Inter-Office Memorandum

To	EPE Group	Date	September 12, 1978
From	Daniel G. Bobrow	Location	Palo Alto
Subject	Some Lisp features I would be loath to do without; or would you take a fish without a bicycle	Organization	CSL

XEROX

Filed on: <Bobrow>lispfeatures.memo

I have been introspecting on how we have used Interlisp to implement and embed KRL, and the following are my distillations of the features used which I would want to have in our EPIC (Experimental Programming Integrated Colossus). To make these next points more specific, here are some fragments from a particular version of KRL-1.

Here is a KRL unit taken from a cryptarithmetic problem as it appears on a file.

 # Column
self:^1 1:Trigger(ToPrint,'(For x in '(top bottom sum)
do (PRINT (SeekMy 'Pointer SomeSignalTable))))
top: An integer
bottom: An Integer
sum:^2 An Integer 2: Trigger(ToFind, '(LispForSum INSTANCE))

Here is a fragment from the middle of some Lisp code that might be run using some mixed KRL and Lisp syntax.

```
(SETQ SUM
   (Seek 'Pointer
         \the summ fromA Column thatIs !Post (CAR columnList)//
         FullMatch))
```

1. Variable Syntax Analysis: That is, procedural escapes in an available parser. The built in Lisp parser (a structure parser that knows about parentheses, brackets, and lexical analysis) also allows access to the basic reading routines (read a character or a symbol or a ...) and allows certain characters to signify procedural escapes.

In the example above there are two different forms of variable syntactic analysis illustrated. In the unit there are examples of the embedding of Lisp expressions; thus a special parser for units must be able to call on the Lisp standard reader. Secondly, in the middle of the code fragment is a "\" indicating an escape to the KRL reader.

2. Programmable Lexical Analysis: Another name for being able to read in atoms under some control (I don't think Lisp's readtables are optimum). Reading results in a unique pointer associated with that string. The unique pointer must be able to be used in various kinds of symbol tables and programs eg

1) To access a procedure definition which can then be run; in the example LispForSum is a procedure to be run.

2) To obtain a value (current under some scoping); In the unit, the variable INSTANCE is assumed bound in the interpretive context.

3) To be compared to a constant in a program; eg the first argument to Seek is a constant which is going to choose a case.

3. Procedural Embedding: This consists of two parts; the use of program text in the context of other structure (the problem of getting in and constructing structures which can be treated as expressions to be evaluated); and the evaluation of said structures in a specified envirionment. In a compile time system, I believe it would be possible to fix the environment as one specific to the interpreter, and/or pass to the interpreted code a data structure which captures the appropriate state. The program structure used should be analyzable and evaluable. Two different kinds of embedding are illustrated: in the unit a prgram fragment in real Lisp as used by the Triggers; and in the code fragment, the ability in Lisp to build a special purpose compiler which can substitute for the Seek expression containing a constant data structure argument some other arbitrary Lisp expression which is then compiled or interpreted by the Lisp system.

4. File Pointers: This is my key phrase for being able to refer to a large extended address space from within Lisp. This is used in KRL for example to point back to the original symbolic version of a unit on a file from the in core version, and the update package to copy symbolics when making incremental changes to a file.

5. Hash Linking: Optional fields in data structures are used often in KRL, and are implemented by using the address of the data structure as a key in a hash structured symbol table. For example, pointers to the triggers for a slot are stored in this way since most slots don't have triggers.

6. Spelling correction: Knowledge by the system of names appropriate for certain environments, and correction of the name I typed to a close one which is appropriate. This is especially useful in correcting names of data fields in units.

7. Evaluation in editing context: To test changes in an edited function, I sometimes select a particular form which I have isolated as having a bug, change it, and want to evaluate it in the context of the program as reflected in the program stack at the time the bug manifested itself.

8. Simple linked data structure. Lists are a super first class data type in Lisp, and their existence should not be ignored, nor the wide range of library functions to manipulate same.

9. Both for lists and other data types we depend on the system maintaining storage for us, providing automatic allocation and deallocation.

Here are my interpetations of some of the comments I got from Doug Lenat about what he likes about Interlisp:

10. The thickness of the manual: There are few indispensable functions, but throwing away any 25% would be a blow to my programming style.

11. Portability of knowledge: AI programmers come knowing some brand of Lisp, and immigration into Lisp is therefore much easier than into some other language.

12. Use of internal structure of names. I explode atom names so that I can search for patterns in them. I also generate plausible names to go with new concepts defined, and programs constructed by my program.

13. Programmable Interrupts: I use the abaility of Lisp to make single characters force a call on a function to monitor and control the the computation in my system.

14. Personalized macros: I added some user macros to personally tailor the editor to my style.

15. Interaction monitoring: I used the easy facility to have typescript files to keep hardcopy records of my interactions.

16. Performance analysis functions: I used **breakdown** (which rewrote my functions to monitor their performance) to tune and understand my system.

17. Programmable debugger: Conditional monitoring of function calls and returns, and firewall protection (ERSETQ) was important in getting my large system up.

Inter-Office Memorandum

To	EPE Working Group	Date	September 14, 1978
From	P. Deutsch (including comments by McDaniel and Swinehart)	Location	Palo Alto
Subject	Unique Smalltalk capabilities	Organization	CSL

XEROX

Filed on: <Deutsch>stcap.memo on Maxc1

This memo is a first stab at listing in detail those capabilities of Smalltalk which are not present in either Lisp or Mesa, and which are important for an EPE. Comments by Gene McDaniel and Dan Swinehart have been incorporated in the memo without attribution.

Language

>>Syntax: Smalltalk has an infix syntax describable in less than 2 pages of equations, including the lexical syntax. It has no reserved words and only about a dozen reserved characters. Smalltalk's syntax provides the following advantages:

(1) The syntax can be learned very quickly by anyone who knows any formal language, and there is never any need to refer to the manual to decide whether a given syntactic construct is legal.

(2) It is possible to design a very simple internal representation of parsed programs, which can be effectively manipulated by programs.

(3) The syntax includes the capability for certain kinds of extension such as user-defined control structures.

(4) There is a uniform notion of "application" which encompasses all language constructs that carry out operations: this means that a future macro-expansion capability will be very easy to add.

(5) The use of punctuation for the common operations (statement grouping and conditionals) enhances compactness without sacrificing readability.

One consequence of this simple syntax, of course, is that a lot of things which one thinks of as being part of the "language" in Mesa become "packages" in Smalltalk (just as in Lisp), but that is just a matter of how the documentation is arranged.

Can the Mesa syntax be simplified to include the above advantages? Can the Lisp infix syntax provide (1)? Can the tangle of Lisp compile-time capabilities (macros, CLISP expansion, etc.) be simplified to provide (4) without giving up (3)?

>>Object-oriented programming: in Smalltalk, all objects interpret the operations legal on them. There is no notion of a free-standing "procedure" or "data structure". This has the following advantages:

(1) The space of operation names is local to each type -- there is no need to qualify names (as in Mesa), or invent names that include the object type in them somehow (as in Lisp).

(2) There is no need to declare the type of the receiver of a message, or to qualify or "open" the names of its components as in Mesa.

(3) All operations are automatically generic, with efficient support from the underlying virtual machine.

A consequence of (1) is that Smalltalk syntax may be a reasonable model for certain application languages as well: in particular, there is no need to provide a command interpreter that looks up a special command set in a table, and eventually spelling correction and type checking could be done by the language mechanisms as well.

Can Lisp be modified to include appropriate mechanisms for name scoping and interpretation for (1) and (2)? Can either Lisp or Mesa provide (3)?

>>Hand coding: Smalltalk provides no way for programmers to write code in any language lower than Smalltalk, and could easily do without the loophole which allows programmers to access arbitrary memory words directly. In Lisp, by contrast, programmers constantly give in to the temptation to write assembly code and/or direct data accesses for efficiency, which result in obscure bugs and machine dependencies.

Can Lisp be modified to provide a sufficiently complete set of non-machine-level data access mechanisms to make assembly code unnecessary? Can Mesa be modified to remove or greatly constrict the loophole mechanisms (including dangling pointers) that compromise program correctness in obscure ways?

Can either Lisp or Smalltalk be upgraded to provide a compiler good enough to reduce the temptation to hand-code to an infrequent, rather than a daily, level?

>>Inheritance of operations through subclass mechanism: Smalltalk makes it relatively easy to create classes of objects which are specializations of more general classes (e.g. Atom specializing String specializing Array), or which supply the implementation for a generically defined class (e.g. Integer and Float implementing Number). Work is going on now to extend this to more complex ways of combining behaviors into new ones. This inheritance idea includes at least some of the Mitchell-Wegbreit "schemes" ideas, as a way to specify shared behaviors only once, and seems to reduce the amount of code one needs to write to make use of existing classes.

Lisp has no class structure at all; can Mesa’s notions of interfaces and modules be extended to provide the flexibility of current and promised Smalltalk subclass mechanisms?

>>No second-class citizens: Smalltalk makes absolutely no distinction between system- and user-defined types or operations, except that certain operations on system-defined types are provided in lower implementation layers.

Can Lisp be extended to provide the benefits of allocation, access, printing, checking, and special compilation of user-defined types and their operations that now are the exclusive province of the built-in types?

>>Future directions: some experiments in future expansion of the notion of programming language are being carried out in Smalltalk now. How hard would it be to do these experiments in Lisp or Mesa?

The proposed template style of two-dimensional object description.

Constraints (Alan Borning's work). Dan Swinehart has been doing some display interfaces (e.g., audio board test program) using Keith Knox's package as a base, where user changes in one entry must cause changes in others -- and vice versa. When combined with the desire to initialize these entries with default values and a few other things, he found the resulting ad hoc programs to be complex and cumbersome. A constraint language would be ideal for specifying these relationships.

The proposals to embed data base linkages as primitive notions in textual objects, such as source programs. (This might be a way to handle some of the linkages needed by Masterscope.)

Tools

>>Browser: all system and user source code is instantly available for browsing, including documentation and table of contents per class, and class- and method-level categories.

Lisp goes part-way towards this, in that it is relatively easy to view the source code of one's own programs. Can Lisp or Mesa make the system code equally accessible? Can Lisp or Mesa incorporate (and require use of) the category system, which greatly helps finding one's way around in one's own and others' programs?

>>Integrated on-line documentation: the programmer can easily insert documentation in the code as it is written or edited. (Off-line documentation is admittedly awful -- it could be improved even by some simple automatic extraction of on-line stuff.)

>>Immediate incremental compilation.

Can the Mesa compiler be made "instantaneous"? (Smalltalk compilations, even on the Alto, are not a major computational bottleneck.) Can the Lisp compilation process be made truly invisible and automatic without giving up the fast turn-around from editing to execution?

Packages

Arbitrary-precision integers with smooth transitions.

Paragraph formatting for display and Press output.

Bitmap and display coordinate manipulation.

Paragraph and picture (bitmap) editor.

Windows with standard behavior -- used by interactive tools.

Menus.

Other

Non-preemptive window behavior (consistent model of interaction). Makes mutually non-interfering applications easy.

28-SEP-78 13:53:02-PDT,17821;000000000000
Date: 28 Sep 1978 1:48 pm (Thursday)
From: Horning
Subject: Re: A reminder
To: Deutsch

Subject: DRAFT Mesa -> Lisp Requirements
To: Programming Environment Interest: Bobrow, Deutsch, Horning, Lampson, Levin, McDaniel, Masinter, Mitchell, Morris, Satterthwaite, Suzuki, Swinehart, Taft, Teitelman;

DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT

[This is a draft, being circulated now for comment. Sept 28, 1978]

[We recognize the considerable merits of the Interlisp programming environment. This document is intended to point out what we consider to be the correspondingly major advantages of the Mesa environment. We feel strongly that these features should be preserved in any EPE. To sharpen the discussion, we have made our remarks quite pointed. We trust that in most cases our "challenges" can be met (with varying difficulty); the reason for posing them is to ensure that they are not overlooked.]

