Inter-Office Memorandum

To	Distribution	Date	December 13, 1978
From	Second EPE working group (names on next page)	Location	Palo Alto
Subject	Report from second Programming Environment Working Group	Organization	CSL

XEROX

Filed on: Ivy <Deutsch>pe2>
pe2.press - the report proper
memos.press - technical memos submitted to the working group (the appendix to the report)
minutes.press - the minutes of the working group's meetings

Distribution:
CSL
SSL: LRG, A. Henderson
SDD Palo Alto: Irby, Liddle, Lynch, Metcalfe, Satterthwaite, Dave Smith, Wallace, Weaver, White, Wick
ASD Palo Alto: Brotz, Elkind, Spitzen, Wegbreit

Non-CSL recipients may distribute further copies as needed

Table of Contents

Summary

The first CSL Programming Environment working group met intensively for a month last summer, and concluded that CSL should launch a project on the scale of the Dorado to develop a programming environment for the entire laboratory, starting from either Lisp *or* Mesa. A second working group has met weekly for nearly three months, exploring in more detail the probable consequences of the alternative starting points. This report presents those consequences in what we hope is sufficient detail for CSL to choose a starting point, and begin the actual construction of an Experimental Programming Environment.

Although the members of the working group are not in complete agreement about the choice of a starting point, there is almost no disagreement about technical issues or long-term goals. The facts and conclusions presented in this report are not in dispute; we have attempted to identify the (few) sources of unresolved disagreements.

As later sections of this report will make clear, the choice between Lisp and Mesa cannot be made solely on the basis of technical considerations. To within the uncertainty of estimation, both starting points present challenges of comparable difficulty, and would lead to systems of comparable utility, with comparable investments, over comparable time spans (two to three years). We are agreed that it is necessary and feasible for an EPE based on either system to support comfortably the programming styles currently associated with Lisp and Mesa. Some differences would remain in the character of the result; the value judgements placed on these differences generated most of the heat in our discussions. Social and political factors -- including the availability of people to fill key roles in the project, and our relations with the rest of the corporation -- are secondary (but possibly decisive) concerns.

Technical issues: Our further investigation of both the Lisp and Mesa alternatives led to two plausible development plans for producing a rudimentary environment on the Dorado (primarily for use by the EPE developers) in about six months, a widely-usable environment in a year and a half, and one superior to either of our present environments in about three years. The final system would provide all the facilities that present Lisp and Mesa users value highly: from Lisp, tools similar to the existing array of tools in the current Interlisp, and the ability to support integrated sublanguages and to delay bindings; from Mesa, the provisions for modularization with explicit interfaces, and the amenability to static checking. However, the system would probably retain some of the "flavor" of its starting point.

Organizational issues: Although the efforts required are about the same, a somewhat larger number of qualified people are available to work on a Mesa-based EPE. This report also discusses a possible structure for the project; after the language choice is made it should be possible to develop an internal operating plan quickly.

External affairs: The rest of Xerox has a fairly large and growing commitment to Mesa, and none to Lisp. Remaining largely compatible with the rest of the corporation has both advantages and disadvantages, but the advantages predominate. Either choice will reduce communication with important (but different) research communities in the outside world: Lisp favors the AI community, Mesa favors the programming language community.

Your working group respectfully asks CSL as a whole to:

endorse the immediate establishment of a large-scale EPE project;

based on the information contained in this report, and such other information as it considers relevant, choose the starting point for the project; and

agree to relegate further pro-and-con language discussions to the status of "printing discussions" for at least a year.

Charter and History

The purpose of this report is to set forth the discussions and conclusions of the second CSL Programming Environment working group. The most active members of this group (those who attended meetings regularly and produced memos or other substantial input for the group's deliberations) were:

Dan Bobrow
Peter Deutsch
Jim Horning
Butler Lampson
Roy Levin
Larry Masinter
Gene McDaniel
Jim Mitchell
Ed Satterthwaite (SDD)
Nori Suzuki
Dan Swinehart
Warren Teitelman

Many other members of CSL also attended some or most of the meetings and/or provided written material for the group.

We began with the conclusions of the first working group, reproduced here from their report:

> We think that CSL can support only one major PE, and that it can be evolved from either Lisp or Mesa; Smalltalk is farther from our goals, though certainly not out of sight. If a new PE is to be developed from Lisp or Mesa, then other work on the existing systems must be kept to a minimum during this development (unless we want half the lab's effort to be devoted to PEs), and they must be phased out as the new environment becomes viable. There doesn't seem to be any reason to attempt a multi-language PE, since an environment with all our ABC features would support any existing style of programming, while a two-language system would require more work, reduce synergy, and create artificial barriers. Our problem, then, is to decide whether we want to pay the price for a new programming environment:
>
> > agreement on what the starting point should be;
> >
> > manpower to build the ABC features which are missing;
> >
> > willingness of users to suffer substantial inconvenience in changing over.
>
> We are clearly talking about a project on the scale of the Dorado, and we should give it just as much careful thought as we did the Dorado project. Lab-wide discussion, and written feedback, seem like appropriate ways to carry out this process.

Our discussions focused on a number of areas in which we felt clearer understanding was needed before we could make a proposal to the lab as a whole about what to do in the PE area:

> What are the key technical issues which arise from each of the possible starting points?
>
> What differences in the ultimate system would necessarily follow from each choice of starting point?
>
> How much effort would be required to reach various levels of result from each starting point? What people would be available to do the work?
>
> What other non-technical and external considerations should play a significant role in our decision about what to do?

The group has been meeting once a week for approximately 3 months. We did not begin with a fixed idea of how long it would take us to reach satisfactory answers to the questions above. However, we feel that we now have sufficiently good answers to consider this phase of EPE activity finished: the next step is to make a final decision about the starting point, which will require discussion by the lab as a whole, and then get down to work on the EPE itself.

Introduction

Why does CSL need a new Experimental Programming Environment?

CSL's impressive productivity over the last few years can largely be attributed to its continuing investment in tools for itself. Dorado is a good recent example. It provides new opportunities for research in many areas by removing hardware capability (speed, memory, address space) as the major bottleneck. For the next few years, our ability to conduct computer system experiments will be largely limited by our programming capabilities.

CSL currently has two major programming environments, Interlisp on Maxc and Mesa/Bravo on the Alto. Each of these has major limitations deriving from its history and hardware base that make it less than ideal for at least some CSL applications. The arrival of Dorados provides an opportunity to remove many of these limitations and provide a *single* programming environment for the entire laboratory.

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; Bravo, Laurel, and KRL (the implementation part) were taken as typical of such experiments, but we believe that it will be important to conduct future experiments more quickly and at lower cost.

A programming environment is more than just a programming language. It is the entire system that is used by the programmer in the process of developing programs. As such, it will certainly contain such tools as text editors, debuggers, prettyprinters, etc. The underlying design principle is that the purpose of a programming environment is to facilitate programming:

> 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). Software bottleneck removal is the implicit justification for many features.
>
> 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.

We don't know how to quantify quality, but we did think about how much more productivity we might expect for sizable projects (e.g., Laurel, Bravo or the travel planner), 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 the high-priority features on which we have agreed.

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 Laurel-sized 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 that a four-person project.

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

Third, the evolution of good packages that 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.

Global technical issues

Consensus principles

In the course of our discussions, a few principles emerged which were not captured as "features" in the original report, or were so fundamental that we thought they needed to be singled out for special mention. These principles included:

> Automatic storage deallocation is an absolute necessity. It produces a qualitative improvement in the ease of programming and the reliability of the results. This need not preclude another class of variables with programmer-managed allocation, but the latter must not be able to destroy the data structures necessary for the former.
>
> Easy use of programs as data underlies many other facilities in the system. Implementing this seems to require having Lisp-style atoms, a run-time type system, and universal pointers (pointers that carry a type with them).
>
> Editing facilities closely coupled with the compiler and the executive interface to the system are essential, both to reduce the turnaround time for minor changes, and to allow easy construction of tools that interact with these facilities.
>
> Capabilities for precisely defining interfaces, and restricting the communication between modules to those interfaces, are essential to reliable and readable programming. This includes, as a special case, the ability to restrict the use of types and names to a local lexical context.
>
> The ability to perform static checking (including, but not necessarily limited to, type checking) over designated program regions is necessary for both security and efficiency reasons.
>
> The present Lisp, Mesa, and Smalltalk programming styles must *all* be supported in a satisfactory way. The same packages, and tools of equivalent power, must be available in all styles. In particular, the Lisp capabilities for embedded variant languages and for programming entirely without type declarations must be supported.
>
> Both large, multi-person and small, single-person styles must be supported well. Bringing a wider range of experiments within the scope of a single person’s effort is an important EPE goal.
>
> The EPE must support a wide range of binding times, including the Mesa and Smalltalk extremes, in a way that allows changes in binding time without structural changes in the program. Different choices of binding time by the programmer may lead to different turnaround times for apparently minor changes, and to different execution efficiencies, but the *functional* behavior of programs must not depend on such choices.

