

			D	H	L	M	S	Total	
(L5)	F	030	6	x	10	20	4	40	Object management—garbage collection/reference counting
(L11)	F	203	3	6	9	7	5	30	Statically checked type system
(L4)	I	000	6	7	*	10	4	27	Memory management—object/page swapping
(L14)	F	302	2	4	5	10	4	25	Abstraction mechanisms; explicit notion of "interface"
(T1)	I	020	3	x	8	5	4	20	Fast turnaround for minor program changes (<5 sec)
(L16)	F	?0?	3/4	6	2	4	3	18/19	Adequate runtime efficiency
	enabled by	(L17)							Inter-language communication
(L1)	F	002	6	7	*	x	5	18	Large virtual address space (≥ 24 bits)

Priority B

			D	H	L	M	S	Total	
(L13)	F	302	3	5	2	4	3	17	Encapsulation/protection mechanisms (scopes, classes, import/export rules)
(L3)	F	333	3	4	8	x	x	15	Well-integrated access to large, robust data bases
(L12)	I	020	4	3	3	x	5	15	Self-typing data (*a la* Lisp and Smalltalk), run-time type system
(T5)	A	203	3	2	3	5	2	15	Consistent compilation
(T6)	I	323	2	x	8	3	2	15	Version control
(T8)	F	000	4	1	*	5	5	15	Source-language debugger
	enabled by	(L9)							Program-manipulable representation of programs
	enabled by	(L10)							Run-time availability of source program information
(P1)	A	121	2	3	3	4	3	15	Text objects and images
(L18)	I	021	2	4	2	4	3	15	Uniform screen management
(L8)	A	202	2/4	3	2	4	3	14/16	User access to the machine's capability for packed data
(L10)	F	111	3/2	3	4	x	4	14/13	Run-time availability of all information derivable from source program (e.g., names, types, scopes)

Priority C

			D	H	L	M	S	Total	
(L2)	A	102	x	6	x	x	4	10	Direct addressing for files (segmenting)
(L6)	I	000	2	5	x	2	1	10	Some support for interrupts
(T2)	I	021	3/2	1	1	2	2	9/8	Compiler/interpreter available with low overhead at run time
(X1)	I	012	2	4	x	x	3	9	Adequate reference documentation
(T7)	A	222	2	1	1	2	2	8	Librarian, program-oriented filing system (incl. Browser)
(L9)	I	012	3/1	3	x	x	2	8/6	Program-manipulable representation of programs

(T9)	I	011	2/3	1	x	2	2	7/8	Dynamic measurement facilities
(P3)	A	222	2	2	1	2	x	7	Scanned (bitmap) objects and images
(P4)	A	120	1	2	1	x	3	7	Formatted document files
(X2)	F	010?	2	1	3	x	1	7	"Efficient" interface for experts
(P2)	A	222	1	2	2	x	1	6	Line objects and images
(P6)	A	001	1	x	3	x	2	6	Remote file storage

Priority D

(L17)	I	202	1	4	x	x	x	5	Inter-language communication
(T11)	F	122	x	2	3	x	x	5	History and undoing
(L15)	F	000	2	x	x	x	2	4	Non-hierarchical control (coroutines, backtracking)
(L20)	F	220	x	x	4	x	x	4	Ability to extend language (e.g., operator overloading)
(L21)	I	333	2/1	x	x	x	2	4/3	Ability to create fully integrated local sublanguages
		enabled by (L9) Program-manipulable representation of programs							
		enabled by (L10) Run-time availability of source program information							
		enabled by (L12) Self-typing data							
(L27)	F	121	1	x	x	3	x	4	Closures
(T10)	I	000	1	x	2	x	1	4	Checkpoint, establishing a protected environment
(L19)	F	321	x	x	3	x	x	3	Inheritance/defaulting (e.g., Smalltalk subclassing; difficulty depends a lot on how much it has to do)
		enabled by (L13) Encapsulation/protection mechanisms							
(T3)	A	111	2	x	x	x	1	3	Cross-reference/annotation capability
(T4)	A	011	1	x	2	x	x	3	Prettyprinter
		enabled by (L9) Program-manipulable representation of programs							
(P15)	A	111	2	x	x	x	1	3	Menus and other standard user interfaces
(P13)	A	020	2	x	x	x	1	3	Document editing
(L7)	F	112	x	x	x	x	2	2	Adequate exceptional condition handling
(T12)	I	021	2	x	x	x	x	2	Editor integrated with language system
(P9)	I	222	1	x	x	x	1	2	Remote procedure call
(T13)	F	003	x	x	x	x	1	1	More optimizing compiler if user willing to bind more tightly—with full compatibility
(T18)	A	011	1	x	x	x	x	1	Access to on-line documentation (Helpsys)
(P8)	A	212	x	x	x	x	1	1	Message transmission system
(P10)	I	111	x	x	x	x	1	1	Event logging
(P12)	A	222	x	x	x	x	1	1	Generalized cache
(P14)	A	222	1	x	x	x	x	1	Forms

