CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Votes on the First Draft Common Lisp Manual

Guy L. Steele Jr.
with contributions by

Alan Bawden
Howard I. Cannon
George J. Carrette
David Dill
Scott E. Fahlman
Neal Feinberg
John Foderaro
Richard P. Gabriel
Joe Ginder
Charles L. Hedrick
Earl A. Killian
John McCarthy
Don Morrison
David A. Moon
William L. Scherlis
Richard M. Stallman
Daniel L. Weinreb
Jon L White

18 November 1981

Spice Document Sxxx

Keywords and index categories: PE Lisp & DS External

Location of machine-readable file: FEEDBACK.MSS.103 @ CMU-20C

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

Introduction

In August, 1981, the first draft of the COMMON LISP Manual was distributed to a number of potential implementors and other interested parties. This draft was dated 13 August 1981, and is known as the "Swiss Cheese Edition" (and was incorrectly identified on its title page as the "SPICE LISP Manual"!).

Comments on this draft were collected and collated into a document whose title was *Discussion of the First Draft Common Lisp Manual.* The comments were grouped by topic, and nearly all were put under the heading of one or another "issue", each issue being a question or suggested change about which there might be debate. For each issue there was a heading, the relevant comments from the respondents, and a set of suggested courses of action (supplied by me). At the end of the document was a ballot. In October, 1981, the collated document was distributed to the commentators, who were asked to vote on the 233 issues and return the ballot.

This document contains the entire original text of the *Discussion* document (with minor typographical corrections), and in addition records the results of the balloting and additional comments received.

One important typographical change: a heading was inadvertently omitted from the *Discussion* document for issue number 226. This heading is included here. As a result, issues numbered 226 through 232 in the *Discussion* document are here numbered 227 through 233.

The people who commented on the draft manual and/or submitted a ballot are identified herein by a short identifier as follows:

ALAN	Alan Bawden	CLH	Charles L. Hedrick
HIC	Howard I. Cannon	EAK	Earl A. Killian
GJC	George J. Carrette	JMC	John McCarthy
DILL	David Dill	DM	Don Morrison
SEF	Scott E. Fahlman	MOON	David A. Moon
CHIRON	Neal Feinberg	WLS	William L. Scherlis
JKF	John K. Foderaro	RMS	Richard M. Stallman
RPG	Richard P. Gabriel	DLW	Daniel L. Weinreb
GINDER	Joe Ginder	JONL	Jon L White

The ballot asked that each vote be a single letter: "Y" or "N" for yes/no questions, or a letter for one of several explicitly labelled choices, or "X" if no choice was acceptable. The intent was to keep the procedure simple, but of course complex issues never fit into a simple mold. Soon after distribution of the *Discussion* document, the following message was broadcast by Scott Fahlman:

> I was just looking over Guy's discussion document, and it occurred to me that there were a lot of issues on which I might vote differently from the rest of you, but about which I really don't care very much. As an aid in resolving the major outstanding issues without undue delay and in

knowing which things should be given priority in any meeting, it might be very useful to know which issues people really care about. At the risk of slightly complicating the feedback procedure, I would suggest that in addition to making a choice on each question you also give it an importance rating from 0 to 5 as follows:

0 Profound indifference.
1 I am indicating a choice, but I really don't care much.
2 Minor preference for the indicated choice.
3 Definite preference for one choice.
4 I think this issue is very important.
5 A make-or-break issue. The wrong decision here will either be impossible to implement or totally unacceptable to my group.

This info can just be filled in on the feedback form. It is optional, but will help a lot in deciding where the real problems are.

Some respondents used this system and some did not; some simply marked issues important to them with a "+", or used other symbols. Also, in the case of multiple-choice questions some respondents indicated more than one choice, sometimes even assigning different weights to the different choices. In a few cases there were even responses such as "D" for a yes/no question, or "Y" for a multiple-choice question, most probably indicating a clerical error in filling out the ballot. Debugging such errors and converting all the reponses into a uniform format has required some subjective interpretation; I can only apologize for any misrepresentation of anyone's intentions.

Each vote is recorded here as one or more letters followed by some number of exclamation points. The numerical scale outlined above was converted as follows: a vote of 0 is simply not shown and otherwise a vote of n is shown as $n-1$ exclamation points. Thus a vote of "A3" is transcribed here as "A!!". For other systems of preference I have tried to translate the sense of importance indicated by the respondent into this form.

For some issues, where there is a clear consensus, I have indicated (in 30-point type!) the conclusive answer (I counted 111 of them). For some other issues, where there is a preponderance of votes for one choice but also either significant opposition or new commentary which I think should be considered before a final decision, I have indicated the preponderant answer followed by a question mark (I counted 56 of these). Where there is no consensus, no conclusion is indicated (I counted 66 of these). In determining consensus I have applied some judgement; as a rule a single dissenting vote is discounted unless quite vehement (at least three exclamation points), and more than one dissenting vote may be discounted if they have no exclamation points at all.

Following the vote for each issue are any additional remarks received on the subject with the ballots or since distribution of the *Discussion* document.

—Guy L. Steele Jr.
November, 1981

Introduction to "Discussion of the First Draft Common Lisp Manual"

This document is a collation of comments on the first draft ("Swiss Cheese Edition") of the COMMON LISP Manual (incorrectly identified on its title page as the "SPICE LISP Manual"!) dated 13 August 1981. Also contained here are my suggestions for specific changes to the COMMON LISP Manual, based on my perception of the consensus of the commentators, or on my own technical judgement where there appears to be no consensus. Sometimes several alternative suggestions are presented where I cannot make up my mind either.

This document is intended to aid in the resolution of problems in the first draft by brining together for inspection the reactions of the several commentators. A second draft will be produced after discussion of the points of disagreement has taken place.

This document is parallel in structure to the first draft manual, containing verbatim all the chapter, section, and subsection headings from the manual. The comments are therefore grouped under the relevant section headings. All of the paragraphs labelled "??? Query:" from the draft manual are also included here.

Numbers appearing in square brackets (for example, "[43]") refer to page numbers in the first draft COMMON LISP Manual. Such numbers with a decimal fractional part "[*m.n*]" refer to page *m*, about 0.*n* of the way down the page; thus "[43.7]" refers to a place on page 43 about 2 or 3 inches from the bottom. Square brackets are also used for bibliographic references. Other material in square brackets represents editorial interpolations of the usual sort, or contextual quotations from the manual.

I am responsible for all text in this document not specifically marked as being written by another person. Comments by other persons are labelled by an identifier as follows:

ALAN	Alan Bawden	MOON	David A. Moon
HIC	Howard I. Cannon	RMS	Richard M. Stallman
GJC	George J. Carrette	DLW	Daniel L. Weinreb
SEF	Scott E. Fahlman	JONL	Jon L White

Most of these comments, however, I have transcribed from terse handwritten notations, and so I have taken the liberty of making such editorial corrections as introducing punctuation, adding font specifications, and correcting grammar. Occasionally, when pictorial devices such as arrows from the comments to the original text were used by a commentator, I have found it expedient to paraphrase the entire remark; in these cases the entire paraphrase is enclosed in square brackets. I apologize if in doing this I have inadvertently misinterpreted or mislabelled anyone's remarks.

My responses to specific questions raised by a commentator, and text which I consider to represent my personal biases rather than matters of fact, are explicitly flagged by "GLS".

The comments are more or less arranged according to the structure of the draft manual, but are also organized by topic. Every topic on which there may be a question of consensus is flagged by a heading proclaiming it to be an "issue":

***** Issue 0: Do pigs have wings?**

Following this heading are arranged the relevant comments, and then one or more suggestions for resolving the issue. If the comments all point to a single solution, then a single suggestion is made, which appears as a paragraph flagged by the word "**** Suggestion**". If there are multiple suggestions, the word "**** Alternatives**"

appears, followed by suggestions labelled by letters of the alphabet. For example:

**** Alternatives:**

A. Pigs should have wings.

B. Pigs should not have wings, but give them antlers instead.

C. Status quo: pigs should not have wings.

In either case, I have freely added suggestions of my own, especially when other commentators have remarked on problems but specified no solutions. Frequently when there are several possible choices I have added an additional choice or two which is particularly radical or off-the-wall, if only not to restrict the collection to too narrow a point of view.

I would like to resolve these issues *tentatively* by a balloting process, voting on the suggestions offered for each issue. This is not meant to force a choice among the given alternatives or to cut off debate, but simply to limit the clerical work required to collate the responses and determine a tentative consensus (or lack thereof) for each issue. If a choice mentioned here reflects more or less what you think is best for COMMON LISP, then you can support that choice with absolute minimum of redundant verbiage.

The results of this balloting will be used to guide a revision of the draft COMMON LISP manual. The cycle will then be repeated: a second draft will be distributed for comments (this time to a larger set of people), a second set of comments collated, and a second ballot taken (or perhaps simply a meeting called, if the set of remaining issues is small). The current ballot is *not* the last chance to comment; therefore I would prefer not to receive extensive commentary or have new issues raised with the ballot, but rather to wait until the next draft comes out. (Similarly, please don't bother to tell me that some of the offered alternatives are stupid. Just vote against them. I will confess that a few jokes have been inserted to amuse you while you wade through the morass.)

This ballot is not intended to produce final decisions on the issues contained herein, but to determine where there is consensus and where there is disagreement. Therefore where there is significant disagreement on an issue, majority rule will *not* obtain; instead, such issues will be recorded and deferred until a meeting can be held. The purpose here is to identify those issues where we all agree so that valuable meeting time need not be wasted on them.

Feedback on this document would be most useful in the following form: a list of issue numbers, each paired with a vote. For a single suggestion, vote "`Y`" or "`N`"; for a multiple suggestion, vote the appropriate letter. If none of the choices even approximately reflects your best judgement, then vote "`X`", and in that case attach English text describing your alternative.

Some comments are not listed under an **Issue** heading, because they seemed not to raise a point of controversy or to identify merely typographical errors. If you wish to remark on such comments, attach such additional remarks (suitably identified) to the ballot.

At the end of this document is a ballot that you can simply fill in and send via U.S. mail; the address is on the ballot. Staple additional sheets of commentary to it if necessary. Alternatively, you may send a facsimile ballot via ARPANET mail to `Guy.Steele @ CMU-10A`.

Thank you for your help in this project.

—Guy L. Steele Jr.
October, 1981

General Questions

***** Issue 1: What Dover font should be used for LISP code?**

MOON: [1.4] Use the LPT font family, available from MOON. This will be in the next Xerox font release, I'm told.

RMS: Why not use the LPT series of fonts for code, instead of GACHA. It has all the M.I.T. character set except possibly the characters hidden under CR, LF, TAB, RUBOUT, etc.

GLS: LPT does correct the problems with underscore, accent acute, and accent grave. The tilde is still really a swung dash, but this is within the variation permitted by the ASCII standard, I think. In any case, it's better than anything else available.

**** Suggestion:** Use LPT as soon as it becomes available. (Anything better around?)

Y

**** Responses**

MOON: Y
DLW: Y!
HIC: Y
GLS: Y
SEF: Y
RPG: Y
GINDER: Y!
CHIRON: Y

MOON: The file `MOON2;LPT FONTS @ MIT-MC` is a binary file containing a PrePress fonts dictionary.

MOON: There are numerous useful things from Lisp Machine Lisp missing. Some examples are `errset`, `third`, `with-output-to-string`, `string-search-set`. It would be interesting to know whether this is intentional, an oversight, due to the incompleteness of the draft manual, or a result of using the black Chine Nual rather than the red one (or, better, the gray one).

GLS: Each of these is correct in some situations.

***** Issue 2: Policy for naming functions in COMMON LISP**

MOON: Many common English words, especially ones which are likely to be used by programmers, are used up for new system functions. I don't think these words should be used for these functions. The most egregious example is `control`...

ALAN: [107.6] [On the names of the generic sequence functions:] In ten years there won't be a word left in the dictionary that isn't the name of a standard LISP function.

GLS: This is a constant problem whenever a language is expanded. Sussman has complained loud and long about the fact that a standard (at that time) LISP utility was named `index`; he wanted that name. On the other hand, "good" names are also desirable for system functions (`search` is nicer than `srchq`, for example). Any other specific suggestions?

MOON: We also need to consider the need to make it continue to be possible to write programs which work in both COMMON LISP and MACLISP. Thus for instance if the meaning of `assoc` is changed the name should be also.

SEF: I hate to cede to MACLISP all the good names just because it managed to make some small error in how to use them. Maybe we continue to redefine things where needed, but provide a source-to-source translation package. I think MACLISP is just about dead anyway.

** Suggestion: Make COMMON LISP the best language we can; shall this include choosing nice names, even though they are English words likely to be desired by a user?

** Responses

MOON: X	RMS: Y!	SEF: Y!	GINDER: Y!!!	DM: X!
DLW: X	HIC: X	WLS: Y!	CHIRON: Y!!	
	GLS: Y!	RPG: Y!!	DILL: Y!	

WLS: Yes, but leave me "`foo`".

RMS: Use common words only if they are specifically semantically appropriate. That is, "search" is very appropriate for a searching function, but "index" can only be specifically appropriate for a function to index into an array, and even then it doesn't clearly say what it will *do* with the index. Does it do indexing? Return an index as a value?

MOON: While we should make COMMON LISP the best language we can, I think there is an important portability constraint. We should not make it unnecessarily difficult to write portable programs. We should not make incompatible changes that are not mechanically detectable, and we should avoid using the same function name to mean something different than what it means in MACLISP, unless there are overriding good reasons to do so. By "portability", here I am referring both to conversion from the current Lisp Machine LISP system to the future, COMMON LISP-compatible one, and also to portability between COMMON LISP dialects and MACLISP-compatible dialects (PDP-10, Multics, and FRANZ LISP). Since there is no COMMON LISP implementation for the PDP-10 in sight, and probably will never be one, MACLISP will unfortunately continue to be used on PDP-10's for some time. Since PDP-10's are wide-spread now, and at least two companies are designing future implementations of that architecture, PDP-10's will unfortunately still be around for some time to come. MACLISP compatibility should not be an overriding consideration for COMMON LISP, but it should not be slighted.

DLW: It has to be handled on a case-by-case basis.

Chapter 1

INTRO

1.1. Purpose

***** Issue 3: Purposes of Common LISP**

DLW: You might mention the basic idea, namely that this is supposed to be a common subset for many upward-compatible dialects.

SEF: Mention *stability* as a goal for COMMON LISP: once it's defined, we expect it to change only slowly and with due deliberation.

MOON: [3.8] [Features which are useful only on certain processors are avoided or made optional.] Not done to the extent that I would prefer.

GLS: Examples?

**** Suggestion:** Add two paragraphs to this section as follows:

Commonality. At least four implementation groups already actively at work on LISP implementations for various machines are considering supporting this dialect. While the differing implementation environments will of necessity force incompatibilities among the implementations, nevertheless COMMON LISP can serve as a common dialect of which each implementation can be an upward-compatible superset.

Stability. It is intended that COMMON LISP, once defined and agreed upon, will change only slowly and with due deliberation. The various dialects which are supersets of COMMON LISP may serve as laboratories within which to test language extensions, but such extensions will be added to COMMON LISP only after careful examination and experimentation.

Y

**** Responses**

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y!	

MOON: Some of the features useful only on certain processors which are not avoided in the draft COMMON

LISP manual, and which I find the most objectionable, include:

- The vector/array distinction.
- The profusion of type declarations, especially in the sequence functions.
- Miscellaneous other kludges here and there to make it easier to open-code things on the VAX.

A great deal of my work over the past five years has been directed at killing off exactly this sort of constraint on the language, through the Lisp Machine LISP approach (which includes SPICE LISP).

DM: [Several paragraphs by DM follow.] A few random flames:

There seems to be some confusion as to what exactly this COMMON LISP standard is. Is it a definition (albeit informal) for a core language, or is it a user manual? It certainly doesn't seem to be a "core" language; it's even bigger than the INTERLISP VM spec (though not so convoluted, I hope).

I think there are two different issues to be addressed here. One is what is the class of relatively primitive operations which define LISP? The other is a library of standard function names to facilitate program interchange. I would rtahers ee them separated, a la red, white, and yellow pages. In principle I suppose there is no reason to do this, but I think it would make discussion much easier. The core will never be gotten right if its semantics are hidden in a morass of complicated names for simple functions `list-nreverse`. There is definitely a need for such a library; it should just be done in parallel to a standard core.

I'd rather see about one-tenth of the current "core" defined as a real core. Implementors could share code for the remainder, at least at first. They might tune it to their implementation. The idea is that there would be definitions of all the rest built out of only the basic core. This sounds like it has no part in a definition; after all, implementors can do that anyway. Well, I think such a partition should be made early on. It will help people trying to understand what COMMON LISP is even more than the implementors. It will especially help the people actually defining the language to know what it is they are defining!

The profusion of data types doesn't really belong in a core. Just simple objects, like vectors. Build more complicated ones like arrays and structures out of the simple ones. This requires having some more mechanism for adding new types (okay, that's acknowledged hair; maybe just a predefined collection of them, rather like the way some antique LISPs had tags for bignums and all the hooks, but they weren't really there till you added a bignum package), with appropriate hooks into the various generic functions (including I/O). With an appropriately flexible package facility each of these guys could reside in a different package.

Sorry, but the current version really gives a feeling of "Well, what's the largest subset of Lisp Machine LISP can we try to force down everyone else's throat, and call a standard?" It would seem far better to try to make the best language possible, especially given that it will be frozen. Folks have lived with enough mistakes of earlier LISPs long enough. Why perpetuate it some more, all in the name of compatibility? I suspect that any good LISP can have MACLISP compatibility mode and the like living in packages, which will do just as well as keeping musty, old definitions around in a supposedly clean, new LISP.

The very notion of "everything else being equal, we'll be compatible with *x*" is a crock. If you can even consider doing it differently than you've done it for years in *x*, there's obviously something at least slightly wrong with the way *x* does it.

***** Issue 4: Division of Common LISP into a core plus modules**

MOON: [4.2] [On the division into a core language and independent modules:] This seems weird. Motivate it. Maybe these modules are optional at the implementation's choice?

SEF: Mention the division into white, yellow, and red pages.

GLS: The intention, originated by SEF, was to impose a discipline to prevent the complicated interweaving of packages which has occurred in INTERLISP. The impression one gets from the INTERLISP manual is that nearly every part of the system has its fingers in some other part, via various global variables, switches, and hooks. The proposed discipline for COMMON LISP is to divide the world into a core language and a set of peripheral packages. The core is described in the COMMON LISP manual; these SEF terms the "white pages". The "yellow" pages describe implementation-independent packages, such as `trace` or a scientific subroutine package. The "red" pages are similar, but document implementation-dependent routines, such as device drivers. Ideally the packages would be theoretically as independent as possible, although one probably would not want to make them arbtrarily optional. Cases of non-independence should be minimized and carefully documented.

**** Suggestion:** Does this discipline make sense? **Y?**

**** Responses**

MOON: N	RMS: N!	SEF: Y!!!	GINDER: Y!!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: X!	GLS: Y	RPG: Y!!	DILL: Y!!!!	

RMS: Keeping things modular is a good goal but don't expect to succeed completely. For example, a winning `trace` facility requires encapsulations, which `defun` or its subroutines must know about.

ALAN: The division only makes a little sense. What good are the red pages? The claim is that they document "implementation dependent" routines. How are they dependent on the implementation? If they have an interface that is the same for all implementations, then they at least belong in the yellow pages. If the interface is different, or if they only exist in some implementations, then why are they documented in the COMMON LISP manual at all? This should be clarified. The reasoning behind the yellow pages is even less clear. Why have these packages been singled out to have their documentation printed on a different color of paper? Again, I just need clarification before I can judge the merits of this idea.

1.2. Notational Conventions

***** Issue 5: Syntax for description of special forms**

MOON: [5.5] I think using & keywords in [the descriptions of] special forms is grossly confusing. See new syntax in Chinual version 4.

DLW: I strongly agree! This aggravates the confusion of thinking that special forms are function calls! Please fix!

GLS: Well, `defmacro` allows `&optional` and `&rest`. Nevertheless, I think the idea is a good one to use some form of BNF.

**** Suggestion:** Modify the descriptions of macros and special forms not to use &-keywords.

Y

**** Responses**

MOON: Y!!!	RMS: N!!!		GINDER: Y	DM: Y!
DLW: Y!!!	HIC: Y	WLS: Y!		
ALAN: Y!	GLS: Y	RPG: Y		

RMS: Many special forms *are* function calls, whether philosophers like it or not. This does not imply that the special forms should be described by giving the arglist of a definition for the function, but BNF is confusing and hard to understand. I definitely prefer the arglists to standard BNF. An extended BNF might make the grade.

CLH: [November] About comments: your convention makes it almost impossible to read in a function with comments. I realize that MACLISP tends not to do that, but some users might like to use a structure editor, and have comments show up. R/UCI LISP uses `{ ; ... }` as a comment, allowing it to be put more or less anywhere in the code. { is defined as a read macro that reads all characters up to the next }, turns them into a string, etc., so that `{ ; abc }` reads as `( ; "abc")`. ; is defined as a function that returns ( ). The pretty-printer puts this back into the original form.

GLS: [November] Of cousre, such a comment can be put "more or less anywhere" only if you are writing FORTRAN-style LISP code. If you use a highly applicative style, such comments can be put more or less nowhere. To meet the goal of truly transparent comments which don't evaporate at `read` time, more cooperation from the interpreter and compiler is needed, or a more clever scheme such as hashing.

Chapter 2

Data Types

***** Issue 6: What useless value should side-effecting functions return?**

MOON: [8.6] [Those functions which have nothing better to return generally return `t` or `( )`.] Not zero values?

GLS: Well, that's an interesting point. Theoretically, that is nicest; however, in certain implementation strategies that might impose unnecessary overhead (strangely enough), because it might require going through some part of a more complicated multiple-value protocol rather than the normal single-value one. (Having a third, specialized, zero-value protocol is probably not worth the trouble.)

**** Suggestion:** Let the current specification stand: return `( )` for a useless value.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!!	GINDER: Y!!!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!		RPG: Y!		

RPG: Perhaps we should make the printing primitives return 0 values, because this is a pure side-effect issue, it is the one thing you don't want to return a value from, and printing is slow anyway so the overhead shouldn't be too bad in most implementations compared to the fixed cost.

MOON: The suggestion disagrees with the quoted statement [8.6], which is not a specification and says "`t` or `( )`".

2.1. Numbers

***** Issue 7: Should the type "scalar" be called "real"?**

DLW: Shouldn't the type named `scalar` be called `real`?

GLS: Mathematically speaking, I would agree. However, there are two problems with using the word "real" for this data type.

- The usual problem that digital computers can't really implement real numbers, but only a countable subset of them (typically the rationals or a subset thereof) and so the term "real" would be incorrect in this application.
- Most other languages which use the term "real", however incorrectly, use it to mean "floating-point number", and so to use "real" to mean floating-point numbers, integers, and ratios would be doubly confusing.

Except for this possibility of user confusion, I do not object to the term "real". As for "scalar", I have consulted a number of mathematical dictionaries, and could reach no definite conclusion regarding it. The term connotes componentlessness or directionlessness, and is usually contrasted with vectors or tensors. Some dictionaries, however, specifically noted that a scalar is often restricted to be real, listing "real or complex" (as opposed to a tensor) as a separate definition. In some applications complex numbers are construed to have direction or components, and in others are taken to be scalars.

** Suggestion: Change the name of the data type `scalar` to be `real`.

** Responses

MOON: Y	RMS: Y!	SEF: N!	GINDER: Y!	DM: N!
DLW: N!	HIC: N	WLS: N!	CHIRON: N	
ALAN: Y!	GLS: Y!	RPG: N!		

WLS: `scalar` loses, but `real` is worse.

RMS: It is true that scalars are sometimes restricted to be real, but this is a special case of restricting scalars to a particular field, which can be any field. They are also sometimes restricted to be rationals, or to be integers mod *p*. In some contexts, scalars can be taken from a ring instead.

GLS: I now agree with RMS. Also, it might be useful to use the term `scalar` to mean a non-sequence, but that is another story.

2.1.1. Integers

ALAN: [10.1] Does anybody have `#B` now? I took it *out* of Lisp Machine LISP years ago.

2.1.2. Floating-point Numbers

*** Issue 8: Bigfloats (arbitrary-precision floating-point numbers)

SEF: Define bigfloats. Are they optional?

GLS: Currently a syntax is defined for bigfloats, but nothing else is said about them.

** Alternatives:

A. Remove bigfloats from the language.

B. Add definitions of operations on bigfloats to the language.

C. Require bigfloats to be a defined data type, and require bigfloat I/O, but leave bigfloat arithmetic to an optional package.

** Responses

MOON: A	RMS: X!	SEF: A!!	GINDER: C!	DM: X!
DLW: C!!	HIC: B	WLS: C!	CHIRON: B!	
	GLS: B!!	RPG: B!!!		

RMS: Make bigfloats an optional feature of the language, and someday define the syntax and operations that people should use if they are going to have bigfloats at all. Do not mention them in the core.

MOON: Bigfloats should be deferred.

WLS: Perhaps allow bogfloat I/O to map to a long-float representation, as controlled by a global variable.

CLH: [November]

I think `E`, `F`, `D`, `S`, `L`, and `B` is going a bit far. Is anyone going to remember which is which? Maybe you should adopt the standard sequence `S`, `P`, `D`, `F`, ... from quantum mechanics? If you are going to specify this many levels of precision, why not do more like PL/I and let the user specify the exact number of digits he wants, and then round up to the nearest thing you supply. The easiest way to do this would be to look at the number of digits he types, but you could also provide some syntax like `#11F13.2` to mean 13.2 with 11 digits of precision. If you want to have multiple letters, then I think you would be better off to specify an exact precision which each is to supply, and let the implementation round up to what it can provide. I think I would prefer a contiguous set of letters, with increasing precision, for example `12.2B12` being least, `12.2C12` next, etc.

GLS: [November] Great! Short, Plain, Double, Fourple, Giant?

***** Issue 9: Terminology "single" and "double" for floating-point numbers**

??? Query: [11.9] There has been some objection to the use of the words *single* and *double*, as they may be misleading to the user or too confining for the implementor. Any suggestions?

MOON: Should the words which are the same as those used in the IEEE standard mean the same things?

GLS: As noted in the draft manual, the terms are meant to encourage implementors to provide floating-point precisions and ranges at least roughly equivalent to IEEE standard. If true IEEE standard floating-point is supported by host hardware, it would seem to be desirable to support them directly for the sake of portability. There would remain many issues of supporting rounding, exceptional values, and so on which are not addressed by the draft manual.

** Alternatives:

A. Status quo: retain the terms "single" and "double".

B. Retain the terms, but also require COMMON LISP implementations to support the IEEE floating-point standard.

C. Rename the terms so as not to cause confusion with the IEEE standard. (In this case, someone please suggest new terminology.)

A?

** Responses

MOON: A		SEF: AC!		DM: C!
DLW: A!	HIC: X	WLS: A!	CHIRON: A!	
	GLS: A	RPG: B!!!		

MOON: COMMON LISP should specify that the "single" and "double" floating-point formats will be as close to the IEEE standard "single" and "double" numbers as the underlying hardware of the implementation permits to be done efficiently. But otherwise we should leave the complex and subtle business of standardizing floating-point arithmetic to people like Kahan.

WLS: Of course implementations should be encouraged to conform, but true support for the IEEE standard seems to be implementation dependent. Could conformity be revealed by a `status` request?

***** Issue 10: Should radix-specifier syntax be immediate or pervasive?**

ALAN: [12.3] [The proposed non-decimal floating-point representation] will screw the Lisp Machine LISP reader's versions of #R, etc. We just bind `base` and call `read` so that `#3R(120 222)` will work. But floating-point read can't look at `base` because it is usually 8.