[Items drafted by Horning:]

>>Static Checking

ISSUES: One of the major advantages of the Pascal family of languages (of which Mesa is a member) is the substantial amount of consistency checking that is possible statically (i.e., independent of program execution). Examples of this are syntax checking, which catches many transcription/input errors, and checking for declaration of all names used in a program, which catches both spelling errors and misunderstandings of the environment. However, probably the most important form of static checking is for type consistency, which is very effective at catching many forms of errors, including many "logical" errors.

There are several reasons for preferring static to dynamic checking:
- -it occurs sooner, thereby giving quicker feedback;
- -errors are more easily related to their sources;
- -many errors can be detected in a single run, rather than being identified one at a time;
- -complete checking is not dependent on executing an exhaustive set of test cases (once a program is accepted, one KNOWS that it has no lurking bugs of this sort);
- -the overhead of checking can be paid once, rather than repeatedly at execution time.

If programs developed in the EPE are to achieve laboratory-wide usage (like Bravo and Laurel), then their robustness cannot depend on run-time interaction with their developers. (Not only is there no debugger in Peoria, there generally isn’t one here, either!)

Many Mesa programmers are willing to accept "long" turnaround times for program changes. Perhaps one reason is that they have much more confidence that their programs will actually work once they are accepted by the Mesa compiler.

CHALLENGE: How can the LISP system be modified to allow a comparable

amount of static checking? In particular, can static tests be performed that ensure freedom from type errors and use of undeclared names?

>>Static Scoping

ISSUES: Mesa and LISP have quite different rules for accessing non-local ("global") variables. In Mesa, the association between name and variable (or, if you prefer, access algorithm) is determinable statically, and is a function of the procedure declaration and its environment. If this association cannot be fixed, it is difficult to develop robust procedures to manipulate data structures that out-live particular procedure activations. It is completely unacceptable for accidental re-use of some name in a calling environment to affect a procedure's access to its non-local data.

Scope rules serve another very useful function. They provide two sorts of limits:

- -they limit the set of names accessible within the program at any given point (the amount of "relevant environment" that the programmer need consider), and
- -they limit the range of text within which a name can be accessed, and hence the amount of text that need be considered when considering a change or searching for a bug involving the named entity.

CHALLENGE: How can LISP provide a satisfactory mechanism for statically associating names and non-local variables? How can names be suitably localized?

>>Encapsulation

ISSUES: Even packages that have a large amount of local state should have narrow interfaces. Ideally, the user should know (and use) only the published interface.

In Mesa it is possible to declare items (data, functions, procedures, types, etc.) to be PRIVATE to a module, and inaccessible outside. This is invaluable in building packages, because it ensures that only the PUBLIC interface is used by clients; assumptions made by the implementation cannot be violated from the outside through either inadvertance or malice.

Mesa allows a program to create multiple instances of a module, supplying, if desired, different parameters to each instantiation. This supports an "object oriented" style of programming, in which each object is referenced only through its associated suite of functions and procedures.

CHALLENGE: How can LISP provide an adequate encapsulation facility for packages? Can LISP support multiple instances of package prototypes, and object-oriented programming?

>>Explicit Interfaces

ISSUES: Mesa provides a special type of module (a definitions module) specifically for the purpose of declaring interfaces. This makes it possible to separate the development of implementing and client modules; with few exceptions, the changes that don't affect the definitions module are invisible to the client, and those that do require corresponding changes in the client.

Explicit documentation of interfaces, of course, is good programming style, and mandatory for effective multi-programmer cooperation. Static checking that

interfaces are adhered to is a bonus.

By separating the definitions module from the implementing module(s), Mesa makes it convenient to test a module in an environment designed for the purpose, and have some assurance that it will continue to work properly when substituted into a larger system with the same interface.

CHALLENGE: How can explicit, enforced interfaces be introduced into LISP?

>>Consistent Version Checking

ISSUE: Programs are developed through many versions. The problems that result from inconsistent versions can be extremely subtle, yet it is not always easy to ensure that a set of components being run together are actually consistent. This program is alleviated (but not completely solved) in Mesa by enforcement of the rule that the components must use the same versions of interfaces (definitions modules).

CHALLENGE: How can LISP prevent the accidental execution of systems built from inconsistent versions of components?

>>Efficiency

ISSUE: One of the key principles of Mesa is that the programmer shall have effective access to the full power of the machine, and that implicit run-time overheads imposed on all programs shall be kept to a minimum. It seems clear that there will always be some experimental programs whose computational requirements will equal (or exceed) the power of the available hardware, and hence that it will always be necessary to avoid implicit overheads.

One advantage of a (reasonably) optimizing compiler is that increased efficiency can be obtained with little or no effect on the program form and semantics by supplying tighter (or earlier) bindings (e.g., by replacing variables by constants).

CHALLENGE: Is there a LISP subset within which the full power of the machine is available and implicit overheads such as garbage collection and indirection on data reference are eliminated? Can the efficiency of some LISP programs be made comparable to Mesa by simply accepting earlier bindings?

>>Exportation of Programs from Environment

ISSUE: Programs developed in the EPE must not be restricted to use within the EPE. Mesa provides a mechanism (image files) for producing essentially standalone systems that carry along with them only those parts of the Mesa world on which they rely. This is important because users may not be Mesa programmers and may not be willing to pay the overhead of keeping the Mesa world in their machine. (E.g., the typical Laurel user should neither know nor care that Laurel is written in Mesa.) Among other things, this means that it must be possible for the program itself to intercept all error reports, and dispose of them appropriately.

CHALLENGE: Can LISP programs be packaged for standalone use? Is it possible to use systems written in LISP without either knowing anything about LISP or paying the overhead of the LISP system? Can LISP programs be made to process all internal and system error reports?

[Items drafted by Mitchell:]

>>Exception Handling

ISSUE: There are three situations in programs that correspond to exceptions:
(a) impossible states from which recovery is unlikely (usually an indication of a bug),
(b) low-frequency states from which the program should be able to extricate itself (e.g., difficulty in reading a page from a disk), and
(c) in interactive systems, user-initiated exceptions (e.g., kill this procedure call and the two previous and then retry).

In case (a), whatever mechanism is used to inform the programmer (assuming he is there) of the error must have the feature that it does not destroy any of the state information that he needs to investigate the bug. In particular this means that the execution stack should not automatically be rolled back as part of raising the error.

In case (b), the errors that a procedure may raise should be part of its interface specifications (they are not currently in Mesa). The program that is selected to handle the exception must be able to receive information (parameters) along with the exception notification, and it must be able to permanently acquire control in order to terminate the action. This will require rolling back the execution state (note that I am very carefully avoiding the phrase "cutting back the stack" - that is not what is needed ina multi-process environment). Such execution unwinding must be done in such a way that each outstanding activation that is to be terminated have a chance to handle the fact that it is about to die; this makes the decisions about what special actions are needed for termination a local one within the activation being killed. The order of cutback must also be reasonable; e.g., in a procedural environment, they must be done in a LIFO order, sort of like backward execution.

Case (c) raises no issues that (a) and (b) have not already proboked. It is included here only for completeness and because an integrated EPE will need to do this.

CHALLENGE: Can Lisp provide an error-handling mechanism with local handling of termination and with the state-preserving property for debugging purposes?

>>Processes and Monitors

ISSUES: Processes and monitors provide a well-engineered way for introducing parallelism in one’s programs. I doubt that they are the ultimately best way for doing so, but they represent the state of the art and there is much experience with using them and with proving properties of programs that use them.

CHALLENGE: What does Lisp have?

>>Coroutines

ISSUES: Some form of coroutines are needed to handle situations such as text formatting. The Mesa control structures are built from a common underlying primitive that allows one to call another program in a uniform manner without being aware of its control regime. The called program could be implemented as a procedure or as a coroutine (there is more than one flavor). This makes a control abstraction facility that is useful for object-oriented programming, since the user of an object can treat all operations as if they are procedures no matter what their exact implementation is.

>>Records, Parameters, Constructors, and Extractors

ISSUES: Record constructors are most frequently used as parameter records to procedures in Mesa. Keyword notation eliminates errors of ordering of two or more parameters of the same type (e.g., COPY[from: x, to: y, nWords: 256], for which I could never remember the correct order of "to" and "from" in the positional notation). Making records first-class values requires that one be able to write record-valued expressions. Programs are clearer when an entire value is specified at once, rather than by a number of assignment statements. Also, writing a constructor enables the compiler to check that one has specified some value (possibly "don't care") for each component and enables one to have default values for some componenets. This also provides a way of having default parameters for procedures as well.

Procedures need to be able to return multi-part values and not be limited to a single value returned in order to avoid some uses of "output parameters." Mesa extractors then allow the programmer to perform multiple assignments of those multi-part values to separate variables for further processing. This is heavily used in Juniper.

CHALLENGE: Can the Mesa record/parameter facilities be provided in Lisp?

>>Efficient Random Access to Files

>>Debugger Access to Source

ISSUES: One of Mesa's strong points as a system programming language is that the programmer can debug at the source level. This requires the ability to specify and set breakpoints by reference to the source program (either by context or by pointing). If the Mesa compiler were to do more global optimization (such as Fortran H's, for instance), the programmer would still need to be able to compile his program in a mode permitting the correspondence between source and object probrams to be maintained.

CHALLENGE: I presume that Lisp has these features normally. What about the block compiler?

[Items drafted by Suzuki:]

>>Readable Programs

ISSUE: It is important that one can express his algorithms clearly so they can be easily understood by others. This is a non-trivial aspect of language design. Mesa has some features that help people to write clear and understandable programs.

Mesa provides the generally accepted control and data structures in forms whose syntax corresponds well with their semantics. It follows many accepted conventions of mathematics and conventional programming languages. Infix operators, precedence of operators, and structural keywords all eliminate the need for many parentheses and reduce the parenthesis matching problems faced by LISP users.

CHALLENGE: Can LISP get rid of the parentheses and enhance readability of programs without falling into ambiguity?

>>Precise Syntax

ISSUE: One advantage of Mesa is that the syntax is rigorously defined so that when one writes a program it is known a priori whether it is acceptable or not and how it is parsed and executed.

In CLISP the syntax is tied to the error checking mechanism in order to implement language extension. Therefore, it is not easy to know in advance whether what one writes is acceptable and even accepted whether it will be parsed as the programer wishes. If the program is not accepted or misinterpreted, then it may be difficult to guess the style of writing which CLISP likes.

CHALLENGE: Can LISP be liberated from error-driven syntax? Can a precise, yet reasonably compact, syntax for acceptable programs be given?

>>Enumerated Types

ISSUE: A very useful tool in programming is the ability to create a type whose elements have programmer-defined names. One example is a parse tree in a compiler where each node belongs to one of a finite number of classes like "assignment" and "addition".

Mesa enumerated types provide this facility. They offer the following advantages:

- -efficient use of storage,
- -usable as array indices and for iteration control,
- -safe when used as array indices (no subrange overflow),
- -compact notation for subsets,
- -combined with Case selection, we can attain an efficient implementation of multiple-branch control mechanism.

CHALLENGE: Provide an equivalent facility in LISP.

>>Case Selection

ISSUES: As noted above, Case selection in conjunction with enumerated and subrange types can lead to efficient multiple branches utilizing the hardware dispatch control mechanism.

Another feature of Mesa SELECT statements is that one can write relation tails as selection criteria so that evaluation of the first operands can be saved.

CHALLENGE: Provide an equivalent selection capability in LISP.

>>Control of Storage Structures

ISSUES: Mesa makes it possible to increase storage utilization (possibly at the expense of access time) by using the PACKED option. Fields can be as small as one bit.

Mesa also enables the precise control of the layout of fields within a record for dealing with externally-defined structures (e.g., interrupt words).

CHALLENGE: Can LISP provide equivalent density and control where required?

>>Constant declaration

ISSUE: Constant declaration is a mechanism to bind a name with a value throughout the execution of a program, so that no statements can overwrite the

value. The mechanism is very useful in constructing reliable software, since this property can be checked locally.

There are two levels of constant declarations in Mesa. One is "module constant" by which a name is bound to a unique constant throughout the life of a module. One cannot even overwrite the value. The other is "local constant" by which a name is bound to a value, which may be different for each procedure invocation, but cannot be overwritten.

CHALLENGE: How can LISP ensure the security of name-constant bindings?