Smalltalk

Fairly early in our discussion we decided that either Lisp or Mesa would be preferable to Smalltalk as a candidate for the EPE base system -- sufficiently preferable that we did not need to consider Smalltalk further. This decision was based on the following observations:

> The present Smalltalk implementation is written in Alto assembly language and intrinsically cannot support more than 128K of real memory or 64K addressable objects. A major redesign and reimplementation of the lowest level of the system would be required to make it suitable as an EPE base. Neither Lisp nor Mesa has this problem.
>
> There is no evidence that large systems can be written in Smalltalk; the Smalltalk view of the world is enough different from that of either Lisp or Mesa that fundamental problems

may be revealed if we try to apply it to our larger projects. Large systems have been successfully written in both Lisp and Mesa.

Significant groups in CSL are familiar with either Lisp or Mesa; there is no CSL user community familiar with the Smalltalk view of the world.

The present direction of Smalltalk evolution is towards smaller machines (NoteTaker), while EPE should take significant advantage of the Dorado's capabilities.

However, there are a number of ideas in Smalltalk, not present in either Lisp or Mesa, which we believe are important to provide in the EPE:

The notion of class-structured behavior, and the ability of more specialized classes to inherit behavior from more general classes.

The user interface style.

The lack of structural distinction between user- and system-defined objects.

The working group spent essentially no time discussing these questions: it is up to the eventual project participants to make sure they receive proper attention.

(EP)E vs. E(PE)

At the beginning of our discussions, we assumed that the facilities required for E(PE), i.e. work on programming environments per se, were a subset of those required for (EP)E, i.e. construction of experimental systems in general. Consequently, we did not discuss the E(PE) parsing separately. However, it should be noted that the plan for a Lisp-based environment was strongly influenced by a desire to facilitate further work in language and system construction, since in the Lisp programming style, every medium- to large-scale project develops some language and system extensions; such considerations were much weaker in the Mesa-based plan. Several members of the working group with strong interests in programming languages and systems feel that neither proposed plan is likely to lead to a system that supports their interests for at least a couple of years.

Facilities

Our discussion of technical issues centered around the priority A, B, and C items from the first EPE report, reproduced below. The columns in the list have the following meaning:

Serial number - a serial numbering of the items, for later reference in this report. These are numbered in priority order, not in the L-P-T-X numbering system of the earlier report.

F/I/A code - the first working group's opinion of how important it was to provide for this item from the beginning in the system design.

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)

Amount of discussion - the amount of time we spent discussing the item in the current working group.

0 - didn't even mention it
1 - mentioned in passing
2 - significant discussion
3 - major issue

L,M,S difficulty codes - the first working group's conclusion about how difficult it would be to implement this item in Lisp, Mesa, or Smalltalk respectively.

0 - available
1 - easy
2 - straightforward but takes time
3 - hard

Serial number	F/I/A code	Amount of discussion	L,M,S difficulty codes	Name of item
Priority A				
(A1)	F	3	030	Object management -- garbage collection/reference counting
(A2)	F	3	203	Statically checked type system
(A3)	I	0	000	Memory management -- object/page swapping
(A4)	F	3	302	Abstraction mechanisms; explicit notion of "interface"
(A5)	I	3	020	Fast turnaround for minor program changes (<5 sec)
(A6)	F	3	?0?	Adequate runtime efficiency
(A7)	F	1	002	Large virtual address space (> __ 24 bits)
Priority B				
(B1)	F	3	302	Encapsulation/protection mechanisms (scopes, classes, import/export rules)

(B2)	F	1	333	Well-integrated access to large, robust data bases
(B3)	I	3	020	Self-typing data (a la Lisp and Smalltalk), run-time type system
(B4)	A	2	203	Consistent compilation
(B5)	I	1	323	Version control
(B6)	F	2	000	Source-language debugger
(B7)	A	1	121	Text objects and images
(B8)	I	1	021	Uniform screen management
(B9)	A	3	202	User access to the machine's capability for packed data (see A6)
(B10)	F	3	111	Run-time availability of all information derivable from source program (e.g. names, types, scopes)
(B11)	F	2	222	CSL control over the system's future

Priority C

(C1)	A	0	102	Direct addressing for files (segmenting)
(C2)	I	0	000	Some support for interrupts
(C3)	I	3	021	Compiler/interpreter available with low overhead at run time
(C4)	I	0	012	Adequate reference documentation
(C5)	A	1	222	Librarian, program-oriented filing system (incl. Browser)
(C6)	I	3	012	Program-manipulable representation of programs
(C7)	I	1	011	Dynamic measurement facilities
(C8)	A	0	222	Scanned (bitmap) objects and images
(C9)	A	0	120	Press files
(C10)	F	0	010?	"Efficient" interface for experts
(C11)	A	0	222	Line objects and images
(C12)	A	1	001	Remote file storage

The two following subsections of this report cover all the topics identified as "major issues" in the above list, as appropriate for the individual language.

Mesa facilities needed in Lisp

The following facilities from the ABC list are present in Mesa but not adequately available in Lisp:

(A2) Statically checked type system
(A4) Abstraction mechanisms, explicit notion of "interface"
(A6) Adequate run-time efficiency
(B1) Encapsulation/protection mechanisms
(B4) Consistent compilation
(B9) (in some contexts) User ability to pack data

In the Lisp tradition, we propose to add these facilities by defining new objects to represent types, name scopes, etc. and associated functions for manipulating them. These objects are accessible to the programmer at run time; however, greater efficiency can be obtained by compiling programs with respect to a particular state of these objects, and paying the price of recompilation (or, more typically, reversion to interpreted or less fully compiled status) if a change is made. A similar comment applies to static checking -- a change in a type or scope requires re-checking the affected parts of the program.

The remainder of this section describes the principal features of a Lisp dialect we have been calling XLisp, which we propose as the EPE base. XLisp includes the current Interlisp facilities as a special case, but it is based primarily on SCHEME, a lexically-scoped Lisp dialect developed at MIT, with additions motivated by the Mesa type and scope system. We intend XLisp not only as a response to the EPE challenge, but as a conceptual framework for thinking about scope and type issues which has some room for future exploration of capabilities not present in either Lisp or Mesa.

To summarize how we propose to provide each of the six facilities listed above:

Static type checking -- by making types be objects, and (as needed) checking that all users of a declared variable refer to identical (EQ) type objects.

Explicit interfaces -- by defining an object called a dictionary that generalizes the notions of parameter list, record, and interface, and by requiring that such an object be associated with every defined or imported function.

Greater run-time efficiency, packed data -- by making the internal representation of a quantity be one of its attributes in the type system, and extending the instruction set to allow more efficient execution when more tightly bound representations are used.

Encapsulation/protection -- by associating protection information with entries in dictionaries.

Consistent compilation -- by extending the notion of type identity and compatibility into the permanent filing system, through an object called a permanent pointer which somewhat resembles Mesa's time stamps.

Dictionaries

A new kind of object called a dictionary is central to the interpretation of names (and, hence, programs) in XLisp. A dictionary consists of one or more name tables, each of which applies to a different syntactic context ("function" or "variable") and specifies the interpretation of a set of names in that context. Each entry in a name table consists of a name (atom) and an associated property list. Dictionaries implement function argument and result lists, record types, and interfaces: the latter uses are discussed more fully under Types below.

Dictionaries are the building blocks for scopes. Conceptually, a scope is an area of a program within which the interpretation of names does not change; implementationally, it is an object which specifies how to determine the meanings of names. (We will freely confuse the concept and

implementation in the discussion that follows.) A scope is made up of components, each of which gives the meaning of some names, and a rule for deciding how to resolve conflicts and what to do with undefined names. Each component may be a scope itself, or it may be a dictionary, which actually associates names and meanings.

Every name in a program must eventually resolve to a meaning of one of three basic kinds: manifest, in which the actual value is found in a dictionary (for example, local and system functions, record names, and what are now thought of as compile-time constants); dynamic, in which the value is found in a run-time data structure (for example, most variables and non-local functions); or unbound, in which only declarative information (if that) is available. In principle, every dynamic name eventually reduces through macro-expansion to a value obtained by applying elementary operations (at roughly the instruction set level) to constants and to pointers which form part of the virtual machine's state, such as the current stack frame pointer. These elementary operations are available as functions in the system, since they are needed for the interpreter and compiler, but their use is deliberately rendered inconvenient since it can lead to violations of scope rules on which not only compiler optimizations but the correct working of programs may depend. For ordinary use, we provide, instead of these primitives, a set of higher-level constructs for use in name tables: these constructs lend themselves better to static analysis and permit enforceable protection boundaries. These constructs are the following:

(MANIFEST value) -- the actual value is present in the name table. The value may be any Lisp object (data structure, function object, etc.). The value is treated as a constant: changing it may require recompiling any function that uses it. If it is a non-scalar value, it is only given out in read-only form.

(SCRATCH value) -- the value is present in the name table, but is not read-only. One can think of this as an OWN structure in the Algol sense.

(MACRO function-object) -- the name table contains a function which is applied to the form (application or atom), and the result is taken in place of the original form. It is vital that macros produce consistent results, i.e. only depend on the form being expanded and the scope in which it is being expanded. We assert that is sufficient to analyze the macro function and all functions it calls to make sure that they refer only to local and manifest variables, and to pass the macro function its arguments in read-only form; then any operations with side effects, if applied to objects not created during this call of the macro function, will cause an error.