GLS: By way of explanation, Lisp Machine LISP allows #R, #0, and other radix specifiers to precede any S-expression, and any integers in the S-expression will be interpreted by that specifier unless an inner specifier shadows it or an explicit decimal point appears. Floating-point numbers are always read in decimal radix. NIL has suggested that non-decimal floating-point input may be useful for very specialized applications. In an attempt to minimize confusion, the COMMON LISP draft requires a radix specifier to immediately precede the representation of a number; it could be argued that this also makes code easier to read by eliminating the possibility that the value of a number might be drastically altered by something very distantly preceding it. However, the pervasive version also has its uses, such as notating a list or array of several hundred octal values (in an assembler, perhaps). (By "pervasive", I mean that the specifier pervades an entire S-expression, affecting all numbers within it.)

** Alternatives:

A. Eliminate non-decimal floating-point numbers; require radix specifiers to immediately precede a number.

B. Eliminate non-decimal floating-point numbers; permit pervasive radix specifiers to affect all integers (except those with a decimal point).

C. Eliminate non-decimal floating-point numbers; permit pervasive radix specifiers to affect all integers (except those with a decimal point) and ratios, but never integers forming the argument for a # construct.

D. Retain non-decimal floating-point numbers; require radix specifiers to immediately precede a number.

E. Retain non-decimal floating-point numbers; require radix specifiers to immediately precede a floating-point number, but permit radix specifiers to pervasively affect all integers (except those with a decimal point).

F. Retain non-decimal floating-point numbers; require radix specifiers to immediately precede a floating-point number, but permit radix specifiers to pervasively affect all integers (except those with a decimal point) and ratios, but never integers forming the argument for a # construct.

G. Retain non-decimal floating-point numbers; permit radix specifiers to pervasively affect all integers (except those with a decimal point), ratios, and floating-point numbers, but never integers forming the argument for a # construct. (The objection to this in Lisp Machine LISP doesn't arise in COMMON LISP because the default input radix for integers as well as floating-point numbers is ten.)

H. Eliminate all radix specifiers!

I. Eliminate #-style radix specifiers, and use the following bizarre scheme. The usual way to indicate a non-decimal radix is by writing the radix in decimal as a subscript. Use square brackets, as is traditional in computer science, to indicate such subscripting. Thus instead of the integer `#0105` one would write `105[8]`; the same number in hexadecimal is `45[16]`. Similarly, `FOO[32]` would be read as the (decimal) value 16185. This is of course harder to implement, because one can't just bind `base` and do the read, but it is closer to standard notation. If the LISP reader is organized so that it first scans over some alphanumeric token and only then decides what kind of token this is, this is no problem (the current COMMON LISP reader specification will allow this). Now, for compatibility, let "." mean the same as "`[10]`" in this context, so that a trailing decimal point always means decimal radix. Now back-generalize this to floating-point numbers: π in decimal notation is either `3.14159265` or `3[10]14159265`, and in octal notation is `3[8]11037552`.

**** Responses**

MOON: C!!!	RMS: X!	SEF: C!!	GINDER: D!	DM: A!
DLW: X	HIC: C	WLS: F	CHIRON: I!!	
ALAN: I!	GLS: I	RPG: G!	DILL: AD!!!	

RMS: Complicated. I like several of the ideas, but only with modifications. Here is the order, best to worst:

I but don't necessarily eliminate other kinds of radix specifiers.

H but keep the default radix 8. H is no good otherwise because octal number are *vital.*

C but I'm not sure exempting arguments to *all* # constructs is right. Exempting radixes is right, but not array sizes in `#A`.

HIC: I vote C, but choice I has some merit.

GINDER: I lean slightly towards retaining non-decimal floating point; and I strongly prefer immediate radix specifications. Alternative I doesn't seem all that bizarre to me!

MOON: COMMON LISP should reserve a syntax for implementation-dependent non-decimal floating-point;

programs using that syntax are non-transportable. `#F` might be a good choice. Alternatively, there could be no syntax and implementations could have a function that converted a character string to a flonum, to be used to set up implementation-dependent floating-point constants.

DLW: We should make `#O`, `#X`, *et al.* be immediate, so that you can say `#xFF` instead of `#x+FF`, but *only* if we provide some way to be pervasive in cases where you really want pervasion with its action-at-a-distance (e.g., Lisp Machine LISP patch files). Suggestion "I" is actually not bad, although `#xFF` is rather easier to type than FF[16]. I don't feel strongly about inclusion of non-decimal floating.

DILL: My primary desire here is to see `#O`, etc., not be pervasive (radix should be a lexical property, not a syntactic one).

2.1.3. Ratios

***** Issue 11: Should rationals be kept in canonical form?**

MOON: [Most arithmetic functions produce rational results in canonical form.] ?

DLW: ? indeed.

**** Suggestion:** Remove this remark from the manual. Implementations may represent raionals however they choose. However, add functions `numerator` and `denominator` which are specified to return the appropriate components of the *reduced form* of a rational number.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: N!!	
ALAN: Y!	GLS: Y!	RPG: Y!		

2.1.4. Complex Numbers

***** Issue 12: Representation of complex numbers**

ALAN: [7.8] Why specify [that complex numbers are Cartesian]? Say just "complex numbers" and leave the representation up to the implementation. I might want to use base $i-1$, for example!

ALAN: [13.4] Why in Cartesian form? I might prefer polar or base $i-1$. This spec should be written to allow other representations than those that store a real and imaginary part.

GLS: This case is not quite parallel to that of rationals, if only because of programming-language tradition. I suspect that there are numerical analysts out there who really depend on knowing the round-off characteristics of floating-point complex arithmetic. Also, if only to be able to interface to other languages, we

want to have type declarations that can speak of Cartesian floating-point complex representations.

** Alternatives:

A. Remove this remark from the manual. Implementations may represent complex numbers however they choose. However, add functions `real` and `imag` which are specified to return the appropriate (presumably approximate) components of a complex number.

B. Retain the remark. Moreover, specify that

```
(eql x (real (complex x y)))
(eql y (imag (complex x y)))
```

This implies that complex numbers have two components, each of which may be any scalar. This allows one to have "Gaussian bignums", complex rationals, and tighter control over numerical issues in the case of complex floating-point numbers.

B

** Responses

MOON: B	RMS: B!	SEF: B	GINDER: B	DM: B!
DLW: B!!		WLS: B!	CHIRON: B!	
ALAN: B!	GLS: B!!	RPG: B!!		

WLS: If not "B", I think complex deserves no special status in the language.

2.2. Characters

*** Issue 13: Definition of string-char data type

??? Query: [14.3] There is a strong assumption implicit in the definition of the `string-char` type about the way character objects are implemented. Is everyone concerned willing to live with that?

** Suggestion: No one has objected to this, so let it stand as in the manual.

Y?

** Responses

MOON: X	RMS: X!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!		WLS: Y!	CHIRON: Y	
ALAN: X!	GLS: Y!	RPG: Y		

RMS: Please explain?? How does this impact Lisp Machine LISP, which does not have character objects?

ALAN: Well, just what *is* this "strong assumption" about the way character objects are implemented? I would rather know what it was before I agreed to live with it.

MOON: What is the "strong assumption"?

GLS: Clearly I wasn't being clear. The main implication is that character objects cannot have any more *code* possibilities than will fit into a string element.

2.3. Symbols

***** Issue 14: Definition of "property list"**

DLW: The definition of *property list* differs from that in the Lisp Machine LISP Manual; see Revision 3, pages 66-67, which is dealing with issues of disembodied plists.

GLS: It seems to me that, while the Lisp Machine LISP manual made a valiant effort to generalize the notion of property lists in a consistent manner in such as to emphasize the fact that property lists have state in a way that a-lists do not, nevertheless it misappropriated the term "property list" and forced upon it a meaning in conflict with its usual usage.

**** Suggestion:** Alter the COMMON LISP manual to use the term "property list" in a way consistent with the usage in the Lisp Machine LISP manual.

**** Responses**

MOON: Y		SEF: N!!!	GINDER: N!	DM: X!
DLW: X	HIC: Y		CHIRON: N!!!	
ALAN: Y!	GLS: N	RPG: Y!	DILL: N!!	

DLW: The criticism of MOON's explanation in the Lisp Machine LISP manual is valid, but we should do *something* to correctly generalize the notion of property lists to include "disembodied" ones, and until we think of a better terminology, I'll push for MOON;'s.

GLS: How about the following arrangement: have three special forms `getf`, `putf`, and `remf`. Each is like the corresponding property-list function, but instead of a symbol takes a `setf`-able location which contains the property list. Then

```
(get x y) <=> (getf (plist x) y)
(putprop x y z) <=> (putf (plist x) y z)
(remprop x y) <=> (remf (plist x) y)
```

So one can use disembodied property lists by saying such things as

```
(putf (cdr d-p-l) x 'foo)
```

`getf` actually doesn't alter the first argument, and so may be a function rather than a special form; it is essentially what is otherwise known as `memq-alternate`.

DM: [Several paragraphs by DM follow.] It doesn't seem right to specify the implementation of a property list (i.e., as a list of alternating indicators and values). It should simply be defined as a mapping *symbols* x *symbols* -> *anything*. The implementor should be free to do his job however he sees fit: as an a-list or hashtable, perhaps. And it would make much more sense for things which deal with disembodied property lists (i.e., what `plist` returns, no matter how it's "really" stored) to use a-lists.

It would also be nice to see something like SSTANDARD LISP's `flag/flagp/remflag`. These have

generally proven to be of great utility. I believe this is why STANDARD LISP implementations have generally used a-lists to implement property lists: an atomic element must be a flag, while a pair (cons) is an (*indicator* . *value*). This would be an appropriate format for `plist` to return.

The order of arguments to `putprop` seems wrong. Shouldn't it be new value last, like most set functions? In fact, the whole language should probably be combed for similar instances to make sure they're all consistent.

And finally, the names `get/putprop` don't strike me as dovetailing as nicely as `get/put`.

***** Issue 15: Preserving of case in symbols, with case-insensitive interning**

> ??? Query: [16.2] How do people feel about the following plan?
>
> Some programmers, particularly INTERLISP people, like to use case in interesting ways, and insist on case being preserved. For example, they like to use names such as `GrossMeOut`. (This is hearsay; the INTERLISP manual certainly shows no examples of this.)
>
> Anyway, it has been proposed that the internal form of a symbol's print name be not upper-case, but whatever case the symbol was first interned in (and therefore in whatever form it was first typed). So if one says
>
> ```
> (Defun GrossMeOut (Hackp) (Cond ...))
> ```
>
> and later types `(grossmeout t)`, this will correctly access the defined function, and `(print 'grossmeout)` will print `GrossMeOut`, not `GROSSMEOUT`.
>
> There is a set of implications here: `intern` must do `string-equal` hashing rather than `string=`. Can use of vertical bars force the existence of distinct symbols differing only in case, and if so which one gets chosen when a symbol is typed whose capitalization differs from any existing one? I think all this can be worked out; what do people think of it?

ALAN: It cannot work.

HIC: I am against it. I agree that the subtleties here are too much to deal with.

MOON: I don't see how this can work. If you have typed in both `Foo` and `\foo`, then how do these print? With or without [back]slashes? And if you have a list `(Foo \foo)` and read back in its printed representation, do you always get a list of two non-`eq` symbols? Regardless of the order they are first seen? There are a lot of subtleties to this. It cannot work, because it is impossible for `print` to decide when to slashify in such a way that things always read back in equally.

SEF: Guess we have reached the same conclusion. Too bad.

GLS: SEF and I have decided it probably cannot be made to work. We were worried about INTERLISP people, some of whom (rumor has it) are used to writing code in mixed case and having it preserved. We tried for a compromise and failed.

MOON: All INTERLISP system functions are uppercase but you must shift for yourself [GLS: !!!] or let DWIM correct it, as I understand.

** Suggestion: Eliminate this idea from COMMON LISP, and revert to the MACLISP style of normally accepting either case in symbols but converting to upper-case those letters not under the influence of a backslash or vertical bar.

** Responses

MOON: Y	RMS: N!	SEF: Y!!	GINDER: X!	DM: X!
DLW: X	HIC: Y!	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!!	RPG: Y		

RMS: Perhaps. It *can* work. Symbols with backslashes in them have to read in as distinct from symbols with no backslashes, always. So `FOO` and `Foo` and `foo` and `fOO` read in `eq`, regardless of which of them is seen first, and `\FOO`, `\Foo`, `F\O\O`, etc., are never `eq` to them. I'm still not sure that this is the right thing to do, but it deserves more consideration based on the fact that it can be made consistent.

GINDER: I don't like reverting to upper case. However, I don't have an alternative that I think deserves to be the standard for COMMON LISP. (Maybe I'll come up with a non-standard package later.)

MOON: Provide a way to set up the readtable to be like INTERLISP's, which I believe means that there is no case-mapping and names of system symbols must be typed in upper case.

DLW: Paul Martin has a new scheme that he explained to me and RPG. It has some features that we may not like, but it should be considered. RPG can tell you about it.

DM: I think the basic idea is a good one. The problem seems to be arising solely because print name and intern name are (implicitly) considered to be the same. I think the idea should simply be that the name used for interning is always raised and the name as typed is retained solely for printing. This latter form should probably contain any escape characters necessary; alternatively, it might be used only by `princ`. `intern`, when encountering a new symbol, would enter the raised version of the name in the symbol's interned-name field, and its as-printed name in the print-name field. On encountering a symbol whose intern-name is already present, it would either do nothing (essentially retaining the eldest printing form, particularly useful if it was first used in a carefully put together file but is now being used from the TTY by a harried typist who can't be bothered to deal with a shift key) or enter the new print-name. There should probably be some sort of switch controlling which action takes place, probably on a per-package basis.

JKF: [Several paragraphs by JKF follow.] DLW said that COMMON LISP will not distinguish cases in function names. This, I feel, is a big mistake and means that I won't be able to run some of my programs on implementations of COMMON LISP (which don't extend the standard). I am sure that you will all agree that existing hardware can support lower case completely. The only reasons I've heard for not allowing multiple case is that:

(1) People don't talk in cases so they couldn't possibly communicate about programs written in both cases.

(2) I've had a bad experience on MULTICS.

(1) is absurd; UNIX systems have multiple case commands, the C language has multiple case variables and functions, and I've yet to hear a complaint about problems with case. People have no problems discussing their programs.

(2) MULTICS did a bad job in selecting names and/or the people who complain about the cases in MULTICS are

so used to single case systems that they get confused.

Given a multiple case system, it is trivial to change the reader to single-caseify input, thus allowing old programs written in single-case systems to be read in.

CLH: Your documentation seems ambiguous on whether \ is part of the name, when it is used to indicate case. It is fairly easy to show that it has to be included in the internal representation, and thus presumably returned by `explode`, etc. From a first reading of your manual, I thought that \ was not included in the name. However that would lead to the following odd result:

```
(remob abc)
(eq 'AB\C 'abc) --> T  ;AB\C creates an atom with name "ABC"
                       ;abc matches ignoring case, so it fits
(remob abc)
(eq 'abc 'AB\C) --> () ;abc creates an atom with name "abc"
                       ;AB\C requires upper case C, which fails
```

I find the method used in R/UCI LISP quite sufficient: `intern` requires the case to match. Lower case is automatically converted to upper case in `read`, except inside strings or when the character is quoted (equivalent to \).

2.4. Lists and Conses

DM: [November] "Cons" as a noun is hideous, especially in the plural. Why not the euphonious, or at least blander, "pair"?

***** Issue 16: ( ) still a symbol in some implementations?**

DLW: [17.6] Please put in a note saying that `(symbolp ())` may be `t` in some implementations. Maybe put this in small type and explain the history. Otherwise you'll confuse many people.

DLW: [26.9] Please note that in some implementations of COMMON LISP, `(symbolp '())` may be `t`.

GLS: Sigh. This has implications for `typep` as well. Okay, so this means that the types `null` and `symbol` may not be disjoint. Can such implementations nevertheless arrange for this symbol to print as "()", and for the string "`nil`" to read as some symbol other than that one?

**** Alternatives:**

A. Permit ( ) also to be a symbol, at least in some implementations. Nevertheless require it to print as "()".

B. Permit ( ) also to be a symbol, at least in some implementations. Make no requirement on how it prints.

C. Forbid ( ) to be a symbol in any implementation.

** Responses

MOON: B!!!!	RMS: A!	SEF: A!	GINDER: Y	DM: B!
DLW: A!!	HIC: B!	WLS: C!	CHIRON: C!!!	
ALAN: B!	GLS: C	RPG: C	DILL: C!!!	

MOON: Presumably a mode switch to force nil to print as either "`nil`" or "`()`" or the implementation-preferred string is required for portability of data files.

GLS: I don't know of any LISP that won't accept `()` in the read stream and treat it as the empty-list/false object, so always printing it as `()` should work.

CLH: Why make `()` be different from `nil`? This is such an inherent piece of LISP tradition that I don't see how you could possibly be gaining enough to justify the change. This could lead to the worst kind of subtle conversion problems.

2.5. Vectors

*** **Issue 17: Vector notation**

DLW: [17.9] [A general vector is notated just like a list, except for the leading #.] And you can't use dots, of course.

** **Suggestion:** Correct the manual to make this clear.

Y

** Responses

MOON: Y		SEF: Y!!		DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: C!	RPG: Y!	DILL: Y	

GLS: Yes, I really wrote "C" on my ballot. I probably meant "Y".

RMS: This assumes my change to vector syntax is not made, doesn't it? It was very disturbing to see this, since it suggested that my suggestion had been discarded. I immediately searched through and found it.

GLS: Many of the issues interact, and a decision on one may affect others. Many comments applied to more than one issue, and I tried to place each comment where it was most applicable, occasionally inserting a comment under more than one issue. Unfortunately, I did not have time to cross-index all the issues. As you observed, your suggestion is included later in the document, and if it is accepted then the outcome of this issue will not be relevant.

DM: How about `[1 foo (3 2 1) ()]` instead of the rather ugly `#(1 foo (3 2 1) ())`?

2.6. Arrays

***** Issue 18: Vectors and one-dimensional arrays?**

MOON: [19.7] Vectors and one-dimensional arrays *will* be the same in Lisp Machine LISP, contrary to this manual.

GLS: Could not Lisp Machine LISP arrange to distinguish between arrays being used to represent vectors, and those which are "really" arrays? SPICE LISP does just the opposite: it views vectors as primitive, and uses them to build arrays. Hence an array is just a specially tagged vector used to contain dimension information and a pointer to a data vector. The COMMON LISP definition tries to avoid requiring this particular model; however, it is important that vectors and arrays be distinguished, because there are performance gains to be had on non-microcoded architectures. A vector is a bare-bones one-dimensional array; it has a dimension and data, and that's all: no leader, no fill pointer, no displacing feature, no multiple dimensions. If you want any of that, you use an array.

**** Suggestion:** Maintain the linguistic distinction between arrays and vectors. An implementation may take either as primitive and use it to implement the other as long as the language semantics are preserved.

Y

**** Responses**

MOON: Y	RMS: Y!!!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
	GLS: Y!!	RPG: Y!!	DILL: Y!!!	

RMS: Lisp Machine LISP should not attempt to distinguish vectors from arrays, because this would be unnecessary complexity for Lisp Machine LISP users. The "vectors" would be an unnecessary extra data type that precisely duplicates a subcase of another data type.

MOON: I don't see how `vectorp` and `arrayp` being different would be useful to an implementation-independent program. Presumably the only reason to distinguish vectors from arrays in a program is for the benefit of some implementations where some operations only work on one or the other.

GLS: RMS's comment indicates to me that there is an unfortunate ambiguity in the statement of the suggestion, and so the apparent consensus may be an illusion at best. The suggestion could be interpreted in at least two ways, depending on whether the term "linguistic distinction" implies the truth of

```
(not (typep (make-vector ...) 'array))
   and
(not (typep (make-array ...) 'vector))
```

My intention was that `typep` could distinguish the results from `make-vector` and `make-array`, but RMS apparently thinks otherwise and yet has voted in favor of the suggestion. As for MOON's question, the proposal is that arrays have fill pointers and leaders and can be displaced, but vectors cannot (and so the overhead for those features is saved). I apologize for all the ambiguity. Clearly more discussion is needed. It should be noted, however, that his remark that vectors are a strict subcase of arrays is true only if arrays as well as vectors are acceptable to the generic sequence operations.

DILL: Of course, if the language semantics require that one-dimensional arrays and vectors be distinguishable at run time, they have to be different types.

2.7. Structures

2.8. Functions

2.9. Randoms

***** Issue 19: Does #<...> syntax imply a random type?**

MOON: Does what is said here mean `(typep x 'random)` is required to be true of all objects that print with a "#<"?

GLS: It was not so intended. What was intended was that implementors be encouraged to use the syntax `#<...>` to print unreadable objects of type `random`. However, non-random objects might also use this syntax, and not all random objects need use it.

**** Suggestion:** Amend the manual to clarify this issue according to GLS's remarks above.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!		

Chapter 3

Program Structure

CLH: [November] R/UCI LISP defaults unspecified arguments to `()`. That turns out to be sufficient for most purposes and is easier to implement. Have you considered using

```
(defun x (a b . c) ...)
```

instead of

```
(defun x (a b &rest c) ...)
```

? I know MACLISP people tend not to care about speed of the interpreter, but I do, and I don't like having to check every atom in the formal list for various magic & tags. If you insist, I will probably make `defun` preanalyze the formal list.

3.0.1. Stuff I'm Not Sure Where to Put It Yet

***** Issue 20: What about the patch facility?**

SEF: Should the patch facility be mentioned? It's not defined in COMMON LISP yet.

**** Suggestion:** Eliminate the reference from the manual unless and until a patch facility is defined for COMMON LISP.

Y

**** Responses**

MOON: Y		SEF: Y!!	
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y
ALAN: Y!	GLS: Y	RPG: Y	DILL: Y!!

CLH: On `defvar`: I would much rather see a separate function where the initialization always happens than have the semantics depend upon whether it is in a patch file or not. Shades of INTERLISP: `(setq defvar-always-happens t)`.

***** Issue 21: Allow `defvar` to provide documentation without initialization?**

MOON: It would be nice to have a way to give `defvar` documentation without an initial value, while retaining the important simple syntax (`defvar` *var val*).

GLS: It sure would! Others have mentioned this. I have no good solution. I do have a very bad one: let `(defvar foo '() "Foo")` initialize `foo` to `()`, but let `(defvar foo () "Foo")` not initialize it. (I don't like this because I make stylistic use of the distinction between `()` and `'()`.)

**** Suggestion:** Here is an even more bizarre idea:

```
(defvar mwf 8 "Maximum warp factor")   ;Initializes to 8.
(defvar mwf . "Maximum warp factor")   ;Does not initialize.
```

Yes or no?

**** Responses**

MOON: Y	RMS: N!	SEF: N!!!	GINDER: X!	DM: X!
DLW: X	HIC: Y	WLS: Y!	CHIRON: N!!	
ALAN: N!	GLS: Y!	RPG: N!!	DILL: X	

GINDER: Use a keyword (e.g., `:initial-value`, or `:documentation`).

DLW: The `'()` idea is truly unacceptable. The bizarre suggestion is barely okay, but it "closes off" the syntax. I have no good suggestions (sorry).

GLS: Another problem with the bizarre suggestion is that it isn't "Lispy".

DM: How about changing the `defvar` syntax to

(`defvar` (*var init*) *documentation*)

This parallels other initialization constructs, and allows

```
(defvar (mwf 8) "Maximum warp factor")    ;Initializes to 8
(defvar (mwf) "Maximum warp factor")      ;Does not initialize
```

The form suggested in the commentary seems rather kludgey, though I suppose it does get the job done.

DILL: Here is a proposal that I mentioned earlier: make `defvar` follow the convention used in `let`-binding: the first argument should be a list, the *car* a symbol, and the *cadr* its initial value. Then the documentation string can be the second argument. If the variable has no initial value, the first argument to defvar can be a singleton list. If compatibility with Lisp Machine LISP is desired, the old syntax can be retained without ambiguity (and uninitialized variables can be written as

```
(defvar (foo) "this is a documentation string.")
```

GLS: Applause! Two great minds thinking alike, obviously. My only reservation is that in most initialization constructs omitting the value means `()`, not undefined. Nevertheless, this is the best I have seen so far, and it is upward-compatible for smooth change-over.

??? Query: [21.9] Actually, the rules for this need to be worked out better? Maybe two kinds, one which always initializes and one which doesn't?

MOON: Seems confused.

GLS: I was.

***** Issue 22: What does `defconst` really mean?**

ALAN: [22] How about specifying a *proceedable* error [when `defconst` is about to alter a value]? It's nice to be able to change constants.

MOON: `defconst` is *not* a declaration to the compiler that the value will never change and it may be wired into compiled code. If that is desired, there should be a separate declaration for it.

HIC: Yes, currently `defconst` means exactly the opposite. Unlike `defvar`, *val* is evaluated every time! Maybe Lisp Machine LISP should change this, maybe not.

MOON: What it says about `defconst` checking for changing the value is terribly confused. `defconst` is *not* a declaration to the compiler that the value of this variable will never change and the constant value may be bound into compiled code. Perhaps there should be another special form which does that. `defconst` is a declaration that the value of this variable will not be changed by the program, only by changes *to* the program; thus changes to the program should be allowed to change it, whereas with `defvar` changes to the program should not reset the variable to its initial value, losing the state of the program.

GLS: Hm. This is an interesting distinction. But the only way to change a `defconst` variable by changing the program is to change the `defconst` itself. Therefore a sophisticated implementation might arrange for `defconst` to recompile things whenever it changes the value. Nevertheless, you're right that I misunderstood the purpose of `defconst`.

SEF: Well, if that's what people think it means, that's okay by me, but `defconst` is definitely a bad name in this case.

RMS: `defconst` should not cause an error if the symbol already has a different value. The only reason this would happen in normal operation is that the user has edited his program and wants a different value. He should be obeyed. The only case in which a `defconst` with a different value represents an error is when there are several nonagreeing `defconst` declarations for one variable. It might be reasonable for a programming system which keeps track of what was defined where to warn about defining the same constant (or function) in different files. Aside from that, there should be no warning or even query, since this event will occur legitimately too often.

** Suggestion: Define `defconst` to be consistent with its Lisp Machine LISP definition.

Y?

** Responses

MOON: Y!!!	RMS: Y!	SEF: Y!	GINDER: Y	DM: Y!
DLW: Y!!!	HIC: Y	WLS: N!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: N!!!	

WLS: No. The semantics and pragmatics seem too fuzzy.

RMS: I think it is legitimate for the compiler or even the interpreter to refuse to allow a `setq` of a symbol that has been `defconst`'d. `set` must be allowed (since `defconst` will use it). `defconst` is not a bad name, because it really does mean that the symbol value is a constant as far as the program is concerned, just not that it is *so* constant even in the programmer's mind that he would never wish to change it. If a simple name that says this more specifically can be found, it is okay with me, but "constantness" is accurate. The distinction is, constant with respect to whom?

HIC: Maybe come up with a better name?

DILL: The complaints here mystify me. Why should `defconst` behave any differently from macro definitions or `inline` declarations of procedures? If you compile things and then change them, you should generally expect to re-compile. The ability to use symbolic constants that get expanded by a compiler is important if you want people to write maintainable code that also runs fast.

Chapter 4

Predicates

***** Issue 23: Terminology: "pseudo-predicates"**

MOON: Didn't "pseudo-predicates" used to be called "semi-predicates"?