Inter-Office Memorandum

To	EPE Working Group	Date	September 21, 1978
From	P. Deutsch, D. Bobrow	Location	Palo Alto
Subject	Mesa -> Lisp reply	Organization	CSL

XEROX

Filed on: <Deutsch>lispreply.memo on Maxc1

This draft responds to the challenges offered in Jim Horning's memo "DRAFT Mesa -> Lisp Requirements" of 9/14/78 (including Jim Mitchell's addendum) and Peter Deutsch's memo "Unique Smalltalk capabilities" of the same date.

Premises

There are certain modifications and additions which we agree must be made to Interlisp (for the Dorado environment), and which impact several of the items in the lists below, so we will enumerate them in advance.

We agree that substantial work needs to be done on the compiler. The byte compiler is much cleaner and simpler than the PDP-10 compiler, and we believe that the additions to handle type declarations (including variant representations) and name scopes are straightforward.

We agree that the notions of interface and scope need to be made explicit in the Lisp system in the form of objects with well-defined behavior -- used by the compiler, Masterscope, the file facilities, etc. Larry Masinter believes that adding a Mesa-like notion of interface to Masterscope is straightforward. In particular, we believe that the Mesa arrangement of separating interface declarations from both client and implementor should be supported. Our current idea for supporting both these notions and fluid variables (see next paragraph) is to introduce an object called a dictionary, which associates interpretations with names in a given scope. (We have a number of design memos that expand on this idea.)

We believe that it is appropriate to replace the current Lisp notion of free variables with a combination of lexical scoping and declarations for fluid variables (variables dynamically bound by name as at present). The MIT SCHEME dialect of Lisp has done this successfully, and an interesting and efficient compiler for this dialect exists. Our present implementation can support this change, but at great cost in runtime efficiency. SCHEME shows one way to improve the efficiency at the cost of a fairly radical implementation change; we do not yet know exactly how we will want to change our implementation.

Both for lexical scoping and for other reasons, we believe we need to provide an efficient mechanism for FUNARGs, or, more generally, for <code, partial environment> objects which can be applied as functions. Such objects are generalizations of current Mesa procedure objects.

We agree it is necessary to support a precisely defined infix syntax in a clean way. Our intent is to provide some general-purpose parsing and unparsing (prettyprinting) mechanisms to handle the syntax, and to define all our semantics in terms of extensions to S-expressions. Depending on how much non-local declarative information is needed for the translation, we may defer translation until compilation, until the user requests it, or until the first attempt to execute the function, but in any case the entire function will be translated at once in a precisely defined way that does not depend on

the dynamic environment.

Horning's items

>>Static checking (for type errors and undeclared names)

Lisp already has a type calculus approximately as rich as that of Mesa (the type system done by Kaplan and Sheil for DECLTRAN), including an S-expression syntax for declaring argument, local variable, and result types. Currently checking is only done within a single function. Adding cross-function checking to Masterscope should be straightforward logically.

Masterscope already has the capability to check for undefined names, both undefined functions and variables which might not be bound.

>>Static scoping

See "Premises" above.

>>Encapsulation

There are some hard issues here -- Lisp doesn't lend itself easily to the kind of total encapsulation that Mesa does. To the extent that lexical scoping conceals internal names, we can do a reasonable job of keeping clients out of implementations. Linked function calls are an existing mechanism for removing external access to local functions.

Multiple instances of modules can be accomplished using FUNARGs. See "Premises" above.

For object-oriented programming, see the discussion below under the Smalltalk list.

>>Explicit interfaces

See "Premises" above.

>>Consistent version checking

We can use the same time stamp scheme that Mesa does.

>>Efficiency

Given the existence of the type system, including the (already existing) provisions for packed records, the compiler can potentially do as well as Mesa for manipulating scalar data. (There are some issues here about instruction set optimization, however.) We believe that a combination of compiler analysis and user declarations can make possible the Mesa-style stack allocation of composite objects with direct addressing, without running the risk of dangling references; we recognize this is not a trivial problem, however.

>>Exportation of programs from environment

We believe that this issue is partly a red herring: after all, even a Mesa.Image file contains copies of all the system modules needed by the program. The irreducible minimum amount of "system" for a Lisp program is about the same as that for Mesa, except for the garbage collector. Given that Masterscope can take a fully bound Lisp program and find all the functions that are callable in any way, we can certainly produce a virtual memory image file that contains only the necessary functions (in fact, we should be able to do better than Mesa since we don't need to take entire "modules").

System overhead during execution is essentially limited to garbage collection and other storage allocation functions. The arguments under Efficiency above suggest that it is possible to incur this overhead relatively rarely, and of course if a program does no dynamic allocation, even the garbage collector (a Lisp program) need not be included in the image file.

Lisp programs can currently intercept all software errors including dynamic type errors, array bounds errors, arithmetic overflow, etc. Lisp programs are not currently able to intercept and recover from hardware errors or running out of storage. Mechanisms to preallocate storage and to pin code and data into real memory, which seem necessary to take care of these, could be added to Lisp with no fundamental change.

Suzuki's items

All of these items are essentially syntactic. All are either already present or very easy to add.

>>Readable programs; precise syntax

See "Premises" above.

>>Enumerated types

The DECLTRAN type system already has this at a semantic level; extending it to the representation level should be easy.

>>Case selection

This is a modest generalization of the present SELECTQ function.

>>Control of storage structures

Lisp already allows tightly packed records. Adding explicit user control over the layout is trivial.

>>Constant declaration

The compiler already provides for compile-time constants. Local constants (not changeable after initial binding) are trivial to add.

Mitchell's items

>>Exception handling

Lisp already funnels all exceptional conditions through a single point (the ERRORX function). These conditions look like a normal function call, so no state is lost. ERRORX can easily be changed to examine the execution environment and call cleanup handlers associated with intermediate frames, in any desired environment (the dynamic environment of the error and/or the environment of the intermediate frame).

>>Processes and monitors

Lisp currently has no process or monitor machinery. Processes are easy to implement with the same mechanism as coroutines (see below). Monitors can be done the same way as in Mesa, provided that object-oriented programming is enforced (see the discussion under Smalltalk below).

>>Coroutines

Lisp currently has a notion of an "environment descriptor" which is essentially an indirect pointer to a stack frame. Functions exist which provide coroutines using a single environment descriptor that gets switched between the participants. A more general coroutine mechanism, that allowed coroutines to look like function calls and returns, could easily be built, and made efficient with a little microcode assistance (similar to the microcode assistance we will want for FUNARGs).

>>Records, parameters, constructors, and extractors

Records already exist. Extractors are a pure syntactic extension. Keyword notation already exists for constructors; adding it for procedure parameters is fairly straightforward, even if no declaration exists for the callee (the names must be matched up at link or call time in that case). Providing multiple return parameters (as opposed to a heap-allocated return record) requires some tinkering with the instruction set.

>>Efficient random access to files

There is no reason why Lisp and Mesa should be any different in this respect -- in fact, it is planned to provide microcode support in Lisp for I/O streams.

>>Debugger access to source

Lisp already provides for access to the source program. There is no current provision for putting a breakpoint in the middle of compiled code: internal breakpoints cause the function to revert to interpreted form. Converting a PC back to a location within the source can be difficult, because Lisp functions often have no statement structure in the Mesa sense (when they do, we can easily produce something like the Mesa fine-grain table): for a non-optimizing compiler we could do it by repeating the compilation and stopping when the PC matched.

Currently, the debugger cannot show frames (either function names or variable values) within a compiled block. It is logically possible to retrieve this information, although about as messy as for the Mesa debugger.

Smalltalk items

>>Simple, extensible, macro-expandable, infix syntax

See "Premises" for syntax. Once we get rid of CLISP and introduce a single uniform way of dealing with non-standard S-expression syntax (like SELECTQ, PRINTOUT, etc.), we will be doing as well as Smalltalk. We believe it is extremely important to do away with the current notion of compiler macros, replacing it by a scheme in which one gets the choice between closed calls, open calls (using the *same* definition), and macro-expansion (an arbitrary user-supplied function, the same for interpretation and compilation, computes the form to be executed).

>>Object-oriented programming (operation names local to each type, all operations generic, no need to declare types, instance state automatically available)

Generic operations, decoded according to the receiver type, can certainly be provided in Lisp now, at considerable execution cost. The real issue is how efficient various degrees of binding must be. The current Lisp approach is to introduce one level of indirection for function names in the normal case, removable by declaration (linked calls), and no discrimination on the receiver type. If an operation is declared generic, we can certainly provide microcode assistance for getting to the right piece of code.

Making the instance state automatically available comes for free with an efficient FUNARG mechanism. However, we believe that the right approach is to go down the Mesa path of trying to eliminate the distinction between variable binding records and other kinds of records. We do not yet know what this means in terms of implementation structures; at the S-expression level, it

certainly means the ability to "open" records for direct addressing exactly as in Mesa. This "open" for instance variables can be inserted automatically when one uses an object-oriented interface.

The scope restrictions associated with object-oriented programming have already been covered under Horning's comments. Private record declarations are the key to this. We do not know exactly how we will make it hard enough for ordinary clients to penetrate an implementation, while making it easy enough to examine (and modify?) state with the debugger.

A more serious question is whether there are benefits from the object-oriented style that one gains only if nearly all programming is done that way. We do not propose to rewrite the Lisp system in object-oriented style, although we think that most new packages will be written this way, and some incremental rewriting will probably occur.

>>Inheritance of behavior through subclass mechanism

We think this can be provided through the dictionary mechanism mentioned under "Premises".

>>No distinction between user- and system-defined types

To the extent that programs are written in object-oriented style, there is no reason why the mechanism for handling operations on integers, say, should be different from that for user-defined types. There is, however, a serious question as to what the default behavior of the compiler should be: should a call to an undeclared function (i.e. a function for which there is no corresponding interface declaration) be treated as generic or not? Addition is presumably an example of such a function, if the argument types are undeclared. Making addition work properly for integers, long integers, floating point numbers, rationals, algebraic expressions, etc. requires generic treatment. We do not have a good answer for this.

>>Future directions: template-style programming, constraints, data base links in text objects

Given Lisp's hospitality to programs that construct and manipulate programs, template-style or constraint-oriented programming should be even easier to implement than in Smalltalk. Data base links in text objects (such as source code) require careful planning of the interface to the text manipulation system, but again Lisp has a better package for this purpose (TXDT) than either Mesa or Smalltalk.

Tools

>>Browser

The Smalltalk Browser depends on having all the system sources and documentation available on-line, and on the 3-tiered classification system (class category, class, method category). Class categories correspond essentially to chapters in the Lisp manual. Classes correspond roughly to the present Lisp "packages"; object-oriented organization of packages would help firm this up. Method categories do not exist and would have to be added (probably as comments in a standard form). The facilities of the Browser are simply another program, probably quite easy to write using DLISP. Note that Masterscope, with its more extended idea of "relatedness", could potentially enable a much more powerful browsing facility.

>>Integrated on-line documentation

HELPSYS is the best on-line reference documentation system available anywhere that we know of. Its drawback is that maintenance of the documentation is a tedious, non-incremental process. It will also need to be re-implemented for the Dorado.

>>Immediate incremental compilation

A very fast, simple, non-optimizing version of the byte compiler already exists. Our first approach will be to divide up the real compiler in a way that reduces the amount of optimization without sacrificing capability, and then explore other alternatives if this turns out to be too slow.

Packages

>>Arbitrary-precision integers with smooth transitions

This can be done as a package, although it requires some modifications to the error handling machinery to field overflow properly, and potentially complicates in-line compilation of integer arithmetic.

>>Paragraph formatting (display and Press)

The existing TXDT package provides most of this for files, and DLISP provides it for the display. Press output is a modest-sized package.

>>Bitmap and display coordinate objects

DLISP provides some facilities for manipulating these objects -- unclear how much, since documentation doesn't exist yet.

>>Paragraph and picture editor

DLISP has some kind of paragraph editor. A picture editor will have to wait for a redesign of the display facilities in the personal-computer environment.

>>Windows

DLISP has them.

>>Menus

DLISP has them.