(FLUID fluid-cell) -- this is the traditional Lisp binding method. Note that function and variable names are different for fluid binding purposes, and that the name itself is not a global atom but an object local to the dictionary, so that the binder and the user of a fluid name must share the dictionary.

(UNBOUND)

(LOCAL location-in-frame) -- this is the SCHEME lexically bound variable. It differs from a fluid variable in that closures which use local variables free retain a shared pointer to the binding in effect at the time the closure is *created*, while a free use of a fluid variable refers to the binding at the time the closure is *applied*. LOCAL names are also used for fields in records: in this case, location-in-frame defines the location of the field within the record instance.

(RELATIVE name scope base) -- this is used to refer to objects found in other scopes. Examples include argument or outer PROG variables referenced from within a PROG, instance variables and abstract operations in a record, and individual entries (usually manifest or fluid) imported from other scopes. A relative entry contains three parts: a name, a pointer to the scope in which to interpret the name, and, if the interpretation is local, an expression (interpreted in the original scope) which gives the base pointer. This base pointer becomes the default base pointer for the purpose of the interpretation, so

cascading of relative entries will work properly. The Mesa OPEN construct amounts to making appropriate RELATIVE entries in the current scope for all the names in the object being opened.

Types

Framework

Types are objects, just like integers or lists. There are three kinds of types: interface or I-types, which describe how objects behave in general; descriptive or D-types, which specify predicates that hold true of objects in particular situations; and representation or R-types, which describe the bit patterns used to represent objects inside the machine. Conceptually every object carries its I-type with it at execution time, but not its R-type (D-types are associated with variables, not with objects). The system provides a default representation for every I-type. The phrases "X is of type T" and "X satisfies type T" are interchangeable -- I- and D-types are predicates.

I-types are built up from elementary types like INTEGER and ATOM by (possibly recursive) applications of the following operators (the syntax is for expository purposes only):

SATISFIES[p:FUNCTION[[t:Type],[Boolean]]] is a type for objects x of type t such that p[x] is true.

ANYOF[t1 ... tn:Type] is a type for objects which can be of any of the types t1 ... tn.

ALLOF[t1 ... tn:Type] is a type for objects which must be of types t1 ... tn simultaneously.

READONLY[t:Type] is a type for objects which are of type t shorn of any operations which produce externally visible side-effects (more on side-effects below).

UNIQUE[t:Type] is a type which is the same as t, except that objects of this type are distinguishable from objects of type t and of any other type created by UNIQUE[t]. This operation is sometimes called "painting".

FUNCTION[argT,resultT:Type] is a type for functions which take arguments of type argT and return results of type resultT. ArgT and resultT must be structure types.

VARID[t:Type] is a type for variable id's of type t. Variable id's are used to link together binders and users of variables, leaving the identity of the variable unbound. (Same idea as the Mesa SIGNAL linking mechanism.)

Structuring -- read on.

The other kind of definable I-type is a structure type. It has named components, each of which may have type declarations. There are two kinds of components: fields, which obey certain rules set forth below that make them act like variables, and operations.

Every field component has two associated operations called fetching and replacing that field. These operations obey the following rules:

(1) Fetching a field has no externally visible side-effects (although it may have internal side-effects like caching or statistic-taking -- obviously "visible" can't be defined rigorously). In particular, it does not affect the contents of any field.

(2) Replacing a field has no externally visible side-effects on any operation other than fetching that same field.

(3) Fetching a field always returns what was last replaced into that field.

(4) No operation other than replacing a field affects what fetching that field will return.

The default R-type for fields is a storage cell: if the programmer specifies a different R-type, its adherence to the above rules is up to him.

Fields in structure types may have names, or an ordering, or both. The ordering is for argument and constructor lists -- it has nothing to do with storage format.

Arrays are a structure type in which the fields, instead of having names, are indexed by a set which may be of any type. If the type is an integer range, the default representation is a traditional array; if not, the default is a hash array.

An interface type, in the Mesa sense, is a structure type with fields which are of various function types. A Mesa module is a group of functions which all agree to use the same instance(s) of the interface types they import. Making an instance of a module consists of making an instance of the interface type it implements, filling in the fields with closures of the exported functions in which the imported interface pointers get bound.

Declarations

A D-type declaration just establishes an invariant for a variable -- checked whenever the variable is stored into, can be assumed whenever the variable is used. D-types can also be used as predicates, and wrapped around expressions. R-type declarations specify the representation of a quantity and hence are only meaningful for variables, as opposed to arbitrary expressions. ("Variables" include record fields, of course.)

Conformity and coercions

Type checking depends only on the I- and D-types of the objects. The conformity relation is straightforward to compute, given the type composition primitives. The compiler does all it can, and then produces code to do the remainder of the check.

Coercions are meaningful at both the I-type and R-type levels. I-type coercions up or down the type lattice are mostly either no-ops or type checks, and can be supplied automatically. Other I-type coercions require user-supplied functions. R-type conversions require user-supplied functions to convert to and from the default R-type, the familiar Lisp typed pointer.

Protection

We implement compile time checking by providing information in a dictionary that specifies the external accessibility on a name-by-name basis. The actual access rights are determined by a function of the originator, the path used to arrive at the dictionary entry, and the information in the entry itself.

Mesa provides four kinds of protection that are not checkable at compile time:

1) Variables, unless otherwise declared, are not accessible at run time by programs outside the lexical scope where they are defined.

2) A client and an implementor must use identical (EQ) versions of an interface.

3) An implementor can define types whose contents are (partially) private, but which the client can use to declare variables.

4) The types of the arguments passed in a control transfer always agree, regardless of how the environment receiving control was obtained (e.g. by binding, through a PORT, through a procedure variable).

In Mesa, all of these are based on intra-module checking by the compiler, and inter-module enforcement based on (2). This, in turn, is accomplished by creating a unique name for an interface when it is compiled, and copying that unique name into both the client and the implementor when they are compiled.

In XLisp, the privacy information in dictionaries and scopes provides (1), since the only way to access a non-local variable is by going from a frame or other object to the dictionary that describes it, and the operations to access objects in general are implemented within dictionaries and check the privacy information.

Our type checker provides (2) by simply using EQ -- XLisp always keeps with any compiled object a pointer to the dictionaries accessed during compilation, which include all imported and exported interfaces. The permanent pointer mechanism described under Miscellaneous below guarantees that the EQ relation is correct even when externally filed data are involved.

(4) is also a type checking question. If the user declares procedure and environment variables appropriately (which is automatic if an interface dictionary is involved), then the check can be made when the interface is bound rather than at the time of the control transfer.

The heart of our protection system is (3). Something like (3) is required to be able to have virtual objects, and objects accessible only through interface procedures (operations). In XLisp one can declare a record type as having sealable instances. What this means is that the record has a piece of code associated with it, and a "sealed" flag. If the record is in unsealed state, anyone may access it freely. If the record is in sealed state, only the associated code may access it. A record must be sealed to be used as the local environment of a function: the operation of calling a function logically breaks down into the steps of creating an instance of its local frame type, filling the parameters into the instance, sealing the frame, and then running the function. Similarly, records which define Smalltalk-like instances are sealed in a way that accepts incoming messages and then executes the implementing function. Such functions can access the sealed record by having a link in their scope structure that says they implement an operation on that record. (This is no less safe than the IMPLEMENTING construct in Mesa.)

A sealable record and its underlying unsealable record are two different types. Coercing the former to the latter (going into an implementor) requires that the sealed flag be set. Coercing the latter to the former is always possible -- the result is always sealed. The implementor can specify an arbitrary error check to be applied to a sealable record in order for the seal to be set: the check and the sealing take place atomically. This can be accomplished by copying the record, doing the check, and then atomically copying the checked result back. Alternatively, the record can be accessed through a level of indirection, and the indirect pointer can be disabled at the start of the check.

Miscellaneous

Filing

XLisp programs contain inter-program references (to dictionaries, in particular) that must be representable on external files, so that a program knows it will have the same environment when run subsequently, rather than some environment made up of objects that happen to have the same strings as their names.

We define a new object called a permanent name (PN) which consists of two parts: a file name (a string), which is to be considered only a hint, and a permanent id (PID, an integer of roughly 64 bits), which is the actual name of the object, and which is guaranteed different for every permanent object. Potentially, many different kinds of objects have permanent names -- function definitions, dictionaries, maybe any object whatever. To find the object addressed by a permanent name, it is necessary to hunt around in the file system, presumably starting at the file named in the PN, until an object with the right PID is found. Each file contains a directory that maps PIDs to text representations of objects in the file.

A PN is only needed for an object if a cross-file reference (or multiple reference within a single file) is made to it. PNs and PN directories are large, so we think it would be satisfactory to assign a PN only when (1) an object was written on a file, and "potentially referenced via a PN" was a property of its type; (2) a program asked for a PN for an object.

PNs are a basic mechanism for constructing permanent pointers. To make them useful, one needs such things as the notion of versions of an object, directories mapping PNs of original versions to PNs of current versions, etc. The crucial question is one of overwriting: how can one change an object without changing its PN? Fortunately most (all?) file storage systems provide some kind of atomic operation which is guaranteed to execute either completely or not at all, and an atomic overwriting operation can be constructed using this.