GLS: No, I cannot find a reference for either! I thought a pseudo-predicate was one which returns ( ) or non-( ) (as opposed to ( ) or t), and a semi-predicate was one which returns t or else never returns. Maybe I'm wedged.

**** Suggestion:** Just eliminate both terms from the COMMON LISP manual.

Y?

**** Responses**

MOON: Y		SEF: N	
DLW: Y	HIC: Y	WLS: N	CHIRON: Y!
ALAN: Y!	GLS: Y!	RPG: N!	

4.1. Data Type Predicates

***** Issue 24: Have a type symbol for "sequence"**

SEF: [24.6] `sequence` should be another type symbol.

**** Suggestion:** Add `sequence` to the list on page 24.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

MOON: [24.7] Unclear to me why it's useful to have one-dimensional bit vectors but not 2D ones. I guess this

is an implementation kludge.

GLS: All vectors are 1D. However, the intention was that for any kind of vector there is a corresponding array type. Thus one can have 2D bit arrays.

***** Issue 25: Functional data types**

RMS: [25.8] Function subtypes, such as `(function (list symbol) list)`, seem to be useless since there is no way to determine the truth of

```
(typep '(lambda (x y) ...) '(function (list symbol) list))
```

If these are intended only for declarations to help the compiler optimize, there should be a comment in the manual to that effect, where these types are defined.

GLS: Right you are, and this is nasty. Bill Scherlis and David Dill here at CMU are working on a consistent and rigorous treatment of data types. Tentatively we can divide data types into three kinds: primitive data types, derived types (which map onto primitive types, as for example a request for a vector of integers mod 9 might actually get you a vector of integers mod 16), and undecidable types. The first kind is what `typep` of one argument might return; the second kind is what you can give as the second argument to `typep`; and the third kind can be reasoned about formally but membership cannot always be effectively tested.

**** Suggestion:** Specify all this in the manual.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!!	DILL: Y!!!!	

***** Issue 26: `typep` of a structure**

MOON: [26.3] What does `typep` return for a structure, or is that undefined?

GLS: In the `defstruct` character I believe it says that one-argument `typep` must return the name of the structure type, not `structure`.

**** Suggestion:** Clarify this issue at this point in the manual also.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y!!!!	

MOON: `rational` type [GLS: `ratio?`] splits into `irat` and `bigrat` in Lisp Machine LISP for the L-machine (this affects the value of one-argument `typep` in a system-dependent way).

GLS: I think this is okay. One-argument `typep` is implementation-dependent. The ways in which it may so depend should be more carefully specified, though.

4.1.1. Specific Data Type Predicates

***** Issue 27: Standard truth value returned by predicates**

JONL: Don't specify that the result of a predicate is the symbol `t`. Virtually nothing is lost by merely specifying that it is some non-null constant value; but if losers know that it's supposed to be some particular value, then they will take advantage of that fact in a non-mechanically-detectible way, and poor NIL will lose since predicates return `#t` there. Note that it would even be okay to return something like 1, and Tom Binford used to do that all the time (i.e., substitute "1" for "`t`").

GLS: You need some way to distinguish it sometimes. How about specifying that, whatever the special value is, normally the symbol `t` has it as its value?

** Suggestion: Specify that system predicates that simply return *true* or *false* return for *true* some particular non-`()` value which depends on the implementation. Moreover specify that `t` initially has this value as its top-level value.

**** Responses**

MOON: Y	RMS: X!	SEF: N!!!	GINDER: Y!	DM: X!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: N!	RPG: Y!	DILL: Y	

RMS: Complicated. The symbol `t` is a perfectly good value for things to return to mean "true". `t` and `()` are not analogous! The reason it is good to have `()` and `nil` be different is so that truth versus falsehood can be determined by the data type only, which is clean. This goal does not require use of anything other than `t` (or any other symbol) for truth. So actually there is no good reason why any implementation should return anything other than `t`! But, there is also no terrible reason to prohibit this. Except: what do I put in a `selectq` to compare against the value `t`? For "nil", I can write `()` and win in all dialects. What do I write for "compare against the standard true value"? Do I have to use "`#,t`"? Alternatively, `#t` could be a standard syntax for "the standard true value". But I think it is easier if all dialects return `t`.

MOON: All right, but why not just fix the bug in NIL that `t` and `#t` are different?

DM: We would rather it be even less restrictive. Any (possibly non-constant) non-nil value for `t`. Forcing it to be a constant kills some possible optimizations on conventional architectures, and buys you very little in return.

GLS: I speculate that DM thinks that we want `(and 3 4)` => `t`, which is of course not the case. In any case, perhaps nothing should be said about any kind of standard value for "true" delivered by a predicate.

SEF: Lots of code breaks if we don't go with `t`. I think NIL should conform here. The funny type issues that surrounded `()` versus `nil` do not come up here.

***** Issue 28: Are all the specialized type predicates really necessary?**

MOON: There are a large number of type predicates; evidently `typep` has been made so hairy that it is considered to be too expensive to use. This optimization would better be left to a compiler than be done by the program writer; it isn't difficult. The separate predicates do have the possible advantage that they can be used as arguments to the large number of functionals introduced, e.g., `(ass-if 'double-floatp list)`.

SEF: The separate type predicates such as `floatp` and `rationalp` are merely included for (imagined?) human convenience. The SPICE LISP compiler does in fact open-code all instances of `typep` that correspond to the separate predicates: `(typep x ':rational)` and `(rationalp x)` generate the same code. I would not object to flushing the separate predicates, though we would want to keep those already in use (`listp`, etc.).

GLS: I don't think `typep` is too expensive to use. Certainly with a constant second argument the compiler can figure things out. The separate predicates were included primarily for completeness, since some of them have to be there for compatibility (certainly `atom` and `numberp`!). Having only about half of them wouldn't be bad, but having *almost* all of them would be significantly worse than having all of them, because that makes it harder for the user to remember which ones don't exist. The point about usefulness in functionals is good. I think these names should be retained.

SEF: [27.8] Should `bigp` be renamed `bignump` to preserve the pattern `(`*xxx*`p z)` <=> `(typep z '`*xxx*`)`?

GLS: The only disadvantage I can see with renaming `bigp` is incompatibility with MACLISP.

MOON: [28.6] Maybe it would be better not to introduce `short-floatp` and friends, and leave it to `typep`.

RMS: Why is there a need for individual new predicate functions for all the new types? `typep` does the job well enough. In principle, there is no need for `listp` or `symbolp` either, but they may be used frequently enough to justify individual names, and also the names are used in existing code. For `rationalp`, neither reason applies.

SEF: Good point. Just keep the ones already in use, use `typep` for the rest.

**** Alternatives:**

A. Eliminate all type predicates in section 4.1.1. Instruct programmers to use `typep` with two arguments.

B. Retain `null`, `symbolp`, `atom`, `numberp`, and perhaps `stringp`, `consp`, and `listp`, out of respect for tradition; but eliminate all others.

C. Retain all except `bigp`, `fixnump`, `ratiop`, `short-floatp`, `single-floatp`, `double-`

`floatp`, and `long-floatp`, the general rule being to retain those which make conceptual distinctions and eliminate those which make distinctions of format or representation.

D. If (`typep foo '`*xxx*) works for some symbol *xxx*, then *xxx*`p` should be a defined predicate of COMMON LISP. As a consequence, rename `bigp` to be `bignump`.

**** Responses**

MOON: C!	RMS: BC!	SEF: BD!!	GINDER: X!	DM: X!
DLW: C!	HIC: C	WLS: BC!	CHIRON: CD!!	
ALAN: B!	GLS: D!!	RPG: D	DILL: D!	

WLS: B or C, but definitely not A or D.

RMS: I vote (`/ (+ B C) 2`). I think all the ones in reasonably common use now should be retained. By the way, `consp` and `symbolp` should return their arguments if they are true. People who use dialects in which that is the case tell me it is *very* useful to operate on the value of (`or (symbolp foo) default-thing`).

GLS: Note that if `symbolp` is to return its argument if the argument is a symbol, then `()` may *not* be a symbol, or else `symbolp` will lie about the symbolness of that symbol!

GINDER: I lean towards D (or at least C) with the following addition: `atomp`. Keep `atom` for consistency with MACLISP, but don't encourage its use.

DM: [Several paragraphs by DM follow.] I don't think you can honestly single out which predicates are kicking around till there is a better notion of types. Exactly which should be available as individual predicates is closely related to this issue of primitive versus derived types, and may be implementation-dependent (I don't like that implementation-dependent business, though). Certainly, however, a definite form for the predicates can be given now, and just which ones are around can be decided later.

The question of functionals is a vacuous issue. It's trivial (and probably desirable) to have an anonymous functional generator around to build up whatever type-checking functions you need to pass. Thus instead of

```
(mapc 'intermediate-foobaric-floatp z)
```

or

```
(mapc #'(lambda (x)
          (typep x ':intermediate-foobaric-floatp))
      z)
```

one would write something like

```
(mapc (typefn ':intermediate-foobaric-floatp) z)
```

Such a functional generator could be used for lots of other name explosion cases, like the zillions of sequence-specific functions. It might be worth trying to formulate a general (about the same generality as `setf`) functional builders for many classes of operations. Of course, I have no concrete suggestions about such a thing: probably a ridiculous idea.

MOON: [27.5] Lisp Machine LISP will change `listp` to be true of ( ).

DLW: I can't wait!

***** Issue 29: Second (optional) argument to `functionp`?**

MOON: [29.8] In Lisp Machine LISP, `functionp` takes a second optional argument.

GLS: This appears in [Weinreb 81a] but not [Weinreb 81b], so it is fairly new. The second argument is a flag, default ( ), which if `t` says that macros and special forms count as "functions". Is this really the right way to kludge that test in?

**** Suggestion:** Allow `functionp` to take two arguments.

**** Responses**

MOON: Y!	RMS: Y!	SEF: N!!	GINDER: N!	DM: N!
DLW: Y!!	HIC: Y	WLS: N	CHIRON: N!!!	
ALAN: N!	GLS: N!	RPG: Y	DILL: N!!!	

SEF: Provide two distinct functions.

WLS: No, but give a way of testing if you have a macro or special form.

DILL: I have a great deal of difficulty understanding exactly what `functionp` is supposed to do (and what it is good for) from reading the draft COMMON LISP manual and particularly the Lisp Machine LISP manual. Maybe if it were defined more precisely, this issue could be resolved directly. In general, I cringe when people try to put switches into functions because they can't decide what they should do.

CLH: [Several paragraphs by CLH follow.] What is the functional value of a variable? Is this the same as `fsymeval`? Clearly something is missing here. You talk about it being the same as if it had appeared in the functional position of a function invocation, but don't define that. Nor do you define what a functional value is. In page 36ff [of the draft COMMON LISP manual], the following concepts are mentioned. I hope they are not all different!

- functional value of a variable
- current value of a dynamic (special) variable
- current global function definition named by a symbol
- local function name (= local function definition, presumably)
- local variable
- static instance of a variable
- dynamic instance of a variable
- lexical variable

Let me say what ELISP does, as I think it is fairly reasonable. A symbol (literal atom) is really a record, with

several fields. Among these fields are its value, and its function definition. These can be set and looked at independently, the value by `set`, `setq`, and `eval`, the function definition by `alias` and `fundef` (which should be changed to `getd` and `putd`, for INTERLISP compatibility). When an atom occurs as an argument, only its value is looked at. The expression in the functional position is supposed to be a `lambda` or `subr`. If it is not, `fundef` and `eval` are used repeatedly as necessary until a `lambda` or `subr` is found. In compiled code, variables not declared special are handled as lexicals. Function calling works the same as in the interpreter, except for one case (which will probably be fixed eventually): `(foo 'bar)`, where `foo` is a special variable bound to something other than the name of a function or special variable (e.g., to a `lambda` expression or an expression that must be evaluated to find the functional object). As far as I know, the only difference between `quote` and `function` is that `'(lambda ...)` would give you the lambda form unchanged, whereas `(function (lambda ...))` (normally abbreviated `(f:l ...)`) would compile the lambda as a function with a `gensym`'ed name, and would be equivalent to `'`*gensym*. It is unclear from your documentation whether you are doing something more complex or not.

I do have one suggestion to make, based on experience with ELISP, and that is to get rid of the concept of fsubr's. I haven't done this yet, but my view is now that there are only four kinds of functional objects (ignoring closures, labels, etc.):

	args bound to names	args put on stack
interpreted	normal `lambda`@\f[lambda] in `lexpr` format	
compiled	`subr`	`lsubr`

(Actually the `lexpr/lsubr` is probably a hack that should be flushed.) The distinction among `expr`, `fexpr`, and `macro` seems fairly clearly to involve the way arguments are bound, not the object itself. To generate code for a function, all you have to know is that it received its arguments in accumulators 5, 6, ... You don't have to know whether it is an `expr`, `fexpr`, or `macro`. (Of course to compile a call to something, you obviously have to know what kind of thing it is.) Thus I would propose that `getd` and `putd` should put forms of the following sort into the function definition cell:

```
(expr lambda (...) ...)
(fexpr lambda (...) ...)
(expr subr #o1001453 2)    ;for a subr of 2 args
(fexpr subr #o1001576 1)
```

In that case, `(function (lambda ...))` should compile code for the `lambda`, and compile into `'(subr` *addr #args*`)`.

HIC: [29.9] Does `subrp` have to be called that? If so, put a hysterical (historical) note here.

4.2. Equality Predicates

***** Issue 30: Definition of equalp**

RMS: [32.0] I can't tell from the documentation whether `equalp` calls itself or `equal` on its recursions. The wording makes it sound as if it looks like:

```
(cond ((stringp x) ...) ((numberp x) ...) (t (equal x y)))
```

I hope not, because `(1)` and `(1.0)` ought to be `equalp` if `1` and `1.0` are.

GLS: This was a result of laziness and confusion on my part. `equalp` is intended to call itself and not `equal` when comparing recursively.

**** Suggestion:** Correct the documentation of `equalp`.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

4.3. Logical Operators

***** Issue 31: Judgements of style**

HIC: [33.3] [The style note on use of `and`] is a matter of *your* taste. I would leave this value judgement out.

GLS: There are notes of style elsewhere in the document. Should they be eliminated also?

MOON: [44.0] Some of the style judgements on use of `cond` are debatable.

**** Suggestion:** Eliminate notes on programming style?

N?

**** Responses**

MOON: Y	RMS: N!	SEF: N!	GINDER: N!	DM: N!
DLW: N!	HIC: Y	WLS: N!	CHIRON: N!!	
ALAN: Y!	GLS: N	RPG: N	DILL: X	

RMS: Tone them down. Do mention alternatives that you think might be preferable, so that people will not use the "bad" one out of ignorance of the other one; but don't pressure them.

MOON: Defer the writing of a book on COMMON LISP style until after the language is defined.

GLS: Touche!

DM: Well, whether these really belong here or not depends on what this document really is. As noted above, it seems rather schizoid; is it a user's manual or a language definition (no matter how informal)? If the former, the comments belong there; if the latter, they don't (and then, neither do the historical notes).

SEF: I think comments like the note on **and** belong in a separate document. But less controversial style notes, especially where they affect portability, can stay in.

DILL: The note on programming style here is necessitated by the use of the example, which is there to illustrate the behavior of **and**. If you use a disgusting example to illustrate the semantics of a program construct, you should also explain why it's the wrong thing to do.

WLS: Maybe soften them a bit so as not to offend.

Chapter 5

Program Structure

5.1. Constants and Variables

5.1.1. Reference

DLW: [36.5] There needs to be a paragraph of English explaining the concept of "functional interpretation of a form".

CLH: [November] I would like to try to do COMMON LISP on TOPS-20 if I can get enough COMMON LISP code that all I have to do is an assembly language kernel. (But we are probably talking about next summer before I could put significant time into it.) The only two things I see that would give me trouble are closures (though I can't really comment on them since I haven't yet read the details, if they are there at all) and lexical binding in the interpreter. We tried lexical binding in the interpreter in an experimental LISP implementation, for the same reason you have stated, to provide compatibility with the compiler. It was considered to be an unmitigated disaster. Some of this is due to the implementation, as it was a warmed-over R/UCI LISP, where a rewrite was really needed. But our users generally considered the cure to be worse than the disease. What they really want is a flag to cause all variables to be special in the compiler, which gets compatibility in a different way. If I announced that I were working on a LISP that had lexical binding in the interpreter, I suspect I would be tarred and feathered. I am curious what implementation you have in mind. Obviously we know a way to do it (since we have a running LISP interpreter that works (ignoring a large number of bugs) that way). But it was a real kludge, and maybe you have thought of something better.

5.1.2. Assignment

***** Issue 32: Value returned by `psetq`**

HIC: [38.1] Why does `psetq` return `()`? Why not be same as `setq`? A little random, I suppose, but surely more useful than `()`.

GLS: In both MACLISP and Lisp Machine LISP, the value returned is the *first* value computed, not the last. This is a consequence of the implementation:

```
(psetq x1 v1 x2 v2 ... xn vn) ==>
   (setq x1 (prog1 v1 (setq x2 (prog1 v2 ... (setq xn vn)...))))
```

The Lisp Machine LISP manual [Weinreb 81a] does not specify what the returned value is. However, the

following comment appears in the Lisp Machine LISP code for `psetq`:

```
;;; Note that the return value of PSETQ is -not- guaranteed.
```

** Suggestion: Retain specification that `psetq` returns `()`. **Y?**

** Responses

MOON: N!		SEF: Y!!	GINDER: X!	DM: Y!
DLW: Y!!	HIC: N	WLS: Y!	CHIRON: Y!!!	
ALAN: X!	GLS: Y!!!	RPG: Y!	DILL: X	

ALAN: I would vote that the return value from `psetq` be undefined.

GINDER: How about returning `t` (or the "truth value")? This seems to more intuitively indicate success (in some sense) to the naive user more than `( )` does. Admittedly this is not all that important.

MOON: Specify that it is undefined.

DILL: `psetq` should return the first argument. In general, we should minimize unspecified things when they will be user-visible.

RMS: [39.3] `makunbound` in Lisp Machine LISP does not act on local variables.

GLS: Okay, I was misinformed. I do believe that `(value-cell-location 'foo)` does "work" though (does it not?), even if `foo` is local in compiled code (although

```
(let ((x 'foo)) (value-cell-location x))
```

does not "work" in the same way).

5.2. Function Invocation

MOON: [39.6] Applying of special forms is used internally in things like `trace` and `advise`. I'm not sure what ought to be said here; certainly the naive reader can only be confused by it. But it's wrong to provide no way at all to do it.

ALAN: [40.0] Lisp Machine LISP calls this `[funcall*]` "`lexpr-funcall`". Lousy name.

GLS: Which is the lousy name?

MOON: `funcall*` requires at least two arguments (*args* may not be null).

5.3. Simple Sequencing

***** Issue 33: Declarations in every implicit `progn`**

MOON: This is somewhat bogus, since everything else with declarations in its body applies those declarations to the variable bindings it is doing around the body.

DLW: Indeed. In fact, the compiler has to go through pain to make this work. I see how, by trying to put this into `progn`, you are trying to simplify things, but it doesn't work!

RMS: MACLISP-style local declarations are not as good as `local-declare` for two reasons. One, they are bad-nesting, in that a subconstruct modifies the meaning of the construct that contains it. Two, they are inconvenient to check for; each function which is an implicit `progn` has to check for the existence of declarations. `local-declare` need be handled only by the definition of `local-declare` itself. Three, what about the *cdr* of a `cond`-clause? Does it allow local declarations there? It is an implicit `progn`. What about the *cddr* of an `if`? The fact that MACLISP-style local declarations resemble ALGOL is a point *against* them. Otherwise, why not get rid of `let` and provide a `bind` function which can be used at the front of any implicit `progn`?

SEF: We are not proposing that a declaration is allowed by any `progn`, are we? I think the proper characterization is that the declaration can be the first thing in any binding form. I admit that the nesting is backwards, but allowing arbitrary local declares means that the declaration context is neither global nor associated with a particular function. This makes big trouble for the scheme to make the interpreter handle locals and specials as the compiler does.

MOON: So how am I supposed to do:

```
(local-declare ((special ...))
  (defun func1 ...)
  (defun func2 ...)
  ...)
```

if there is no `local-declare`? Is there some awful variant of `(progn 'compile ...)`?

GLS: Okay, I was trying to do something here; maybe it's not what we want, but I still think it's a pretty good idea. Yes, the intent was to draw a closer parallel between `progn` and the ALGOL **begin-end** construct, which allows declarations at the front. (I feel that COMMON LISP will be the better for it whenever, other things being equal, it can parallel existing programming-language ideas, features, and culture; but *only* if other things are equal, of course!) Granted, as DLW points out, this doesn't really simplify the implementation; however, I suggest that it may nevertheless simplify the user's model of the language. The rule is: any implicit `progn` may have declarations, and if this implicit `progn` is the body of some construct that binds variables then the declarations apply to those bindings too. As RMS points out, they aren't properly nested in the S-expression sense. However, they do preserve the property that a `defun` stays against the left margin, and doesn't have to be embedded in a `local-declare` (which I find a bit ugly). On the other hand, one can get the effect of any `local-declare` by writing, for example:

```
(progn (declare (...))
       (declare (...))
       body)
```

for

```
(local-declare ((...) (...))
  body)
```

For MOON's specific case, one might write:

```
(eval-when (load eval)
   (declare (special ...))
   (defun func1 ...)
   (defun func2 ...))
```

if you really want to. (Recall that `eval-when` has an implicit `progn` for its body.) Now, if the `progn` is not part of a binding form then non-pervasive declarations don't make sense. Another possible use is:

```
(cond ((numberp x)
       (declare (type (mod 8) x))
       ...))
```

which allows one to say that if `x` is a number then in fact it is known to be one of the integers 0 through 7. This could be done with a `local-declare` too, but with greater nesting depth. I agree it is not very "Lispy", but it does allow easier deletion, insertion, and commenting-out of declarations using most text editors (a weak argument, granted). I'm not rabid about this idea, but I did want to point out the motivation and few merits.

** **Suggestion:** Shall this idea of declarations in every `progn` be retained?

** **Responses**

MOON: Y	RMS: N!	SEF: N!!!!	GINDER: N!	DM: N!
DLW: N!!	HIC: X	WLS: Y		
ALAN: Y!	GLS: Y	RPG: Y!!	DILL: Y	

RMS: [Several paragraphs by RMS follow.] I think that local declarations whose contexts are neither global nor the entire function are desirable, and I have designed how to make them work in the new lexical Lisp Machine LISP interpreter I am writing.

I think that a `local-declare` at top level surrounding multiple functions is indeed ugly. The right solution, I think, is to provide an alternative to `defun` that provides a place to put the local declaration, and also an alternative to `lambda`. The scope of a local declaration (no matter how notated) should be lexical, so that strictly speaking `(local-declare (...) (defun ...))` should not work! And neither should `(progn (declare ...) (defun ...))` if your syntax were used. I think that `defun-declare` and `lambda-declare` are needed, as a way to sandwich a declaration lexically outside the lambda variables but within the new lexical context of the function or lambda. I plan to make this work in the new Lisp Machine LISP interpreter too.

There is some value in having a declaration insertable at the front of a `cond` clause, but why only at the front? Why not anywhere within an explicit `progn`, and applying to all the rest of it? And it is equally convenient, for the same reasons, to have a `bind` function to introduce a new local for the scope of the implicit `progn`, because a `bind` is easier to insert or delete, or comment out, than a `let` would be. So if we have this kind of declaration, we should have `bind` as well. `bind` is to `let` as `declare` is to `local-declare`.

Just suppose that `bind` were allowed in a `cond` condition, and would apply for the duration of the `cond` [GLS: clause?]!

```
(cond ((symbolp (bind x (foo)))
       (print x)))
```

or

```
(and (listp (bind x (foo)))
     (eq (car x) 'foo)
     (listp (bind y (cdr x)))
     do-something)
```

GLS: These last examples are not analogous because the `bind` forms do not appear at the top level of the implicit `progn`. However, the analogy between `declare` and `bind` is in general well-taken. I don't understand why you assume that `(local-declare ... (defun ...))` would not work: the `defun` is lexically within the `local-declare`. Apparently we disagree about the precise effects of a "new lexical contour". I assume that declarations are precisely as pervasive as bindings, and would also expect

```
(let ((x 43))
  (defun add43 (y) (+ x y)))
```

to define `add43` to be a function that adds 43 to its argument; therefore I would expect that `defun` within the `local-declare` to work.

MOON: Isn't there a big confusion here between declarations about variables and declarations about values? (I'm not sure what would be better, though)

SEF: This whole issue of local declarations needs to be totally re-thought, especially how it will interact witht the notion that specials and locals are observed by the interpreter. Arbitrary nesting of `local-declare` makes this so hairy that we may have to abandon this goal for the interpreter. Generally, I favor the use of a declaration form just inside each binding form in order to get declarations out of the variable or lambda list. However, I don't like the idea of putting declarations in non-binding `progn`'s. I disagree strongly with the idea that we should emulate ALGOL.

5.4. Environment Manipulation

***** Issue 34: Macros or special forms?**

MOON, DLW: Whether `let` (for example) is a macro or special form can be left up to the implementation. As far as this definition is concerned, `let` is a special form.

GLS: The same might be said of `push` or `setf`. While in some sense the user shouldn't care, the user who is writing a program-transforming programn does care in the following way. A program (such as the compiler, or perhaps an indexer (MASTERSCOPE?)) which operates on other LISP programs does not have to know about any particular macro, but only how to perform macro-expansion. On the other hand, it must know about each and every non-macro special form specially: it must understand `quote`, `cond`, `lambda`, `return`, and so on, all as primitives. The smaller the set of such primitive special forms, the easier the task of writing such a program. Thus I would urge a policy of leaning towards documenting forms as macros wherever possible.

** Suggestion: Document `let` as being a special form rather than a macro, and more generally document

most system-provided macros as special forms also.

** Responses

MOON: Y	RMS: Y!	SEF: N	GINDER: N!	DM: X!
DLW: X	HIC: X	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: N	RPG: Y	DILL: Y	

DM: I gather that the reason for preferring "special forms" to "macros" is that macros are considered something confusing for the novice Lisper. Surely special forms are at least as confusing! You can't pass special forms around as functional values, and their semantics are very complicated: when, if ever, do you evaluate their arguments, and in what environment, etc. Perhaps you need two levels of tags: a general one which simply says this guy is a non-function, and one for the more sophisticated user which promises that this is a macro, so he needn't worry his little head about it when he writes a program transformation tool. Absolutely, keep the number of special forms to a minimum. Twenty sounds like about the right sort of upper limit. You don't have to specify the definition of the macro (that's up to the implementor), but you should guarantee that programs that grind on LISP programs don't all have to know about 273 different special forms. And the definition of COMMON LISP has to spell out just which 20 these are, so these programs are portable.

DLW: I understand GLS's point, but the counterpoint is that it is desirable to give the implementor flexibility regarding whether to implement various special forms as macros or not. We recently changed `let` from a macro to an fsubr, for example. Tough issue.

GLS: Why not have it both ways? Presumably an implementor chooses to use an fsubr rather than a macro primarily to speed up the interpreter (except for that irreducible few such as `quote`). Typically the compiler uses macros for `do` and `let` even when the interpreter uses an fsubr; the only problem is that the fsubr definition is the "public" one, and the compiler has a private stash of macros. I suggest that things be arranged so that the fsubr and macro can coexist in the function cell, combined into a single object. If the interpreter inquires of this object "are you an fsubr?", it says "yes"; if anyone inquires of it "are you a macro?", it also says "yes". The interpreter simply checks for fsubr-ness before checking for macro-ness; all other comers will get the macro definition. Given this strategy, we can define all but a very small number of special forms to be macros as far as the user manual is concerned; the implementation is free, however, to provide fsubrs for as many of these macros as it chooses.