Other

>>Non-preemptive window behavior

The DLISP scheduler is not documented yet. It clearly has some ability to keep track of several windows all waiting for input, but the cross-window selection capability suggests that it requires more cooperation between windows than Smalltalk does.

Inter-Office Memorandum

To	PE-Interest	Date	October 9, 1978
From	Lampson, Levin	Location	CSL, Palo Alto
Subject	Mesa Response to Lisp Challenge	File	[Ivy]<Lampson>mesaresponse.memo

XEROX

We culled five major (read "potentially difficult to implement") facilities from the memos by Bobrow and Masinter:

1) Programs as data
2) Variable syntax analysis
3) Dynamic variable binding
4) Safe language (subset)
5) Automatic storage allocation

This memo summarizes the analysis presented at the September 21st meeting.

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. This seems straightforward in principle, though we haven't looked at the details. We think that a representation other than parse trees should be explicitly defined, permitting the representation of parse trees to change as the compiler/language evolve.

Atoms. The symbol tables built by the Mesa compiler currently include a mapping between source identifiers and internal unique ids (atoms). At present, these mappings exist only on a per-compilation unit basis, implying that they will need to be merged into a global map when the compilation unit "enters the environment". A similar merge presently occurs inside the compiler when the source program accesses an included module (i.e., the symbol table for that module is opened and the identifiers (used by the source program) are remapped into the current atom table). A merge into a global atom table therefore seems feasible, at least in principle.

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. It seems possible to extend the techniques used to ensure uniqueness of record types to accommodate general types. Run-time efficiency is vital, so the check that two type IDs are equivalent must be fast. We anticipate using the LISP "spacing" technique, which places all data structures of the same type within a well-defined region of the address space (e.g., each page allocated for such dynamic structures can hold objects of precisely one type). This scheme avoids keeping any type ID bits with the pointer, but makes it difficult to point inside a composite data structure with a universal pointer. Two possible remedies have been suggested: (1) an explicit type identifier immediately preceding the (sub)structure, or (2) an indirection mechanism in the data structure itself. In any event, access to statically allocated variables (i.e., those in local and global frames) via universal pointers will have to be prohibited, unless special steps are taken to prevent problems with garbage collection.

2) Variable 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 a piece of program in S-expression form. 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.

3) Dynamic variable binding.

Mesa, at present, can only support 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. Such a table could reside in the code segment, with a bit in the associated global frame(s) to indicate its existence. 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. We will also consider a relaxation of this condition in which certain *regions* of the storage may not be safe in this sense; as long as values stored in such unsafe regions do not themselves contain pointers to safe regions, the damage which can be caused is confined to the unsafe regions.

There are three ways in which a pointer variable *p* may acquire an improper value

1) Lack of initialization - in this case the value of *p* is just the random bit pattern which happens to be in the storage occupied by *p* when it is instantiated.

2) Incorrect computation - in general a pointer value can only be the value of some pointer variable (which could of course be a record component) or the result returned by a storage allocation. If pointer values can be computed in other ways, e.g. with LOOPHOLEs or their equivalents, or by arithmetic, then there is no guarantee that such values point to storage containing values of the proper type.

3) Premature deallocation - in this case *p* is pointing at a region of storage which once contained a value of type T, but is now being used for some other purpose.

If none of these bad things can happen, then induction tells us that the language is safe with respect to pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer. Finally, array descriptors require the same treatment as pointers.

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 did I forget

Preventing premature deallocation has two aspects: dynamic (explicitly allocated) variables and stack variables. For the latter, a simple and reasonable rule seems to be to disallow the use of @ on local variables. As partial compensation, it seems desirable to introduce **var** parameters (call by reference); these can be thought of as pointers to stack variables which are *always* dereferenced, so that it is not possible for the called procedure to make a copy of the pointer which will outlive the call. We also need to enforce a restriction that **var** parameters are not permitted across process and coroutine calls, since in those cases there is no guarantee that the caller will live longer than the called procedure (again, this seems too strong, but it isn't clear how to weaken it).

For dynamic variables some support from an automatic deallocation mechanism seems essential, since otherwise there is no way to tell when the programmer says Free(*p*) whether other pointers to *p*^ still exist. We see three possibilities

a) Reference counting. This could be added to Mesa easily enough, but the cost during execution is rather high, and it is probably not a good idea in view of the other alternatives.

b) Deferred freeing. The idea is to make a list of all the storage freed by the programmer, and at suitable intervals to invoke the scanning phase of a garbage collector and check that there are no outstanding references to the storage queued to be freed. At the end of the scan any storage which has passed this check can be actually freed.

c) Automatic deallocation, discussed below.

Quite possibly deferred freeing will be worthwhile, since it is fairly easy to implement and has substantially lower running costs than fully automatic deallocation. With plenty of address space there should be no serious objection to a long interval between the programmer's call of Free and actual availability of the storage.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Relative pointers have several appealing properties, one of which is particularly relevant in this context

a) They allow an entire complex structure to be moved in storage, or between storage and a file, as long as all its contained pointers are relative to the structure.

b) They can save substantial amounts of space, especially as the address space grows and pointers become longer.

c) 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 relation 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 age of the region 2^n; an especially nice value for *n* is 16.

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.

Inter-Office Memorandum

To	EPE working group	Date	October 26, 1978
From	P. Deutsch	Location	Palo Alto
Subject	XLisp/EPE plan	Organization	CSL

XEROX

Filed on: <Deutsch>xlispplan1.memo on Maxc1

As requested by the EPE working group, this memo contains a gross (~5 item) plan for how we would get from where we are to an EPE XLisp, including preconditions, and a statement of how we would use 'warm bodies' to help us get there.

Since the overall XLisp design has only been presented orally, there may be steps in the plan whose meaning is not clear. A memo is in preparation which summarizes the relevant features of the design.

Plan

XLisp can evolve from our current systems in a series of steps, with a working (although not stable) system at each step. Here are the major milestones:

1) Get Alto Lisp running on the Dorado. Major work on XLisp is not possible on any other hardware -- the Alto is too slow and Maxc is too small. In particular, we need a Dorado Lisp that uses the IFU and hardware map, and all of the Dorado's real memory. This almost certainly means converting the kernel from Bcpl to Mesa. These issues are addressed in separate memos.

The next two steps can be done in either order, but not in parallel.

2a) Change the system to use shallow binding and have invisible indirection for variables. These are necessary for the lexical scoping rules and closure capability of XLisp. This has some impact on the compiler and a major impact on the functions that handle the stack.

2b) Add dictionaries to the system, but not lexically-scoped closures. This replaces the LOCALVARS and LINKFNS capabilities of the current system. It has major impact on the compiler, interpreter, and record and file packages, and minor impact in many places such as the editor and stack functions. Changing the way macros and special syntactic forms are handled can be done later, but we want to eliminate error-driven interpretation early. This strongly affects CLisp and user-defined macros and NLAMBDAs.

3) Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances.

4) Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking. A good deal of this is in the Decl package already.

5) Implement compile-time type checking. This affects the compiler, interpreter, and in general any function that manipulates function definitions (e.g. GETD/PUTD, APPLY*).

6) Implement variant representations. This has major impact on the instruction set (and therefore the compiler), and on all functions that manipulate records or the stack.

The following steps are necessary eventually, but do not affect the basic functional capabilities of the system:

7) Implement an infix front-end. This requires writing (or stealing) a text-oriented editor and a prettyprinter, and interfacing them to the Lisp editor and file package.

8) Implement permanent pointers. This involves completely rewriting the file package.

Using volunteers

Our ability to use volunteers depends somewhat on the ordering of the above steps, and on the qualifications of the volunteers. In particular, doing step 7 early might mean we could get some volunteers who only really felt at home in Mesa syntax. Here is how we could use people to help with each step:

1) Converting the Alto Lisp kernel to Mesa is mostly a transcription job, and could be done by anyone who knows both Bcpl and Mesa with relatively little briefing required. The other people-resource issues in Alto Lisp are discussed elsewhere.

2a) The shallow binding conversion requires people familiar with Lisp, with Lisp implementation, and with the Alto Lisp internal data structures. 'Warm bodies' are unlikely to be useful.

2b) Adding dictionaries is a tricky job; much of the work on the compiler and record package could be done by anyone familiar with Lisp, but much of it could only be done by the implementors of the programs in question.

3) Lexically scoped closures are a small task if done in a straightforward way. Any PARC Lisp programmer could do it. We know from SCHEME (and even from GEDANKEN) that Mesa-style instances can be built on top of lexically closed FUNARGs.

4) Implementing the type calculus requires understanding it, but little else. Whether we build on existing Decl code depends mostly on who does the work.

5) Implementing pre-execution type checking requires access to global knowledge of the Masterscope variety. Getting it done right probably requires an experienced person with an ongoing interest in type systems and Masterscope-like issues.

6) Variant representations are also tricky enough to require an experienced and dedicated person.

7) Anyone can write a decent parser and prettyprinter, and an adequate editor.

8) Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project. The basic design, and an implementation of the lowest level operations for using them, can be done by anyone experienced.

XLisp: a Lisp answer to the EPE language challenge

- What are our goals?

- What technical methods will we use to achieve those goals?

- How can we achieve the goal in a step-by-step way?

- What do we need (in time and resources) to put this plan into effect?

Goal: upgrade (Dorado, Inter-)Lisp to include important EPE attributes as listed in the EPE report

We need to retain:

- **Totally automatic storage management**
- **Easy program manipulation of programs**
- **"Instant" turnaround for program changes**
- **Lists and symbols as primary objects**
- **Runtime type system**

We aim to add:

- **Type system with static checking capability**
- **Abstractable interfaces**
- **Improved runtime efficiency**
- **Scope-oriented protection**
- **Consistent compilation, version bookkeeping**
- **Uniform notion of calling, instantiating, closures**

Method:

Define system entirely in terms of dynamic objects -- name scopes, types, etc. are objects

Static checking & optimization come through added declarations

Dictionaries for name interpretation -- unifies notions of argument list, record, interface

Maps names to attributes such as type (abstract & representation), protection, manifest value, etc.

Protection is associated jointly with the dictionary and the path from the accessor

Dictionaries describe how to create instances of records

Calling = instantiating (the frame) + copying (the arguments) + running

Closure formation = instantiating + copying

Coroutine transfer = copying + running

Type system includes ideas of record formation (via dictionaries), specialization by adding attributes, extension (subclassing), multiple representations

Coercion between representations (e.g. tagged vs. untagged) or degrees of specialization is automatic

Checking is done with EQ (identity)

Permanent pointers extend the EQ concept to the file system

Plan:

1) **** Get Alto Lisp running on the Dorado**
(more about this next week!)

2) *** Add dictionaries for scoping only, not generalized closures**

3) * Add generalized closures**
(at least one implementation of this already exists)

4) ** Add a type calculus, but do all checking at run time
(existing Decl package does a lot of this)

5) ** Add compile-time type checking**

6) * Add variant representations (e.g. unboxed numbers)**

7) * Add an infix (Mesa-like) front end

8) ?? Implement permanent pointers

Resources:

Think of * = 3 person months

All but (7) and perhaps (8) require experienced people who understand the system implementation

We think critical mass is 4 essentially full-time professionals

Providing Lisp strengths in Mesa

There are two levels

Upper: Integrated editor
Masterscope
Programmer's assistant

Lower: Programs as data
Variable syntax analysis
Dynamic variable binding
Safe language
Automatic storage deallocation

- **Programs as data**
  - **Atoms**
  - **Standard S-expression representation**
  - **Universal pointers**
    - **Unique type IDs**
    - **"Spacing"**
    - **Discrimination by CASE or by assignment**
- **Variable syntax analysis**
  - **At READ (scanner) time (read tables)**
  - **At compile time (syntax macros)**
  - **At execution time -- not in Mesa**

- **Dynamic binding**
  - **Needs a table which maps**
    - **VariableID => (location in frame, type)**
  - **Mesa already does this for signals**

Safe language

- Pointers into regions of storage which contain system data structures or other such pointers must be handled correctly.
- There are three problem areas:
  - Initialization
  - Computation -- no LOOPHOLE, UNSPECIFIED, computed or overlaid variants involving pointers, or pointer arithmetic; bounds checks for array references
  - Dangling reference -- forbid deallocation
- Relative pointers to data which doesn’t contain absolute pointers need not be safe

Automatic deallocation

Use the Deutsch-Bobrow incremental collector, as in Alto/Dorado Lisp

Symbol table information

- Atoms -- print name <==> internal ID
- Name => (location, type) map for frames
- Unique type identifiers
- Type ==> (location, type) map for pointer-containing types

Inter-Office Memorandum

To	EPE working group	Date	November 7, 1978
From	P. Deutsch	Location	Palo Alto
Subject	XLisp/EPE plan, revised	Organization	CSL

XEROX

Filed on: <Deutsch>xlispplan2.memo on Maxc1

As requested by the EPE working group, this memo contains a copy of the 8-step plan presented in last week's Dealer for how we would get from where we are to an EPE XLisp, including preconditions, time estimates, and a statement of what we would have at each step. This memo supersedes the previous memo <Deutsch>xlispplan1.memo of October 26.