(X4) F 222 1 x x x x 1 Uniformity in command interface

Priority E (no votes at all)

VIRTUAL MACHINE / PROGRAMMING LANGUAGE

(L2b) ? 222 An enormous virtual address space ($\geq$ 48 bits)

(L22) A 100 User access to the machine's capability for multi-precision arithmetic

(L23) I 202 Good facilities for processes, monitors, interrupts

enabled by (L6) Some support for interrupts
enabled by (A15) Abstraction mechanisms

(L24) F 120 Simple, unambiguous syntax (including infix notation)

(L25) F 302 Control over importation of names

enabled by (A15) Abstraction mechanisms

(L26) F 331 User packages as "first-class citizens"

(L28) F 333 Full-scale inter-language communication

enabled by (L17) Inter-language communication

(L29) A 222 User microprogramming

(L30) ? 333 Clean data and control trapping mechanisms

enabled by (L6) Some support for interrupts
enabled by (L23) Good facilities for processes, monitors, interrupts
enabled by (L31) "Good" exceptional condition handling

(L31) ? 333 "Good" exceptional condition handling

enabled by (L7) Adequate exceptional condition handling
enabled by (L13) Encapsulation/protection mechanisms
enabled by (L14) Abstraction mechanisms
enabled by (L15) Non-hierarchical control structures

TOOLS

(T14) A 222 Aids for incremental development (stubs, outstanding task list)

enabled by (L9) Program-manipulable representation of programs

(T15) A 222? Regression testing system

enabled by (L9) Program-manipulable representation of programs

(T16) A 333? Random testing aids

(T17) F 022 (High capability) Masterscope

(T19) A 333 Static analyzers: verifier, performance predictor

enabled by (L9) Program-manipulable representation of programs

PACKAGES

(P5) A 022 More elaborate screen management

(P7)	A	221	Small data base manager
(P11)	F	323	Background processing
(P16)	A	011	History lists
(P17)	I	111	User access to full bandwidth of disk
(P18)	A	222	(English) dictionary service
(P19)	A	222	Teleconferencing
(P20)	A	222	Audio
(P21)	A	111	User access to full bandwidth of networks
OTHER			
(X4)	F	222	"Self-teaching" interface for beginners
(X5)	I	222	"Good" introductory documentation

3.2 Discussion of difficulty estimates

In this section we discuss the difficulty estimates for the highest-priority features. Nearly all these features fall into one of three difficulty patterns:

Considerably easier in Mesa than in either Lisp or Smalltalk (e.g., L11, L13, T5).

Considerably easier in either Lisp or Smalltalk than in Mesa (e.g., L5, T1, L12).

Comparable in difficulty in all three systems (e.g., L4, L3, T8).

This is not terribly surprising, given the underlying philosophical similarities between Lisp and Smalltalk.

Priority A

(L5) 030 *Object management—garbage collection, reference counting*

Reference counting, which relies almost entirely on *local* information, and garbage collection, which needs a *global* map of all potentially traceable data, present quite different problems for Mesa. For reference counting, it might be adequate to add an attribute to the Mesa type system so that reference counting or transaction queuing can occur when the program stores into a pointer that might point to an automatically managed object, and arrange for a user-supplied finalization procedure to be called when a count becomes zero. A further refinement might be automatic generation of the procedure by the compiler based on declarations. All this would be difficulty level 2.