***** Issue 35: Variables without *init* forms in a `let`?**

MOON: What about variables alone without inits in `let`? People seem to use that a lot in Lisp Machine LISP.

GLS: Pretty much the only use of that is to be able to `setq` it later. I find such programming style (let me not say "abominable") disheartening. Such use undermines what I understand to be the main intent of a `let`: "let this variable have this value, throughout its scope". I suspect such lone variables usually get tossed in as afterthought patches, to avoid the work of introducing a new `let` where appropriate. (*End of flame.*)

GLS: Should `let*` also permit such lone variables?

** **Suggestion:** Allow a variable to appear in a `let` where normally a variable-value 2-list appears, meaning to let that variable have the value ( ).

Y?

** Responses

MOON: Y	RMS: Y!!!	SEF: Y!!!	GINDER: N!	DM: Y!
DLW: Y!!	HIC: Y!	WLS: Y!	CHIRON: N!	
ALAN: Y!	GLS: N!!	RPG: Y!	DILL: X	

RMS: I disagree with GLS's taste about binding a variable and `setq`'ing it later. In addition, there are cases where this is the only way. If the `setq` is inside a condition, and the value is wanted in one or both arms of the conditional, there is no way to bind the variable where the value is produced. Also, I have used this in lots of code and I don't want to flush it.

DILL: If you allow uninitialized variables in a `let`, they should be *unbound* until someone explicitly `setq`'s them. Unspecified `&optional` parameters should also be *unbound.*

DLW: You may not have the same variable bound twice in a `let`, but you may in a `let*`.

GLS: I'll clarify the documentation.

*** **Issue 36: The "obvious" macro definition of `let*`**

??? Query: [42.9] There is a problem with the interaction of this definition of `let*` with declarations; if one does things in the obvious manner, declarations cannot apply to any variables except *warm.* This seems unfortunate. Any suggestions?

DLW: Indeed! This definition is no good. I went through this whole thing while writing the L-machine compiler. My suggestion is that you not define it this way.

SEF: I vote to flush the description of `let*` in terms of nested lambda-expressions.

MOON: Don't call it a macro.

** **Suggestion:** Flush the description of `let*` as a macro.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: X!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

5.5. Conditionals

***** Issue 37: The `select` special form**

MOON: [45.4] Where did `select` go?

GLS: I think I thought that

```
(select foo ((si:gross-out si:confusion) ...) ...)
```

might be replaceable by

```
(selectq foo ((#,si:gross-out #,si:confusion) ...) ...)
```

but perhaps I am wrong.

**** Suggestion:** Add `select` as in Lisp Machine LISP.

**** Responses**

MOON: N	RMS: N!	SEF: N	GINDER: Y!	DM: N!
	HIC: Y	WLS: Y	CHIRON: Y!	
ALAN: N!	GLS: N	RPG: Y!		

RMS: I think `select` is ugly. I'm not sure whether it is very useful. The main uses seem to be in inner parts of the Lisp Machine LISP system where it is necessary to compare numeric byte values against names for particular values. The `#,` alternative would work for this. If there are important applications I don't know of, that exist for users in general, put in `select`, but my expectation is that there are not.

MOON: I don't think it would hurt to flush `select`, it seems to have been a mistake. Also see issue 42.

***** Issue 38: Optional predicate function to `select` and `selectq`**

MOON: [45.4] I would suggest that `selector` be flushed, and both `select` and `selectq` allow an optional predicate between the key and the clauses. It defaults to `eql`.

GLS: I don't understand how this could be disambiguated:

```
(selectq foo (function ...) ...)
```

sure looks like I'm checking whether `foo` has `function` as its value. Could we see a more concrete proposal?

**** Suggestion:** Table this idea pending a more concrete proposal.

Y

**** Responses**

MOON: X		SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!!	DILL: Y	

MOON: I was assuming the predicate would only be a symbol, not a functional expression. That was dumb, of course.

***** Issue 39: What does a null `selectq` clause return?**

ALAN: [45.6] Lisp Machine LISP returns `t` [when a `selectq` clause has no consequents, because the clause expands into the `cond` clause:] `(cond ... ((eq ...)) ...)`.

GLS: Unfortunately the "test" is not apparent to the (human) reader.

**** Alternatives:**

A. Require a null `selectq` clause to return `t`.

B. Permit a null `selectq` clause to return an arbitrary value.

C. Status quo: require a null `selectq` clause to return `()`.

**** Responses**

MOON: B	RMS: C!	SEF: AC!		DM: C!
DLW: C	HIC: A	WLS: B	CHIRON: C!!	
ALAN: B!	GLS: C	RPG: C!	DILL: A	

***** Issue 40: What numbers are acceptable to `selectq` and `caseq`?**

MOON: [46.3] Should `selectq` and `caseq` accept any numbers, or only integers?

GLS: Well, rationals are exact, but the presumption is that floating-point numbers are not. My inclination would be to discourage bugs by not permitting floating-point numbers. And what of complex numbers?

**** Alternatives:**

A. Permit only symbols, `()`, and integers in `caseq` and `selectq`.

B. Permit symbols, `()`, and rationals.

C. Permit symbols, `()`, and complex rationals.

D. Permit symbols, `()`, and all scalars.

E. Permit symbols, `()`, and all numbers.

**** Responses**

MOON: C	RMS: A!	SEF: AB!!	GINDER: C!	DM: A!
	HIC: B	WLS: B!	CHIRON: B!!	
ALAN: C!	GLS: X!!	RPG: C!	DILL: X	

RMS: I don't think types which attempt or pretend to model continuous data are useful for this sort of

comparison. Allowing rationals is not too bad, as long as `4` equals `4/1`.

MOON: You forgot characters.

GLS: I forgot that, in the all above suggestions, characters should also be permitted! I will interpret the votes as if this had been understood.

DILL: `selectq` should accept any LISP object, and use `eql` comparisons with the clauses. Ambiguity problems will have to be dealt with anyway.

***** Issue 41: Should `selectq` and `caseq` signal an error if no clause succeeds?**

RMS: I don't think that a `selectq` ought to err if the tests all fail. It is true that it is useful to be able to get that behavior, but that is true of `cond` as well; so whatever is done to make it easier to get that behavior when it is desired ought to be something which applies also to `cond`, especially since `selectq` is strictly less flexible, so a program written using `selectq` may have to be changed to use a `cond`. I find that I write `selectq` rarely, and much more often I write a `cond` most but not all of whose clauses could be turned into clauses of a `selectq`. Here is a way of providing the feature conveniently for use in `selectq` and `cond`: use

```
(error-restart (cond ... (t (error-in-value x "info"))))
```

which says that if the user continues from the error, set `x` to the value he supplies and then throw back to the `error-restart`.

SEF: My vote is to flush the whole silly thing and just let these guys return `( )` if they drop through. I just can't get excited about having an easy way to restart with a different key; I can always do this from the debugger if I am running interpreted code. On the other hand, I don't violently object to Guy's scheme, I just don't like it much.

MOON: I don't at all like the proposed change to `selectq` which signals an error if it drops off the end (instead of returning `( )`). I would much prefer that the default be left the way it is, and some uniform syntactic way be found to tell `cond`, `selectq`, `caseq`, and `typecaseq` (and any other similar forms) to do this. This could be something you write in place of a clause, or it could be a different function name (e.g., prefixing it with the letter `e`). RWK (Robert W. Kerns) suggests that if the last clause is a string (instead of a list), then that is the error message, and if that clause is reached an error is signalled. This seems pretty flavorful to me. The important point is that the error is proceedable and if proceeded, starts over at the top with a new value for the key (except in `cond` where there is no key) (if it is a variable, does the variable change? This is desirable, at least for `typecaseq`, isn't it?), which is a little hard for the user to write himself. The other important point is that it be consistent across all the "dispatching" special forms.

SEF: Looks like this is nearly unanimous. Just let these fall through.

GLS: The idea from RWK isn't bad, but will it grind well? Ideally one could provide a `format` string, with arguments (maybe the failed thing would be an implicit first argument). How about a keyword instead?

```
(caseq (car x)
   ((2 3 5 7) 'prime)
   ((1 4 9) 'square)
   ((6) 'perfect)
   ((8) 'cube)
   (error "~S not a known digit"))
```

Note that `error` and `otherwise` are mutually exclusive options. This may need more polishing. I'm not sure that `cond` really belongs in this class with the others; `cond`, unlike the others, does not have a single item which is "obviously" at fault about which to complain and replace for a retry. I would definitely recommend some simple way (other than programming it yourself, which is not simple) to get a correctable error that can retry the dispatch. It is very easy to cause bugs by blithely assuming that one of the cases must hold, and when an error check is put in it is harder to make it correctable than not.

** Alternatives:

A. Forget the whole idea; require the user to program a retry for failed dispatch constructs.

B. Let failed dispatch constructs automatically signal a correctable error.

C. Let failed dispatch constructs produce `()`, but let a keyword `:error` (an alternative to `:otherwise`) take an error message and use that to signal an error.

** Responses

MOON: X	RMS: AC!!!	SEF: A!!	GINDER: C!	DM: A!
	HIC: A!	WLS: C!	CHIRON: C!!!	
ALAN: C!	GLS: BC!!	RPG: C!	DILL: A!!	

RMS: I think it is important to provide an easy way to request a correctable error, and that my previous suggested way is not easy enough, but it is important to include `cond`. Yes, the `cond` does not intrinsically imply that there is one variable being examined, but that is true nonetheless for many uses of `cond`. Here is my way: define

`(bad-value` *format-string varname . additional-args*`)`

to report the error, allow the user to set the specified variable, and if he does so, return to the beginning of the innermost containing `selectq`, `caseq`, `typecaseq`, *or* `cond`.

MOON: I don't understand what "will it grind well?" is worrying about. There is no reason to provide elaborate error facilities in `selectq`, such as `format` strings, since nothing stops you from writing an explicit call to `error` (or whatever the name of the general error function is). The goal is to have something which is so simple and easy and inexpensive-looking that you don't fail to provide for the error case. That's why I'd almost prefer to call it somthing like `eselectq`, but probably putting a string in place of a clause is better.

GLS: The point is to make it easy not just to signal an error, but to make the error correctable and retry the dispatch in the obvious manner. Calling `error` isn't enough to do this.

MOON: The syntactic words for `selectq` (mainly `otherwise`) do not have colons any more than names of special forms do.

***** Issue 42: The "q" in "typecaseq"**

MOON: By the way, why does `typecaseq` have a "q" in its name? Oops, I got faked out by the inconsistency I myself am responsible for (at least for inheriting from INTERLISP), namely that in these functions "q" means "quoted" whereas in most other functions with a suffix "q" it means "eq". Do we want to do anything to remove this inconsistency?

GLS: I'd love to get rid of all the "q" names by flushing non-"q" constructs and then removing the letter "q" from the names. What primarily prevents this is `select` and `selectq` in Lisp Machine LISP.

**** Suggestion:** Remove the "q" from the names of `selectq`, `caseq`, and `typecaseq`.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
	HIC: N!	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

MOON: How about making a new form

`(select` *predicate item clause* `...)`

and define `selectq` as an abbreviation for `(select 'eql ...)` or `(select 'eq ...)`, and call the type-dispatch one `typecase` or `typeselect`. It doesn't make sense to give a user-supplied predicate for `caseq`, since it's whole point is that the system decides whether the predicate is `eq` or `=`. I think it's all right to get rid of the Lisp Machine LISP `select`. This issue needs more thought, however.

5.6. Iteration

5.6.1. General iteration

***** Issue 43: Initial values for "uninitialized" do variables**

ALAN: [47.7] Well, [if the *init* form is omitted for a `do` variable and] if you declare it `fixnum` then it defaults to 0 in compiled code. Perhaps it is really undefined?

ALAN: [56.4] `prog` has the same problem as `do` vis-a-vis declarations. [That is, should a `prog` variable without an *init* form be considered undefined, because of a problematical interaction with type declarations?]

GLS: Well, it seems to me that that hack to `do` in MACLISP was intended to compensate for the fact that `prog` did not allow explicit init forms. I don't think we want local variables with no values floating around. Better to require the user to initialize the variable explicitly to 0 if he is going to declare it `fixnum`.

**** Suggestion:** Retain the currently documented (and slightly MACLISP-incompatible) semantics: no *init*

forms *always* means the initial value is ( ). If that implies a type conflict, the compiler should complain.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: X!	DM: Y!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: N!	GLS: Y!!	RPG: Y!	DILL: Y	

GINDER: Require an *init* form to be specified, as in the `let` proposal.

*** Issue 44: What about the MACLISP (`do` *varspecs* ( ) ...) syntax?

DLW: [48.6] [An end-test clause of ( ) meaning one iteration has been flushed as a crock.] But it is upward-compatible, right? I'd like to keep this stupid crock, for compatibility, though I could be convinced [otherwise].

GLS: Well, I wouldn't want to encourage it in new code. What it does is effectively let the user have `prog` with initialized variables. But it was easier to modify `do` this way in MACLISP than to modify `prog`.

** Alternatives:

A. Make the single-iteration do syntax part of COMMON LISP.

B. Do not make it part of COMMON LISP, but add an implementation note that implementations may also accommodate this syntax for the sake of MACLISP compatibility.

C. Status quo: forget the single-iteration do syntax.

C?

** Responses

MOON: C	RMS: C!	SEF: C!!	GINDER: C!	DM: C!
DLW: B!!	HIC: C	WLS: C!	CHIRON: C!!!	
ALAN: C!	GLS: B!	RPG: C!!	DILL: C!!	

GLS: For the record, I dug up the original announcement of this feature from the old `LISP ARCHIV` file:

```
MONDAY  OCT 15,1973  FM+4D.3H.22M.30S.  LISP 623  - GLS -
THERE IS A THIRD FORMAT FOR DO NOW:
      (DO <VAR-SPECS> NIL
          BODY...)
   THIS IS EXACTLY LIKE A NEW-STYLE DO (MULTIPLE INDICES)
   EXCEPT THAT THE PREDICATE/RETURN VALUE CLAUSE IS NIL.
   (NOT (NIL), BUT NIL!!!) THE MEANING OF THIS IS THAT
   THE VARIABLES SHOULD BE INITIALIZED AND THE BODY
   PERFORMED EXACTLY ONCE. NOTE THAT STEPPERS FOR THE VARIABLES
   ARE ILLEGAL IN THIS MODE, SINCE STEPPERS CAN NEVER BE
   EXECUTED. EXAMPLE:
      (DO ((A 0) (B 1)) NIL
          (PRINT 'A=)
          (PRIN1 A)
```

```
        (PRINT 'B=)
        (PRIN1 B))
PRINTS THE FOLLOWING:
    A=0
    B=1
AND THEN RETURNS NIL. IN THIS WAY ONE GETS THE EFFECT OF
THE FAMOUS "PROG WITH INITIALIZED VARIABLES" FEATURE.
```

***** Issue 45: Result returned by `do` for a singleton *end-test* clause**

MOON: [48.2] [On `do` returning the value of the *end-test* in a singleton *end-test* clause:] Many users will be unhappy at this, since their code will stop working with no error indication from the syetem. This is just the sort of change which causes the most weeping, wailing, and gnashing of teeth.

MOON: The reason *end-test* is not returned [in MACLISP and Lisp Machine LISP] was probably only that the current behavior is much easier to implement with a macro.

ALAN: [48] [For total compatibility, if you adopt the rule that the value of the *end-test* is returned if there are no result forms for a `do`, then it really takes *four* pairs of parentheses to convert old-style to new-style: `(do` *var init step test . body*`) ==> (do ((`*var init step*`)) (`*test*` ()) .` *body*`)`.]

GLS: Well, many users have complained about the failure to behave like a `cond` clause. Does any code really depend on the MACLISP/Lisp Machine LISP behavior? (We'll probably never know.)

**** Suggestion:** Make `do` with a singleton *end-test* clause return `()`, as in MACLISP and Lisp Machine LISP, rather than the value of the *end-test*.

**** Responses**

MOON: Y	RMS: N!!!	SEF: Y!	GINDER: N!!!	DM: N!
DLW: Y!!!	HIC: Y	WLS: N!!!	CHIRON: Y!	
ALAN: Y!	GLS: N!	RPG: N!	DILL: N!!	

RMS: I know I have written lots of code that depends on this. Every time I write some sort of search function, it tends to return a value of `()` with a singleton end-test clause in a `do`. The end-test and value forms of a `do` are *not* really a *cond* clause. They are a condition followed by things to put inside a `(return (progn ...))`.

GLS: I believe that RMS, at least, meant to vote yes rather than no, probably confused because the title of the issue states the case backwards from the way the suggestion states it. This was bad writing on my part, for which I apologize.

GLS: Here's the evidence on whether or not the *endtest* clause of a `do` is "really" a `cond` clause. Not only [Weinreb 81a], [Weinreb 81b], and [Weinreb 78], but also [Moon 74] note the resemblance of this clause to a `cond` clause. The original announcement of new-style `do` in MACLISP was as follows (this and following quotations were culled from the file `LSPMAI;LISP OARCHI @ MIT-MC`, which contains the oldest parts of

the old LISP ARCHIV file):

```
3/17/72  - JONL -
AN EXPANDED FORM THE MULTIPLE-INDEX DO HAS BEEN ADDED TO THE SYSTEM
        (DO INDEXLIST (ENDTEST RETURNVALUE) DOBODY)
THE ITEMS OF AN INDEXLIST MAY BE OF FORMS:
          (X XINIT XSTEPPER)    WHERE X IS INITIALIZED TO XINIT
                                AND MODIFIED AFTER EACH PASS THROUGH
                                DOBODY BY (SETQ X XSTEPPER)
          (X XINIT)             X IS INITIALIZED TO XINIT,
                                AND MAY BE USED LIKE A PROG VAR
          (X)                   LIKE (X NIL)
AN ALTERNATE FORM FOR (ENDTEST RETURNVALUE) IS (ENDTEST), WHICH IS
TAKEN TO BE (ENDTEST NIL).
```

Note that no mention of an implicit `progn` is made here. The next note about `do` is:

```
11/14/72  - JONL -
DO AND IOG HAVE SLIGHTLY MORE GENERAL FORMATS NOW.
    (IOG C E1 E2 . . . EN) WORKS AS BEFORE, BUT ALL OF E1 TO EN
   ARE EVALUATED, WITH THE VALUE OF EN BEING RETURNED.
    (DO ((Z INITIALVALUE STEPPERFUN) . . .)
        (ENDTEST E1 E2 . . .EN))
   WORKS LIKE THE USUAL EXTENDED DO FORMAT, EXCEPT THAT THE ENDTEST-
   RETURNVALUE PAIR NOW LOOKS LIKE THE GENERALIZED COND CLAUSE.
```

Note here that the new format is simply described as being like a `cond` clause, and the result of `(ENDTEST)` is not explicitly stated.

***** Issue 46: Include `loop` in the COMMON LISP core**

ALAN: [49.5] [Regarding the use of `nreverse` as a "standard idiom" with `do`:] Use `loop`!

MOON: I would like to see `loop`, extended to support sequences, adopted as a standard package in COMMON LISP, with roughly the same status as defstruct. Not everyone likes this sort of syntax, but it does make sequence iterations much easier to write and a clear readable style for using it seems to be evolving.

GLS: Is this style a matter of syntax only (indentation, for example)? Or does the style involve avoidance of certain features or combinations of features? I've known some users to be frightened off by `loop`, even though the simple cases are indeed easy. (`defstruct` has the same problem, as does `format` (*mea culpa*).)

SEF: From what I have seen of `loop`, I consider it an abomination and a blight on mankind. It belongs in the yellow pages, along with CGOL, MLISP, and the FORTRAN simulator.

GLS: I'd be happier with `loop` if there were some defined intermediate form a program could easily manipulate, halfway between the surface syntax (which requires some hair to parse) and the full expansion (which is a hairy `prog` whose looping structure is difficult to analyze from scratch).

GLS: For those unfamiliar with the Lisp Machine LISP `loop` construct: the documentation is 24 pages long, because the construct is quite complex and quite powerful. Here is an example of its use (in the Lisp Machine

LISP definition of the `psetq` macro):

```
(DEFMACRO-DISPLACE PSETQ (&REST REST)
  (LOOP FOR (VAL VAR) ON (REVERSE REST) BY 'CDDR
        WITH SETQS = NIL WITH PSETQS = NIL
        DO (IF (AND (NULL PSETQS)
                    (LISTP VAL)
                    (MEMQ (CAR VAL) '(1+ 1- CDR CDDR))
                    (EQ (CADR VAL) VAR)
                    (NOT (MEMQ VAR SETQS)))
               (SETQ SETQS (CONS VAR (CONS VAL SETQS)))
               (SETQ PSETQS (CONS VAR (CONS VAL PSETQS))))
        FINALLY
          (SETQ PSETQS (PSETQ-PROG1IFY PSETQS))
          (RETURN (COND ((NULL SETQS) PSETQS)
                        ((NULL PSETQS) (CONS 'SETQ SETQS))
                        (T '(PROGN ,PSETQS (SETQ . ,SETQS)))))))
```

** **Suggestion**: Add `loop` to the core of COMMON LISP.

N?

** **Responses**

MOON: Y	RMS: X!	SEF: N!!!!	GINDER: N	DM: X!
DLW: N!	HIC: N!	WLS: N!	CHIRON: N!!!	
ALAN: Y!	GLS: N!!	RPG: N!	DILL: N!!!	

RMS: I hate loop, and find it unclear. But given that it exists, we might as well tell people how it works so that they don't create incompatible ones. It should not be part of the core, but could be documented as an add-on package.

HIC: I never did like `loop`...

DM: [Several paragraphs by DM follow.] `loop` certainly doesn't belong in a core, any more than `defstruct` does. But the functionality of `loop` should be avialable somewhere, and standardized. It eliminates the need for all those zillions of different mapping functions which no one will remember.

The current definition of `loop` leaves much to be desired, however. The semantics are just great, but the syntax is too much. It's ridiculous to have to really parse LISP. That's one of LISP's great strengths. I'd much rather the various iterators like "for U in L" were each their own little sublist (maybe "`( in u l)`"), and they were themselves collected into a list of some sort. The only place for a keyword is in the function position right after a left parenthesis. It would be nice if the "body" of the thing were an implicit `progn`, too. Not sure how to specify the means of handling the return value (a al `do`, `collect`, etc.).

In general, sticking syntax into LISP is a bad idea. Another place where COMMON LISP sticks too much syntax in is declarations (which should follow the usual scope rules instead of going up and diddling their parents in the tree). Another case of too much syntax was a suggestion for function types incorporating the diphthing ->. Bletch.

Syntax is not just ugly. Program transformation tools or programs which emit LISP code are much happeirs dealing with something which feels like a parse tree. And mixing semantics and syntax just leads to trouble

whenever you try to extend something. That's a frequently recurring problem at Utah where folks seem to like laying ALGOL syntax on top of LISP. If somebody's already forced some complicated syntax down your throat it just makes it that much harder to put what you want on top. And this is a real issue. While most Lispers may (rightly) prefer syntax-free (or, I guess, it's actually syntax-simple) LISP, any effort to try to get LISP used [outside?] its traditional areas of application is sure to meet resistance based solely on what is perceived as an arcane syntax. At Utah much effort has been spent interesting CAGD and VLSI fols to use LISP. A major selling point has been the claim that it's trivial (well, one does have to exaggerate) to put whatever syntax is desired on top of LISP. Just think how many MACSYMA users have been [un?]able to hack at the LISP level simply because they're put off by the syntax. It may be a stupid bias, but it's there.

SEF: The `loop` construct will not become part of the core of SPICE LISP. This is not negotiable.

5.6.2. Simple Iteration Constructs

***** Issue 47: The *result* form in `dolist` and friends**

ALAN: [51.2] [On the *result* part of `dolist`:] I guess.

MOON: Is the *result* just one form, leaving space for possible future expansion (I am dubious), or is it an implicit `progn`? Say explicitly because users (and implementors!) might easily assume one or the other.

GLS: I had intended it to be a single form, but am amenable to change.

**** Suggestion:** Document the *result* part of `dolist` and friends to be a single form, *not* an implicit `progn`.

Y

**** Responses**

MOON: Y		SEF: Y!!		DM: X!
DLW: Y!!		WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

DM: Document the result part of `dolist` and friends to be an implicit `progn`.

***** Issue 48: Zero or negative count to `dotimes`**

MOON: [51.7] What about a zero *count* in `dotimes`? Should a negative *count* be defined to iterate no times? Probably better than error.

DLW: Zero and negative should be defined to iterate no times.

GLS: I'd prefer to signal an error for a negative number of iterations, but I'd settle for leaving the effect of a negative count undefined; an implementation can optionally signal an error, or iterate no times, or whatever. This is to encourage the programmer to think carefully about boundary cases.

** Alternatives:

A. Define `dotimes` to iterate zero times if the *count* is zero or negative.

B. Define `dotimes` to iterate zero times if the *count* is zero; signal an error if the *count* is negative.

C. Define `dotimes` to iterate zero times if the *count* is zero; the behavior is undefined if the *count* is negative.

D. Define `dotimes` to iterate once if the *count* is zero or negative, for compatibility with FORTRAN 66.

A?

** Responses

MOON: A	RMS: A!!!	SEF: AB	GINDER: BC!	DM: A!
DLW: A!!	HIC: A	WLS: B!	CHIRON: A	
ALAN: A!	GLS: C!!	RPG: B!!	DILL: A!!	

DILL: BLISS decided to do this with `incr/decr` because such loops were frequently generated by macros, which would otherwise have to deal with all the boundary cases explicitly. I suspect the same argument applies here.

*** Issue 49: Effect of modifying the *dotimes* control variable

ALAN: [51.8] [On the specification that altering the *var* of a `dotimes` will not affect the number of iterations:] Well, OK, sigh.

MOON: Complete loss; define it to be undefined. This complicates the implementation to provide something useless to the user.

GLS: Thinking about it, I agree.

** Suggestion: Define the effect of `setq`'ing a `dotimes` control variable to be unpredictable.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!		

RMS: What happens if you `setq` something used in computing how many times to do it? Does that change the number of times? Probably in Lisp Machine LISP now, it does.

GLS: I would certainly not like for obscure side effects on the *count* expression to change the number of iterations! According to [Weinreb 81a], page 42, in Lisp Machine LISP the *count* is computed exactly once and saved in a hidden variable.

5.6.3. Mapping

***** Issue 50: Change to `mapc` not to return the first argument**

ALAN: [53.2] [On the incompatible change to `mapc` not to return the first argument:] Right!

MOON: This seems like a pointless incompatibility.

GLS: The motivation is explained in a compatibility note in the draft manual.

**** Suggestion:** Retain the COMMON LISP specification that `mapc` (and `mapl`) return `t`.

Y?

**** Responses**

MOON: N		SEF: Y!!	GINDER: Y!	DM: X!
DLW: N!		WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!		

GLS: Maybe this whole thing matters a little less now that `dolist` exists.

DM: Why `t` instead of `()`? This goes for a lot of other functions, too. I get the feeling it's simply a case of "it's always been this way", or am I missing something? I think it's well worth changing so that everything which can't find somthing useful to return, returns `()`. In any case, *don't* return a functional from `mapc` and `mapl`.

GLS: Oops! The term "first argument" should have been "first sequence argument".

CLH [53.6] [November] You seem to have a tautology: "the mapping functions should be used wherever they are clearer, since they are clearer in such cases".

GLS: [November] Well, aren't they?

***** Issue 51: Use of `return` within a `for`*xxx* construct**

ALAN: [54.2] [On the body of a `forxxx` being a `progn` rather than a `prog` body:] Why not? Seems like a likely thing to want to do.