Lisp facilities needed in Mesa

We distinguish two kinds of facilities presently available in Lisp and absent from Mesa which are needed in an EPE Mesa:

> *Foundations*: basic, low-level facilities needed to match the functionality of Lisp: atoms, variable syntax analysis, dynamic variable binding, a safe language subset, and automatic storage deallocation.
>
> *Features*: equivalents for the Lisp Masterscope, file package, programmer's assistant, Helpsys and similar features, in a more general setting, together with improvements to the debugger and editor, facilities for specifying the construction of complex systems, and other "environmental" capabilities.

We propose to provide a few of these facilities, which we lump under *Preliminaries*, by initially constructing an interim integrated Mesa environment (IME) which provides quick edit-compile-run turnaround for the existing Mesa language and system, as well as rudimentary access to the Dorado's large virtual memory. This system serves as a base for later developments.

The following facilities from the ABC list are present in Lisp but not currently available in Mesa:

> (A1) Automatic storage deallocation
> (A5) Fast turnaround for minor program changes
> (B3) Run-time type system, self-typing data
> (B10) Run-time availability of all source program information
> (C3) Compiler/interpreter available with low overhead at run time
> (C6) Program-manipulable representation of programs

Two other Foundation facilities emerged from our discussions as being extremely important to the Lisp way of doing business:

> Dynamic variable binding
> Programmable syntax analysis (for both programs and data)

Of all these facilities, we will deal with (A5) under Preliminaries, and we think (C3) is covered by our proposals for (A5) and (B10); the rest are Foundations. We will deal with Features separately.

Preliminaries

This section describes the steps required to obtain an interim integrated Mesa programming environment (IME). The intended user community for this environment is a small corps of experts who will use it to construct a more lasting integrated Mesa world.

Phase I

Phase I represents the bulk of the effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The facilities provided in Phase I are:

> Microcode and system extensions to support a 24-bit address space and virtual memory, with some crude protection between major entities (compiler, debugger, editor, user programs). Some of this work has already been done for Extended Memory Alto/Mesa.
>
> A window-based display interface for interaction with the debugger, editor, and other tools. Such an interface already exists in the Mesa debugger and in the Mesa Tools Environment, and one is planned for Chameleon, a package for market probes being developed by AOS.

A packaged text editor of about the functionality of the Laurel editor. The Tools group already have such an editor. Chameleon will have a similar editor. Alternatively, it wouldn't be much work to write our own.

Changes to the compiler and debugger to allow the symbol tables and program being debugged to reside in the address space for fast access.

Additional commands in the debugger to allow it to assume the role of the Alto Executive.

Phase II

Phase II will provide two extensions to the Phase I system:

Replacing an existing module by a new version without losing execution state. This involves changing the compiler to remember the decisions it made the last time it compiled the module, and is only possible if the structure of the module (global variables and entry procedures) has not changed. The loader must detect and invalidate extant frames that point into the global frame being replaced.

A dependency analyzer that determines what modules might (logically) require recompilation as a result of a change. SDD already has a version of this program called the Include checker.

Foundations

1) Programs as data

The ability to treat programs as data requires three major facilities within the Mesa environment: (a) an S-expression-like representation of source programs, (b) a means of using atoms (in the LISP sense) uniformly throughout the environment, and (c) a universal pointer mechanism.

S-expression representation. We propose to define a representation other than parse trees, so that the representation of parse trees can change as the compiler/language evolve.

Atoms. The Mesa compiler currently builds symbol tables that include a mapping between source identifiers and internal unique IDs (atoms) for an individual module. The Lisp atom capability requires a global map, into which the individual map would be merged when a compilation unit "enters the environment". This is very similar to the merge now done by the compiler for included modules.

Universal Pointers. We will need pointers that carry the type of their referent with them. This in turn requires a scheme for universal type IDs, which do not currently exist in Mesa. We also intend to add a facility for coercing a universal pointer to a pointer of more specific type; the coercion must be accompanied by a run-time check.

2) Programmable syntax analysis

LISP provides three opportunities for the programmer to affect the parsing/interpretation of input text: (a) at scanning time, (b) at compile time, and (c) at function call time. (a) may be accomplished by providing scanner escapes to user-written code, which will replace some appropriate substring of the current parsing context (i.e., the as-yet unclosed lists) with an S-expression. It appears straightforward to provide a facility comparable in power to LISP's readtables -- a more elegant mechanism might be desirable. (b) may be accomplished by more-or-less conventional macro expansion along the lines of the Mesa mechanisms for in-line procedures.

(c) is deemed undesirable and will not be provided.

Lisp allows free mixing of code and (structured) data, and parses both in essentially the same way; programs frequently read in both at run time. Consequently, the scanner and parser must be packaged for run time use.

3) Dynamic variable binding

Mesa, at present, only supports static binding of identifiers. To be able to bind a name to a variable in an enclosing *dynamic* context, we need to know what variables are to be found in each frame. Instead of the LISP default, we propose that variables to which dynamic binding may occur be declared as such at compile time. The compiler can then produce a table of entries of the form

atom -> <frame,type>

for each compilation unit. Dynamic binding would then occur much in the same way the signal-to-catch phrase binding occurs (but more efficiently).

4) Safe language

By a safe language we mean here one in which a pointer value of a given type POINTER TO T is guaranteed to point to a region of storage containing a value of type T.

There are three ways in which a pointer variable may acquire an improper value: lack of initialization, incorrect computation (e.g. by pointer arithmetic), or premature deallocation of the object pointed to. Array and string descriptors require the same treatment as pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer.

Preventing uninitialized pointers is straightforward, by generating code to initialize all pointers in procedure frames when they are allocated, and in dynamic variables of record and array types when they are allocated. Mesa 5 will have language constructs for allocating dynamic variables, so it is known where to do the initialization.

Preventing incorrect computation requires excluding all the dangerous constructs from the language

a) Loopholes into a pointer type or a pointer-containing type (PC for short)

b) Computed or overlaid variants in which the variant part is PC

c) Pointer arithmetic

d) Use of UNSPECIFIED (this restriction is probably stronger than necessary)

e) What is missing?

Preventing premature deallocation has two aspects: dynamic (explicitly allocated) variables and stack variables. For the latter, we propose to eliminate the ability to obtain an explicit pointer to a local variable, and as partial compensation introduce **var** parameters (call by reference). For dynamic variables some support from an automatic deallocation mechanism seems essential. We see three possibilities, discussed in the appendix: the one that comes closest to Lisp is fully automatic deallocation based on an incremental garbage collector.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Misuse of a relative pointer can only damage data in the region it refers to; if none of the storage in this region contains any external pointers, then no global damage can occur. Embedded external pointers are still possible if they are treated as array indices, i.e. go through fingers. The advantage is that the programmer can forgo the safety precautions for such relative pointers without risking the integrity of the entire system. Of course a bounds check is still required (analogous to an array bounds check), but in many cases of interest it can be provided by storing the pointers in n-bit fields and making the size of the region 2^n.

5) Automatic deallocation

Our proposal is to have automatically deallocated (AD) types, and to use the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an AD type, but the techniques are well-known and straightforward.

Features

This section attempts to characterize the "programmer's assistant" facilities of a Mesa-based EPE, in terms of a more or less direct replacement for existing Interlisp functions. However, we must bear in mind that we have the opportunity to redesign the user's interface to the system, taking the high bandwidth display into account. The goal will be to provide a consistent, integrated default interface for program development and for control of user applications that do not have exotic interactive requirements. The right facilities for such an environment will look substantially different from previous ones, as environments like the Smalltalk browser, DLisp, and the Mesa Tools are beginning to show us. These issues deserve a great deal of additional design work once the project is underway.

Interpreter

We advocate extending the compiler to provide the equivalent of incremental compilation for single statements in specified contexts. An evaluator within the debugger would generate the appropriate linkages to get the statements executed. Sufficient information would be saved to provide the History List features (below.) An appropriate default global context could be made available for the execution of statements corresponding to the Lisp "top level".

Break and Advise

Only modest extensions to the current conditional breakpoint code, using the incremental compiler invented just above, are necessary to obtain a break/advise facility the equal of Interlisp's. The standard breakpoint handler could take care of providing the linkage to the system-produced procedures containing the advice.

History Lists and their Use

A history list that records top-level user actions in external (text) form, along with an indication of the contexts in which they were executed, are sufficient to implement the *Redo* command and the function-composition capabilities that it enables. To provide *Undo*, however, requires internal information as well. Once automatically allocated lists, programs as S-expressions, atoms and data with attached types, and the other *Fundamentals* are available, implementation of both commands should be straightforward. Probably both would be done, as in Lisp, by retaining S-expression representations of the original statements and their reversing functions.

In Mesa even simple programs consist of several independent global contexts. Virtually every user action will have to be interpreted with respect to a specific environment. History list entries will have to include the corresponding environmental information, and the commands will have to know when the corresponding environments no longer exist, or will have to arrange, as Interlisp can, to retain the context frames as long as they are referenced (requires garbage collector to operate on frames.) Contexts running as concurrent processes could also cause trouble if more than one of the processes becomes involved in user history activity. The history provisions would want to be more fully integrated with the complex context structures than are the Interlisp models.

Implementating the equivalent of the TESTMODE(T) facility in Interlisp -- saving the old contents for *every* store other than into local variables -- could cause both time and space problems, because Mesa is much more an assignment-based language than Lisp.