Plan

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 I believe this order leads to the least total work. I also believe little parallelism is possible *between* any two of the steps, although, as my 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.

The total of the figures below is: 87 work months, 27 1/2 elapsed months.

1) Dorado Interlisp
Work months: 18
Elapsed months: 6?

People: Bobrow, Deutsch, Haugeland, Teitelman; Birrell, Fikes, Goldstein, Kaplan, M. Kay; Laaser, Masinter, Thompson

Get Alto Lisp running on the Dorado. Major work on XLisp is not possible on any other hardware -- the Alto is too slow and Maxc is too small. In particular, we need a Dorado Lisp that uses the hardware map, and all of the Dorado's real memory. Danny Bobrow has written a detailed memo on the subject of getting a Dorado Lisp up, including assignment of people to subtasks.

After step 1 we have a usable Interlisp system running on the Dorado.

1') Shallow binding
Work months: 3
Elapsed months: 1

People: Bobrow (stack fns & interpreter), Deutsch (compiler & loader), Haugeland (microcode)

Change the system to use shallow binding and have invisible indirection for variables. In addition to performance improvements, these changes help support the lexical scoping rules and closure capability of XLisp. This has some impact on the compiler and a major impact on the functions that handle the stack.

Step 1' produces no externally visible changes in the system, except that GLOBALVAR declarations are no longer necessary.

This step can actually be done at any time; it is easiest to do here, when less code will need to be changed. It requires people familiar with Lisp, with Lisp implementation, and with the Alto Lisp internal data structures.

Step 1' may not be necessary at all if microcode support for dynamic binding proves fast enough. I am inclined to think this will not be the case, especially if GLOBALVAR declarations are eliminated.

1") Clean up macros
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

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. This strongly affects CLisp and user-defined macros and NLAMBDAs. Relatively small changes are also required in the compiler, interpreter, and Masterscope.

The current MACRO scheme does not fit into the scoped view of the world, and has caused many obscure bugs; NLAMBDAs make separate MACRO definitions necessary, and require internal EVALs, which we would like to eliminate since they will require elaborate run-time checking. We can arrange things so that after step 1" 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.

This too could be done later, but it makes things a lot simpler if the system doesn't have to support run-time DWIM correction while dictionaries are being installed. Any good Lisp programmer can go over all the system code to change NLAMBDAs and macros to conform to the XLisp scheme; only Warren can make the necessary simplifications in DWIM and CLisp.

2) Dictionaries
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

Add dictionaries to the system, but not lexically-scoped closures. This replaces the LOCALVARS and LINKFNS capabilities of the current system. It has major impact on the compiler, interpreter, and record and file packages, and minor impact in many places such

as the editor and stack functions. The hardest part of this step is probably designing the proper interfaces to dictionaries-as-virtual-objects.

After step 2 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.

Dictionaries are the largest single change to the system, requiring substantial design work. Some of the work on the compiler and record package could be done by anyone familiar with Lisp, but most of it could only be done by the implementors of the programs in question.

Steps 2 and 3 can be done in either order, but the presence of dictionaries, which guarantee that most names are only known within a limited lexical scope, makes it possible to do much better optimization on closures. See Guy Steele's thesis on RABBIT (a compiler for SCHEME, a lexically scoped Lisp dialect) for details.

3) Closures

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)

Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances. This affects all programs that deal with the representation of dynamic state, such as the stack functions and the BREAK package.

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.

Careful design is needed here to produce reasonably efficient results. We will probably want to change the instruction set at some point to make record access faster. Implementation wizards, including someone who really understands the Mesa and Smalltalk way of doing things, are needed for this step.

4) Type calculus

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

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

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

Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking. A good deal of this is in the Decl package already.

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 step 5.

Getting the type calculus right will require help from people with particular insight in this area such as Horning and Scott. Meshing the type system with the compiler, interpreter, and Masterscope is not believed to be a very big job, given the Decl experience.

Note that in the absence of permanent pointers (step 8), some method involving unique type IDs recorded in compiled code files will be required to do checking properly, as in Mesa. Modulo this question and the related question of protection, most of step 4 could be done in any order relative to steps 2 and 3.

5) Compile-time type checking
Work months: 12
Elapsed months: 3

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

Implement compile-time type checking. This affects the compiler, interpreter, and in general any function that manipulates function definitions (e.g. GETD/PUTD, APPLY*).

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.

Getting pre-execution type checking done right probably requires an experienced person with an ongoing interest in type systems and Masterscope-like issues. A hard issue we have not examined is just how simple we can make the part of the checking system that must be absolutely correct: for example, it would be nice if it didn't need to include all of Masterscope.

6) Variant representations
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)

Implement variant representations. This has major impact on the instruction set (and therefore the compiler), and on all functions that manipulate records or the stack.

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.

This step, like the previous one, requires someone with compiler experience and careful scrutiny to see how well we can isolate the code that must be absolutely correct.

The remaining steps are necessary eventually, but do not affect the basic functional capabilities of the system.

7) Infix front end
Work months: 3

Elapsed months: 1

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

Implement an infix front end. This requires writing (or stealing) a text-oriented editor and a prettyprinter, and interfacing them to the Lisp editor and file package.

Anyone can write a decent parser and prettyprinter, and an adequate program editor. This step could be done much earlier, and might help us get people who like Mesa syntax to work on the system.

8) Permanent pointers
Work months: 6?
Elapsed months: 2?

People (design): Deutsch, Horning, Sturgis

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

Implement permanent pointers. This involves completely rewriting the file package.

Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project. The basic design, and an implementation of the lowest level operations for using them, needs someone who understands the considerations that go into file systems like Juniper.

Inter-Office Memorandum

DRAFT - DRAFT - DRAFT - DRAFT

To	PE Group	Date	November 9, 1978 1:09 PM
From	Deutsch, Bobrow	Location	Palo Alto
Subject	XLisp design -- a detailed response to the Mesa challenges	Organization	CSL

XEROX

Filed on: <Deutsch>xlisp1.memo on Maxc1

****** Note: this is a document-in-progress. It is not stored in the PE file on <Deutsch>pe2>. Always check the date and time in the "Date" field above to make sure you have the latest version. ******

Introduction

This document describes part of the design for a system we are calling XLisp. It is a dialect of Lisp, based largely on SCHEME (a lexically-scoped Lisp dialect developed at MIT), and adding many capabilities derived from Mesa and other modern statically typed languages. The goals of XLisp are roughly the following:

> Keep the current ability of Lisp to manipulate programs and to carry out computations with no type or scope declarations required, at approximately current efficiency.
>
> Add the ability to achieve greater efficiency and statically checkable protection, by adding declarations.
>
> Develop 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.

The purpose of this document is not to exhibit a design that we think we could start to implement tomorrow: a lot of details are missing. Our purpose is rather:

> To demonstrate to the Mesa-oriented community that we really know how to achieve the valuable capabilities of Mesa within Lisp, and that our view is not only implementable but conceptually clean (hopefully, cleaner than current Mesa).
>
> To persuade the Lisp-oriented community that the changes they will have to make in the way they do things are a small price to pay for the new power they will obtain through this design.

This document needs, but currently lacks, a tremendous amount of motivational and explanatory material (including examples). One kind of material is needed for Lisp-oriented readers to answer the question, "Why is this changed?" Another kind is needed for Mesa-oriented readers to answer the question, "What Mesa challenge does this respond to?" In this edition of the design, because of time pressure, we have opted for a presentation of the end result with little indication of how we got there.

Organization of this document

This document is not organized properly yet. The right order of presentation is probably:

Types excluding records

Record types and their dictionaries

Interface types and their dictionaries

Function objects and environments

Miscellaneous

Types

****** This section is still confused. I- and D- types have come and gone at least twice. There is a section on declarations, assertions, and invariants near the end of the memo which indicates some of the source of the confusion. ******

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 using the following operators (the names and Mesa-like syntax are for expository purposes only):

ANY[] is a type which all objects belong to.

FLOAT[] is a type for floating point numbers of one particular precision and range defined by the "hardware".

ATOM[] is a type for atoms.

INTEGER[i1,i2:IntegerOrNIL] is a type for objects which can take on integer values from i1 through i2 inclusive. NIL means there is no limit on that end.

CONSTANT[x] is a type for objects which can take on only the value x. This is not as useless as it sounds: for example, NIL occupies a special place in the type space.

SATISFIES[p:FUNCTION[[ANY[]],[Boolean]],t:Type] 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.)

Structure types are discussed in a following subsection.

[How do variant R-types get defined??]

Declarations

An S-type declaration just establishes an invariant for a variable -- checked whenever the variable is stored into, can be assumed whenever the variable is used. S-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.)

Structure types

The other kind of definable S-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.

****** Fields require an instance. Smalltalk operations require an instance. Interfaces don't necessarily require an instance. How to resolve this best? Mention "fetch", "replace", "perform"

syntax for things that need an instance.

Conformity and coercions

Type checking depends only on the S-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 S-type and R-type levels. S-type coercions up or down the type lattice are mostly either no-ops or type checks, and can be supplied automatically. Other S-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.

Here are the coercions available among S-types. The notation t1 -> t2 means that an attempt is being made to convert a quantity U of type t1 to a quantity V of type t2.

t -> ANY[]: no checking required
ANY[] -> t: requires a type check

FLOAT[] -> INTEGER[i1,i2]: may require a range check
INTEGER[i1,i2] -> FLOAT[]: always allowed

CONSTANT[x] -> t: requires checking that TYPEP[x,t]
t -> CONSTANT[x]: requires checking EQUAL[U,x]

INTEGER[i1,i2] -> INTEGER[j1,j2]:
If i2<j1 or i1>j2, not allowed.
If j1<=i1 and j2>=i2, allowed with no check required.
Otherwise requires a range check on U.

SATISFIES[p,t] -> t': use algorithm for t -> t'
t -> SATISFIES[p,t']: p[U] must be true, and t -> t' allowed

ANYOF[t1 ... tn] -> t: for some j, must have TYPEP[U,j] and tj -> t allowed
t -> ANYOF[t1 ... tn]: for some j, must have t -> tj allowed

Functions, closures, bindings, and environments

Function and environment objects

Current Interlisp encourages the user to confuse function names, list structures representing functions, and function objects, but does not treat them uniformly: for example, the first argument of APPLY* may be a function name or a list structure, while the second argument of PUTD may be a list structure or a compiled or SUBR function object.

In XLisp we distinguish between code objects, which are merely different forms of program text, and environment objects, which may contain values for some or all of the variables of the text. There are three kinds of code objects: primitive (representing SUBRs and microcoded functions), compiled, and interpreted functions. Conceptually, every code object contains a pointer back to the naming environment (called "scope" below) in which it is embedded: an interpreted code object is thus not simply an S-expression, but rather a pair <S-expression, scope>.

In XLisp the only thing that can be executed is an environment object, or EO: EOs, and only EOs, are legal as function definitions. An EO is either a code object or (recursively) an EO plus bindings for some of its unbound variables. The state of an EO logically always contains two components:

What kind of data the environment expects when control next arrives: none, a particular

argument list, or an operation id and associated arguments. Note in particular that the continuation, i.e. the identity of the calling EO, is an argument.