Garbage collection in Mesa is more difficult, for two reasons:

> The type information needed to locate all pointers is not present in the actual data, but must be derived from the symbol table produced by the compiler.
>
> The presence of LOOPHOLEs, pointer arithmetic, overlaid variants, and relative pointers makes it impossible for a garbage collector to find or trace all pointers to structure to be saved.

The former problem is only one of efficiency: one way around it might be to arrange for the compiler to generate a tracing procedure for each structure that might participate in garbage collection. The latter, however, is a fundamental difficulty. We believe that the proper approach to overcoming it is to discover a subset of the Mesa language that does not use any of the LOOPHOLE-like features mentioned above, and find a way to draw a protection boundary around a program written in this subset so that it can have automatically managed data. This is a hard problem.

(L11) 203 *Statically checked type system*

The basic problem with static type checking in both Lisp and Smalltalk is that very little is known at compile time about the referents of names.

Lisp currently has a facility called DeclTran, which uses embedded declarations (both of types and of arbitrary predicates) to check the types of operands at runtime and to generate more efficient code within an individual function. This facility applies to both interpreted and compiled code, but is somewhat machine-dependent and does not exist for the Alto/Dorado Lisp system. Also, it has no ability to deal with static checking of interfaces between user-defined functions (as opposed to the interface from user-defined functions to built-in operations like arithmetic). If mechanisms, such as those of (L13) and (L14), were added to Lisp to control exporting and linkage of names, it is likely that DeclTran could be extended to inter-function type checking. We are inclined to think that this is not a good approach in the long run and that the type system should be more tightly integrated with the underlying language system, but it seems like a workable way to obtain the desired result.

Smalltalk has no facilities at all for static declaration of types. Furthermore, because every operation is decoded at execution time by the object that is its first operand (even elementary operations such as "+"), there is in principle no way of knowing what code will actually get executed in advance of execution time without some combination of type declarations and searching the entire system to discover what classes implement what operations. The situation is complicated by Smalltalk's subclass structure, which makes it very desirable that, for example, a method requiring an argument of type Number should accept values of type Integer (a subclass of Number). Smalltalk has a fairly simple compiler and name structure, but a lot of new machinery would have to be built, some of which would require careful thought about what "type" means in the Smalltalk environment.

(L4) 000 *Memory management—object/page swapping*

Lisp and Mesa already use a straightforward paging scheme, and Smalltalk an object-swapping scheme.

(L14) 302 *Abstraction mechanisms; explicit notion of "interface"*

As discussed under (L13) below, Lisp currently has only a very weak mechanism for explicitly dealing with interfaces between a package and its clients. In addition, Lisp has no mechanism for taking a group of functions and controlling how its imported names are to be linked: such a facility is necessary if interfaces are to be separated from their implementations.

Smalltalk is somewhat better off since its invocation mechanism does hide the implementation from the caller. A type system for Smalltalk, as discussed under (L11) above, would also go far towards providing an explicit notion of interface, since the current Smalltalk notion of "abstract class" can be viewed as either a type or an interface description.

(T1) 020 *Fast turnaround for minor program changes (less than 5 seconds)*

Lisp has an interpreter, and a fully integrated editor. (We have discussed replacing the interpreter with a high-speed, low-code-quality compiler, which already exists for the Alto.) Smalltalk has a compiler which we believe will run fast enough on the Dorado, and an integrated editor.

Mesa offers several medium-size obstacles to fast turnaround for changes:

> Since the present editor is not integrated or even properly packaged, considerable machine and human overhead is involved in getting in and out of it. We recommend replacing it by a packaged editor available in the programming environment.
>
> The compiler is not designed to compile anything smaller than an entire module. We recommend some work to modularize the compiler (it is already broken up quite well) and to make sure enough information is available for it to be able to compile single procedures.
>
> The current Mesa system does not provide for safe, incremental replacement of modules or procedures by new versions without losing execution state. This feature must be provided.

(L16) ?0? *Adequate runtime efficiency*

The Lisp instruction set is actually quite similar to that of Mesa: for example, it can use the Dorado instruction fetch unit. Sources of additional overhead include the fact that all data are doublewords, and considerable runtime type-checking is required. We believe that even the present Lisp system will perform adequately on the Dorado. Further significant gains in efficiency would be possible with some redesign that would use the Dorado processor stack, and with the extensions required for (L11), static type checking.