Given Mesa's complex syntax, it is not clear how best to provide undoable versions of each language

primitive that can produce a storage modification (e.g., a /_ b ?). Languages like Alphard provide a model for how this might be done, by regarding infix operators as abbreviations for function application, but this also requires eliminating the special role of primitive generic functions in Mesa. More generally, the maintenance of a syntax that is acceptable to both people and LALR machines will present problems throughout the system that are not encountered in the pure Lisp implementation.

Editor

If the editor chosen for the *Preliminaries* implements an undo/redo capability, integration with the more general version should be straightforward. Otherwise, these commands would have to be built from scratch, possibly at considerable cost (depending on the editor design.)

In a display-based editor, structure-oriented operations can be largely implemented as part of the selection mechanism. To the extent that the text files are in communion with the underlying S-expression representation, it will be easy to produce the needed hierarchical selections.

Dwim

Some spelling correction and the repair of minor typographically induced syntax errors could be incorporated into the compiler. The user interface to these tools are likely to be quite different in a system based on compilation and display editing.

MasterScope

The structures needed for the *Fundamentals* implementations are adequate for the development of a Masterscope-style database. An extension of the interface used for the other tools would provide user access. The compiler, binder, or an intermediate agent, would notice change and update the data base.

Character of result

We have agreed that very similar results could be obtained starting from either Lisp or Mesa. We also agree that the EPE must accommodate a range of styles including the present Lisp, Mesa, and Smalltalk styles, and our goal is to support these styles without prejudice. However, we expect that a variety of influences will cause both the intermediate and final results to differ depending on the starting point, primarily in terms of the efficiency with which they support early versus late binding. The following two sections set forth what we think the "flavor" of a Lisp- or a Mesa-based EPE would be -- the strengths (and weaknesses) that would result from starting with each of the two possible base languages.

Lisp-based

Aside from the AI effort, a number of other projects in the lab (such as the original wirelister, the Dorado timing analyzer, the PARCHIVE program) have been implemented in Lisp. Perhaps the single most important observation about the use of Lisp is that as users become more experienced, they start building tools within the system to help them. These tools may span many projects, or they may be very specialized; they range from complete languages (KRL) to minor extensions (automatic checking for undefined functions when one leaves the editor). The AI group in particular has made heavy use of this ability to extend the Lisp system: Lisp has been uniquely suited to their approach, which is to solve problems by first building a new language in which to express significant parts of the solution, and then building the solution (simultaneously extending the language). The Humus effort within Mesa offers a contrasting example of an attempt to build an extension on a system that did not support it well: Juniper, for example, probably could have benefited substantially from some extensions of this kind.

It is important that an EPE support the working style which builds tools and even languages as part of the problem-solving effort. Lisp supports this style well in part because it is:

> Incremental: Small changes require only a small amount of work (both mental effort and real time). This is true for both ordinary Lisp programs and programs written in embedded languages. (Current implementations, unfortunately, have relatively weak tools for discovering whether one’s changes are consistent.)
>
> Open: The basic facilities of Lisp are open to change at a very low level: one can modify the operation of the entire system from a user program. There is no distinction between system and user code, variables, name spaces, etc. (This is also a weakness, in that it is not uncommon to find that parts of the system make assumptions about each other that casual modifications violate.)
>
> Integrated: The user can slip relatively gracefully from procedures written in an embedded language back to procedures written in Lisp itself, and vice versa. Since the parser and interpreter are packages, code (in a variety of languages) can be stored in data structures and executed when retrieved.
>
> Aware of user activities: Standard packages in Lisp keep track of new objects added to the system by the user on-line, and changed objects, and help the user keep track of his complex environment. Because all transactions with system objects (such as editing, creating new variables, etc.) are handled through an active intermediary and a set of functional interfaces, it is easy to provide all the "hooks" for complex assistants like Masterscope. .
>
> Abstraction-based: The user is completely freed from concern with basic questions like the representation of integers and symbols, and management of storage. (Most Lisp implementations, including the present Interlisp, have a related weakness, in that they provide few mechanisms for attaining better efficiency even when the user knows full generality is not needed.)

Mesa-based

The "Mesa style" tends to place greater emphasis on structure than on unconstrained flexibility. Probably its two most important aspects for an EPE are its emphasis on static checkability and its provision for explicit interfaces. Both are important in speeding up the programming process and in improving the quality of the result; they become even more so if the units that programmers manipulate are large (packages or subsystems), rather than small (statements or functions); the advantages will be small for programs whose "characteristic times" (design, programming, checkout, existence, total execution) are all measured in minutes, large if they are measured in weeks or months. In an environment where programs are undergoing rapid change, however, mandatory checking mechanisms tend to introduce unnecessary overhead by requiring complete internal consistency at every step of the development process.

Static checking: The present Mesa style encourages (and often requires) relatively early binding of many aspects of programs that in Lisp are typically bound during execution. Mesa also incorporates redundant information (e.g., declarations) that can be checked for consistency. These properties make it possible to routinely check statically for program faults that in Lisp would normally only be detected at run time. (A good example is Mesa's type-checking.) It is generally better to locate faults by static checking, because

> Many faults can be identified in a single run of the checker, rather than surfacing one at a time in debugging runs.
>
> There is experimental -- as well as anecdotal -- evidence for the proposition that program faults located statically are diagnosed and removed more quickly than those located dynamically.
>
> Passing the static test ensures that *all* faults in a given class have been removed; in general, no finite set of test cases gives such assurance with dynamic checking.
>
> "Correctness" is a static property of the program text; it is hard to ensure that a program that relies heavily on dynamic properties actually does what is intended.

A Mesa-based EPE would make provision for binding at a variety of times, but delayed or dynamic bindings would occur at the programmer's request, rather than by default. Tools would also be available to exploit program redundancy to *infer* suitable declarations for programs written without them, and otherwise make it easy to convert programs originally written in a delayed-binding style to earlier bindings. (It should be noted that such tools have already been developed, experimentally, in a Lisp-like environment.)

The belief that this style actually speeds programming (measured as problem solutions per unit time) is intimately tied to the observation that most programmers spend more time worrying about the possibility of programming errors -- and coping with their consequences -- than they actually spend writing new code. This style does ____ require more planning before a running program is created -- some would consider this a disadvantage. A definite weakness of this approach is that it will be relatively difficult to add flexibility that was not anticipated in the program design, thus restricting the range of experiments that can be performed easily.

Explicit interfaces: Much of the effort in the design of Mesa went into the development of facilities for a modular style of programming. Definition modules are program components used to specify interfaces between clients and implementors of abstractions -- permitting them to work independently, and make changes independently, as long as they respect the interface. (Ideally, they would completely liberate each side from any knowledge of the internals of the other, but Mesa has not quite attained this ideal.)

It seems quite clear that one of the major thrusts of our future research must be building systems out of standard components, and decoupling the implementors and users of such components; Mesa provides our best effort so far in this abstraction-based direction. The need to specify interfaces in advance may be cited as either a strength or weakness of the Mesa approach, depending on circumstances and prejudices. The present need for vast recompilations whenever a fundamental

interface is changed is a weakness that can certainly be reduced, and probably eliminated by incremental recompilation on demand.

Other notes:

A Mesa EPE would allow programming to be, in a very real sense, incremental, because much more of the burden of propagating the consequences of any change to a program could be assumed by the compiler and other EPE tools; conceptually small changes would not generally require large amounts of reprogramming.

One happy consequence of relatively early binding is that decisions made statically are generally much less expensive than those made (repeatedly) dynamically. *This consequence should not be interpreted as the motivation for encouraging early binding in the EPE.* Some of the present efficiency of Mesa would definitely be sacrificed in those situations where greater flexibility and later bindings are introduced. However, it should not be hard to retain an "efficient subset" (roughly SDD Mesa).

Effort and schedule -- Lisp base

A Lisp-based EPE (here called XLisp) can evolve from our current systems in a series of steps, with a working (although not stable) system after each step. The steps listed below need not be done in exactly the order given, although we believe this order leads to the least total work. We also believe little parallelism is possible *between* any two of the steps, although, as the assignments of people indicate, a lot of parallelism occurs *within* a single step.

The names attached to steps other than the first should not be taken too literally -- the named people have not been consulted. We have assumed the worst, namely, that almost no volunteers from outside the current Lisp community will participate in this work.

The total of the figures below is: 87 work months, 27 1/2 elapsed months. This total does not include work on Masterscope, but we assume that this work will be done eventually by Larry Masinter, who is not currently available. At the end of this period, we have a system which includes all the present Lisp facilities plus the critical Mesa-like facilities discussed earlier in the Facilities section.

The memo on which this section is based, which contains a fuller discussion of the sequencing and rationale for the various steps, appears in the appendix to this report.

Preliminaries

Dorado Interlisp

The first step is to get Alto Lisp running on the Dorado. This work began in early November and is well underway. For more details see a memo by Bobrow, [Ivy]<Bobrow>dlisp-status.memo.

After this step we have a usable Interlisp system running on the Dorado.

> *Work months: 18*
> *Elapsed months: 6?*
>
> *People: Bobrow, Deutsch, Haugeland, Teitelman; Fikes, Goldstein, Kaplan, M. Kay; Laaser, Masinter, Thompson*

Main effort

Shallow binding

Change the system to use shallow binding and have invisible indirection for variables.