Whether a new instance of something needs to be created when control arrives. (A new instance is always created if the environment expects an <operation, arguments> pair.)

Naturally, when an EO transfers control outside itself, it specifies what kind of data it is passing to the new EO: none, an argument list, or <operation, arguments>. If the sender does not pass itself as the continuation, then (in effect) it has lost one reference to itself, and if there are no more references it can be freed. An example of this is a function return.

A traditional Lisp function is an EO which expects an argument list and creates a frame to hold it. A closure (more on these below) is an EO which already has one or more frames as part of it. A Smalltalk instance or Mesa module instance is an EO which expects <operation, arguments> and creates an appropriate frame.

Some kinds of EO cause errors if an attempt is made to execute them with any of their variables unbound. For EOs that do allow such execution, then an error occurs if an attempt is made to actually reference an unbound variable. Clearly we will want several different representations of EOs depending on how much and what kind of state information they include, corresponding to different kinds of functions, closures, and environments.

Closures

****** Need to talk here about local and fluid variables; how interfaces get hooked up; various kinds of instantiation (Mesa & Smalltalk modes). How VARID types with the extra level of indirection do SIGNAL binding and Lisp dynamic variables.

Given the existence of fluid variables, there are two different kinds of state one might wish to capture in a closure: the current bindings of the lexically enclosing variables, and the current fluid binding environment. An XLisp closure always captures the first of these (but only the bindings for those variables actually used free within the function being closed -- this is determinable statically); optionally, the programmer can specify whether the fluid binding environment is to be captured as well. (Individual fluid bindings can be captured by rebinding them to local variables, and then rebinding the fluid variables to the captured values inside the function -- of course this can be done with a macro.)

Syntax

In current Interlisp, LAMBDA and NLAMBDA play an anomalous role as syntactic markers, different from any other syntactic context. XLisp follows the SCHEME convention: NLAMBDA does not exist, and LAMBDA is a *function* which takes an argument list and a body and returns a function object. This leads to few changes from the user's point of view: the main one is that the standard representation for a functional argument becomes simply (LAMBDA --), although (FUNCTION (LAMBDA --)) can still be acceptable by making FUNCTION a macro.

The form (NLAMBDA var . forms) can be converted to a MACRO with the definition

```
(LAMBDA (FORM)
  (SUBPAIR 'REST (CDR FORM)
    '((LAMBDA (var) . forms) (QUOTE REST)) )),
```

and similarly for NLAMBDAs which spread their arguments, but this leads to very inefficient execution. There is no way to duplicate Interlisp's handling of NLAMBDAs exactly, since MACROs are not functions and cannot be APPLY'ed or APPLY*'ed. We see nothing wrong with this.

****** Need to do something about multiple return values. Maybe (RESULTS form var-for-val2 ... var-for-valn)?

Protection

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).

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.

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 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. For programs like editors that must be able to make changes in dictionaries, we provide a way to make a less-protected (but non-EQ) copy of a dictionary, and an operation that replaces the entire contents of a dictionary with that of another, rejecting it if it does not satisfy appropriate protection requirements. This operation also decides how much current state is invalidated by the change.

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.

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 (currently, only "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 (litatom) and an associated property list. We have already discussed dictionaries above -- they are what defines a structure type. Every function has an associated dictionary whose fields are the function's arguments (functions of an arbitrary number of arguments have numbered, rather than named, fields).

****** What's on the property list?

Names vs. atoms

In current Interlisp, global values and function definitions are associated with litatoms. In XLisp, they are associated with <name, dictionary> pairs, where the name is a litatom. For fluid variables known globally, the distinction can be ignored, but the functions to manipulate values and definitions (GETD, GETATOMVAL, etc.) will all be redefined or extended to take an environment (scope plus execution environment) as argument.

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.

Scopes are built up and modified in four ways: by declarative information that is part of the program text; as a side effect of certain program constructs such as LAMBDA and PROG; by editing or otherwise operating on the program; and by direct calls on the functions that operate on the representation of scopes. [How do we confine the last of these to safe uses??] Scopes are not a "compile-time" phenomenon -- they apply equally to interpretation, analysis, and all modes of compilation -- but compilation can be viewed as a process of binding scope-derived information into the code.

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. Unbound names must be bound before a function using them can be compiled, or before a reference to one can be completed successfully during interpretive execution; however, editing, printing, analysis, and interpretation of other parts of the function are possible even in the presence of unbound names.

In principle, every dynamic name eventually reduces 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. Thus, in principle we only need one kind of name table entry: a way to macro-expand a name into a form eventually containing only constants (since ultimately the primitive operations are represented by constant function objects). An appropriate set of primitive operations would be:

(FLUIDREF fluid-cell) -> field-reference
(LOCALREF variable-number [frame-pointer]) -> field-reference
(FIELDREF field-descriptor [base-pointer]) -> field-reference
(FETCH field-reference) -> value
(REPLACE field-reference value) -> value
(ENCLOSING [frame-pointer]) -> frame-pointer (the lexical access link)
(CLOSURE function-object var1 ... varn) -> function-object
(APPLYFN function-object fluid-binding-link arg1 ... argn) -> value

(A field-reference is a new kind of reference object which can be used for either fetching or storing.) Indeed, these primitives are available in the system, since they are needed for the interpreter and compiler, but their use is deliberately rendered inconvenient (except for APPLYFN) since it can lead to violations of scope rules on which not only compiler optimizations but the correct working of programs may depend. (The mechanism for isolating the use of these primitives, like the even lower-level primitives like CLOSER, is to install the corresponding primitive function objects, not in the global dictionary, but in a private dictionary. Scopes using these operations must mention this dictionary explicitly.) As is usual in implementing protection systems, we provide, instead of these primitives, a set of higher-level constructs that cover all programming needs above the lowest level of system implementation, which lend themselves better to static analysis and permit more enforceable protection boundaries.

Here, then, are the constructs actually provided in the form of different kinds of name table entries:

(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, and editing requires copying the affected part and then recopying when installing in the name table.

(SCRATCH value) -- the value is present in the name table, but it 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 litatom 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*. (See the discussion of Closures under Functions, above, for more detail.) 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.

Miscellaneous

Pointers

In current Interlisp, the only way to protect a structured object which the system allows the user to access is to copy the object when it enters the protected area and copy it again on any request to see it. We choose to provide a more efficient mechanism that has many other uses as well, namely, to introduce a read-only bit into pointers: this still requires copying going in, but not going out. The read-only bit is propagated, in that any pointer retrieved through a read-only pointer is made read-only.

The MIT Lisp machine uses a combination of shallow binding and "invisible indirection" to achieve its efficient implementation of lexical scoping and correct semantics of closures. We intend to use the same technique. This requires being able to specify in a local variable binding in a stack frame that a fluid cell is to be used instead, which is a limited form of invisible indirection. We don't currently plan to provide this in other contexts.

Syntax

The syntax for programs is ultimately an S-expression syntax. There are two ways that this syntax gets extended:

There will be a preprocessor that converts an infix notation into S-expressions, and a prettyprinter that inverts this process, so that corrections made to the S-expressions can easily get reflected in the infix source. We don't have any prejudice about which notation programmers will prefer to use -- both will be supported for listings, editing, etc. Programs, however, will always operate on the S-expressions.

Macros can produce alternative syntax at the S-expression level. AND/OR, PROG, SELECTQ, and CLisp constructs like the iterative statement are good examples of this. Some such constructs (like the iterative statement) are complex enough to be packages in their own right. However, such alternative syntax is *always* introduced by a well-defined (macro) name in the function position. We may support the CLisp technique of caching the translation of such constructs, to speed up interpretation.

We feel that CLisp-style "error-driven interpretation" is a mistake: the language should be defined in such a way that if an execution-time error occurs, it is the result of an erroneous computation or source program, not of a construct that the user thinks of as valid but that just happens to get executed through the error machinery.

Even in present Lisp, "program syntax" is more than the syntax of expressions -- it really includes DECL constructs, BLOCKS declarations, fileCOMS syntax, etc. ****** We haven't yet really dealt with the question of syntaxes for defining linkage and dictionary construction.

Filing

External filing systems provide capabilities (e.g. backup, archiving, scavenging, sharing, large capacity, very high reliability) that we do not understand how to provide entirely within a programming system. 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 (like the Interlisp filemap) 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.

Declarations, assertions, and invariants

There seem to be two kinds of declarations/assertions:

> Those which are associated with *control*, i.e. which hold precisely when control reaches a particular point in the program.
>
> Those which are associated with *objects*, i.e. which hold throughout an object's lifetime independent of the flow of control.

Type declarations for fields are a kind of object assertion. They are easy to enforce, by simply checking whenever one stores into the field that the value being stored is of the right type. However, more complex invariants relating various fields of an object cannot be enforced this way. What seems to be needed for this case is a way to make updating operations apparently atomic. The accepted way of implementing this is to use monitors. An alternative method with slightly different consequences is an atomic multiple-store operation in the language. This avoids the necessity of ever having the object in a programmer-visible state that contradicts the invariant, at the cost of some programming awkwardness.

One place where this problem arises in particular is in the initialization of objects. Invariants cannot be guaranteed to hold until the object has been initialized properly. We first encountered this problem in the context of argument records: the function may have complex criteria for argument legality, which cannot be checked until all the arguments have been computed. In this case there is normally no problem, since the construction of the argument record and the storing of the argument values is an atomic operation from the programmer's viewpoint. The right way to take care of this seems to be the same as in Alphard, namely, to allow implementor-specified initialization code.

Topics not covered adequately

How do scopes get built out of dictionaries -- search rules.

Inter-Office Memorandum

To	EPE working group	Date	November 15, 1978
From	Lampson, Levin, Satterthwaite	Location	Palo Alto
Subject	Plan for an Interim Integrated Mesa Environment	Organization	CSL

XEROX

Filed on: [Ivy]<Levin>PE2>IME.bravo

This memo describes the steps required to obtain an interim integrated Mesa programming environment. We establish the following ground rules:

1) 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. Consequently, the proposed system is not idiot-proof and occasionally delivers its function in less-than-ideal ways.

2) The primary purposes of this environment are to speed up the edit/compile/debug/edit loop by removing the Alto Executive as the link between weakly-communicating programs and to exploit the large real and virtual memories of the Dorado.

3) This system is only intended to work on the Dorado.

We divide the development into two phases. Phase I will be concerned exclusively with the construction of the environment proper and a fast edit/compile/debug/edit loop. A change to a module will require recompilation of that module and any other modules that depend upon it. Configurations whose constituent modules change will have to be rebound and reloaded, causing data in their frames to be lost. Phase II will produce a system that relaxes these requirements in certain common situations, thereby eliminating unnecessary recompilations and loss of data in these cases.

Approximate Effort to Construct Phase I System: 9.5 work-months
Approximate Effort to Construct Phase II System: 4 work-months

Phase I Development

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 components of the Phase I system are listed below with estimates of the effort required to produce them.

Microcode

The present Dorado Mesa microcode supports Mesa 4.1 and precisely emulates the Alto behavior. It is therefore extremely inconvenient to access more than 64K of primary memory. The microcode will have to be extended to be functionally compatible with the present D0 implementation, which requires the addition of instructions that manipulate long pointers (24 bit addresses) and page

faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

The Alto/Mesa runtime environment will be carried over to the Dorado more-or-less intact. The actual movement of code requires only recompilation (to take advantage of the microcode discussed above). Functional changes are required 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 XMesa (the version of Alto/Mesa that exploits the extended memory of a wide-bodied Alto). The only additional substantive change is a virtual memory maintenance facility. We expect to implement a simple-minded scheme in which the virtual memory is mapped onto a contiguous region of the disk, thereby eliminating the need for paging tables. *Approximate effort*: 1 work-months.