The Smalltalk instruction set, in principle, is also similar to that of Lisp and Mesa. In addition to doubleword data and runtime type-checking, Smalltalk also suffers somewhat from substantially less ability to compile things like arithmetic and structure access in-line, and the requirement for doing some kind of hash lookup on each invocation. We believe that Smalltalk could also be brought up to acceptable efficiency through a high-performance design.

(L1) 002 *Large virtual address space ($\geq$ 24 bits)*

Lisp and Mesa already provide 24-bit address spaces. The difficulty code for Smalltalk reflects Dan Ingalls' estimate that the current system could be modified (without changing its basic structure) in 3 to 6 months.

Priority B

(L13) 302 *Encapsulation/protection mechanisms—export control*

Lisp already has one mechanism for controlling the export of names from a group of functions: the block compiler. However, this mechanism has many disadvantages:

Using it requires giving up a good deal of debugging capability, since non-exported function and variable names are completely lost.

It is not compatible with either the interpreter or the normal compiler.

Its application is limited to a single level, e.g., a package cannot use this mechanism to have subfunctions local to an individual interior function or group of functions.

We believe that adding export control to Lisp would require introducing notions of nested or otherwise controlled lexical name scopes similar to those of Mesa; we would expect this to have ramifications throughout the system, and some impact on user programs that take advantage of the fact that there is a single space of names that encompasses both all of the system, their own programs, and their own data. (We note that in the past, the lack of export control has often been seen as an advantage, since it made it easy for a somewhat knowledgeable user to extend or modify the system as desired, but we believe this is outweighed by the problems it causes.)

Smalltalk has two different export control mechanisms. For variables, there is a dual lexical hierarchy based on lifetime (method variables, instance variables, and static class variables) and on specialization (subclassing). For operations, each class defines the names of operations it will accept. However, Smalltalk is deficient in several respects:

There is a loophole, fairly easy to use, that allows access from outside to any variable of any object.

A class cannot restrict purely internal operations to internal use—all operations within the class are accessible to all clients.

Subclasses have access to all operations and data of their superclass.

We believe that remedying these defects would require significant additions to Smalltalk but no changes in its basic structure.

(L3) 333 *Well-integrated access to large, robust data bases*

This is an area requiring considerable thought. No general-purpose programming language currently offers this capability as we intend it to be interpreted. We have no *a priori* reason to think it would be harder to implement in one language than in another—it will be a challenge in any of them.

(L12) 020 *Self-typing data (*à la *Lisp and Smalltalk), run-time type system*

Mesa, unlike Lisp and Smalltalk, segregates type information from data at runtime by relegating the former to symbol table structures* put out by the compiler, which are normally not part of the runtime environment. Extending the type system into the runtime environment in a reasonable way seems to require adding types as a type in the language, and arranging things so that the runtime representation of types can match up properly with the information in the symbol table. Some of the design for this already exists, since the debugger must manipulate types internally. It is worth noting that Mesa's facilities for variant records already address a small part of this problem.

(T5) 203 *Consistent compilation*

Mesa ensures consistent compilation by placing time stamps on source and object files, and by recording in each object file the complete list of time stamps for the files that produced it. For Lisp and Smalltalk, the basic problem is that identified in (L13) and (L14) above, namely, lack of mechanisms for identifying the dependencies between modules.

(T6) 323 *Version control*

Mesa contains most of the mechanisms for the history part of version control (knowing exactly how to reconstruct any object file). Some things are missing, such as a record of the compiler switches, and also the Binder does not provide this information in the files it constructs. For Lisp and Smalltalk, the problem is that objects do not carry any kind of identifying stamp, nor do compilation processes record the information they used. In the present Lisp system, the problem is almost insoluble, because compilation takes place in the context of whatever random collection of macros, flags, and other declarations happens to be lying around at the time of compilation. (The problem is not the large amount of state on which compilation depends, but the fact that there is no mechanism for recording the relevant parts of it with the compiled file.)