This step produces no externally visible changes in the system, except that GLOBALVAR declarations are no longer necessary.

> *Work months: 3*
> *Elapsed months: 1*
>
> *People: Bobrow (stack fns & interpreter), Deutsch (compiler & loader), Haugeland (microcode)*

Clean up macros

Change the way macros and special syntactic forms are handled, to eliminate "error-driven interpretation". In particular, eliminate DWIM corrections that require altering or examining the state of the interpreter.

We can arrange things so that after this step the system will still run existing Interlisp programs, but MACRO properties and NLAMBDAs will produce warning messages. On the other hand, packages like the iteration and pattern match statements will fit into the system much more cleanly.

Work months: 6
Elapsed months: 1

People: Bobrow (interpreter), Deutsch (compiler), Teitelman (DWIM/CLisp), any 3 Lisp programmers (scan the system); defer work on Masterscope

Dictionaries

Add dictionaries to the system, but not lexically-scoped closures. The hardest part of this step is probably designing the proper interfaces to dictionaries-as-virtual-objects.

After this step we have a system which supports localization of names. Individual packages can be rewritten to take advantage of this at any time. Some lame-duck code must remain in the file package to be able to dump and load individual definitions in a global dictionary so that existing programs will run.

Work months: 15
Elapsed months: 2 (design), 3 (implementation)

People (design): Bobrow, Deutsch, Masinter, Horning/Lampson

People (implementation): Bobrow (stack & interpreter), Deutsch (compiler), Fikes (record package), Kaplan (dictionary objects), Teitelman (file package & miscellaneous); defer work on Masterscope

Closures

Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances. Careful design is needed here to produce reasonably efficient results.

After this step the system will support instantiable packages and Smalltalk-style object-oriented programming. Old programs are not affected; however, some existing packages should be recast in instantiable form, and there will be a period of confusion and trauma while this happens.

Work months: 9
Elapsed months: 1 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Levin

People (implementation): Bobrow (stack functions), Deutsch (compiler), Haugeland (microcode), Teitelman (BREAK et al)

Type calculus

Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking.

The result of this step is to make the type system a first-class citizen, rather than an intermittent visitor as in the Decl package. Programmers can start writing type declarations in the knowledge that they will reap the benefits after the next step.

Work months: 9
Elapsed months: 1 1/2 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Horning/Lampson, Morris, Scott?

People (implementation): Bobrow (interpreter), Deutsch (compiler), Kaplan (type objects)

Compile-time type checking

Implement compile-time type checking.

This step gives us a system that has nearly all the functional capability of a "safe" Mesa, although it is considerably less efficient in its representation of some kinds of objects, e.g. bounded integers and set elements. It is not clear what to do about the Binder.

Work months: 12
Elapsed months: 3

People: Deutsch, Levin, Satterthwaite (compiler), Kaplan (PUTD, APPLY, etc.)*

Variant representations

Implement variant representations, such as unboxed numbers.

This system should be competitive with Mesa, aside from the questions of LOOPHOLEs, pointer arithmetic, etc., and of course it has a lot of capabilities not in current Mesa.

Work months: 6
Elapsed months: 1 (design), 1 (implementation)

People (design): Deutsch, Satterthwaite

People (implementation): Bobrow (stack functions), Deutsch (compiler), Fikes (record package), Haugeland (microcode)

Further work

Infix front end

Implement an infix front end, i.e. an infix parser and prettyprinter.

Work months: 3
Elapsed months: 1

People: Goldstein (prettyprinter), Swinehart (editor), Teitelman (integration)

Permanent pointers

Implement permanent pointers. Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project.

Work months: 6?
Elapsed months: 2?

People (design): Deutsch, Horning, Sturgis

People (implementation): Sturgis (primitives), Teitelman (file package)

Effort and schedule -- Mesa base

As discussed under Facilities above, the plan for a Mesa-based EPE has three major stages: Preliminaries, Foundations, and Features. The stages must be done in approximately the indicated order, although some overlap is possible. An important property of the plan is that growth of the system is highly incremental: even within a stage, there are many points at which a stable system with improved function is available.

This section of the report is a condensation of a memo submitted to the working group. For more detail, see the full memo in the appendix to this report, and the earlier discussion under Facilities.

Preliminaries (IME)

Phase I

Phase I represents the bulk of the IME effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The components of the Phase I system are listed below with estimates of the effort required to produce them.

Approximate Effort to Construct Phase I System: 9.5 work-months

Microcode

Extend the microcode to handle long pointers and page faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

Change the Alto/Mesa runtime environment to handle page faults and support (code) segment allocation in the 24-bit address space. Nearly all of this work has already been done for the Extended Memory Alto version of Alto/Mesa. *Approximate effort*: 1 work-months.

Provide rudimentary protection through separate MDSs for major entities. *Approximate effort*: 2 work-months.

Interactive Base

Acquire a window-oriented display manipulation package, from either the Mesa debugger or the Tools Environment. *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

Alternatively, acquire the Chameleon package from AOS; this would supply a data base package and a text editor as well (see below). *Approximate effort*: 0.5 work-months. (That is, we won't adopt this package unless it appears stable and we can use it with only minor tweaks.)

Text Editor

One possible source for an adequate text editor (i.e. about as good as the Laurel editor) is an existing editor that runs in the Tools environment. *Approximate effort*: 1 work-month.

Chameleon is an alternative source for a comparable editor. *Approximate effort*: 1 work-month.

(The effort estimates above are intended to be upper bounds. Butler believes we can "hack together an editor in 2 work-months", so we probably shouldn't invest more than 1 work-month in

some other package.)

Compiler

Maintain a cache of symbol tables in the virtual memory space. *Approximate effort*: 2 work-months.

Debugger

Extend the debugger command language to include editing, compilation, binding, etc. Also change the debugger to use the common symbol table cache. *Approximate effort*: 1.5 work-months.

Binder

We intend to make no functional changes to the binder. *Approximate effort*: 0.5 work-months.

Phase II

Phase II will provide two extensions to the Phase I system. These extensions are independent and could be performed in either order.

Approximate Effort to Construct Phase II System: 4 work-months

Replacing an Existing Module

Make the compiler retain more information about the structure of a compiled module, so that a slightly changed module can often (almost) be slipped into the system as a replacement for the old version without altering the current state. *Approximate effort*: 2 work-months.

Change the loader to find and alter extant frames affected by loading a new version of a module. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

Implement the DeSoto dependency analyzer, which ensures that all modules that must be recompiled will actually be recompiled. *Approximate effort*: 1 work-month.

Foundations

As discussed under Facilities above, there are five major basic language/system facilities which need to be added to existing Mesa. These facilities comprise the Foundations stage of the Mesa-based EPE work.

Approximate total effort for Foundations stage: 20 work-months, including 4 work-months for cleanup of the current Mesa compiler. These estimates result from consultation with Ed Satterthwaite for compiler-related things, and with Peter Deutsch for things copied from Alto/Dorado Lisp.

1) Programs as data

S-expression representation. *Approximate effort*: 1 work-month.

Atoms. *Approximate effort*: 1 work-month.

Universal Pointers. *Approximate effort*: 4 work-months.

2) Variable syntax analysis

Implement programmer-controlled parsing at input time, using the existing Lisp read-table design. *Approximate effort*: 3 work-months.

Implement programmer-controlled extension at compile time, using macro expansion, based on Mesa in-line procedures. *Approximate effort*: 1 work-month.

3) Dynamic variable binding

Make the compiler produce a table of entries of the form

atom -> <frame,type>

for each compilation unit, and use this to implement Lisp-style dynamic variable binding. *Approximate effort*: 1 work-month.

4) Safe language

Prevent uninitialized pointers and array descriptors. Do bounds checking on arrays. Exclude dangerous constructs (e.g. LOOPHOLE) from the language. Replace @ on local variables by **var** parameters. Replace explicit freeing by deferred freeing or automatic deallocation. Make relative pointers safe. *Approximate effort*: 2 work-months.

5) Automatic deallocation

Implement the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an automatically deallocated type. *Approximate effort*: 3 work-months, including necessary compiler changes.

Features

This section attempts to estimate the cost of the "programmer's assistant" facilities of a Mesa-based EPE. Here we consider the activities required to provide a more or less direct replacement for existing Interlisp functions. As discussed in the Facilities section above, we do not believe that this is the best approach to providing these facilities in the EPE.

We assume that the work described under *Preliminaries* and *Fundamentals* has been completed. The text editor and debugger described in those sections, perhaps augmented by Mesa Tools, would certainly provide an adequate interface. The time estimates are presented in the same spirit of optimism that pervades the rest of this document.

Interpreter

Extend the compiler to handle single non-declarative statements in specified contexts. An evaluator within the debugger would generate the appropriate linkages to get the statements executed. Save sufficient information to provide the History List features (below). *Approximate effort:* compiler extension: 2-4 work months; evaluator, top-level environment: .5-1.0 work months. If all or part of this work is implicit in providing the "programs as S-expressions" fundamental capability, the estimates can be reduced accordingly.

Break and Advise

Extend the Mesa debugger's current conditional breakpoint code to obtain a break/advise facility at least as good as Interlisp's. *Approximate effort:* 1-3 work months.