GLS: The intent was to reserve these constructs to mean "we are mapping down a sequence straightforwardly, and do not intend to abort early". If you want to abort early, use `do`, or use `throw` from within the `for`.

**** Suggestion:** Allow `return` to exit a `for`*xxx* construct.

** Responses

MOON: Y		SEF: N!!	GINDER: N!	DM: N!
DLW: N	HIC: Y	WLS: N!	CHIRON: N!	
ALAN: Y!	GLS: N!	RPG: Y!	DILL: Y	

*** Issue 52: All the `for`*xxx* constructs

DLW: [On `forlist` and friends:] I really think the keyword-oriented "`loop`" is preferable to having all these random special cases. But if you really want them in, it's okay.

MOON: [54.3] Requiring the result of `forvector` to be a general vector seems like an unnecessary restriction.

MOON: [On `forvector` and friends:] In general, these are crocks. Suppose I want to map down a string and return a bit vector? (Certainly a frequent thing to want to do.)

MOON: What's the point of having `forlist` and `forlists`? There shouldn't be both; this is worse than old-style `do`. These are a crock anyway since you don't get to control the type of the return sequence (I propose changing them and `map` so that you do). Is it worth having both `forlist` (and related things) and `map`?

SEF: I kind of like the idea that we supply the `map` series (tradition!) and `forlist` (a simple and useful form), but that we flush `forlists` and let the user write a `do`. We can't anticipate every possible control structure that anyone will ever want to use, and this looks to me like it's a good place to draw the line.

GLS: The reason for `forlist` was Lisp Machine LISP compatibility, and `forlists` because I, at least, who code lots of interpreters, frequently want to write

```
(dolists ((var (lambda-vars fn))
          (val args))
   (crock-out var val))
```

Now I forgot to stick in `dolists`, but `forlists` was added for parallelism. It seems to me that this explosion is as ridiculous as that of the sequence functions, and ought to be resolved in a similar fashion. For example, if all the type-specific sequence functions go away, then so should the type-specific looping functions?

** Suggestion: Do not have both `forlist` and `forlists`. Resolve the type problems (both argument and result) in a manner compatible with the resolution of the `map` function.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!		WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!		

MOON: `forlist` can't be for Lisp Machine LISP compatibility; there is no such function in Lisp Machine

LISP.

GLS: My explanation was confused. I meant to add `dolist` for compatibility, then also `dolists`; then added `forlist` and `forlists` by analogy and actually edited them into the draft manual; then forgot to edit in `dolists`! Oh, well.

5.6.4. The Program Feature

***** Issue 53: &-keywords in `prog` and elsewhere**

RMS: [55.3] There is an example of a `prog` with a lambda-list keyword in its variable list. I think it is bad for `prog` to look at lambda-list keywords. For that matter, they are bad in lambdas too, but it is harder to get rid of them there. Right now lambda-list keywords are not meaningful anywhere but in lambda lists, in Lisp Machine LISP; in particular, they are not meaningful in `let`. I think it would be an unimprovement to add them.

GLS: The keyword involved was `&special`. The alternative to writing the keyword there would have been a `declare` form in the `prog` body.

**** Suggestion:** Tentatively disallow &-keywords in `prog` lists. (The final decision must await development of the documentation on declarations and on lambda-expressions.)

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!!!		

***** Issue 54: Permissible scope of `go`**

MOON: [55.8] What about a `labels` function returning from a `prog` in its parent?

GLS: Well, `labels` isn't in the draft COMMON LISP document (though I would like to see it in COMMON LISP). However, a similar case is:

```
(prog () (map #'(lambda (x) (if x (car x) (return 3))) berfi))
```

In either case, the point is that while it is a *necessary* condition for legality is that a `go` be lexically within its `prog`, this is not a *sufficient* condition.

**** Suggestion:** Clarify these problems in the manual.

Y

** Responses

MOON: X		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!!		

MOON: The sentence just before the suggestion makes no sense in terms of the example given. Note that in Lisp Machine LISP, when there are lexically enclosed functions, `go`'s and `return`'s from those functions to `prog`'s in their parent will work in all cases.

GLS: How about this case:

```
(declare (special x))
(defun lose ()
  (prog ()
    (setq x #'(lambda () (go a)))
    (return)
    a
    (print 'barf!)))
(progn (lose) (funcall x))
```

Will that last form actually cause the printing of "`barf!`"? How about when the new `lexical-closure` construct is used?

5.7. Multiple Values

5.7.1. Constructs for Handling Multiple Values

***** Issue 55: Semantics of multiple-value "`let`" and "`setq`" forms.**

SEF: I would like the Lisp Machine LISP versions of `multiple-value` and `multiple-value-bind` instead of, or in addition to, `multiple-value-let` and `multiple-value-setq`.

MOON: It does *not* say explicitly in the manual, and it took me three readings to get this, but I suspect that the intention is that if the callee returns fewer values than the caller expects, an error is signalled, and the caller must use the `&optional` keyword to prevent this. If so, this is a major change, since it means that the caller cannot be written as

`(and` *predicate* `(values ...))`

relying on returning as many `()`'s as the caller expects when the predicate is not satisfied, which is very common now. It also doesn't say what happens if the callee returns more values than the caller expects, except in the case where the caller was expecting one value. This needs to be cleared up. Does value-receiving behave like lambda-binding, or does it behave as in the Lisp Machine LISP at present but augmented by &-keywords? The manual must be extremely explicit about this, both because it is easy to get confused and because there are lots of different preconceived notions already going around.

MOON: I feel that while erring if too few values are returned superficially sounds good, the analogy of function returning to function calling is a false one, because the syntax in the callee of a function return is not at all like the syntax in the caller of a function call. Not allowing values to default to () leads to clumsier code when conditionals are involved. I think that making all values optional leads to a clearer programming style, even though you might naively expect just the opposite.

** Alternatives:

A. Provide Lisp Machine LISP-style multiple-value-receiving forms. These ignore extra values, and default missing ones to ().

B. Provide the forms specified in the draft COMMON LISP manual, which have a variables-list like a lambda-list; one can use `&optional` to specify default values, and `&rest` to get a list of all extra values. It is possible to get wrong-number-of-values error.

C. Provide both kinds.

** Responses

MOON: AC	RMS: A!!!	SEF: X!!!!		DM: B!
	HIC: C	WLS: B!	CHIRON: C!	
ALAN: A!	GLS: B!	RPG: B!!!	DILL: AB!!!	

DLW: This is just plain to hard for me to decide. Argh!

WLS: Definitely don't default missing values, but perhaps allow extras to be ignored.

SEF: I favor providing *only* the Lisp Machine LISP forms `multiple-value` and `multiple-value-bind`, plus a `multiple-value-call` form. The latter takes a function as its first argument, then a values-returning form. The function is called with the values as arguments. This gives checking for minimum and maximum numbers of arguments and default values without adding much hair. Now that I have thought about the implementation problems, I am adamantly opposed to including `multiple-value-let` and `multiple-value-setq` as described in the manual. I swallowed multiple values only on the condition that they would be kept *simple*.

JMC: [Several paragraphs by JMC follow.] The proposals for multiple valued functions in the draft COMMON LISP manual outlaw the neatest ways of writing certain multiple-valued functions by virtue of the restriction that only the first argument is used when such a function is used as an argument. It would be best not to define the multiple-value conventions yet until the issues can be decided. We give some examples to illustrate the point.

(1) The following function computes the gcd of two numbers `m` and `n` with `n` the smaller. It also computes the coefficients `a` and `b` such that the gcd is `am+bn`.

```
gcd(m,n) <=
   if n=0 then m,1,0
          else {m/n}[lambda q r.
                        {gcd(n,r)}[lambda g a b. g,b,a-qb]]
```

This is in internal notation; in list notation, this is

```
(defun gcd (m n)
  (if (zerop n)
      (values m 1 0)
      ((lambda (q r)
         ((lambda (g a b) (values g b (- a (* q b))))
          (gcd n r)))
       (/ m n))))
```

Herein the fucntion `/` is considered to produce two values (quotient and remainder), and the `gcd` itself produces three (the gcd itself and the two coefficients).

(2) The function `(mapalong f l u)` like `(mapcar f l)` in making a list obtained by applying the function `f` successively to the elements of the list `l`. However, `f` has two arguments, the second of which is a state `u`, and two outputs, the second of which is a new state. Thus we have

```
(defun mapalong (f l u)
  (if (null l) (values nil u)
      ((lambda (e u1)
         ((lambda (l1 u2) (values (cons e l1) u2))
          (mapalong f (cdr l) u1)))
       (f (car l) u))))
```

(3) `mapalong` is used to make an expression rewriter that keeps a list of subexpressions that can't be rewritten any more. We have

```
(defun rewrite (e u)
  (if (member e u)
      (values e u)
      (if (atom e)
          ((lambda (e1) (if (= e1 e)
                            (values e (cons e u))
                            (rewrite e1 u)))
           (rewtop e))
          ((lambda (l u1)
             ((lambda (e1)
                ((lambda (e2)
                   (if (= e2 e1)
                       (values e1 u1)
                       (rewrite e2 u1)))
                 (rewtop e1)))
              (mkexp (op e) l)))
           (mapalong (components e) rewrite u)))))
```

In the above, `(rewtop e)` rewrites the expression `e` at the top level, `(component e)` is a list of the subexpressions of `e`, `(op e)` is the operation or function part of `e`, and `(mkexp op l)` makes an expression with given op and components.

(4) `(count x n u)` counts the number of cells in a possibly re-entrant list structure `x` considering the cells listed in `u` to have already been counted and `n` to have been found. We have

```
(defun count (x n u)
  (if (memq x u)
      (values n u)
      (atom x)
      (values (add1 n) (cons x u))
      (count (car x) (count (cdr x) n u))))
```

Notice that the inner occurrence of `count` serves as two of the arguments of the outer occurrence.

Note of September 24: I am no longer sure that the examples prove my point. In fact, it now seems to me that only the last example is excluded as written, and it can be made acceptable to the COMMON LISP conventions by a small modification. On the other hand, the restrictions still seem inelegant.

GLS: Note that, in JMC's notation, `if` of more than three forms is defined as follows:

```
(if p1 e1 p2 e2 ... pn en else)
  <=>
(cond (p1 e1) (p2 e2) ... (pn en) (t else))
```

I explained to JMC that one can get the effect of

```
((lambda vars body) mvform)
```

by writing

```
(multiple-value-let vars mvform body)
```

This is a bit awkward in the last example, though. I would propose to generalize `multiple-value-call` as outlined above by SEF to allow more than one multiple-value-producing form; their collective results are concatenated and passed to the function. Then one would write

```
(multiple-value-call #'count (car x) (count (cdr x) n u))
```

instead of the more awkward (?)

```
(multiple-value-let ($ %) (count (cdr x) n u)
  (count (car x) $ %))
```

***** Issue 56: Syntax of multiple-value "`let`" and "`setq`" forms.**

MOON: Instead of `multiple-value-let`, I would prefer `(with-values ((`*var1 var2 ...*`)` *form*`)` . *body*).

MOON: The name `multiple-value-setq` is *not* an improvement!

MOON: Rather than the present

```
(multiple-value-bind (var1 var2 ...) form . body)
```

or the same syntax with the name `multiple-value-let` (which loses because it does not have the same syntax as `let`), I would prefer to follow the `with-` convention and introduce the new special form

```
(with-values ((var1 var2 ...) form) . body)
```

GLS: It seems to me that the answer to this question depends on the outcome of the semantics issue.

**** Suggestion:** If the semantics are like those of Lisp Machine LISP, use the Lisp Machine LISP syntax for compatibility; otherwise defer the question.

Y?

** Responses

MOON: Y	RMS: X!	SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y	HIC: Y	WLS: X!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!!!		

RPG: About multiple values: I think that the analogy between multiple values returning and function calling is legitimate, though there are obvious asymmetries (of course, with the right tail-recursion rules it isn't true that you know who has returned the values). The name `multiple-value-let` was chosen, I thought, because we didn't realize enough that to the Lisp Machine LISP crowd `let` isn't a macro on `lambda`, which meant a `let` binds less powerfully than `lambda`. `multiple-value-setq` is a bad name, but there has to be some name that a naive user relates to assignment rather than to binding.

RMS: Neither. `(let (((values a b) form)) body)` is better.

*** Issue 57: Result of `multiple-value-setq`

??? Query: [59.7] Fooey. Why not just say it [`multiple-value-setq`] returns ( ) [rather than that assigned to the first variable]?

RMS: `multiple-value-setq` should return the first value, always, unless the called function returned none. This is regardless of which of those values the user puts into variables. This is what `multiple-value` and `(setf (values ...) ...)` currently do on the Lisp machine, and is much simpler than the proposed definition.

GLS: The confusion is entirely due to my laziness or losiness; I said "first value assigned" when I really meant "first returned value". If the value returned is not always ( ), then it should be the first returned value.

** Suggestion: Let `multiple-value-setq` (or whatever replaces it) return the first value returned by the multiple-value-supplying form, or ( ) if it returns zero values.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

DM: I think a careful comparison should be made of all the various `setq`-style constructions. It would be nice if they were all cognates of one another, and returned "the same thing", whatever that may mean.

5.7.2. Rules for Tail-Recursive Situations

*** Issue 58: Restrictions on behavior of multiple values

> ??? Query: [60.2] The Lisp Machine LISP manual states: "The exact rule governing passing-back of multiple values is as follows: If *X* is a form, and *Y* is a sub-form of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then all the multiple values returned by *Y* are returned by *X*. In all other cases, multiple values or only single values may be returned at the discretion of the implementation; users should not depend on this. The reason we don't guarantee non-transmission of multiple values is because such a guarantee would not be very useful and the efficiency cost of enforcing it would be high. Even `setq`'ing a variable to the result of a form, then returning the value of that variable might be made to pass multiple values by an optimizing compiler which realized that the `setq`ing of the variable was unnecessary."
>
> I'm not sure the implementation should be allowed this caprice. In particular, a compiler smart enough to optimize out a `setq` can just as well leave behind code to enforce the single-value-returning semantics. I believe it is more important to have a dependable definition here.
>
> Opinions? For now the following documentation makes some clear requirements. These are not incompatible with Lisp Machine LISP, but merely requirements on implementations to make certain choices which Lisp Machine LISP leaves open.

MOON: Does this mean no implementation is allowed to have an optimizing compiler? It's debatable that these restrictions are useful.

GLS: You can optimize all you want, as long as the language semantics are preserved. In any case, I don't think the tail should wag the dog.

> ??? Query: [60.7] Should `prog1` and `prog2` return multiple values or not? It can be tricky to compile. Lisp Machine LISP causes them to return single values only. In SPICE LISP it happens to be easier to return multiple values. On the S-1 the issue is unclear.

MOON: This is an example of why it should be undefined.

SEF: I vote for `prog1` to return multiple values.

JONL: [60.9] Multiple values should be permitted from singleton `cond` clauses. I'm sure that the implementation will have no trouble receiving many values back, but making the predicate check by merely testing the nullity of the first one. Not only does this prevent the singleton case from being an anomaly, but it shows that we must consider *all* functions as returning multiple values even though predicate actions still test only the first return value. (But I'm not sure what to say about a predicate that returns zero values: probably an error.)

MOON: I don't like the changes to multiple values being considered, to force exactly one value to be returned in certain situations, and to return multiple values from the first subform of `prog1`. These seem to put tight restraints on the implementation, and certainly the first of the two gives no corresponding benefit to the user. This is something I will have to think about more. (Having thought about it more, I still don't like it.)

GLS: Well, you don't mention which are the "certain situations". I can only think of one offhand, namely the case of a terminal singleton clause in a `cond`, so I'll respond to that. I proposed that such a clause (*foo*) return exactly one value, and therefore *not* be equivalent to (`t` *foo*), for consistency: this way, the number of values returned by a `cond` clause is independent of its position in the `cond`, and does not change if more clauses are appended. Moreover, there is a trivial way to get multiple values returned: write (`t` *foo*) explicitly. (An ulterior motive is that I, perhaps somewhat tyrannically, would like to strongly discourage the slovenly habit of omitting that "`t`", which makes code harder to read for two reasons: first, it obscures the

fact that there is no drop-through case; second, it is harder to see that there is in fact only one item there, causing the value of the predicate to be returned. Those two extra characters "`t `" make both of these issues immediately clear.)

SEF: I consider the restriction that just one value be returned when the values end up in an argument or predicate test to be essential to the sanity of the world. I can't imagine how it would cause trouble to return multiples from `prog1` or `prog2`, but don't care much eiother way.

GLS: I don't much care whether `prog1` returns a single or multiple value, but I think it is *very* important that it not be left undefined. Worrying about blocking a small compiler optimization is putting a (probably small) efficiency ahead of the consistency of compiled code with interpreted code. I for one consider such consistency for *correct* programs to be paramount (programs in error need not be as rigorously consistent).

** Alternatives:

A. Require singleton `cond` clauses and `prog1` to return a single value.

B. Require singleton `cond` clauses and `prog1` to return multiple values.

C. Permit singleton `cond` clauses and `prog1` to return single or multiple values at the implementor's (and even compiler's) discretion.

D. Require `prog1` to be single-valued, and singleton `cond` clauses to be multiple-valued.

E. Require `prog1` to be multiple-valued, and singleton `cond` clauses to be single-valued.

F. Require `prog1` to be single-valued, but leave singleton `cond` clauses to the implementor's discretion.

G. Require `prog1` to be multiple-valued, but leave singleton `cond` clauses to the implementor's discretion.

H. Require singleton `cond` clauses to be single-valued, but leave `prog1` to the implementor's discretion.

I. Require singleton `cond` clauses to be multiple-valued, but leave `prog1` to the implementor's discretion.

** Responses

MOON: C!!!!	RMS: I!	SEF: E!!!!	GINDER: E!	DM: C!
DLW: E!!	HIC: X	WLS: B!	CHIRON: A!!!	
ALAN: C!	GLS: A!!	RPG: E!!!	DILL: X!!!	

RMS: I am assuming that by "singleton `cond` clauses" you mean only when they are at the end of the `cond`. A singleton `cond` clause at the end is fundamentally different from one in the middle, because it is not in fact a conditional. `(or a b c)` ought to be equivalent to `(cond (a) (b) (c))`. `or` returns multiple values only from `c`, and so should the `cond`.

GLS: If `or` treats its last argument form differently, then there is no reason why its translation to `cond`-form should be the same. So the correct model for `(or a b c)` is not `(cond (a) (b) (c))`, which implies equivalent treatment of the three argument forms, but rather `(cons (a) (b) (t c))`, which indicates the special (multiple-value) treatment of the last form.

MOON: `unwind-protect` is an "expensive" construct and `prog1` is a "cheap" one. We could change that or put in a variant. The issue is that you *cannot* tell from context whether multiple values are expected when evaluating a form, when you are compiling code for it. When the value of the form is being returned from a function, it depends on the caller of the function at run-time. The reason for the Lisp Machine LISP rules on multiple values is that alternative rules, while apparently simpler and more consistent [I think this is partly only apparent], would require that code always be compiled for the most expensive case, which in practice is almost never used. Thus for any `and` at top level in a function, code would have to be compiled to save the extra values of a clause somewhere while the first value was tested. I think it's much better not to compromise efficiency for consistency *when* the consistency does not actually provide any benefits to users. Similarly I don't think it is worth generating extra code to throw away possible extra values from final clauses of `cond`, since this is not going to benefit anyone. I haven't worked out the details, but I think it is possible to define the semantics in a simple, consistent way that makes all this natural. Thus `and` can be defined in terms of `if` and a primitive `andp` which is only allowed to be used as a predicate, for example.

DM: I don't think I understand the subtleties of multiple value returns. I can certainly see their use, though. Modulo what I really don't understand, I think the version proposed we (Utah) can live with, but I wouldn't want to stake my life on it. Whether people here can really be persuaded to put them in is another question, but I would like to see it. The full Lisp Machine LISP version (with automatic defaulting of missing values to ()) is totally out of the question. I don't think we can do it efficiently on a wide variety of conventional machines. I believe that inclusion of Lisp Machine LISP-style multiple value returns would probably pretty much shoot down all hopes of us following the COMMON LISP drummer. The mechanism should certainly be kept as simple as possible. For instance, funny business with `cond` and `progn` will probably also screw us.

SEF: I cannot live with any decision *requiring* singleton `cond` clauses to be multiple-valued. It would screw SPICE LISP totally. I prefer to bind both of these issues and to allow `prog1` to pass multiple values, but the importance is much lower: "!!" and "!" respectively.

DILL: Any of alternatives not allowing the implementor to decide is fine.

MOON, DLW: [61.4] Lisp Machine LISP is upward compatible with COMMON LISP as of a month ago on the matter of a multiple-valued form in a `return`.

5.8. Non-local Exits

5.8.1. Catch Forms

***** Issue 59: A catch *may* or *must* have a tag?**

MOON: Must?

DLW: Must!

** Suggestion: Must.

Y

** Responses

MOON: Y		SEF: Y!!		DM: Y!
DLW: Y!!!	HIC: Y!	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y	

*** Issue 60: Must a catch tag be a symbol?

MOON: Why must a catch tag be a symbol?

DLW: I'd prefer to restrict it to symbols if there's no reason not to.

MOON: What about catch tags that aren't symbols? I am thinking mostly of structures and instances being used as tags. Also, MACLISP allows a (possibly-empty?) list of catch tags; Lisp Machine LISP currently does not. Does COMMON LISP want to accept this? And are catch tags compared with `eq` or with `eql`, i.e., can numbers be used as catch tags?

SEF: Let's just go with symbols and `eq`. It's a big enough pain already, and the tag searcher certainly doesn't want to do arithmetic or grovel structures.

GLS: I think the intent was to do `eq` comparisons on structures. It's just that sometimes you want to have a catch keyed on something no one else has access to, and often a structure that is hanging around anyway is just the ticket.

** Alternatives:

A. Require catch tags to be symbols.

B. Let catch tags be anything, but specify that throwing does `eq` tests.

C. Let catch tags be anything, but specify that throwing does `eql` tests.

D. Let catch tags be anything, but specify that throwing does `samepnamep` tests.

B?

** Responses

MOON: B	RMS: BC!!!	SEF: AB!!!	GINDER: B!	DM: A!
DLW: B!!	HIC: B!!	WLS: C!	CHIRON: B!	
ALAN: C!	GLS: B!	RPG: B!!	DILL: A	

RMS: Lisp Machine LISP makes use of this in the implementation of `prog` and `return` in the interpreter.

MOON: MACLISP allows a list of catch tags and does `memq` on it during `throw`. The reason Lisp Machine LISP doesn't have this is not a conscious decision, but simply that no one told us when the idea was dreamed up and put into MACLISP, and no one ever asked for it later. The list of catch tags thing should either be put into COMMON LISP or tossed out, with a compatibility note in the manual. I don't care which.

*** **Issue 61: Flush special meanings of `t` and `()` as catch tags**

MOON: Flush the special meanings for `t` and `()` as catch tags. This is a historical accident of Lisp Machine LISP.

GLS: Well, the ability to catch anything seems worthwhile. Or are you saying that `unwind-protect` and similar things should identify themselves by some mechanism other than special tags?

** **Suggestion:** Flush the special meanings for `t` and `()` as catch tags. **Y**

** Responses

MOON: Y		SEF: X!!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

SEF: Allow `t` and `()` internally as tags, but maybe flush them at user level in favor of `catch-all`, etc.

*** **Issue 62: What to do about `catch` and `*catch`?**

MOON: `catch` and `*catch` seems like a pointless duplication of functions.

DLW: I agree.

MOON: I think having both `catch` and `*catch`, and both `throw` and `*throw`, with the difference that the non-* ones don't evaluate their first argument, is a total crock. I would propose to have only the non-* versions; make them evaluate their argument. (How hard is it to type a "'" mark, versus remembering this special case?) Also the behavior with respect to multiple values has been changed (`catch` now returns all the values of the body if exited normally, or all the values of the second subform of `throw` if thrown to). The * versions could be kept around for a while in Lisp Machine LISP, working the old way, to lessen the shock of this change. The new multiple values behavior may not be implementable on the A-machine; I will have to think about this. I will accept the new multiple values behavior for the L-machine, as opposed to the less consistent thing I was planning to do.

SEF: I agree about `catch` versus `*catch`. Just evaluate the tag and forget about compatibility.

ALAN: [62] I would prefer that `catch` *also* evaluate the tag and we can then flush the losing name `*catch`. [64] Similarly for `*throw`.

** **Suggestion:** Flush `*catch` and `*throw`, and let `catch` and `throw` always evaluate the *tag*.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!!	GINDER: Y!!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!!!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y!!!!	

*** Issue 63: Lexical catch and throw?

RMS: Wild idea: get rid of named `prog` and friends, and re-introduce `catch` and `throw` with lexical tags! `*catch` and `*throw` would be the same as now, with dynamic tags, and `catch` and `throw` would replace named `prog` and `return-from`. I think it was a mistake that I connected the named escape operation with `prog` at all. For that matter, I think that it is a shame that `prog` needs to be used so often just to do `return`. It would be nice if there were something else, just like `progn` but allowing `return`.

GLS: I like the idea. The proposed naming convention might be a bit confusing, however. How about `block` and `exit` for the lexical versions?

SEF: I tend to favor `prog` as in the current manual. There is this implicit weird assumption that `prog` costs more than `progn`.

GLS: I happen to believe that `prog` is *cognitively* more expensive than `progn`. When I see a `progn` in someone's code, I know what's going on. When I see a `prog`, anything goes; I must stop and psyche out whether he's "really" doing a `do`, a `cond` in some odd form, something really clever like a doubly-nested search loop with two exit points, or just transcribing FORTRAN code.

** Alternatives:

A. Introduce some kind of lexical catch and throw, in addition to named `prog` and `do`.

B. Introduce some kind of lexical catch and throw to replace named `prog` and `do`.

C. Have no lexical catch and throw.

** Responses

MOON: C	RMS: A!	SEF: C!!	GINDER: B!	DM: B!
DLW: B!!	HIC: A	WLS: B!	CHIRON: B!!!	
ALAN: B!	GLS: A	RPG: A!!	DILL: A!!	

SEF: I would be interested in seeing a more concrete proposal for lexical catch and throw. If it's clean enough, I would probably prefer it to named `prog` and `do`. Until I see this, I vote for the status quo.

RMS: Document the named `prog` and `return-from` as required but obsolete. Do not introduce named `do`, which is not in any existing code, and can easily be expressed using named `prog` and unnamed `do`.

WLS: Only if a concise syntax is possible.

DM: I like it. However, I expect I'm the only one in Utah who will. It can certainly be lived with, however.

After all, you can always build a `prog` (modulo `go` and labels) from `catch` and `let`. The notions of `go`, `return`, and variable should be factored out from one another. One should also have a construct which exists solely to hold `go`'s and labels. If you want to do a `go` inside some new bindings you have to stick one of these jiggers in a `let`, and similarly for a `do`.

***** Issue 64: Funny extra values from `*catch`**

MOON: The third and fourth arguments returned by `*catch` in Lisp Machine LISP are a result of historical accident and irrelevant. They are no longer documented.

**** Suggestion:** Eliminate them from the COMMON LISP manual.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

??? Query: [64.3] The Lisp Machine LISP manual regards it as a bug that Lisp Machine LISP doesn't handle multiple values from `unwind-protect`. I agree. So if we can do it for `unwind-protect`, why not for `prog1`?

MOON: The `unwind-protect` problem was fixed in Lisp Machine LISP some time ago.

***** Issue 65: Names for `catchall` and `unwindall`**

MOON: By the way, spelling these `catchall` and `unwindall` rather than `catch-all` and `unwind-all` is inconsistent with the general naming conventions.