An rudimentary protection scheme is required to keep the various programs from clobbering each other. We intend to give each major entity (compiler, debugger, editor, etc.) its own MDS (main data space - a 64K region of the virtual address space that is implicitly addressed by short (16-bit) pointers). Code segments and symbol tables (see below under Compiler) will reside outside all MDSs and thus be accessible only via long pointers. Since any program can fabricate a long pointer and use it, we cannot prevent arbitrary access, but by using the write-protect facility of the hardware memory map, we can prevent arbitrary modification. When a particular MDS is executing, all other MDSs, all code segments, and all symbol tables will be write-protected. Switching control from one MDS to another requires changes to the map entries for all pages in each MDS. Code segments and symbol tables remain write-protected except when the compiler and debugger find it necessary to modify them (e.g., setting breakpoints, cache update). *Approximate effort*: 2 work-months.

Interactive Base

We envision a window-oriented display manipulation facility as the basis for interaction with the various programs in the environment. The package presently used in the Mesa debugger supplies adequate function, albeit in a somewhat awkward fashion. It is likely that this interface will be replaced by the one used in the Tools Environment, possibly within the next few months. (This conversion would be done by the Mesa group and become part of the official, released debugger.) *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

An alternative package (called Chameleon) is being developed by AOS to support market probes. This interface will probably share many of the ideas of the Tools interface, but package them somewhat differently. In particular, Chameleon expects to find a data base package (AOS calls it Iliad) and understands a variety of scenarios for interacting with the data base. Chameleon will supply 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

There are (at least) two sources for a text editor of adequate functionality (i.e., about what the Laurel editor provides). The Tools group have an editor that runs in the Tools environment. Obviously, if this environment becomes the interactive base of IME, we can adopt this editor virtually intact. (Smokey believes we can alter the command set easily to provide global substitution, etc., if we want.) The editor uses a quick-and-dirty form of piece tables (every piece is an integral number of lines), which means it is probably somewhat wasteful of disk space. *Approximate effort*: 1 work-month.

Chameleon will also provide a text editor, whose functions will resemble those of the Laurel editor (but cleaned up a bit). Doug Brotz is slightly skeptical about unplugging the Chameleon editor and inserting our own, but agrees that there should be no problem "in principle". The Chameleon editor will understand about data base records and manipulating fields of such records, a feature we might want to play with, but don't really need. *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

In Mesa, every module has an associated symbol table. Both the compiler and the debugger maintain a cache of the symbol tables they manipulate, though the limited memory space on the Alto tends to restrict the effective size of these caches. We propose to maintain this cache in the virtual memory space and change the compiler and debugger to share a single cache, whose entries are accessed by long pointers. Ed Satterthwaite believes this compiler change is nothing more than a recompilation with an altered definitions file. Once this separate cache exists, the compiler's "front-end" processing of the DIRECTORY statement (which maps symbol table names to Alto files) has to be reworked in a straightforward manner. *Approximate effort*: 2 work-months.

Debugger

In IME, the debugger will be "in control". That is, the command language (such as it is) will be interpreted in the debugger. Thus, the existing command processor will be extended to include commands that trigger editing, compilation, binding, etc. This will require that the Alto/Mesa debugger's interface to the "debugee" (currently, Inload/Outload) be replaced by the MDS-switching mechanism described above. Data in the debugee will then be accessed by long pointers instead of reading Swatee. The debugger will also be changed to use the common symbol table cache. *Approximate effort*: 1.5 work-month.

Binder

The Mesa binder combines configurations (of which the compiler output is a degenerate case) to form new configurations. In the Phase I system, when a configuration changes, all configurations in which it appears must be rebuilt. The binder will therefore be required as a part of the process that takes a multi-module program from editing to execution. We intend to make no functional changes to the binder. In particular, since the binder does not read symbol tables (except in a unimportant, Alto-specific case), it doesn't have to be modified to use the shared symbol table cache. *Approximate effort*: 0.5 work-months.

Phase II Development

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

Replacing an Existing Module

Small changes to a module should not necessarily require recompilation and rebinding of all configurations in the dependency chain. In particular, if the compiler is able to honor the existing structure of the module when recompiling it, the resulting new version can (almost) be slipped into the system as a replacement for the old version without altering the current state. Specifically, the compiler must retain the arrangement of variables in the global frame and the order of procedures

in the module's entry vector. (Additional entries can be tolerated as long as they don't require allocation of a larger global frame.) This ensures that links in other modules will not have to be changed. Obviously, the compiler cannot always satisfy these constraints, but when the module has been changed only slightly, the compiler has a good chance of doing so. *Approximate effort*: 2 work-months.

There are one complication in the loader, however. At the time that the (new version of the) module is loaded, the loader must locate all extant local frames that point to the global frame being replaced. The saved program counters in these frames are now nonsense, since the code has been recompiled. Therefore, these frames cannot be allowed to execute. Two options are available: (1) cut back the stack to the point of such a frame (i.e., simulate a RETURN from that frame), or (2) mark the frame so that a trap occurs when control returns to it. Both options are of equal difficulty. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

SDD has, in the past, worked on a system called DeSoto, whose purpose is to automate the process of maintaining a consistently compiled and bound set of modules. DeSoto actually consists of three separable parts: (1) a dependency analyzer, (2) a symbol table patcher, and (3) a file system maintainer. The dependency analyzer is a straightforward program that follows the graph of module inclusions to determine what modules might (logically) require recompilation as a result of a change. The symbol table patcher has a detailed knowledge of symbol table structure and can determine whether a particular change actually requires recompilation or whether it can be accommodated by patching the symbol table (and associated .bcd). The file system maintainer makes sure that consistent source and object files are stored in designated places, and controls the other two parts of DeSoto.

We propose to implement only the dependency analyzer as part of the Phase II system. (In fact, the Mesa group has a stand-alone version of this program, called the Include checker.) The dependency analyzer by itself ensures that all modules that must be recompiled are in fact recompiled, though it will induce unnecessary recompilations. We consider the effort to implement the symbol table patcher to be of marginal benefit for IME. *Approximate effort*: 1 work-month.

Inter-Office Memorandum

To	EPE working group	Date	November 16, 1978
From	P. Deutsch	Location	Palo Alto
Subject	EPE consensus principles	Organization	CSL

XEROX

Filed on: <Deutsch>epe-consensus.memo on Maxc1

At the request of the working group, here is a memo which sets down certain "consensus principles" which are implicit in the EPE Report, although not explicitly stated, and which were made more explicit in the presentations that Butler and I gave at Dealer on Nov. 1.

Here are what both presentations mentioned as important strengths in Lisp (to be preserved in XLisp, and added to EPE Mesa):

- Automatic storage deallocation
- Easy use of programs as data
  - Atoms
  - Universal pointers / run-time type system
- Integrated editor / instant turnaround

The remaining items on our lists did not coincide, although they all appear on the feature list in the EPE Report. My presentation mentioned Mesa strengths to be added to Lisp, but Butler's did not mention Mesa strengths to be preserved (presumably because EPE Mesa is meant to be upward compatible with present Mesa in a much more direct sense than XLisp is upward compatible with Interlisp).

Here are some mechanisms we both viewed as appropriate for representing information about programs:

- Tables mapping name to <location, type> for frames
- An atom table associating a print name (string) and internal ID
- Tables for each pointer-containing type giving <location, type> for each pointer field
- Globally unique type identifiers
- "Spacing" tables mapping (fixed-size) ranges of virtual addresses to the type of objects allocated in that range

Inter-Office Memorandum

To	PE-Interest	Date	November 16, 1978
From	Lampson, Levin	Location	CSL, Palo Alto
Subject	Mesa-based EPE plan	File	[Ivy]<Lampson>EpeMesaPlan.memo

XEROX

The plan for a Mesa-based EPE has three major stages:

> *Preliminaries*: 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.
>
> *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.

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.

Preliminaries

This section describes the steps required to obtain an interim integrated Mesa programming environment (IME). We establish the following ground rules:

1) 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. Consequently, the proposed system is not idiot-proof and occasionally delivers its function in less-than-ideal ways.

2) The primary purposes of this environment are to speed up the edit/compile/debug/edit loop by removing the Alto Executive as the link between weakly-communicating programs and to exploit the large real and virtual memories of the Dorado.

3) This system is only intended to work on the Dorado.

We divide the development into two phases. Phase I will be concerned exclusively with the construction of the environment proper and a fast edit/compile/debug/edit loop. A change to a module will require recompilation of that module and any other modules that depend upon it. Configurations whose constituent modules change will have to be rebound and reloaded, causing data in their frames to be lost. Phase II will produce a system that relaxes these requirements in certain common situations, thereby eliminating unnecessary recompilations and loss of data in these cases.

Approximate Effort to Construct Phase I System: 9.5 work-months
Approximate Effort to Construct Phase II System: 4 work-months

Phase I Development

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 components of the Phase I system are listed below with estimates of the effort required to produce them.

Microcode

The present Dorado Mesa microcode supports Mesa 4.1 and precisely emulates the Alto behavior. It is therefore extremely inconvenient to access more than 64K of primary memory. The microcode will have to be extended to be functionally compatible with the present D0 implementation, which requires the addition of instructions that manipulate long pointers (24 bit addresses) and page faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

The Alto/Mesa runtime environment will be carried over to the Dorado more-or-less intact. The actual movement of code requires only recompilation (to take advantage of the microcode discussed above). Functional changes are required 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 XMesa (the version of Alto/Mesa that exploits the extended memory of a wide-bodied Alto). The only additional substantive change is a virtual memory maintenance facility. We expect to implement a simple-minded scheme in which the virtual memory is mapped onto a contiguous region of the disk, thereby eliminating the need for paging tables. *Approximate effort*: 1 work-months.

An rudimentary protection scheme is required to keep the various programs from clobbering each other. We intend to give each major entity (compiler, debugger, editor, etc.) its own MDS (main data space - a 64K region of the virtual address space that is implicitly addressed by short (16-bit) pointers). Code segments and symbol tables (see below under Compiler) will reside outside all MDSs and thus be accessible only via long pointers. Since any program can fabricate a long pointer and use it, we cannot prevent arbitrary access, but by using the write-protect facility of the hardware memory map, we can prevent arbitrary modification. When a particular MDS is executing, all other MDSs, all code segments, and all symbol tables will be write-protected. Switching control from one MDS to another requires changes to the map entries for all pages in each MDS. Code segments and symbol tables remain write-protected except when the compiler and debugger find it necessary to modify them (e.g., setting breakpoints, cache update). *Approximate effort*: 2 work-months.

Interactive Base

We envision a window-oriented display manipulation facility as the basis for interaction with the various programs in the environment. The package presently used in the Mesa debugger supplies adequate function, albeit in a somewhat awkward fashion. It is likely that this interface will be replaced by the one used in the Tools Environment, possibly within the next few months. (This conversion would be done by the Mesa group and become part of the official, released debugger.) *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

An alternative package (called Chameleon) is being developed by AOS to support market probes. This interface will probably share many of the ideas of the Tools interface, but package them somewhat differently. In particular, Chameleon expects to find a data base package (AOS calls it Iliad) and understands a variety of scenarios for interacting with the data base. Chameleon will supply 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

There are (at least) two sources for a text editor of adequate functionality (i.e., about what the Laurel editor provides). The Tools group have an editor that runs in the Tools environment. Obviously, if this environment becomes the interactive base of IME, we can adopt this editor virtually intact. (Smokey believes we can alter the command set easily to provide global substitution, etc., if we want.) The editor uses a quick-and-dirty form of piece tables (every piece is an integral number of lines), which means it is probably somewhat wasteful of disk space. *Approximate effort*: 1 work-month.

Chameleon will also provide a text editor, whose functions will resemble those of the Laurel editor (but cleaned up a bit). Doug Brotz is slightly skeptical about unplugging the Chameleon editor and inserting our own, but agrees that there should be no problem "in principle". The Chameleon editor will understand about data base records and manipulating fields of such records, a feature we might want to play with, but don't really need. *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

In Mesa, every module has an associated symbol table. Both the compiler and the debugger maintain a cache of the symbol tables they manipulate, though the limited memory space on the Alto tends to restrict the effective size of these caches. We propose to maintain this cache in the virtual memory space and change the compiler and debugger to share a single cache, whose entries are accessed by long pointers. Ed Satterthwaite believes this compiler change is nothing more than a recompilation with an altered definitions file. Once this separate cache exists, the compiler's "front-end" processing of the DIRECTORY statement (which maps symbol table names to Alto files) has to be reworked in a straightforward manner. *Approximate effort*: 2 work-months.