History Lists

Implement Undoing, as in Lisp, by retaining S-expression representations of the original statements and their reversing functions. Integrate history facilities with Mesa's more complex environment structure. Add language syntax for undoable operations. *Approximate effort:* 2-4 work months to implement the mechanism; development of primitives and system procedures with undoable versions as time goes on.

Editor

If the editor chosen for the *Preliminaries* implements an undo/redo capability, integration with the more general version should be straightforward. Otherwise, these commands would have to be built from scratch, at considerable cost (depending on the editor design). *Approximate effort:* 1 work month or ? work months.

Implement structure-oriented operations in the editor. *Approximate effort:* 1 work month.

Dwim

Incorporate some correction of minor spelling and typographically induced syntax errors into the compiler. *No estimate*.

Masterscope

Extend the interface used for the other tools to provide user access to a Masterscope-like data base. Put code in the compiler, binder, or an intermediate agent to notice changes and update the data base. *Approximate effort:* the data base facilities, no estimate; the interface, 2 work months.

Participating people

All the people listed at the beginning of the report are candidates for participation in the actual EPE effort, and most of them have indicated a willingness to contribute in some way regardless of whether a Lisp or Mesa base is chosen. However, nearly all of these people have substantially more experience in one of the two languages than in the other. To achieve our goals, it is essential that people with both kinds of experience participate in the project in a major way: we are very concerned that if a Mesa base is chosen, people with Lisp backgrounds might not feel highly motivated to contribute their unique experiences and insights to the project, and conversely if Lisp is chosen.

The following are individual responses to a message we sent out to everyone who had indicated an interest in the programming environment area by putting themselves on an on-line distribution list. The absence of someone from the following list does not mean they are not interested in working on EPE.

Birrell

I would probably be interested in low-level work in EPE (i.e. the parts which might be considered as an operating system), and in at least keeping track of language design.

Bobrow

If it is a Mesa based EPE, I expect to put about a third of my time in on it. If it is a Lisp based EPE I expect to put in about 2/3.

DaveSmith

I cannot participate in actual implementation design or coding. I guess that puts me in the class of kibitzers. However I have several ideas on programming languages and programming environments that I would like to share. (Most of these ideas came out of the LISP70 project that Enea, Tesler and I did at Stanford.) I am anxious for EPE to succeed because PARC is not currently at the state-of-the-art in programming environments. In the short term, some simple extensions would benefit us all (such as Lampson's and Levin's intermediate Mesa PE ideas). But in the longer term, I hope EPE turns out to be a laboratory for basic research into programming languages and techniques. I think there's at least a 5-10 year project waiting there.

Deutsch

I expect to spend about half my time on EPE through mid-1979. What happens after that depends a lot on how well we are doing at genuinely incorporating the strengths of all three languages into a single system. If things are going well, I expect EPE to continue as my largest single interest for at least another year after that.

Gifford

I am interested in the tradeoffs between protection mechanisms and type systems for builing layered systems; I am especially interested in the design of a flexible protection scheme for the D* machines that would enable the construction and debugging of large systems. This interest also encompasses hardware/microcode support for a run time type system. I strongly favor Mesa over Lisp as the EPE base.

I plan to spend the first part of January reviewing the EPE material in detail. It may be possbile to view part of my work in the comming year as EPE related, or my work as part of the EPE project, or both; it depends on the intersection of my thesis interests with EPE needs.

Guibas

Here is a very selfish reply. At this late hour it is perhaps best to paint matters as black or white, rather than gray.

An (incomplete) list of epe areas that interest me is:

arithmetic, text formatting, graphics, pattern matching, and searching.

My motivation is twofold: to have a friendly environment in which to do my own research, and to make sure certain fundamental facilities are provided at an acceptable level of quality.

My time commitment would depend heavily on my research interests and the congeniality of epe to my working style. For example, a very important attribute for me is consciseness of notation. Has it been a serious epe desideratum?

With regard to base language choice, my deep-down feeling is that a lisp based environment would be infinitely more appealing to me than a mesa based one.

Then again, don't underestimate the power of persuasion...

Horning

I consider myself committed to EPE as my major technical effort for the next couple of years, independent of starting point. My background implies that my contribution to early phases of the project will depend on the starting point: with Lisp, I would expect to contribute to requirements definition; with Mesa, I could get involved in the system programming.

Lampson

I expect the EPE to be my major research interest for the next 18 months. There is some chance that I will change my mind about this if Lisp is the base. I expect to be able to spend a lot of time on design, but am less optimistic about begin able to do any substantial programming jobs.

Levin

I consider that, if EPE is Mesa-based, I will consider it my second-priority project (after the distributed transport mechanism). That translates to about 30% of my time over the next year. After that, I can't say. I would expect my involvement with EPE to be greatest during the earlier phases, when fundamental structures are being built. I probably couldn't contribute much to a Lisp-based EPE.

Masinter

I plan (if I am hired by CSL) to devote full time to EPE starting July 1 working on those pieces of system which are prerequisite to building Masterscope-like facilities for Mesa. I imagine that actually getting to actual Masterscope facilities will be in the rather distant future: working on either an S-expression representation (easier to analyze), data base facilities (shared data base server), access to remote files, etc. will take first priority.

Morris

I might sign up to work on the design and implementation of a more powerful static checker for the language. I guess about half-time would be my level of commitment.

Satterthwaite

If Mesa is chosen as the base for CSL's EPE, the implications for SD will be carefully considered. We have no current position on the numerous policy questions that would be raised by direct participation, but contributions to subprojects of mutual interest seem likely to the extent that SD's goals and schedules permit.

My principal interests re the EPE issues are (a) designing a "strong" but much more flexible type system, (b) (for Mesa) designing a safe language subset and storage reclamation techniques, (c) implementation of (a) and (b).

Suzuki

Database:

There are two issues concerning database in EPE. One is the database for interaction, namely masterscope level thing. I would like my database system to be considered for the use in EPE system along ASD and masterscope. I will demonstrate the system tomorrow so many of you may have a chance to evaluate. The other is the access from the programmers. This may be some type extension and packages. If the previous line is chosen I will spend quite a few time of mine devoting bringing up the user interface. If not I will be interested in the latter subject which I would be interested in spending some of my time.

Basic microcode:

I am willing to support Dorado Mesa and its extension. Birrel may want to do this in connection with Pilot. In that case I am willing to help him.

Subrange checking, verification, etc.

Probably not much to do for a while.

Compiler, etc.:

I have quite a few experience in writing compilers. But there are so many people who can do it, so unless there is some urgent need I may not particularly want to do it.

Swinehart

I'm willing to play a substantial role in what you've called the Tools segment of the EPE project. I cannot in all honesty call it my primary activity, however, since it seems extremely important to further the voice communications work. Percentages are probably misleading, but I'd expect to devote 30-50% of my time to it. Goals: unified (default) approach to user interface for program development, control of applications.

Taft

I expect to spend a majority of my time on the Laurel transport project during the next 6 to 9 months, so I can commit only a fraction (say, 25%) until that is done, but perhaps more later (50 to 75%).

As for what I might do: if the EPE is Lisp-based, one obvious task would be to implement a Pup package. If it is Mesa, I am less certain what to do, though one possibility is to extend the file system to access IFS or Juniper remotely. I am also prepared to contribute effort in whatever other ways seem appropriate, e.g., writing Dorado microcode. Since I am not familiar with the insides of either Lisp or Mesa, and I am only superficially familiar with Lisp as a whole, it is probably not appropriate for me to work on the basic kernel of EPE.

Teitelman

I expect significant portions of my time will be involved in epe, regardless of which base they start from. If mesa, then would be in implementing various user level packages not in mesa, such as programmer assistant, file package, dwim type stuff, etc. If starting from lisp, I have a less clear

picture of what I would be doing, at least initially, since a lot of things have to be changed down in the guts of lisp, but I would think I would be fairly involved simply because I am currently custodian of so much of the system, and can also make inputs about how various changes will affect the style of lisp programming.

Project organization

The working group did not spend any time on organizational questions. However, we identified two areas in which people would be needed almost immediately, and noted the possible relevance of the current work on Dorado Lisp. In addition, a 3-person coordinating committee for the EPE project has been appointed by CSL management (Bob Taylor). This committee (Deutsch, Horning, Lampson) has met briefly and produced some organizational ideas described below.

Coordination with SDD

If a Mesa base is chosen, we will need someone to function as a continuing liaison with SDD. This person will need to be familiar with the language and system work being done in both places, to bring both potential divergences and opportunities for sharing code to the attention of the appropriate people doing the work. We believe that our goals and SDD's are similar enough, at least at the language level, that such a liaison could be very helpful.

IME (Integrated Mesa Environment)

As mentioned in the earlier section on Effort, we believe it is worthwhile to make a short-term investment in improving certain gross aspects of the current Mesa environment, if a Mesa base is chosen. We view this as an initial phase of the EPE (even though much of the code is likely to be superseded later) and hence it should be coordinated with the rest of the project.

Dorado Lisp

The current project to produce an Interlisp system for the Dorado is independent of EPE. However, if a Lisp base is chosen, the starting system will be Dorado Interlisp. In this case it will be to our advantage for people with EPE interests to become involved in the Dorado Lisp project sooner rather than later. We also expect that in this case the current CSL Lisp community will be the initial core of the EPE design and implementation effort.