GLS: I agree, but what is in the COMMON LISP manual is more like the `catchall` of NIL than the `catch-all` of Lisp Machine LISP; I was just trying to prevent confusion.

**** Suggestion:** Rename `catchall` and `unwindall` to be `catch-all` and `unwind-all`.

Y

**** Responses**

MOON: X				DM: Y!
DLW: Y!	HIC: Y		CHIRON: N!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

MOON: Rename `catchall` to `catch-all` but flush `unwindall` (I can't think of a use for it, and on page 63 of the draft COMMON LISP manual it says essentially that you shouldn't use it).

5.8.2. Throw Forms

***** Issue 66: Error-handling for throw**

MOON: The draft COMMON LISP manual is very unclear about error checking in `throw`. Unseen-throw-tag errors should be signalled in the environment of the `throw`, not in an environment where the entire program state has been unwound. Thus throwing needs to be a two-pass operation. Clearly the first pass succeeds when it finds a `catch` for the specified tag, and ignores `unwind-protect`. What about `catchall` and `unwindall`? Does a `catchall` satisfy the first pass of all throws inside it, or of none of them? Does `unwindall` behave like `catchall` or like `unwind-protect`?

** Suggestion: Adopt MOON's rules above for error-checking in `throw`. The search for a tag ignores `unwind-protect`, but considers `catchall` or `unwindall` to satisfy it.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

MOON: [The requirement on `throw` and multiple values] makes `throw` not a function and introduces a possible hidden efficiency cost. I want to think about this before agreeing to it. The analogy with `return` makes some sense.

MOON: [November] It's okay for `throw` not to be a function.

***** Issue 67: Keep `*unwind-stack` in the core language?**

??? Query: [65.1] Perhaps this [`*unwind-stack`] belongs not here but in a chapter on semi-compatible low-level stuff?

MOON: This does not belong in the core language, especially when you see what else you need to make it useful.

SEF: Get rid of it. (Move it to red pages?)

** Suggestion: Flush `*unwind-stack`.

Y

** Responses

MOON: Y		SEF: Y!!!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!!!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

Chapter 6

FUNC

Chapter 7

MACRO

Chapter 8

Declarations

8.1. Declaration Syntax

***** Issue 68: Change the name of global-declare?**

MOON: `global-declare` is a fairly poor name.

GLS: I agree; it is at least as bad as `local-declare`.

** Suggestion: Rename `global-declare`. But to what? How about renaming it back to `declare`, and renaming what the COMMON LISP draft now calls `declare` to be `dcl` (thereby confusing all PL/I programmers)?

** Responses

MOON: X	RMS: Y!	SEF: N!!!	GINDER: N!	DM: X!
DLW: N!	HIC: X	WLS: Y!		
		RPG: Y!		

WLS: Yes. Nice! If it's concise, people will use it.

DM: `global-declare` is definitely ugly. Even `declare-globally` sounds better. But a squinty little name like `dcl` is even worse.

MOON: I think it is better for the global and the enclosed-in-contour forms of `declare` to have the same name, and for the distinction to be made by whether or not it is enclosed in a body. A global declaration is just one whose enclosing contour is the whole universe. Also, referring to the example on page 23, I don't think `eval-when` should constitute an enclosing contour (however, `progn` should). What are we doing about `(progn 'compile ...)`?

***** Issue 69: Pervasiveness of declarations**

MOON: There is an important issue with local declarations which has been ignored up to now. The COMMON LISP manual is ambiguous about this also. To use an example,

```
(let ((x 1))
  (declare (:type :fixnum x))
  ...
  (let ((x (append a b)))
    ...))
```

does the declaration apply to the inner `x` (as well as the outer one), or does it not? Suppose the inner `let` started life as an open-codable function which was then bodily incorporated as a lambda-combination, and just happened to call its argument `x`, the same symbol as its caller was using for something else? After talking this over with some people, I've decided that you have to have your choice, of whether the declaration is truly local or applies to all instances of variables with this name inside here. I'm not sure of the best syntax for distinguishing them. The default probably wants to be truly local, unfortunately the opposite of the (only) choice that MACLISP provides. Except for `special` the default wants to be the other way. But having it default one way for some declarations and the other way for the rest is clearly a bad idea. In addition there should be a declaration that turns off inherited declarations. This is related to the thing you put around a form to make it not be inside the lexical environment where it appears, which is used for such things as `defstruct` slot initializations (forms which are copied from where the user wrote them to some other place by a macro).

GLS: If you are being really careful, then you need one kind of `special` declaration which makes a *binding* special, and another kind which makes *references* legitimately special. The latter is needed when the binding is not lexically apparrent.

SEF: It seems to me that a `declare` just inside a local binding context applies to that instance of the varaible, but does not establish a lexical declaration scope for nested instances. Then we provide `global-declare` to set up a default for instances without such an immediately-embedded declaration. The default default for bindings is `local`, but for unbound variables it is `special`. References look at the lexical binding context to see if they are local or special references.

GLS: `special` and `local` declarations are not the only kinds there are. While a type declaration probably ought to apply only to the immediately present binding of a variable, an `inline` declaration probably ought to be pervasive, applying to all subexpressions within its scope. One idea is to divide declarations into two types, immediate and pervasive; however, those in a `global-declare` are always pervasive.

** **Alternatives:**

A. Specify that `special`, `local`, `type`, and `ftype` declarations are immediate in scope, and that `inline` and `notinline` are (lexically) pervasive in scope. Declarations appearing in a `global-declare` are always pervasive, in effect establishing defaults. As a special case, a `special` declaration for a variable not bound in the construct containing the declaration is pervasive.

B. Specify that `special`, `local`, `type`, and `ftype` declarations are immediate in scope, and that `inline` and `notinline` are (lexically) pervasive in scope. Declarations appearing in a `global-declare` are always pervasive, in effect establishing defaults. Introduce a pervasive `specref` declaration for declaring references to free variables.

C. Use `declare` to effect immediate declarations, and `proclaim` to effect pervasive ones.

** Responses

MOON: C	RMS: X!	SEF: X!!!!	GINDER: A	DM: C!
DLW: X	HIC: A	WLS: B!	CHIRON: C	
	GLS: C	RPG: B!	DILL: X	

RMS: Complicated. All local declarations should be pervasive, but open-codable functions should contain declarations sufficient to override anything that may come in from outside. This could be either a single declaration which says "block any declarations from outside", or simply no-op declarations of all variables used.

GLS: It would be naive to assume that one can open-code a function simply by plopping the text of its definition into the place of reference. This is the very substitution problem which eluded such logicians as Tarski and for the elucidation of which Church invented the lambda-calculus. The blocking of irrelevant declarations should not be the task of the author of such a function, but of the open-coding mechanism itself.

DLW: The existing L-machine compiler implements my favorite scheme: `declare` forms are completely local, and `local-declare` is pervasive. I vote for this.

DM: The whole business of declarations going up a level and goosing the surrounding construct seems totally wrong. A declaration should apply to what's contained in it. I get the feeling it's simply a case of "this is the way we did it in MACLISP, so why not here, too". That's probably not a fair assessment, but it's the first thing I think when I see

```
(prog (n)
  (declare (:fixnum n))
  ...)
```

or whatever it is. I think my vote would be for `proclaim` at top level, and `declare` for a pervasive declare-locally. To turn it off you need another, inner `declare` which declares it something else. This requires a `declare` keyword corresponding to "nothing special about this guy".

SEF: This needs to be re-thought into a coherent proposal that allows the interpreter to win.

DILL: After reading this chapter of the manual carefully for the first time, I think that it is going to be necessary to take out some time and think about redoing the whole thing. Some of this awaits elaboration of the type system, but there are a number of other problems here. `declare` is being used for at least three different purposes: indication to the interpreter what type of binding to use, making assertions about types (which should be checked at run-time in the interpreter), and offering advice to the compiler. There is no particular reason to believe that these things want to appear in the same places in various constructs, or have the same scope. Making incremental changes to the MACLISP declaration stuff is probably not going to do the trick.

EAK: I think using `(declare ...)` is wrong because of the known problems with macros. Can't you put the information in the lambda list?

***** Issue 70: Declarations and top-level code**

MOON: What happens with code typed in at top level? If I do

```
(setq x (frobboz *))
(setq z (nthcdr (- (length x) 2) x))
```

and things like that, do I get a lot of "undeclared variable assumed to be special" warnings from the interpreter, as I would from the compiler, since they now behave identically? (Perhaps this issue is trivial? Then again, perhaps it isn't? It's fairly easy to see what to do with `setq` typed in at top level, but will it be really convenient with `let` typed in at top level? And in what environment are forms evaluated by `break` evaluated? The best thing would be to have them evaluated in the *lexical* (as well as dynamic of course) environment of the guy who called `break`, whether he is interpreted or compiled. This could be difficult in some implementations.)

GLS: Well, it's not that the interpreter and compiler are truly identical, but that they impose the same semantics on correct programs, which is another matter. Anyway, we clearly do not want to get gobs of error messages when typing interactively. On the other hand, it may be reasonable to require (at least in principle) declaration of variables used free in top-level forms in files to be compiled; the compiler may or may not choose to check this.

**** Alternatives:**

A. The interpreter as well as the compiler should give a warning on reference to an undeclared special variable.

B. The interpreter should never give a warning, but the compiler should be permitted to warn about undeclared references in top-level forms.

C. Status quo: The interpreter never gives error messages for undeclared special variables, and neither does the compiler for top-level forms.

**** Responses**

MOON: C	RMS: C!!!	SEF: X!!!!	GINDER: C!	DM: X!
DLW: B!!	HIC: X	WLS: B!	CHIRON: C	
ALAN: B!	GLS: B!	RPG: B!	DILL: BC	

HIC: Can the interpreter give a warning if inside an interpreted function, but not if evaluating the top-level form?

DM: The compiler and interpreter should behave the same, and in general give messages about undeclared free variables. However, there should be some sort of switch controlling these (of course), and it should normally be suppressed when reading from the terminal. When reading from a file, one can assume the programmer has had time to figure out who's used freely and to declare it. After all, why wait until he compiles it to tell him what's wrong; he can use the information much more when he's debugging his code interpretively!

8.2. Declaration Keywords

***** Issue 71: Are names of declarations keywords?**

> ??? Query: [72.6] It seems to be that declaration types should be keywords. The old MACLISP crock of just evaluating declaration forms is not *necessary* now that `eval-when` exists, and it may not be *desirable* because it makes it harder to deal with arbitrary implementation-dependent declarations. On the other hand, all those colons are pretty ugly. What do people think?

MOON: If you don't want the colons, have declarations names looked up by `string-equal`.

GLS: Then, by the same token, perhaps all keyword-style applications should use `string-equal`. (I suppose one difference here is that the lookup only occurs at compile time, not run time.)

ALAN: I think the colons are ugly. `(declare (foo ...))` should examine the property list of `foo` for something and funcall that on `(foo ...)`. Things inside a `declare` are pseudo-special-forms and should look that way. Flush the colons.

**** Alternatives:**

A. Keep the colons; declaration names are keywords.

B. Flush the colons; use `string-equal` to determine what a declaration is.

C. Flush the colons; think of declarations as a kind of special form, like `cond` (which also has no colon).

C?

**** Responses**

MOON: C	RMS: A!	SEF: AC!!	GINDER: C!	DM: A!
DLW: A	HIC: C	WLS: BC!	CHIRON: A!	
ALAN: C!	GLS: C!	RPG: C!	DILL: C	

RMS: Names of local-declarations are already keywords in Lisp Machine LISP. However, the typical ones require no colons since they are names of global functions.

***** Issue 72: Should there be a converse for special declarations?**

MOON: There should be an `unspecial` declaration.

SEF: Should there be a `:local` declaration to complement `:special`?

**** Alternatives:**

A. Introduce a `local` declaration, meaning "not special".

B. The same, but call it `unspecial`.

C. Status quo: introduce no such declaration.

A?

** Responses

MOON: A	RMS: A!	SEF: A!!		DM: A!
DLW: X	HIC: A	WLS: A!	CHIRON: A!	
ALAN: A!	GLS: A	RPG: A	DILL: A!!	

DLW: The same, but call it `lexical`.

*** Issue 73: Provision for more concise type declarations

MOON: The word `type` in a type declaration should probably be elidable when the type name does not conflict.

** Suggestion: If `(declare (type` *xxx* `...))` is a valid type declaration, and *xxx* is a symbol, then make `(declare (`*xxx* `...))` also be a valid type declaration.

** Responses

MOON: Y	RMS: N!	SEF: N	GINDER: N	DM: N!
DLW: Y!!	HIC: Y	WLS: Y	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: X	

RMS: This would be convenient for the user but rules out the easy kinds of implementations.

GLS: Ooops, I just realized that the suggestion as stated blithely encompasses type names created by `defstruct`, which would indeed get hairy. I had meant to include only built-in type names such as `integer`.

*** Issue 74: Syntax for declaration of types of functions

SEF: I would prefer a more pictorial declaration form for functional types. Rather than

```
(declare (:ftype (:function (:integer :list) t) nth nthcdr))
```

something more like

```
(declare (:function (nth :integer :list) t)
         (:function (nthcdr :integer :list) t))
```

Also, or instead, how about a little arrow to make the mapping clearer?

```
(declare (:ftype (:function (:scalar :scalar) -> :scalar :scalar)
                 trunc round floor ceil))
```

GLS: By the way, note that the functional type syntax was designed to allow one to declare individual types for the several multiple values a function might return. However, perhaps as a convenience the MACLISP style of declaration should also be permitted:

```
(declare (:type :integer (ackermann :integer :integer)))
```

instead of

```
(declare (:ftype (:function (:integer :integer) :integer ) ackermann))
```

** **Alternatives:**

A. Provide an alternative syntax as suggested above by SEF.

B. Replace the `ftype` syntax with SEF's suggestion.

C. Allow the MACLISP-style functional type declaration in addition to `ftype`.

D. Allow the MACLISP-style functional type declaration and SEF's suggestion; flush `ftype`.

E. Allow all three kinds.

F. Status quo: retain the `ftype` declaration and omit the others.

G. Eliminate all three kinds. A compiler that wants to be that smart can just darn well look at all the functions before generating any code. (This makes a one-pass file compiler hard.)

** Responses

MOON: BD!!	RMS: X!	SEF: B!	GINDER: B!	DM: F!
DLW: A		WLS: F!	CHIRON: B!	
ALAN: C!	GLS: D	RPG: A!	DILL: X	

RMS: The user or the system can provide macros that take SEF's syntax and expand into the standard one.

DM: All the fancy syntax is a real loser. I think I know what

```
(:function (:scalar :scalar) -> :scalar :scalar)
```

really is (as an S-expression), but there is a string temptation not to think of "->" as a symbol, but just a syntactic bracket, like "(".

*** **Issue 75: Should compilers be required to warn of ignored declarations by default?**

MOON: Yes, the default mode should be for a compiler to provide warnings of ignored declarations.

** **Suggestion:** Impose this requirement, but note that a compiler may provide a switch to override this default.

Y

** Responses

MOON: Y		SEF: N!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

SEF: I would *encourage* compilers to warn by default, with an override switch, but not *require* it. It may be hard to warn in some cases.

***** Issue 76: May implementation-dependent declarations exist?**

MOON: Note that implementations will add their own declaration keywords in most cases.

GLS: If so, then the warning mode proposed above becomes even more important.

**** Suggestion:** Permit implementation-dependent declaration types.

Y

**** Responses**

MOON: Y!!!!		SEF: Y!!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y	RPG: Y!!	DILL: Y	

Chapter 9

Symbols

9.1. The Property List

9.2. The Print Name

9.3. Creating Symbols

***** Issue 77: Does make-symbol copy the given string?**

ALAN: [79.8] [Is the string argument to `make-symbol` *copied*?]

GLS: An advantage of not copying is to save run time and space, maybe; a disadvantage is the danger of later clobbering the string. An advantage of copying is that it gives the implementation a chance to put the string in a better place (such as read-only storage).

**** Alternatives:**

A. Define `make-symbol` not to copy its string argument, but to use it directly as the print name.

B. Define `make-symbol` always to copy its string argument.

C. Leave it undefined and at the implementor's option; therefore the programmer may not depend on it.

**** Responses**

MOON: C	RMS: C!	SEF: A		DM: C!
DLW: C!	HIC: A	WLS: C!	CHIRON: B!	
ALAN: C!	GLS: C!	RPG: B!!	DILL: B	

DILL: Pnames of symbols should be read-only, so any writeable string should be copied. If a string is already read-only (like another pname), it would be okay not to copy.

***** Issue 78: Existence of si:*gensym-prefix and si:*gensym-counter**

MOON: [80.2] Do you really want to document the names `si:*gensym-prefix` and `si:*gensym-counter`?

GLS: No.

**** Alternatives:**

A. Document `si:*gensym-prefix` and `si:*gensym-counter` as the official way to get at the insides of `gensym`.

B. Allow `gensym` to take an argument as now to reset the counter or prefix, but don't document the insides.

C. Provide no way to reset the `gensym` counter or prefix.

B

**** Responses**

MOON: B		SEF: BC!!	GINDER: B!	DM: B!
DLW: B	HIC: B	WLS: B!	CHIRON: B!	
ALAN: C!	GLS: B!	RPG: B!!		

***** Issue 79: What is gentemp for?**

SEF: What is `gentemp`?

GLS: Gensyms tend to be used in two different ways. In one usage, they serve as data objects, usually in order to use the property list as a structure. The different names are useful primarily for debugging, so that a person can recognize them by sight. However, it is not algorithmically crucial that the names be distinct. In the other usage, they may be interned. Therefore the names must be distinct. To this end it is desirable both that the counter not wrap around and that an arbitrary prefix be specifiable. It is also desirable that the name be easy to type, so it shouldn't require vertical bars to type. `gentemp` is intended for this second type of usage, as `gensym` is for the first.

**** Suggestion:** Add `gentemp` to the language. It is similar to `gensym`; one may reset the count or prefix. However, `gentemp` permits an arbitrary prefix, and will not wrap the count around.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: X!
DLW: Y	HIC: X	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

HIC: Why not just flush `gensym` then?

DM: Just have `gensym`, but give it the semantics of `gentemp`.

MOON: The `gentemp` prefix should not be an optional argument and should not be remembered from call to call.

***** Issue 80: Rename get-package to be symbol-package?**

MOON: [80.7] Should `get-package` be called `symbol-package`? That's what Lisp Machine LISP calls it. (We'll probably be changing this name, though.)

**** Suggestion:** Rename `get-package` to be `symbol-package`.

**** Responses**

MOON: Y	RMS: Y!	SEF: N!		DM: N!
DLW: Y!	HIC: Y		CHIRON: N	
ALAN: X!				

ALAN: Well, I don't like `get-package` if it isn't what Lisp Machine LISP calls it. But I don't like `symbol-package` either if Lisp Machine LISP is going to change its mind later. I would prefer to put the issue off until Lisp Machine LISP makes up its mind (or until Lisp Machine LISP agrees to something concrete). I wonder what's wrong with `symbol-package` anyway?

MOON: Lisp Machine LISP will be renaming this function, probably. Packages in COMMON LISP should be deferred completely until Lisp Machine LISP gets its act together, which is predicted to be soon.

Chapter 10

Numbers

***** Issue 81: Complex numbers**

MOON: [82.2] What do `plusp` and `minusp` do for complex numbers?

MOON: [83] What do `<`, `>`, `<=`, `>=`, `max`, and `min` do for complex numbers?

MOON: [84] What does `abs` do for complex numbers?

ALAN: [85.9] Complex works [as an argument to `exp`], I assume?

MOON: [86.0] Does `expt` of an integer to a negative integer return a ratio? Does `expt` of a ratio to a positive integer return a ratio, or a flonum?

ALAN: [86.8] Do these [`sin` and friends] work for complex? How about `sinh` and `cosh` in that case? (You can lose using $(e^x - e^{-x})/2$, or whatever it is.)

SEF: [87.0] For `atan`, y and x should be scalars.

GLS: [The following paragraphs are all by GLS.] There are two approaches one can take to adding complex numbers to LISP. One is to try to separate the real and complex domains, to prevent the user from accidentally stumbling across complex numbers. The other approach is simply to make everything generic.

A particularly subtle point is illustrated by the following example:

```
(expt -8 1/3)
```

Should this produce `-2` (or `-2.0`) or should this produce the principal complex root `#C(1.0 1.732050808)` = $1+sqrt(3)i$? If one were confined to the real domain, one might want `-2`. However, this is inconsistent with the usual definition for complex numbers (and is in fact horribly discontinuous). This question actually doesn't arise in most programming languages, because if the result is to be real, a negative number can only be raised to a rational power with an odd denominator, and languages using binary representations cannot represent any such rational exactly. (In APL, however, the expression `¯8*1÷3` actually does yield `¯2` in some implementations; the exponentiation function "*", when confronted with a negative left argument and a right argument which is not (approximately) integral, will try to approximate the right argument by a rational with a small odd denominator. On the other hand, a study of all APL programs ever published in *APL Quote Quad* and in proceedings of APL conferences (1704 functions containing 12,434 lines of code) found not one use of this feature [Penfield 79].)

I have found four other widely-known programming languages that have supported complex numbers in a big way: FORTRAN, ALGOL 68, PL/I, and APL. It is instructive to examine them.

The FORTRAN 77 full (not subset) standard [ANSI 76a, ANSI 78] allows the following generic operations to apply to complex numbers: `+`, `-`, `*`, `/`, `**` (exponentiation), `.EQ.`, `.NE.`, `REAL`, `ABS`, `SQRT`, `EXP`, `LOG`, `SIN`, and `COS`. The four arithmetic operators and the two relationals may take operands of mixed type, provided, however, that one may not combine a complex quantity with a double-precision quantity (there being no double-precision complex numbers in the language). The type-specific operations `CABS`, `CSQRT`, `CEXP`, `CLOG`, `CSIN`, and `CCOS` are also provided; they behave identically to the generic ones but restrict the argument and result types. In addition, `AIMAG` and `CONJG` are non-generic operations requiring a complex argument. (Note that therefore the real-part operator is generic, and may apply to non-complex numbers, but the imaginary-part operator is not.) There is no phase function on complex numbers. Exponentiation does not permit the exponent to be complex; if the base is complex, the exponent must be an integer (this of course completely avoids the issue of principal values). The other logarithmic, trigonometric, and hyperbolic functions do not permit complex arguments or results, even where they would be mathematically well-defined; examples are `LOG10`, `TAN`, `ASIN`, and `SINH`. Complex numbers are always pairs of "real" numbers; there are no complex integers.

The revised version of ALGOL 68 [van Wijngaarden 77] allows the following generic operations to take complex numbers: `+`, `-`, `*`, `/`, `=`, `≠`, `leng` (convert to the format of next-higher precision), `shorten` (convert to the format of next-lower precision), `abs`, and `**` (exponentiation). Exponentiation of integer, real, and complex operands is restricted to integral exponents; the definition causes `0**0` to be `1`, and zero to a negative power causes a division by zero. It is not permitted to raise an integer to a negative power. The four arithmetic operators and the two relationals may combine a complex with an integer or real. The following non-generic operations are provided on complex numbers: `re` (real part), `im` (imaginary part), `arg` (argument, or phase), and `conj` (conjugate). The phase of zero is undefined. The trigonometric functions are restricted to real operands, as are *exp* and *ln*; no hyperbolic functions are provided. Complex numbers are always pairs of "real" numbers; there are no complex integers.

In PL/I [IBM 70, ANSI 76b] the following generic operations may take complex numbers: `+`, `-`, `*`, `/`, `**` (exponentiation), `ABS`, `ATAN`, `ATANH`, `COS`, `COSH`, `EXP`, `LOG`, `SIN`, `SINH`, `SQRT`, `TAN`, and `TANH`. In addition, `REAL`, `IMAG`, and `CONJG` are non-generic operations requiring a complex argument. (In FORTRAN, `REAL` converts any argument to be a floating-point number; for a complex number, the real part is taken. In PL/I, `FLOAT` is used for conversion, and `REAL` is used only to extract a real part.) `ACOS` and `ASIN` (which exist in [ANSI 76b] only), `LOG10`, `LOG2`, `COSD`, `SIND`, and `TAND` forbid complex arguments. Complex numbers may have fixed-point (including integral) or floating-point parts, as long as both parts are of the same format.

The following rules apply to exponentiation $x{**}y$ of real numbers in PL/I:

$x{**}y = 0$ if $x=0$ and $y>0$.
$x{**}y$ = error if $x=0$ and $y\leq 0$.
$x{**}y = 1$ if $x\neq 0$ and $y=0$.
$x{**}y = x^y$ if $x>0$ and $y>0$.
$x{**}y = 1/x^{-y}$ if $x>0$ and $y<0$.
$x{**}y$ = error if $x<0$ and y not integral.

For complex numbers the PL/I rules are:

$x**y = 0$ if $x=0$ and *realpart*$(y)>0$ and *imagpart*$(y)=0$.
$x**y$ = error if $x=0$ and *realpart*$(y)\le 0$ or *imagpart*$(y)\neq 0$.
$x**y = cexp(y*clog(x))$ if $x\neq 0$.

APL did not originally include complex numbers. Various extensions were proposed over the years, and between 1977 and 1979 a series of articles [Penfield 77, Penfield 78a, Penfield 78b] explored possible design alternatives. Comments from the APL community were solicited and summarized [Penfield 78c]. Finally, a complete and coherent proposal for complex APL appeared [Penfield 79]; it is this proposal that I will describe here. I. P. Sharp Associates, a vendor of time-shared APL, has implemented this proposal with minor changes [McDonnell 81].

Because there is no agreeable way to order the complex plane, such functions as monadic ⍋ and ⍒ (sort), dyadic ⌈ (max) and ⌊ (min), and dyadic <, ≤, ≥, and >, are all restricted to non-complex operands.

The following mathematical functions have extensions to the complex domain which are either well-known or easily derived:

Monadic functions		Dyadic functions	
−	negation	−	subtraction
÷	reciprocation	÷	division
!	factorial	!	choose
○	multiplication by π	○	trignometric/hyperbolic
⌹	matrix inversion	⌹	matrix division
|	magnitude	+	addition
*	antilogarithm	×	multiplication
		⊥	base encoding

The generalization of ! is based on the gamma function of a complex variable (see [Kuki 72], for example). For obscure reasons, the unary | function (magnitude) may also be written as 1⍟0 [Penfield 79] or 10○ [McDonnell 81].

For dyadic * (exponentiation), the negative-numbers-to-rational-powers crock is eliminated; the result is always the principal complex value. The precise rules are:

$0*0 \leftrightarrow 1$.
$0*Y \leftrightarrow 0$ provided that the real part of Y is positive.
$X*Y \leftrightarrow *Y\times\circledast X$ otherwise (that is, $e^{y \log x}$).

A new function, *arc* or *phase*, written as 1⍟0 [Penfield 79] or as 12○ [McDonnell 81]), returns the principal value for the angle part of the polar form, in the range $(-\pi, \pi]$; the phase of zero is defined to be zero. The real-part function is 12○ [Penfield 79] or 9○ [McDonnell 81]; the imaginary-part function is 11○ [Penfield 79] [McDonnell 81].

Monadic ⍟ (natural logarithm) returns the principal complex value, always, with an imaginary part in the range $(-\pi, \pi]$. Dyadic ⍟ is defined by $X*Y \leftrightarrow (\circledast Y)\div\circledast X$, that is, $(\log y)/(\log x)$. Principal values for the trignometric and hyperbolic functions are not discussed in the proposal; Penfield tells me, however, that he has a paper in the APL 81 conference on the subject of consistent choices of principal values and branch cuts

for all these functions, and will send me a copy.