Aside from C/Mesa and the Binder, none of the three systems has any mechanism for describing how parts are put together to form a whole. C/Mesa fails to cover certain crucial aspects of initialization and parametrization—there is no way to pass parameters from a C/Mesa program to initialization code, for example, in lieu of supplying them directly when the program initializes itself. Lisp allows programs to load other programs as part of their initialization, and several large systems have more elaborate mechanisms of their own for doing this, but there is no system-wide machinery of this kind.

(T8) 000 *Source-language debugger*

All three systems have adequate source-language debuggers. All of them could profit from additional work, since there are ideas in each one that are not present in the others: for example, the REVERT feature in Lisp, the ability to point directly to the source text on the screen in Mesa, and the Smalltalk method for displaying local variables.

(P1) 121 *Text objects and images*

Lisp already has a package (TXDT) for manipulating documents with formatting on individual characters (font, bold/italic, etc.). However, TXDT has no provisions for paragraph-level formatting or tables or for conversion to or from standard document file formats. These defects are minor. Facilities for manipulating text images as images are provided within DLisp. Documentation for the formatting aspects of TXDT, and for all of DLisp, does not currently exist, although the latter is being worked on.

No comparable package exists in Mesa, although several application systems have had to provide some of these functions.

Smalltalk provides the notion of a formatted paragraph, and facilities for editing, displaying, and selecting within it, in the standard system. Smalltalk also provides conversion to and from document file formats. Smalltalk does not provide paragraph-level formatting or tables, and its facilities for handling text images in any but the most straightforward way are poor. Again, these defects are relatively minor.

(L18) 021 *Uniform screen management*

Lisp already possesses an elaborate and functionally rich screen management package, DLisp. As noted above, DLisp is still being documented. It does seem to satisfy our principal desire, namely that programs access the display only through an interface that makes sharing and allocation of the physical screen invisible to them; we are not certain whether it imposes too much structure on the user's view of the display, to the extent that some kinds of experiments might be difficult or impossible (no such difficulties have arisen in the few applications attempted thus far).

Several attempts have been made at creating a screen management package in Mesa; all have failed to gain acceptance, for a variety of reasons. We believe that it will be necessary to start from scratch, taking into account the experience with the failed models and with the more successful ones of Lisp and Smalltalk.

Smalltalk has the opposite problem: accessing the screen in an anarchic way has been too easy. Some moderately successful interfaces have emerged—one for windows, one for documents, and one under development for graphics—but there is still a substantial need for unification of what has been learned.

(L8) 202 *User access to the machine's capability for packed data*

Lisp and Smalltalk, unlike Mesa, take the view that all data accessible to the programmer are pointers. Intrinsically non-pointer data, such as string characters, are handled with various subterfuges. To obtain the benefits of packed data within these two systems would require extending the storage management system to be able to allocate and deallocate such objects, and extending the compiler to generate instructions to access them under safe conditions. The Lisp "record package" accomplishes some of this, but in a way that is poorly integrated with the rest of the storage management system and makes use of what we consider impermissible loopholes in the compiler. Smalltalk has no such facility at all, although the Smalltalk implementors have discussed it and believe they understand in principle how to do it without structual changes in the system.

(L10) 111 *Run-time availability of all information derivable from source program (e.g., names, types, scopes)*

All three systems make most of the source information available in some form at runtime. In Lisp, the facilities for retrieving source information are very good, but they do not handle declarations and variables with the same ease as functions. In Mesa, the relevant data structures (in the symbol tables produced by the compiler) are not currently documented for public use. In Smalltalk, the information required to correlate the dynamic environment with the names used in the program exists but not in convenient form. Mesa's facilities for correlating the program counter with a point in the source program are much better than either Lisp's or Smalltalk's.

4. Conclusions

We are somewhat surprised at how little disagreement remains in our rather mixed group. Our priority ranking of PE features is numerical evidence of this, and the minutes of our discussions provide further, qualitative support for the same point. There are, of course, differences about the relative importance of many features, but hardly any about the five-level priority assigned to each feature, or about how hard things are to do. We therefore urge that discussion should focus on the value of an EPE and how it is to be achieved, rather than on detailed argument about individual features.