Proposal for EPE project organization

We (the coordinating committee) believe the project should be organized into three areas: language/system, tools, and packages. Naturally these areas overlap; one of our functions is to facilitate communication among them.

In the language/system area, we believe the work will demand a fairly small, closely knit group of 3-4 people to work on the compiler and run-time system. At least one person from the coordinating committee should be a member of this group.

In the tools area, we believe that more autonomy of individual projects is possible, although certain decisions with system-wide consequences (such as display management, low-level hooks for programmer assistance, and data base interfaces) must be made at a global level.

In the packages area, we would like benevolent anarchy to prevail, subject to some planning for things we know will be widely used throughout the system (e.g. network communications) or require special expertise (e.g. high-quality mathematical routines). Our idea is to set up a standards committee, which would develop and publish a set of criteria that would be applied to all packages submitted for inclusion in the official library. Packages would be read by the committee for agreement with these standards, and by one other person (not the implementor) for appropriateness of design and implementation. Since this procedure would introduce a certain amount of inertia into the system, we envision the existence of an unofficial library, with the understanding that a package is only allowed to exist in this library for a limited amount of time (say six months), and that anyone who uses the package during that time may have to rewrite some code if the package's interfaces change as a result of the approval process.

Foreign affairs

The choice of Lisp or Mesa as the starting point for an EPE project within CSL has implications for other projects within CSL, for other organizations, and for CSL's relations with those organizations. Although we are agreed that "foreign affairs" should be given less weight than CSL's own needs, they are a legitimate concern in making the decision, and may prove to be one of the balance tippers.

The following discussion enumerates many of the areas of concern and indicates the probable effect of either decision. To a high degree, the effects are "formally symmetric," i.e., for every advantage of one choice, a corresponding advantage of the other choice can be found. However, CSL may wish to give higher weight to relations with the rest of Xerox than to its relations with the world outside Xerox. For intra-company relations, somewhat more advantages follow from the choice of Mesa.

Separate sections discuss the effect of the decision on the importation and exportation of people, ideas, and code, and on CSL's "image."

Importation

Importation of ideas

From within CSL

Issue: There is a lot known about programming environments within the lab, and people who are not directly involved in the project are still potential sources of good ideas.

If Lisp is chosen: The people currently working in and on the Lisp environment will likely be within the EPE project. Some effort will be needed to ensure that Mesons concerned with interfaces, static checking, verification, and optimization provide enough input to ensure that they can profitably build on the EPE as it becomes available.

If Mesa is chosen: The Mesons mentioned above will likely be within the EPE project. Some effort will be needed to ensure that the Lispers concerned with programmer assistance, programs as data bases, and integrated sublanguages provide enough input to ensure that they can profitably build on the EPE as it becomes available.

From within Xerox

Issue: Other parts of Xerox, most notably SDD, are also concerned with developing programming environments (although with somewhat different characteristics than EPE requires). We should be prepared to shamelessly steal good ideas.

If Lisp is chosen: The relationship to SDD efforts will be somewhat more distant, the application of their ideas somewhat less obvious.

If Mesa is chosen: Much of what SDD does will be directly relevant; there will be continual pressures to remain compatible, and to import code, rather than just ideas.

From the world at large

Issue: We would like to track the state of the art in the world at large.

If Lisp is chosen: Much of the "automatic programming" and "programmer's assistant" type of work is done in various Lisp dialects, and should be easily applicable. This group of researchers is currently being encouraged to focus their attention on the development of a programming environment for ADA (formerly DoD/1); however, much of their research work is likely to

continue to be in Lisp.

If Mesa is chosen: Much of the specification and verification work is directed towards Pascal dialects, and should be easily applicable. ADA, like Mesa, is based on Pascal, and work on ADA environments should be highly relevant.

Importation of people

Issue: We want people to move into the project easily, and are even more concerned that it be easy for users to convert to the *use* of the EPE.

From within CSL

Lisp and Mesa have roughly comparable numbers of "hard core" users, and the issues of migration seem roughly symmetric.

From within Xerox

Mesa will be much more widely known and used within the corporation.

From the world at large

Generic Lisp is more widely known than Mesa; Generic Pascal is more widely known and used than Interlisp. If there is a systematic difference, it probably lies primarily in the communities from which we expect to recruit.

Importation of code

From within CSL

Issue: Existing service packages should be made compatible with, or absorbed into, EPE to the greatest extent feasible. We want to avoid fragmentation of both user and maintenance effort. (In addition, importing people implies importing their research systems.)

If Lisp is chosen: The Interlisp packages come over almost intact. Will Bravo and Laurel users accept them as substitutes, or will functional equivalents of those systems be needed?

If Mesa is chosen: Provision of functional equivalents of the Interlisp packages will be a major chore. Laurel comes over, but something would still need to be done about Bravo.

From within Xerox

Issue: We want to stand on the shoulders of SDD and ASD wherever possible, both in terms of service packages/systems, and program development aids. CSL will want to experiment with prototypes and systems built elsewhere, and would benefit from the ability to incorporate outside packages and subsystems into our experimental systems.

If Lisp is chosen: Nothing comes over.

If Mesa is chosen: The amount of actual code that we can import depends both on the degree to which we force our EPE Mesa to remain equivalent to SDD Mesa, and the extent to which the solutions they choose to provide match the problems we need solved. Transferability will probably require that EPE Mesa be a strict superset of SDD Mesa.

From the world at large

Issue: It would be nice not to have to replicate large system-building projects from elsewhere.

If Lisp is chosen: There is significant precedent for Interlisp system facilities being imported from elsewhere. The amount of actual code that we can import in the future depends both on the degree to which we force our EPE Lisp to remain equivalent to Interlisp, and the extent to which the systems built elsewhere solve the problems we have. There is apparently relatively little history of importing application-level code in Lisp.

If Mesa is chosen: No code comes in.

Exportation

Exportation of ideas

To the rest of CSL

Issue: We want to "talk their language," even before conversion to EPE is complete.

Again, the Lisp and Mesa situations seem quite symmetric; special efforts at communication will be needed in either case to avoid alienation.

To the rest of Xerox

Issue: We would like to be able to demonstrate feasibility of ideas, and let the development groups take them over and produce the production systems.

If Lisp is chosen: The credibility gap will be somewhat larger, particularly if adequate performance of a demonstration system depends on the full power of the Dorado.

If Mesa is chosen: We will be under continual pressure to export not just ideas, but code, and then to maintain that code for the whole Xerox world.

To the world at large

Issue: As part of the commerce of ideas, we must give as well as take. (See also the discussion under importation of ideas.)

If Lisp is chosen: We communicate easily with the AI and theorem-proving communities.

If Mesa is chosen: We communicate easily with the programming language, system implementation, and structured programming communities.

Exportation of people

To the rest of CSL

Issue: None of us sees EPE as his life work.

However, regardless of starting point, we expect EPE to be *the* major programming tool within CSL for some time to come; there should be no major problem moving from EPE to any other CSL programming project.

To the rest of Xerox

Issue: One of the best ways to transfer ideas is often to transfer the people who carry them. We would like to make (temporary) transfers easy and productive.

If Lisp is chosen: In addition to the problem of moving to a less powerful programming environment, there will be a definite (but, one hopes, temporary) linguistic gap of the same order of magnitude as the effort required within CSL to convert a Lisper to a Mesa-based EPE. This will significantly increase "swapping overhead."

If Mesa is chosen: The transfer may be almost too easy; we need to ensure that we get people back!

To the world at large

Issue: Do we really want to encourage this, except on a temporary basis?

As with importing people, this largely depends on the communities with which we want to interact most strongly.

Exportation of code

To the rest of Xerox

Issue: Some systems that we build primarily for our own use may prove to be so good that we want them used widely within the corporation (e.g., Bravo); in other cases we may need a large community of users to enable our own research (e.g., message transport system).

If Lisp is chosen: This won't happen, *unless* provision is made for "severable" systems (analogous to Mesa image files) and programs can be tuned to run with acceptable efficiency on D0's.

If Mesa is chosen: This depends on degree of source-language compatibility for packages and subsystems. It is probably simpler at the system (image file) level, as with Laurel.

To the world at large

If Lisp is chosen: It probably won't happen, except for a few research sites whose interests include both Lisp and high-capability personal computing.

If Mesa is chosen: It definitely won't happen.

Effect on Image

In the rest of Xerox

If Lisp is chosen: Software development groups throughout the corporation are likely to perceive this as a CSL "abandonment" of Mesa just as it is being adopted elsewhere. This may tend to lessen our credibility, and somewhat reduce the morale of Mesa proponents and users.

If Mesa is chosen: CSL would probably be perceived as doing research of greater relevance to the corporation. There would be a greater chance of development organizations converting to environments similar to EPE as its advantages are demonstrated (and as more capable hardware becomes cheaper).

In the world at large

Issue: Either choice will increase the "black hole" perception in some communities relevant to us.

If Lisp is chosen: The programming language, system implementation, and structured programming communities will find it harder to communicate and cooperate with us.

If Mesa is chosen: The AI, automatic programming, and theorem-proving communities will find it harder to communicate and cooperate with us.