Another new function, nameless (written as ¯130 in [Penfield 79] and as ¯120 in [McDonnell 81]), computes e^{ix} given x. I have always known this function as *cis*, an acronym for "cosine-*i*-sine", because $e^{ix} = \cos x + i \sin x$; however, I have not located a source for this name. For real x, *cis* x is a complex number with unit magnitude and phase x mod 2π.

Monadic + (unary plus: a no-op) is compatibly extended and renamed "complex conjugate"; for obscure reasons this may also be written as ¯120 [Penfield 79] or ¯100 [McDonnell 81]. Monadic × (signum) is compatibly extended and renamed "direction"; the result is 0 if the argument is 0, and otherwise $\times X \leftrightarrow X \div |X$, that is, `(/ x (abs x))`, a complex number with unit magnitude and pointing in the same direction as X within the complex plane (that is, argument and result have the same phase). Another way to write this (for a non-zero argument) is as ¯130130X, that is, `(cis (phase x))`, but this is probably slower and less accurate for floating-point computation.

Note that in APL the real-part, imaginary-part, conjugate, and phase functions are completely generic, as are all APL operations; APL goes to great lengths to *hide* internal representations from the user, converting back and forth as necessary.

Having discussed these four languages, let me now make some observations about the nature of numerical functions in COMMON LISP so far. They are generic, in the sense of accepting all numeric types that make sense. They do not discriminate by type except to determine result type, in the sense that if a function accepts integers and also floating-point numbers, then giving the function `equalp` arguments produces `equalp` results. Thus `(sin 3)` and `(sin 3.0)` produce the same numerical result; `(+ 3 4)` produces 7 and `(+ 3.0 4.0)` produces `7.0`, and 7 and `7.0` are `equalp`. (This property is not true of `quotient` in MACLISP (and indeed most LISP systems): `(quotient 3.0 2.0)` => `1.5`, but `(quotient 3 2)` => `1`, and `1.5` and `1` are *not* `equalp`! COMMON LISP has of course fixed this for good reasons, and this, I argue, should be consistently maintained.)

Now let us consider, for example, what `(sqrt -1.0)` or `(log -1.0)` should do in COMMON LISP. I can suggest four ways to handle this:

** Alternatives:

A. Let `sqrt` and `log` accept scalars only, and produce scalar results. Provide no irrational functions on complex numbers.

Pro: This is easy to implement, and shields the naive user from unwanted complex results when working in the scalar domain. The expert user can write his own complex irrational functions or open-code them. This is the attitude of ALGOL 68, and to a lesser extent of FORTRAN (regarding the trigonometric and hyperbolic functions).

Con: It is counterintuitive not to implement mathematical functions where they are well-defined in a language which is otherwise generic. It is inconvenient for users to have to write their own mathematical functions; moreover, coding such functions is tricky, and ought to be done once and correctly.

B. Let `sqrt` and `log` accept scalars only, and produce scalar results. Require use of the functions

`csqrt` and `clog` to accept complex arguments or to produce complex results.

Pro: This helps to shield the naive user from unwanted complex results when working in the scalar domain.

Con: This introduces two copies of many functions, including potentially all the trigonometric and hyperbolic functions. Admittedly, FORTRAN and PL/I in effect take this approach, but it is not user-visible because the user can write simply `SQRT` or `LOG` and compile-time type analysis (usually) determines which run-time routine to use. However, COMMON LISP is not a strongly typed language.

C. Let `sqrt` and `log` accept scalars and complex numbers. However, if the argument is a scalar, then the result must be scalar or an error is signalled. The user can force a complex result from a scalar argument by explicitly converting the argument to be a complex number. Thus `(sqrt -1.0)` => error, but `(sqrt (complex -1.0))` => `#C(0.0 1.0)`.

Pro: This helps to shield the naive user from unwanted complex results when working in the scalar domain, and also avoids having two versions of every function. This is how FORTRAN and PL/I effectively behave for most purposes. (One cannot, however, in FORTRAN pass `LOG` as a functional argument, to be applied to a real or a complex with the caller not knowing which; the caller must name `ALOG` or `CLOG` explicitly.)

Con: This spoils the principle of type-nondiscrimination. If `sqrt` accepts 3 even though the result cannot be expressed as an integer, why should it not also accept `-1.0` even though the result cannot be expressed as a scalar?

D. Allow `sqrt` and `log` to produce results in whatever form is necessary to deliver the mathematically defined result. Thus `(sqrt -1.0)` => `#C(0.0 1.0)` and `(log -1.0)` => `#C(0.0 3.14159265)`.

Pro: This preserves the principle of type-nondiscrimination, and does not produce multiple copies of functions. This is most in the spirit of genericism. APL takes this approach. A survey of APL programs indicate that such functions as logarithm and sine are little-used anyway, and those who use them know what they're doing.

Con: This may have pitfalls for the unwary user. Also, it may be harder to compile (consider the S-1, which has a floating-point `log` instruction!).

D?

** Responses

MOON: CD	RMS: D!	SEF: A!!	GINDER: D!
DLW: X	HIC: D	WLS: D	CHIRON: X
ALAN: D!	GLS: D!!!	RPG: D!!!	

RPG: This has to be done right first, consistent with other languages second. The mathematical result should be the generic situation, but perhaps there should be special purpose operators for language compatability.

MOON: Provide a global variable which selects between alternatives C and D, with C being the default. I

guess, since it is less surprising. I don't think it is possible to win without a mode on this issue.

GLS: See [Penfield 78b] and [Penfield 78c] for reasons why a mode switch was deemed not desirable for APL.

DLW: I discussed this with Gosper at length; as of 15 November we still don't know what to recommend.

DM: Does anyone really understand all this stuff? I get a strong feeling of *deja vu*: feels just like PL/I, we're going to solve *everyone's* problems. All this stuff doesn't belong in a core language, but rather as a standardized add-on package, at the level of `loop` or `defstruct`. I also doubt that Lispers are really the people to decide how to do this sort of thing. Get some outside opinions from the non-LISP community. The concern with matching APL is, I guess, a step in that direction.

CHIRON: Hmm... seems what you really want to do is have a complex math package which you can optionally place between the current package and the usual LISP package. I think while a complex data type should exist, and simple operators should know about them (+, -, *, /), trgi, log, and power functions should ask you to load the complex part of the system. I think complex arithmetic should indeed be coded, but not specified in COMMON LISP. Let the implementor decide. Perhaps have it be a `(status feature)` entry. Keeping things simple and efficient are my primary reasons for wanting this, and also to keep the implementations flexible.

SEF: Complex numbers are not going into SPICE LISP for a long time (as long as I can make it). I don't care what is done about them as long as they *never* bother the user who does not bother them. Also, the less they clutter up the manual, the better. [GLS: Since this was written, SEF told me verbally that he would be happy to take complex-number code if someone else would write it and hand it to him free.]

***** Issue 82: Branch cuts and boundary cases in mathematical functions**

GLS: Questions arise in the definition of the irrational and transcendental functions: where shall branch cuts occur, and how shall boundary conditions be handled? The conservative approach to boundary cases, taken typically by FORTRAN and ALGOL 68, is that odd cases such as 0^0 and *phase*(0) shall be undefined. The advantage of this is catching as many program errors as possible. The liberal approach, taken typically by APL, is to define useful, if not unique, values for these cases; thus APL assigns the values `1` and `0` to the given example cases. The advantage of this is that many (though not all) useful identities continue to hold for boundary cases; also, this allows vector operations to succeed even though a few components may be odd cases. Penfield has written a paper on consistent choices of branch cuts for mathematical functions, intended for adoption by APL. I suggest we take a look at that and decide whether or not to adopt it.

**** Suggestion:** Tentatively consider compatibility with APL on the subject of branch cuts and boundary cases. **Y**

**** Responses**

MOON: Y	RMS: Y!	SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!!!	RPG: Y!!!		

GLS: Penfield's paper has appeared [Penfield 81]. It is also available as an MIT VLSI Memo.

10.1. Predicates on Numbers

10.2. Comparisons on Numbers

***** Issue 83: Fuzzy numerical comparisons**

SEF: Make "fuzzy equal" a function distinct from =.

DLW: Bill Gosper really wants = to work on more than two arguments. Maybe that would be more useful. You could have a separate function called `fuzzy=`?

GLS: Should there be other operations as well: `fuzzy<`, `fuzzy>`, `fuzzy-floor`, and so on?

**** Alternatives:**

A. Have a new function `fuzzy=` which takes three arguments: two numbers and a fuzz (relative tolerance).

B. The same, but the third argument is optional and defaults to some constant.

C. The same, but the optional third argument defaults in a way that depends on the precision of the first two arguments.

D. Have no fuzzy comparison functions.

C?

**** Responses**

MOON: C		SEF: A!	GINDER: C!	DM: BC!
DLW: X	HIC: B	WLS: B!	CHIRON: C!!	
ALAN: C!	GLS: C!	RPG: C!		

DLW: Consider (`fuzzy=` *tolerance* `&rest` *numbers*)?

***** Issue 84: Should = take more than two arguments?**

DLW: Bill Gosper really wants = to work on more than two arguments. Maybe that would be more useful.

SEF: I don't much care for the idea of extending = to multiple arguments, to about the same extent that I didn't like doing it with >. I'll go along if everyone else wants this. Then there's `eq`, `eql`, ...

GLS: If we had a ≠ function as well, that could be extended to more than two arguments as well. However, the current model would probably have to change. Right now, we think of (? *a b c d*) as meaning

```
(and (? a b) (? b c) (? c d))
```

for ? being any of the five relationals =, <, >, <=, and >=. However, this is not a suitable interpretation for ≠: we would probably like (≠ 3 4 3) to yield (), not t. The solution is that *all* pairs of elements must be compared, not just adjacent ones. Happily, this extends to the other five relationals as well; the adjacent-pairs rule is just an optimization made possible by transitivity.

** Suggestion: Allow = to take more than two arguments, in which case it is true if and only if the arguments are pairwise equal.

Y

** Responses

MOON: Y	RMS: Y!	SEF: N!	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: Y!	RPG: Y!!		

DM: If `equal` takes more than two arguments, then make sure `eq`, `eql`, etc., do too!

EAK: Is COMMON LISP going to be sensible and provide a <> function, or is it going to lose like NIL?

GLS: [Several paragraphs by GLS follow.] Well, I assume that you mean for (<> x y) to mean the same as (not (= x y)). This is as in PASCAL. However, when you make the extension to complex numbers, the name becomes inappropriate. On a totally ordered domain such as the reals, "not =" and "< or >" do happen to mean the same thing, by the law of trichotomy. However, complex numbers are not ordered, and so < and > are not well-defined, and so <> should also not be well-defined (although "not =" is well-defined).

Similar problems arise with IEEE proposed standard floating-point. The result of a floating-point comparison is tetrachotomous: the result may be <, >, =, or *unordered* (the latter may arise because the domain includes objects designated "not a number" which are produced by such situations as dividing by zero). In this domain <> and `not=` are distinct predicates, because the former is false and the latter true when the relationship is in fact *unordered.* (The S-1 supports this kind of arithmetic, although it is not true IEEE standard.)

PASCAL itself runs afoul of this problem! It allows = and <> to be used on sets, even though < and > are not defined on sets!

In conclusion, I would look very favorably on including a "not =" predicate in COMMON LISP, but I would be greatly opposed to calling it "<>". Perhaps "/=" would be an acceptable name? (This is one of three names for this operation permitted by ALGOL 68 [van Wijngaarden 77], the other two being a true not-equals symbol and the reserved word "ne".)

EAK: As you might expect, I've no great love for the name <>, but I do care about having the brevity of a simple form for the function. Besides <>, and /= which you suggest, there's PL/I's ↑=, and C's !=.

GLS: Of course, the ↑= of PL/I and != of C make sense within those languages because ↑ and ! are the respective logical negation operators for those languages. However, there is no precedent for this in LISP. (PL/I's ↑ has the additional disadvantage that its logical-not character is not in ASCII! It is transliterated into ASCII as a circumflex, and so appears as one here.)

EAK: Gee, if we weren't stuck with such losing terminals, we could use =<backspace>/. I wonder if that's why

they left ≠ out of ASCII in the first place? (very unfortunate that they did). Following the ↑= and != reasoning, there's `not=` for LISP. Sigh. As I said, pick for favorite (`/=` is fine with me), and document it.

ALAN: [83.3] Please find a better description [for `<=` and `>=`] than "monotonically nondecreasing" and "monotonically nonincreasing". I still can't figure out if these are correct.

10.3. Arithmetic Operations

***** Issue 85: Make rational a required data type for all implementations**

MOON: [85.2] Division of two integers should never produce a flonum. The result should always be a rational. Make rationals a required language feature rather than an optional one; they are at least as easy as bignums. In fact they were recently implemented for the Lisp Machine in three half-days.

DLW: Yeah!

SEF: Make rationals required. If someone will program them, I'll take them.

DLW: [7.8] I think it [provision of rational numbers [GLS: and also complex?]] should be required! It's very easy to do and quite valuable; if it's optional, it is far less useful.

GLS: The only reason for allowing `(/ 3 5)` => `0.6` was to avoid requiring implementations to support rationals. If everyone will support rationals, there is no problem with `/`. (By the way, ALGOL 68 [van Wijngaarden 77] always produces a real result when dividing two integers; you have to feed that to `round` or `entier` to get an integer out.)

**** Suggestion:** Make rationals (actually, ratios) a required data type of the language. **Y**

**** Responses**

MOON: Y!!!!	RMS: N!	SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!!!	HIC: Y!	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!!	RPG: Y!!!	DILL: Y!!!	

RMS: I vote no because bignums shouldn't be required either!

***** Issue 86: Why do `add1` and `sub1` still exist?**

MOON: Why are `add1` and `sub1` present if `plus` and friends are flushed?

GLS: I suppose I left them in because, after all these years, I still find `1-` incredibly tasteless and confusing. Every LISP manual including it has had to explain carefully that `(1- x)` computes $x-1$ and not $1-x$. I

suspect that LISP 1.5 had `add1` and `sub1` more for efficiency than cognitive reasons, but I may be wrong. I would be glad to eliminate all four of them.

** Alternatives:

A. Eliminate all four of `add1`, `sub1`, `1+`, and `1-`.

B. Keep `add1` and `sub1`; eliminate `1+` and `1-`.

C. Keep `1+` and `1-`; eliminate `add1` and `sub1`.

D. Keep all four of them.

E. Eliminate all four, and replace them with `inc` and `dec`.

** Responses

MOON: CE!!	RMS: D!	SEF: CD!!!	GINDER: E!	DM: E!
DLW: A	HIC: C	WLS: Y!	CHIRON: B!	
ALAN: C!	GLS: A!	RPG: E	DILL: ABCE!	

RMS: I vote D, but I now agree with you that `add1` is nicer than `1+`. `inc` does not subsume `add1`.

*** Issue 87: Add the function `signum`?

GLS: Add `signum` to the language. For a scalar, the definition is:

```
(defun signum (x)
  (cond ((zerop x) 0) ((plusp x) 1) (t -1)))
```

Following Penfield [Penfield 78a, Penfield 78c, Penfield 79], one can extend this to complex numbers as follows:

```
(defun signum (x) (if (zerop x) x (/ x (abs x))))
```

(Another way to define it is:

```
(defun signum (x) (if (zerop x) x (cis (phase x))))
```

where `(cis z)` computes e^{iz}, but I suspect this is numerically less accurate.) Penfield calls this function "direction"; it yields a unit vector in the direction of `x`, or zero if `x` is zero. Hence the range of the function is the union of the origin and the unit circle. I propose to retain the name `signum`.

** Suggestion: Add `signum` to the language as described above. **Y**

** Responses

MOON: Y	RMS: Y!	SEF: Y		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

MOON: Two-argument `signum` a la FORTRAN?

GLS: MOON evidently means the "transfer of sign" function called `SIGN`.

```
(SIGN a b) <=> (* (ABS a) (SIGNUM b))
```

except that this would define `(SIGN a 0)` to be 0, but in FORTRAN it is not defined in that case.

***** Issue 88: Add `numerator` and `denominator` to the language?**

MOON: There should be functions `numerator` and `denominator` which accept rationals (ratios and integers) and return the appropriate component of the lowest-terms representation. The value of `denominator` is always `plusp`.

GLS: Define "lowest terms" to mean that `(gcd (numerator x) (denominator x)) => 1`, always. (Among other things, this implies that `(denominator 0) => 1`.)

**** Suggestion:** Add functions `numerator` and `denominator`.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y	

***** Issue 89: Add least-common-multiple function?**

GLS: Add `lcm`, defined to be equivalent to:

```
(lcm x) <=> x
(lcm x y) <=> (/ (* x y) (gcd x y))
(lcm x y . more) <=> (lcm (lcm x y) . more)
```

**** Suggestion:** Add the function `lcm` of one or more arguments.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y

***** Issue 90: Extend `gcd` to complex numbers?**

GLS: For any two Gaussian integers, there exist four Gaussian integers such that they are largest in magnitude of all Gaussian integers which evenly divide the two given ones. (A Gaussian integer is a complex number whose real and imaginary parts are both integers. A Gaussian integer evenly divides another if the quotient is

a Gaussian integer.) These four numbers moreover are distributed one per quadrant, any three being equal to the fourth times some power of *i*. One can define *the* gcd of two Gaussian integers to be the one in the first quadrant (including the real axis and excluding the imaginary one). This can also be extended to rationals and complex rationals.

** Alternatives:

A. Extend `gcd` to Gaussian integers.

B. Extend `gcd` to rationals.

C. Extend `gcd` to complex rationals.

D. Status quo: `gcd` accepts only integers.

** Responses

MOON: AD	RMS: C!	SEF: D!	GINDER: X!
DLW: X	HIC: A		CHIRON: D
	GLS: C!	RPG: C!	

GINDER: I'd like to be able to get `gcd` of complexes when I want them, but not have to worry about getting them unexpectedly.

GLS: The properties of `gcd` are such that you can't get a ratio out if you put in only integers, and you can't get a complex number unless you put a complex in.

MOON: Factorization is meaningless for rationals.

GLS: Sort of, but the function is "greatest common divisor", not "greatest common factor" (a quibble). Anyway, the `gcd` of two rationals *x* and *y* is that rational *z* of largest possible magnitude such that *x*/*z* and *y*/*z* are both integers. Example: `(gcd 12/5 9/4) => 3/20`.

DLW: You must realize that we're getting into PL/I disease here. These things are nice, but their presence will discourage people from writing new COMMON LISP implementations because it will look too hard to do all this stuff. Can we say that we'll provide a "virtual machine" specification so that you can get all this stuff from us for free on a tape, written in COMMON LISP, or something?

GLS: Yes, such a specification and program data base is desirable; but it doesn't belong in the language definition as such, I think.

10.4. Irrational and Transcendental Functions

*** Issue 91: Arguments and results of irrational and transcendental functions

MOON: Except as noted, these functions accept any kind of arguments, but always return a floating-point result. If the argument is floating, the result will be of the same precision. (What about multi-argument case?)

GLS: If the argument is rational, to what floating-point format shall it be converted? I would suggest *single* format, on the grounds that that is probably adequate for the naive user, and expert users can insert an appropriate coercion function where appropriate: `(sqrt (double-float x))`.

GLS: In the multi-argument case, the rule given on page [81] should probably be followed: convert the shorter numbers to the precision of the longest one. Thus:

```
(atan 3.0s0 3.0d0)
   <=> (atan (double-float 3.0s0) 3.0d0)
   <=> (atan 3.0d0 3.0d0)
   <=> 0.7853981673974483d0
```

** Suggestion: Specify that, for irrational and transcendental functions: (1) arguments not in floating format are converted to *single* floating format; (2) all arguments are converted to the format and precision of the argument whose precision is largest; (3) the result has the precision of the arguments.

Y

** Responses

MOON: Y		SEF: Y	GINDER: Y!	DM: N!
DLW: Y!!	HIC: N	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!		

HIC: Am I confused? When is it ever okay to get extra precision?

*** Issue 92: Should `log` take two arguments?

??? Query: [86.5] Most LISP implementations, as well as other programming languages (such as FORTRAN), call the natural-logarithm function `log`. Mathematicians usually call this `ln`, however. It would be useful to have a two-argument logarithm function. One could let `log` serve for both the one-argument and two-argument versions, but then `&optional` arguments could not be used in the obvious way if one puts the arguments in the order normally used in mathematical notation, because it would be the first argument which is optional. Opinions?

ALAN: I haven't figured it out, but is it likely that `(log b x)` is going to lose less accuracy than `(/ (log x) (log b))` (or `(/ (ln x) (ln b))`)? If so, then having the order of arguments be `(log` *scalar* `&optional` *base*`)` with *base* defaulting to `10` would seem best to me. If not, then I would say to punt the issue and have `log` be *only* base `10`.

GLS: The only other programming language I've found with a two-argument logarithm function is APL, which puts the base argument first: `2⍟8` ↔ 3. However, I would still argue for it on the grounds of convenience, so the user doesn't have to remember the formula, and so we don't have to provide `log10` and `log2` as separate functions.

GLS: I have discovered that ALGOL 68 [van Wijngaarden 77] calls the natural-logarithm function *ln* and not *log*. By the way, if `log` is to accept two arguments, then an advantage of *not* letting one be optional is to help catch the incompatibility with MACLISP.

** Suggestion: Retain the definitions on the draft manual: `ln` is the natural logarithm, and `log` takes two required arguments.

Y?

** Responses

MOON: N	RMS: Y!	SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: N!	GLS: Y!	RPG: Y!!		

*** Issue 93: Complete set of trigonometric functions?

GLS: How about the missing trig functions? For example, `(/ (sin x) (cos x))` is a terrible way to compute a tangent.

** Suggestion: Add `asin`, `acos`, and `tan`.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

*** Issue 94: Degree-style trignometric functions

MOON: Lisp Machine LISP introduced the degree-style trig functions, but I think they probably ought to be flushed.

DLW: Not clear.

SEF: Harmless enough to keep them.

GLS: PL/I and LOGO have them.

** Alternatives:

A. Flush trig functions that take arguments in degrees.

B. Status quo: retain `cosd` and `sind`.

C. Retain `cosd` and `sind`, and add `tand` also.

D. Have all of `cosd`, `sind`, `tand`, `acosd`, `asind`, and `atand`.

D?

** Responses

MOON: AD!	RMS: A!	SEF: AD!	GINDER: D!	DM: D!
DLW: D!!	HIC: D	WLS: D	CHIRON: D!	
ALAN: A!	GLS: D!	RPG: C!!		

RMS: I think providing `pi` is good enough.

***** Issue 95: Hyperbolic functions**

ALAN: [86.8] How about `sinh` and `cosh`? (You can lose using $(e^x - e^{-x})/2$, or whatever it is.)

GLS: Actually, I believe it is `tanh` that is very subtle to do correctly using floating-point numbers. If we can get them right, it might save some loser some grief.

**** Alternatives:**

A. Status quo: no hyperbolic functions.

B. Add `sinh`, `cosh`, and `tanh` to the language.

C. Add `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, and `atanh`.

**** Responses** **C**

MOON: AC!	RMS: C!		GINDER: C!	DM: C!
DLW: C!!	HIC: C	WLS: BC!	CHIRON: C	
ALAN: C!	GLS: C!	RPG: C!!		

***** Issue 96: Are several versions of pi necessary?**

MOON: [87.8] Is `short-pi` not equal to `(short-float pi)`? I would assume that if the rounding is done properly, these would be `=`, in which case the other four variable names are superfluous. Flush all these variables for `pi`; just have one. If you aren't going to have variables for `pi/2` and so forth, the user needs to assume these constant computations are done at compile time anyway.

SEF: Sounds good to me.

GLS: The one language that gives you π in a choice of precisions is ALGOL 68 (one may write "*long long long pi*", for example). So in effect one does have just a single variable, and one writes a precision modifier in front of it if necessary. A difference is that *pi* is by default the *shortest* one available, and you have to modify it to get longer ones. However, the longer ones are required to be accurate to their format's precision. APL also provides π if you write `○1` (π times 1), but precision is not controllable in APL.

**** Suggestion:** Eliminate the variables `short-pi`, `single-pi`, `double-pi`, and `long-pi`, retaining only `pi`. Encourage the user to write such things as `(short-float pi)`, `(single-float (/ pi 2))`, etc., when appropriate.

Y

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: X!
DLW: Y!	HIC: Y	WLS: X!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!		

*** Issue 97: Other constants besides pi?

ALAN: [87.9] How about `e` [as well as `pi`]?

GLS: Shall we also have other constants, such as Euler's constant and the fine-structure constant? Where do we stop? Also, `e` seems to be too good a name to gobble.

** Suggestion: Define the variable `e` to be `2.718281828...L0`.

N

** Responses

MOON: N	RMS: N!	SEF: N	GINDER: Y!	DM: X!
DLW: N!!	HIC: N!	WLS: X!	CHIRON: N	
ALAN: N!	GLS: N	RPG: Y!		

HIC: What about something like (`constant` *key format*)? Then, since the constant is identified by a keyword, we can add as many different constants as we want without eating up names.

MOON: A variable for *e* would be all right if it had a better name. Otherwise `#.(exp 1)` will probably suffice.

DM: Instead of having the variables (what's going to happen when somebody transcribes his physics calculation that uses a (free) variable named `pi` to hold a matrix or something?), and all the other interesting constants, why not have a constants function? Thus instead of `pi` you use `(const ':pi)`. Perhaps dimensional quantities could be handled, too, where you supply optional arguments to specify the units you want. For example, `(const ':c '(/ :m :s))` for the speed of light in meters per second, or some such.

10.5. Type Conversions on Numbers

*** Issue 98: Optional second argument to `float`

MOON: [88.2] `float` should take an optional second argument. If present, the first argument is floated to the same precision as the second. Among other things, this is used for functions which are supposed to return results of the same precision as their argument (but do the intermediate computations in extended precision).

** Suggestion: Allow `float` to take an optional second argument as described.

Y

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
	GLS: Y!!	RPG: Y!!		

MOON: Gosper suggests calling it `(coerce x y)` instead of `(float x y)`, and have it fix `x` if `y` is fixed, rationalize if rational, etc. The name `coerce` is too general, but this does sound like an improvement.

*** Issue 99: Is the function `rational` useful, given `rationalize`?

MOON: [88.7] Is `rational` useful for anything vis-a-vis `rationalize`?

GLS: It's faster, and closer to the idea of straightforward type coercion.

** Suggestion: Retain `rational`.

Y?

** Responses

MOON: Y		SEF: Y	GINDER: Y!
DLW: X	HIC: Y	WLS: Y!	
ALAN: N!	GLS: Y!	RPG: Y!!	

DLW: Gosper points out that `rational` and `rationalize` need to be defined more strongly and carefully. For example, what happens for `(float (rational[ize] x))`? Does it = *x*?

*** Issue 100: Optional argument to `rationalize` to specify precision

MOON: [88.7] Should `rationalize` take an optional second argument of the number of bits of precision? (If negative, the number of least-significant bits to discard.) It would very often be used with a second argument of `-1` for rationalizing results of floating computations.

GLS: Why `-1` rather than `-2`? Also, I am a little suspicious of measuring floating-point numbers in "bits". While that may be easy to implement, it would seem to be more general to specify a relative tolerance.

SEF: Hmmm. I am reluctant to get into the business of diddling precision bits within a particular flonum format. Maybe provide an optional package that really does all this stuff right, carrying around the precision of everything and allowing big flonums, etc.

** Alternatives:

A. Status quo: let `rationalize` take no second argument.

B. Allow `rationalize` to take an optional second argument, an integer, which is the number of bits of precision to observe (if negative, a number of bits of precision to ignore).