Debugger

In IME, the debugger will be "in control". That is, the command language (such as it is) will be interpreted in the debugger. Thus, the existing command processor will be extended to include commands that trigger editing, compilation, binding, etc. This will require that the Alto/Mesa debugger's interface to the "debugee" (currently, Inload/Outload) be replaced by the MDS-switching mechanism described above. Data in the debugee will then be accessed by long pointers instead of reading Swatee. The debugger will also be changed to use the common symbol table cache. *Approximate effort*: 1.5 work-month.

Binder

The Mesa binder combines configurations (of which the compiler output is a degenerate case) to form new configurations. In the Phase I system, when a configuration changes, all configurations in which it appears must be rebuilt. The binder will therefore be required as a part of the process that takes a multi-module program from editing to execution. We intend to make no functional changes

to the binder. In particular, since the binder does not read symbol tables (except in a unimportant, Alto-specific case), it doesn't have to be modified to use the shared symbol table cache. *Approximate effort*: 0.5 work-months.

Phase II Development

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

Replacing an Existing Module

Small changes to a module should not necessarily require recompilation and rebinding of all configurations in the dependency chain. In particular, if the compiler is able to honor the existing structure of the module when recompiling it, the resulting new version can (almost) be slipped into the system as a replacement for the old version without altering the current state. Specifically, the compiler must retain the arrangement of variables in the global frame and the order of procedures in the module's entry vector. (Additional entries can be tolerated as long as they don't require allocation of a larger global frame.) This ensures that links in other modules will not have to be changed. Obviously, the compiler cannot always satisfy these constraints, but when the module has been changed only slightly, the compiler has a good chance of doing so. *Approximate effort*: 2 work-months.

There are one complication in the loader, however. At the time that the (new version of the) module is loaded, the loader must locate all extant local frames that point to the global frame being replaced. The saved program counters in these frames are now nonsense, since the code has been recompiled. Therefore, these frames cannot be allowed to execute. Two options are available: (1) cut back the stack to the point of such a frame (i.e., simulate a RETURN from that frame), or (2) mark the frame so that a trap occurs when control returns to it. Both options are of equal difficulty. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

SDD has, in the past, worked on a system called DeSoto, whose purpose is to automate the process of maintaining a consistently compiled and bound set of modules. DeSoto actually consists of three separable parts: (1) a dependency analyzer, (2) a symbol table patcher, and (3) a file system maintainer. The dependency analyzer is a straightforward program that follows the graph of module inclusions to determine what modules might (logically) require recompilation as a result of a change. The symbol table patcher has a detailed knowledge of symbol table structure and can determine whether a particular change actually requires recompilation or whether it can be accommodated by patching the symbol table (and associated .bcd). The file system maintainer makes sure that consistent source and object files are stored in designated places, and controls the other two parts of DeSoto.

We propose to implement only the dependency analyzer as part of the Phase II system. (In fact, the Mesa group has a stand-alone version of this program, called the Include checker.) The dependency analyzer by itself ensures that all modules that must be recompiled are in fact recompiled, though it will induce unnecessary recompilations. We consider the effort to implement the symbol table patcher to be of marginal benefit for IME. *Approximate effort*: 1 work-month.

Foundations

We culled five major (read "potentially difficult to implement") facilities from the memos by Bobrow and Masinter, and discussions among the EPE group:

1) Programs as data
2) Variable syntax analysis
3) Dynamic variable binding
4) Safe language (subset)
5) Automatic storage allocation

These seem to be the major basic language/system capabilities which must be added to Mesa to make a Lisp-like programming style possible. Doubtless there are also a number of minor points which will need to be improved, but we believe that these are in the moise for the current level of planning detail.

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.

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. This seems straightforward in principle, though we haven't looked at the details. We think that a representation other than parse trees should be explicitly defined, permitting the representation of parse trees to change as the compiler/language evolve.

Approximate effort: 1 work-month.

Atoms. The symbol tables built by the Mesa compiler currently include a mapping between source identifiers and internal unique ids (atoms). At present, these mappings exist only on a per-compilation unit basis, implying that they will need to be merged into a global map when the compilation unit "enters the environment". A similar merge presently occurs inside the compiler when the source program accesses an included module (i.e., the symbol table for that module is opened and the identifiers (used by the source program) are remapped into the current atom table). A merge into a global atom table therefore seems feasible, at least in principle.

Approximate effort: 1 work-month.

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. It seems possible to extend the techniques used to ensure uniqueness of record types to accommodate general types. Run-time efficiency is vital, so the check that two type IDs are equivalent must be fast. We anticipate using the LISP "spacing" technique, which places all data structures of the same type within a well-defined region of the address space (e.g., each page allocated for such dynamic structures can hold objects of precisely one type). This scheme avoids keeping any type ID bits with the pointer, but makes it difficult to point inside a composite data structure with a universal pointer. Two possible remedies have been suggested: (1) an explicit type identifier immediately preceding the (sub)structure, or (2) an indirection mechanism in the data structure itself. In any event, access to statically allocated variables (i.e., those in local and global frames) via universal pointers will have to be prohibited, unless special steps are taken to prevent problems with garbage collection.

It will be possible to discriminate universal pointers with the existing Mesa facilities for discriminating variant records. We also intend to add a facility for coercing a universal pointer to a pointer of more specific type; the coercion must be acocmpanied by a run-time check. The intended model is the Lisp IPLUS function, which coerces its arguments to integers. This coercion is applied during assignment; since argument-passing in Mesa is treated exactly like assignment, this seems to work out just fine.

Approximate effort: 4 work-months.

2) Variable 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 a piece of program in S-expression form. 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.

Approximate effort: 4 work-months (3 for (a), assuming the existing Lisp read-table design, and 1 for (b)).

3) Dynamic variable binding.

Mesa, at present, can only support 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. Such a table could reside in the code segment, with a bit in the associated global frame(s) to indicate its existence. Dynamic binding would then occur much in the same way the signal-to-catch phrase binding occurs (but more efficiently). It might be better to use some other form of variable identifier than its atom; the existing machinery used to bind signal codes should be suitable for binding such identifiers also.

Approximate effort: 1 work-month.

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. We will also consider a relaxation of this condition in which certain *regions* of the storage may not be safe in this sense; as long as values stored in such unsafe regions do not themselves contain pointers to safe regions, the damage which can be caused is confined to the unsafe regions.

There are three ways in which a pointer variable *p* may acquire an improper value

1) Lack of initialization - in this case the value of *p* is just the random bit pattern which happens to be in the storage occupied by *p* when it is instantiated.

2) Incorrect computation - in general a pointer value can only be the value of some pointer variable (which could of course be a record component) or the result returned by a storage allocation. If pointer values can be computed in other ways, e.g. with LOOPHOLEs or their equivalents, or by arithmetic, then there is no guarantee that such values point to storage containing values of the proper type.

3) Premature deallocation - in this case *p* is pointing at a region of storage which once contained a value of type T, but is now being used for some other purpose.

If none of these bad things can happen, then induction tells us that the language is safe with respect to pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer. Finally, array descriptors require the same treatment as pointers.

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, a simple and reasonable rule seems to be to disallow the use of @ on local variables. As partial compensation, it seems desirable to introduce **var** parameters (call by reference); these can be thought of as pointers to stack variables which are *always* dereferenced, so that it is not possible for the called procedure to make a copy of the pointer which will outlive the call. We also need to enforce a restriction that **var** parameters are not permitted across process and coroutine calls, since in those cases there is no guarantee that the caller will live longer than the called procedure (again, this seems too strong, but it isn't clear how to weaken it).

For dynamic variables some support from an automatic deallocation mechanism seems essential, since otherwise there is no way to tell when the programmer says Free(*p*) whether other pointers to *p*^ still exist. We see three possibilities

a) Reference counting. This could be added to Mesa easily enough, but the cost during execution is rather high, and it is probably not a good idea in view of the other alternatives.

b) Deferred freeing. The idea is to make a list of all the storage freed by the programmer, and at suitable intervals to invoke the scanning phase of a garbage collector and check that there are no outstanding references to the storage queued to be freed. At the end of the scan any storage which has passed this check can be actually freed.

c) Automatic deallocation, discussed below.

Quite possibly deferred freeing will be worthwhile, since it is fairly easy to implement and has substantially lower running costs than fully automatic deallocation. With plenty of address space there should be no serious objection to a long interval between the programmer's call of Free and actual availability of the storage.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Relative pointers have several appealing properties, one of which is particularly relevant in this context

a) They allow an entire complex structure to be moved in storage, or between storage and a file, as long as all its contained pointers are relative to the structure.

b) They can save substantial amounts of space, especially as the address space grows and pointers become longer.

c) 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 relation 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 age of the region 2^n; an especially nice value for *n* is 16.

Approximate effort: 2 work-months.

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.

Approximate effort: 3 work-months (for Lisp-style transaction garbage collection, including necessary compiler changes).

Features

xxx

XEROX
PALO ALTO RESEARCH CENTER
Computer Science Laboratory
December 8, 1978

To PE-Interest

From Dan Swinehart

Subject Mesa-Based EPE Programmer's Assistant

Filed as [Maxc and Ivy]<Swinehart>Assistant.bravo, . . .Assistant.Press

This section attempts to characterize the "programmer's assistant" facilities of a Mesa-based EPE, and to estimate their cost. Here we consider the activities required to provide a more or less direct replacement for existing Interlisp functions.

However, one 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.

Prerequisites

It is assumed that the work described under *Preliminaries* and under *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

Before anything remotely resembling the Interlisp interactive flavor can be achieved, either the Mesa debugger's interpreter must be extended to encompass the entire language, or a compiler-based statement evaluator must be produced. In this latter, preferred, approach, the compiler would be extended to provide the equivalent of incremental compilation for single non-declarative 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, if necessary, for the execution of statements corresponding to the Lisp "top level". Suggested implementation: enclose the statements in a procedure body, compile and execute the procedure.

Approximate effort: interpreter, 4-6 somewhat mythical work months; compiler extension, 2-4. 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

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 -- more powerful, in fact, when combined with the Mesa display-oriented methods for indicating points of interest. (In fairness, DLisp presumably restores the balance.) The standard breakpoint handler could take care

of providing the linkage to the system-produced procedures containing the advice.

Approximate effort: 1-3 work months.

History Lists and their Use

The top-level interpreter, a history list that records top-level user actions at the text level, 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. But to provide *Undo* requires that internal information be used 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. The following points prevent a placement in the easy category:

> 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.
>
> Mesa programs, because Mesa is an assignment-based language, produce more temporary side-effects, on average, than Lisp programs. The information stored to provide undo capabilities might be more extensive. Because of the large virtual memory, space problems will probably not arise, but performance problems might. However, an implementation the equivalent of the TESTMODE(T) facility in Interlisp -- saving the old contents for *every* store into memory -- could cause both time *and* space problems.
>
> Mesa, possessing a syntax, is susceptible to syntax abuse. Such could occur when trying to provide the undoable versions of each language primitive that can produce a storage modification (e.g., a /_ b ?) Some care will be required to integrate this basic capability into the language. This is just an instance of the efforts that are needed anyway to eliminate the special role of primitive generic functions in Mesa. Languages like Alphard provide a model for how this might be done.
>
> (More generally, as mentioned elsewhere in this report, the maintenance of a syntax that is acceptable to both man and LALR machine will present problems throughout the system that are not encountered in the pure Lisp implementation. It will be extremely difficult to produce a set of interactive facilities in Mesa whose use is as natural and compelling as those in the current Interlisp.)

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.

In a display-based editor, the structure-specific stuff can be largely implemented as part of the selection

mechanism. To the extent that the text files are in communion with the underlying parse tree or S-expression representations, it will be easy to produce the needed hierarchical selections.

Approximate effort: 1 work month.

Dwim

Some spelling correction and the repair of minor typographically induced syntax errors could be incorporated into the compiler. It would be useful, given room, in both interactive and standard versions of Mesa. The user interface to these tools are likely to be quite different in a system based on compilation and display editing. No estimate.

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.

Approximate Effort: the database facilities, no estimate; the interface, 2 work months.