As we said initially, the purpose of a PE is to improve the productivity of programmers or the quality of programs. We don't know how to quantify quality, but we did think about how much more productivity we might expect for sizable projects, as compared to the current state of affairs in either Lisp or Mesa. We guess that a factor of four is possible, about half from relaxing current space and time constraints by moving to the Dorado, and half from a PE which has our A, B, and C priority features and a respectable sprinkling of the others.

How will all this productivity be applied? We anticipate three major effects.

First, many more interesting things will be within the scope of a single person's efforts. Hence, the number of ambitious one-person projects can be expected to increase dramatically; not only are they much less work to do, but it is much easier to organize a one-person than a four-person project.

Second, much more elaborate things will be feasible, and hence will be attempted.

Third, the evolution of good packages which can be easily used without disastrous time, space, or naming conflicts will cause a qualitative change in the nature of system-building: the current Interlisp system gives us our few hints of what this change will be like.

References

[Bobrow & Winograd, 1977]
Daniel G. Bobrow and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, vol. 1 no. 1, 1977.

[Cousot & Cousot, 1977]
Patric Cousot and Radhia Cousot, "Static Determination of Dynamic Properties of Generalized Type Unions," *Sigplan Notices*, vol. 12 no. 3, March 1977.

[Geschke, *et al.*, 1977]
Charles M. Geschke, James H. Morris, Jr., and Edwin H. Satterthwaite, "Early experience with Mesa," *Communications of the ACM*, vol. 20 no. 8, pp. 540-553, August 1977.

[Ingalls, 1978]
D. H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, 1978.

[Kay, 1977]
Alan C. Kay, "Microelectronics and the Personal Computer," *Scientific American*, vol. 237 no. 3, pp. 231-244, March 1977.

[Lampson & Pier, 1980]
Butler W. Lampson and Kenneth A. Pier, "A Processor for a High-Performance Personal Computer," *Proceedings of the 7th International Symposium on Computer Architecture*, May 1980.

[Lampson & Redell, 1980]
Butler W. Lampson and David D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, vol. 23 no. 2, pp. 105-117, February 1980.

[Lauer & Satterthwaite, 1979]
Hugh C. Lauer and Edwin H. Satterthwaite, "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering*, 174-182, 1979.

[Levin, 1977]
Roy Levin, "Program Structures for Exceptional Condition Handling," Carnegie-Mellon University technical report, June 1977.

[Milner, 1978]
Robin Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, 348-375, 1978.

[Mitchell, *et al.*, 1979]
James G. Mitchell, William Maybury, and Richard Sweet, "Mesa Language Manual, version 5.0," Xerox PARC technical report CSL-79-3, April 1979.

[Mitchell & Wegbreit, 1978]
James G. Mitchell and Ben Wegbreit, "Schemes: a High Level Data Structuring Concept," *Current Trends in Programming Methodology* (Raymond T. Yeh, ed.), vol. 4, Prentice-Hall, 1978.

[Newman & Sproull, 1979]
William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, second edition, McGraw-Hill, 1979.

[Popek, *et al.*, 1977]
G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London, "Notes on the Design of Euclid," *Sigplan Notices*, vol. 12 no. 3, March 1977.

[Redell, *et al.*, 1979]
D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell, "Pilot: an Operating System for a Personal Computer," *Proceedings of the Seventh Symposium on Operating System Principles*, December 1979.

[Shaw, *et al.*, 1977]
Mary Shaw, William A. Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," *Communications of the ACM*, vol. 20 no. 8, August 1977.

[Stoneman, 1980]
"Stoneman: Requirements for Ada Programming Support Environments" (John N. Buxton, ed.), U. S. Department of Defense, February 1980.

[Teitelman, 1977]
Warren Teitelman, "A Display-oriented Programmer's Assistant," *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pp. 905-915, August 1977.

[Teitelman, 1978]
Warren Teitelman, *et al.*, "Interlisp Reference Manual," Bolt, Beranek & Newman and Xerox Palo Alto Research Center, 1978.

[Thacker, *et al.*, 1979]
C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, "Alto: a Personal Computer," *Computer Structures: Readings and Examples* (Siewiorek, Bell, and Newell, eds.), 1979.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304