C. Allow `rationalize` to take an optional second argument, a relative tolerance. The effect of a negative number of bits can be obtained if one assumes that one can obtain from the system the relative tolerance for the precision of a given floating-point format.

** Responses

MOON: B!!!		SEF: A!!		DM: C!
DLW: X	HIC: C		CHIRON: C	
ALAN: B!	GLS: C	RPG: C!		

MOON: Maybe provide both alternatives B and C, where a fixnum is a number of bits and a flonum is a relative tolerance. This applies to issue 102 also. B is the usually desired case, since the reason you want `rationalize` rather than `rational` is to compensate for lack of perfect accuracy in the floating-point computations, not because there is some particular desired precision in your program, usually.

DLW: See remark on previous issue.

SEF: Specify that `rationalize` returns a rational number for which the given floating-point number is the best available approximation *of its format.*

*** Issue 101: Rename `remainder` to be `rem`?

MOON: I sure wish `remainder` could be called `rem`.

SEF: What's wrong with `remainder`? Use auto-abbreviation if you don't like to type.

GLS: PASCAL and ADA both call this operation `rem`.

** Suggestion: Rename `remainder` to be `rem`, conditional on the sequence functions being juggled so that the name `rem` is not needed there.

Y?

** Responses

MOON: Y	RMS: Y!	SEF: Y	GINDER: N!	DM: N!
DLW: Y	HIC: Y!	WLS: Y!	CHIRON: N!!	
ALAN: Y!	GLS: Y!	RPG: Y	DILL: N!	

RMS: This by itself is not important enough to be considered a reason for flushing the existing function `rem`. Besides, I'd rather change \ to \\.

*** Issue 102: Optional precision argument for `mod` and `remainder`?

MOON: In the Lisp Machine LISP proposal, `mod` and `remainder` also take an optional argument which is the minimum number of bits of precision acceptable in the result. An error is signalled if the result would have

fewer. The comparison is allowed to be approximate (e.g., simple comparison of exponents). Consider `sin`, which starts by `mod`'ing its argument with 2π. Note that this number of bits is *not* an implementation-dependent parameter. This got dropped somehow from my proposal which was the source of `floor` and so forth.

SEF, GLS: May have same problems as for `rationalize` above?

** Alternatives:

A. Status quo: let `mod` and `remainder` take no second argument.

B. Allow `mod` and `remainder` to take an optional second argument, an integer, which is the number of bits of precision to observe (if negative, a number of bits of precision to ignore).

C. Allow `mod` and `remainder` to take an optional second argument, a relative tolerance. The effect of a negative number of bits can be obtained if one assumes that one can obtain from the system the relative tolerance for the precision of a given floating-point format.

** Responses

MOON: B!!!		SEF: A!!		DM: C!
DLW: C!	HIC: C		CHIRON: C	
ALAN: B!	GLS: C	RPG: C!		

DLW: C, I guess, but please make it very clear in the manual just what this means; I don't grok it. How is the "amount of precision" in the answer determined so that you can compare it?

*** Issue 103: Extend `floor` and friends to complex numbers?

GLS: The APL community has been exploring extensions to `floor` and friends to complex numbers. If x is complex, then (`floor` x) is a Gaussian integer (a complex number whose real and imaginary parts are both integers). Given `floor`, then `ceiling` and `mod` are easily defined. There are two proposals outstanding: [McDonnell 73] and a paper by Forkes to appear in the APL 81 conference proceedings. McDonnell omitted extension of floor in his complex APL implementation [McDonnell 81] pending resolution of these conflicting proposals.

** Suggestion: Extend `floor` and friends to complex numbers in a manner compatible with APL, as and when the APL community gets its act together and presents a non-bogus proposal.

Y

** Responses

MOON: Y		SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: Y	RPG: Y!!		

GLS: The paper by Forkes has appeared [Forkes 81]. The results do not appear to be at all conclusive, so I would recommend punting this issue for now, and watching the APL community fight this out.

10.6. Logical Operations on Numbers

***** Issue 104: Restore `boole` to the language?**

RMS: [91.5] It looks like you plan to flush `boole` and replace it with explicit names for all the two-argument logical operations that are useful to write into source code. However, it is very useful that `boole` operations include the trivial ones, so that a graphics operation can provide a parameter which is a `boole` code ("ALU function") and one value of the parameter says to copy the new data onto the screen, ignoring the old data entirely. If there is no `boole`, it is necessary to pass a function such as `logior` as an arg instead, so there must be functions provided for all six two-argument logical operations which don't really use both arguments. In addition, `(funcall alu-function x y)` is likely to be a lot slower than `(boole alu-function-code x y)`. So I think it is better to keep `boole`, and provide standard names for all the possible first arguments; and keep only `logand`, `logior` and `logxor`. I'm not sure `lognot` is needed since `(logxor -1 ...)` is so easy.

GLS: Indeed, I was only thinking of writing fixed constants in the code, and not of parametrized applications. I agree: bring back `boole`.

SEF: Sounds good to me. However, we should keep `lognot`, since it is more perspicuous than `(logxor -1 x)`. I would enjoy flushing `andc2` and friends, replacing them with a `boole` whose arguments are keywords, not numbers.

**** Suggestion:** Define `boole` as in MACLISP, but let it apply to all integers. Do not specify what the values for the first argument are, however. Instead, define sixteen global variables:

`logclr`	`logset`	`logxor`	`logeqv`
`logand`	`logior`	`lognand`	`lognor`
`logandc1`	`logandc2`	`logorc1`	`logorc2`
`log1`	`log2`	`logc1`	`logc2`

whose values are implementation-dependent but cause specified things to happen when given as a first argument to `boole`.

**** Responses** **Y**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y!	WLS: X!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!!	DILL: Y	

RMS: It is better to define what the values of the keywords are. One should not leave something undefined just out not seeing offhand why anyone would want to know. I agree that it will be better for programmers to use the keyword names, and whenever they write a fixed one they should use the name; but it may be useful to compute one. Consider a PDP-10 simulator that wants to do `(boole (ldb ...) ...)` to simulate the logical instructions! In other words, do not be to quick to ask "does the user *need* to know". Instead ask, "does it do much good not to tell him".

GLS: The reason I left it undefined is that it may be good not to tell the user on the grounds that the implementation may be able to exploit the freedom. If I recall correctly, the PDP-10 and the standard TTL 74181 ALU do not use the same codes! Let the PDP-10 simulator written in COMMON LISP use the more transparent

```
(setq pdp-10-boole-table (vector logclr ... logset))

(boole (vref pdp-10-boole-table (ldb ...)) ...)
```

MOON: I don't like the names `log1` and `log2`. I don't have a better suggestion yet. Why not make the first argument to `boole` a symbol rather than a number?

GLS: I don't like the names either. The suggestion doesn't say the first argument is a number; it says the values are implementation-dependent. The reason for that in turn is so that they can be numbers *if desired*, for speed (as for graphics).

MOON: [93.8] [`haulong` performs the computation *ceiling*(log2(*abs*(*integer*)+1)).] Maybe this expression should be written in LISP? The expression is in fact correct, but unnecessarily obscure?

GLS: This was written before `ceil` was added. Now it can be written as

```
(ceil (log 2 (+ (abs integer) 1)))
```

This may be obscure, but it is accurate, and I haven't found a better way to express it. (Actually, maybe I shouldn't write it in LISP form. That implies the use of floating-point numbers of potentially insufficient precision, whereas `haulong` always produces an accurate result.)

10.7. Byte Manipulation Functions

***** Issue 105: Reverse the order of arguments to `byte`?**

MOON: The arguments to `byte` should be *size, position* rather than *position, size*. There is a typo at the top of page 94 which supports my contention that this is a more natural order.

** Suggestion: Reverse the order of arguments to `byte`.

Y

** Responses

MOON: Y!!!	SEF: Y	GINDER: Y!	DM: Y!
	WLS: Y!	CHIRON: N!	
	RPG: Y!	DILL: N!	

10.8. Random Numbers

***** Issue 106: Definition of `random` of one argument**

MOON: Is it the case that `(random n) <=> (mod (random) n)`?

GLS: I don't see why this restriction needs to be imposed. It may be a valid way to implement it, but the user needn't know that.

**** Suggestion:** Define `(random n) <=> (mod (random) n)`, always.

N?

**** Responses**

MOON: N		SEF: N	GINDER: N!	DM: N!
DLW: N	HIC: Y	WLS: Y!	CHIRON: N!!	
ALAN: N!	GLS: N!	RPG: N!		

DM: `(random (* 0.66 largest-fixnum))` has a rather different distribution from `(mod (random) (* 0.66 largest-fixnum))`. I don't suppose many implementors currently take note of that, but they should. If the argument to `random` is larger than some tolerance (say 1% of the largest fixnum) something sexy should be done to try to keep the distribution reasonably flat.

GLS: DM is quite right. Indeed, inasmuch as the draft COMMON LISP manual specifies that `(random)` may return an integer from some limited range, and `(random n)` must draw with approximately equal probability from the integers from zero to $n-1$, `(mod (random) n)` cannot possibly be a valid implementation if n is greater than the range of `(random)`, because some numbers between zero and $n-1$ would never be chosen.

***** Issue 107: Random floating-point numbers**

GLS: Perhaps `(random x)` where `x` is a positive floating-point number should return a floating-point number of the same format, between zero (inclusive) and the number (exclusive).

**** Suggestion:** Allow `random` to accept a floating-point number as described.

Y?

**** Responses**

MOON: Y		SEF: N!!		DM: N!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: Y	RPG: Y!		

SEF: Actually, I am happy to accept this, but only if someone hands me a working implementation *before* this goes into the manual.

MOON: Should also generate random complex numbers, presumably within a specified circle.

GLS: Or square? I don't know what FORTRAN people use. I do know that in the FORTRAN world random floating-point numbers with either uniform or Gaussian distribution are much more used than random integers.

***** Issue 108: Random number initialization and seeding**

MOON: Need a way to reset the random number generator to a known state, and a way to seed it from a random source, to do anything non-trivial with it.

DLW: The names for resetting and seeding in Lisp Machine LISP are losers but the idea is right.

**** Suggestion:** ??????

**** Responses**

MOON: Y		SEF: ?	
DLW: X	HIC: X	WLS: Y!	CHIRON: X
	GLS: Y	RPG: ?!!	

DLW: I'll try to have a proposal ready at the meeting.

CHIRON: By seeding to a number *x*, you are resetting it to a known state. Have a function `makeseed` which reads the clock, the process number, and other implementation-specific information to make an arbitrary seed; and a function `seed-generator` to seed the generator with.

Chapter 11

Characters

***** Issue 109: Character set and representation independence**

DLW: Mention that no particular character set is needed and that some implementations may use fixnums to represent characters.

MOON: [The predicate `characterp` may be used to determine whether any LISP object is a character object.] Even in implementations that don't have them?

MOON: [29.0] In Lisp Machine LISP `characterp` will be the same as `fixnump`.

GLS: Well, the document "Character Standard for LISP" which I wrote allowed for characters not to be a separate data type. The COMMON LISP manual as it stands is at best silent on the issue. Can Lisp Machine LISP not be persuaded to implement a character data type? Lisp Machine LISP (though not COMMON LISP in general) could permit arithmetic on such objects, if upwards compatibility is a problem. Not having character be a separate data type has ramifications for `typep` and `typecase` that I would rather avoid.

**** Suggestion:** Do not require `character` to be a distinguishable data type, but make it a "virtual" data type, like "a-list". Continue to permit `(typep x 'character)`; in some versions of COMMON LISP this might be true of integers or even of conses.

Y?

**** Responses**

MOON: X	RMS: Y!!!	SEF: Y!	GINDER: N!	DM: Y!
DLW: Y!!!!	HIC: Y!	WLS: Y!	CHIRON: N!!!	
ALAN: Y!	GLS: N!	RPG: Y!	DILL: N!!	

RMS: Character objects are entirely unnecessary.

GLS: I worry that users will be seduced into writing non-portable code by performing arithmetic on fixnum-represented characters without using the appropriate conversion functions as interfaces.

MOON: My main objection to character objects is that there have to be two copies of every function and every array type, as well as that you can't do arithmetic on them. If those crocks could be gotten rid of, I wouldn't mind putting them into Lisp Machine LISP.

***** Issue 110: Having bits and font components in the same character**

MOON: It really does not make sense to have bits and fonts in the same character; keyboard characters and display characters are two rather different things.

GLS: A counterexample: consider an editor somewhat like Stanford's E, which uses the META key to distinguish inserting from overwriting. Now suppose the keyboard also has an Italic or Greek key, which might be most conveniently interpreted by the keyboard interface as a font specification, say 1 for italic, 2 for Greek, and 3 for italic Greek [is there such a thing??? if not, then say APL characters]. Then the character `#1\K` would be a command to overwrite the current character with a "*K*", but `#1\Meta-K` would be a command to insert a "*K*".

**** Suggestion:** Do not have bits and fonts in the same character; have two kinds of character as in Lisp Machine LISP, one with bits and the other with fonts.

**** Responses** **N**

MOON: N	RMS: N!	SEF: N!!!	GINDER: N!	DM: X!
DLW: N!		WLS: Y!	CHIRON: N!!	
ALAN: Y!	GLS: N!!	RPG: N!!		

RMS: Implementations must be permitted not to allow bits and font in the same character, but I don't think it does much harm to allow them. It also doesn't do much good to allow them. So, if the whole system of character functions can be made simpler if this is not allowed, then do that. Otherwise, continue to allow them.

MOON: What I said about bits and fonts in the same character wasn't very reasonable; I guess the real issue was packing into 16-bit bytes, which can be handled and anyway shouldn't be a consideration at this level.

DM: This whole sort of extended character notion rather bothers me. It seems wrong that implementation considerations force us into the bizarre situation of saying that only a subset of the possible characters may occur in strings. I'd rather view characters as the simple, old-fashioned things, and build up to these nifty new things with extra shift keys or fancy fonts as structures which contain a "primitive" character.

11.1. Predicates on Characters

***** Issue 111: Choice of standard character set**

MOON: I don't understand the motivation for picking those four special characters [to be standard: <tab>, <form>, <return>, and <rubout>.]

GLS: Those four characters have equivalents in EBCDIC, except posibly <form>. The first three are a minimal set, with <space>, of whitespace characters for formatting programs into pages, and <rubout> is needed for input editing. <backspace> seemed less useful, as terminals disagree on whether or not one can use it to do

overstriking.

DLW: Do you mean to say that all implementations, ASCII or not, must have these characters in their character sets, and the others are optional?

GLS: Right. An implementation must provide the standard characters, or transliterations of them; one needs to know the minimum number of distinguishable characters in which to write code, and what their purposes are. Additional characters may be provided; the code/font/bits organization provides a framework for their inclusion.

** Suggestion: Retain the current definition of standard characters for COMMON LISP.

Y?

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: X	HIC: X	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!!		

DLW: I don't understand what these are for and how they are defined; in particular, what is <tab>? Systems vary in how many columns wide they are. <backspace> is hard to define, too. <space> is clear enough. More important, can we arrange things so that things like `#/Abort` can be defined to *read* without error in all COMMON LISP implementations, so I can run-time conditionalize its use?

GLS: This does need clarification, but I think it isn't hard. We can define <tab> to be a whitespace character other than <space>, which has implementation-dependent effect on the cursor when printed; the main reason for making it standard is to ensure compatibility of program text files, which are likely to contain <tab> characters (and likewise for <form> characters). <backspace> is not a standard character. <space> is perfectly clear, and indeed absolutely transparent.

*** Issue 112: Should font 0 be specified to be fixed-width?

MOON: Isn't the specification that font-0 graphic characters are all the same width rather outside the domain of a language definition?

GLS: The possibly misguided intention was that programs could depend on this property in order to do tabular formatting, as with `format`.

** Suggestion: Remove the specification that font 0 be fixed-width from the language.

Y?

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: N!
DLW: N!!	HIC: Y		CHIRON: Y	
ALAN: N!	GLS: N	RPG: Y!	DILL: Y	

11.2. Character Construction and Selection

***** Issue 113: Null arguments to `character` and `code-char`**

MOON: [102.1] For both `code-char` and `character`, it would be better to signal an error than to return `()`, both for better robustness of carelessly-written programs and to permit type declaration and associated optimization on conventional machines.

GLS: The intent was to have a way to discover whether the implementation could support a given combination of character components; thus these two functions serve as predicates. Maybe this was a bad idea? Perhaps the three components should be made completely orthogonal.

**** Suggestion:** Disallow results of `()` from `character` and `char-code`.

**** Responses**

MOON: Y	RMS: N!	SEF: Y!		DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: N	RPG: N!!	DILL: Y	

RMS: I agree with your reasoning. Does this become any easier if characters cannot have both bits and font?

***** Issue 114: Make `code-char` a generic function?**

MOON: Why not just let `code-char` accept either a number or a character as its first argument?

GLS: The intent was that `code-char` and `character` be easy to open-code. Maybe `code-char` should just be generic, as most of the language is turning out to be (but then the name should be changed, as it is then even less symmetric with `code-font` and `code-bits`?).

**** Suggestion:** Make `code-char` be generic.

**** Responses**

MOON: Y	RMS: Y!	SEF: N!!	GINDER: Y!	DM: Y!
DLW: N	HIC: Y	WLS: Y	CHIRON: N	
ALAN: Y!	GLS: N	RPG: Y!	DILL: Y	

***** Issue 115: Change the meaning of the `character` function?**

MOON: The function `character` is incompatible with the usual interpretation as a type-coercion function. `character` of a non-character is an error and furthermore `character` of a character does not necessarily return the same character.

** Suggestion: Make `character` be a type-coercion function which converts arguments of certain types to be a character.

Y

** Responses

MOON: Y	RMS: Y!	SEF: N	GINDER: Y!
DLW: Y!	HIC: Y	WLS: Y!!!	CHIRON: N!
ALAN: Y!	GLS: Y	RPG: Y!	

11.3. Character Conversions

***** Issue 116: Should `digit-char` be allowed to return `()`?**

MOON: As with `code-char`, `digit-char` should produce an error rather than return `()`.

GLS: I think the argument that `digit-char` is a useful predicate is very strong here, stronger than for `character`.

** Suggestion: Require `digit-char` to error rather than produce `()`.

** Responses

MOON: Y	RMS: N!	SEF: N!	GINDER: N!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: N!	RPG: N!!	DILL: N	

***** Issue 117: What are eof objects?**

DLW: What is `#\eof`? What is its type? Can there be more than one of them?

GLS: `#\eof` is a piece of nonsense whose purpose I have forgotten. I think the idea was that it would be `-1` in MACLISP and something else in Lisp Machine LISP; it was a compatible way of referring to the implementation's standard eof value, and pandered to MACLISP's needs for limited numeric type declarations. I think the whole concept is best eliminated from COMMON LISP.

** Suggestion: Eliminate `#\eof` from COMMON LISP.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!!	HIC: Y!	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

11.4. Character Control-Bit Functions

*** Issue 118: Names for character control-bit functions

ALAN: [104.8] [`control` and friends are] fairly good names to gobble down for such trivial functions.

MOON: Many common English words, especially ones which are likely to be used by programmers, are used up for new system functions. I don't think these words should be used for these functions. The most egregious example is `control`, a function which sets the control-shift bit in a character. This function should have "`char`" or "`character`" in its name: use `char-control` or `control-char`, etc. Maybe there should be one function which takes multiple arguments rather than a separate function for each key.

MOON: Why do `control` and friends produce `()` if given `()` as an argument?

DLW: I bet it's so you can nest two of these! But I agree that this is a job for exceptional conditions.

GLS: The main reason I had for using `control` instead of `control-char` was that a nice syntax was needed for reading and printing control characters. Because we can't count on every COMMON LISP implementation having Greek characters available, I had thought that perhaps something like `#,(control #\A)` would be acceptable for Control-A, or `#,(control (meta #\Space))` for C-M-Space (DLW's guess is therefore quite correct). With the invention of the much better syntax `#\Control-Meta-Space`, this reason disappears. We could rename these functions `control-char`, `meta-char`, etc.; or we could get rid of some names by using a two-argument function `(control-bit` *char integer*`)`: *integer* should be `0` or `1`, which is installed as the control-bit of char and returned.

** Alternatives:

A. Rename `control`, `uncontrol`, and `controlp` to be `control-char`, `uncontrol-char`, and `control-charp`. Similarly for `meta`, etc.

B. Replace them by a single function `control-bit`: `(control-bit c)` returns `0` or `1` according to whether the control bit of `c` is clear or set, and `(control-bit c n)` sets the control bit of `c` to be `n`.

B?

** Responses

MOON: X	RMS: B!	SEF: A!!	GINDER: A!	DM: A!
DLW: B!!	HIC: B	WLS: A!	CHIRON: B!!	
ALAN: B!	GLS: B!	RPG: B!	DILL: A	

RMS: The user does not need to know about two-arg `control-bit`, since he can use `setf` on the one-arg form.

MOON: I would prefer `(control-char c)` => `t` or `()`, and `(control-char c z)` => `c1`, where `z` is `t` or `()`. Let's avoid multiple representations for true and false.

WLS: Why not separate one-argument `control-bit-p` and two-argument `control-bit`?

Chapter 12

Sequences

***** Issue 119: Generic sequence coercions**

SEF: Various conversion functions are missing, such as `vector-to-string`, `list-to-bit-vector`, `to-list`, and so on.

JONL: Regarding generic coercions. Regardless of whether or not we have functions like <type1>-to-<type2> (e.g., `vector-to-list`), we will have to have the general coercion to each type, which does the straightforward things for converting from one sequence type to another (we can fight later about how to convert a `fixnum` into a `vector`!). Since MACLISP and NIL already have `to-list`, `to-vector`, `to-string`, `to-bits`, `to-symbol`, `to-character`, `to-upcase` (and a few other "internal" ones), then it is trivial to make something like `vector-to-list`:

```
(defun vector-to-list (v)
  (check-type v #'vectorp 'vector-to-list)
  (to-list v))
```

Analogously, `subseq` (or "`subsequence`" if you must) can subsume all the `sub-`<type> functions.

GLS: It is unfortunate that two conflicting conventions have become somewhat well-established: for non-decomposable objects, the name of the type is a coercion function (`float`, `rational`), while for decomposable objects the name of the type is a constructor (`list`, `vector`).

**** Suggestion:** Define generic coercion functions `to-list`, `to-vector`, `to-bit-vector`, `to-string`, and so on.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!!	RPG: Y!		

DM: They are a little (?!) excessive. On the other hand, they're all things somebody or other needs at some time or other. Rather than having each guy that needs one writing it himself, and giving them some totally random name (how many functions do you know with people's initials embedded in them like `dfmappend` or nice mnemonics like `foofun`?), it would be nice to have some fairly standard function he could use. here again, I think the idea of a standardized library wins. In fact, with a suitably flexible package system, you

could accomplish wonders.

***** Issue 120: Sequence functions and multidimensional arrays**

ALAN: [107.4] [On sequence functions handling multidimensional arrays in row-major order:] This gross "sequence" generalization has always made me uncomfortable, but including 2D arrays this way seems more square-peg-in-round-hole than the rest. The operations are *frequently* going to be *very useless.* [108.9] What does `subseq` do with a 2D array?

GLS: While `subseq` may be useless on 2D arrays, `position` or `map` may be more useful. There is a consistent model for what's going on which is similar to that in APL, and people have found APL's row-major order convenient (though you have to explicitly ravel an array before you can treat it as a vector). The model is that an array is a pair of two things: a linear vector of data items, and a linear vector of dimensions. To select an array element, one provides an index vector which matches the dimension vector in a certain way (in particular being of the same length), performs a mathematical calculation to determine a single integer index, and uses that to index into the data vector. Thus linear vectors are, in this model, more primitive than arrays. Indeed, a simple vector is a pair of a linear vector of data and a single dimension which is an integer rather than a vector. Just as an array subscript must be a vector conforming to the array's dimension vector, so a vector subscript must be an integer conforming to the vector's dimension integer. (This is why it is sensible theoretically, as well as implementationally, to distinguish vectors from 1D arrays.)

**** Suggestion:** Continue to permit generic sequence functions to operate on multidimensional arrays by effectively using the underlying row-major order data vector.

**** Responses**

MOON: N		SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!!	HIC: N	WLS: Y!	CHIRON: Y	
ALAN: N!	GLS: Y!	RPG: Y!!	DILL: N	

WLS: Yes, but only if you say what `subseq`, `remove`, and `concat` do.

MOON: Provide a `ravel` function.

GLS: Call it `bolero`? Seriously, that sounds great to me. I'd change my vote to no if that were added. More robust. Maybe specify that when given a multidimensional array it just creates a displaced array so that they share data?

***** Issue 121: All those ridiculous sequence functions!**

MOON: I sent a message some time back suggesting that there are too many names, the name-abbreviation is not always consistent enough to be easy to use, and optional arguments can be used [to pass flags and predicates]. I repeat the suggestion (although I can see arguments against it).

MOON: The hundreds of functions which are the same as the general sequence functions except for a built-in data type declaration are ridiculous. Optimizing special cases like this should be the business of the compiler and the implementation, not the programmer. There should be some general data-type declaration wrapper which allows the programmer to declare type information in this case (and the many other cases not provided for at all in the draft language); a generalization of MACLISP's `fixnum-identity` special form.

SEF: Something like `(length (type ':list foo))` is indeed the cleanest logically, but I find it ugly. I think our only other option is `(length foo)` for true generics and `(typed-length ':list foo)` if we want to declare.

JONL: There are indeed just too many names for sequence functions. Maybe we ought to first let some development take place to see how many deserve "advertising" in the language; we could still offer an optional package which links up the full set of names to appropriate subroutines. For example, we could totally flush the "dimension" of names which prefix the data type; I know that I may have been instrumental in maintaining this in the past, but all things considered now, the open-coding options appear to be less often used than one might have thought, and there is still an option for the compiler to detect mechanically most open-codable cases. However, we may want to keep all the names `string-`*xxxx* which are in Lisp Machine LISP, mostly for their utility in making ZWEI-like editors, but also partly for the slightly-incompatible nature of these functions (admitting symbols, etc.). Also, we would have to leave things like `bit-ior`, `bit-and`, etc., since these aren't generic.

RMS: I am not sure that there is any need for all our favorite functions such as `memq` and `nreverse` to become generic, but if they do, I am sure that there should not be functions `list-nreverse`, etc. The only purpose for those additional functions is to save a run-time type check whose cost is very small compared with the cost of the function itself in any of the possible cases. So the gain is very small. It is better for people to use the generic function all the time. This is the same as the MACLISP distinction between + and `plus`, except there the saving is significant because the operation is so simple. If you don't think that non-generic arithmetic functions are worth keeping, then you certainly shouldn't want to add non-generic structure functions. If you do want to have a way for the user to indicate that the argument is a list, provide a function `i-am-a-list`, and let the user do `(nreverse (i-am-a-list foo))`. Then the user can trivially define as macros any functions such as `list-nreverse` that he wants to have.

SEF: Guess I agree that we just go ahead and use the generics most of the time. Provide two forms for each, `(length x)` and `(typed-length ':list x)` where the `':list` can be any type-expression.

GLS: The only problem with wrappers is that

```
(vector! (concat (vector! x) (vector! (car y)) (vector! z)))
```

(where I have used `vector!` to mean "`vector-identity`" to keep it short) is a lot worse-looking than `(vector-concat x (car y) z)`.

GLS: One thing I've been concerned about is having to depend too heavily on a complex data-type processing facility to get good compiled code. While I am willing to do this work for the S-1, the Lisp Machine LISP or SPICE LISP may not need it or want it. If I'm wrong, then fine. Perhaps eventually a portable compiler front-end will be available to do the requisite type analysis, but I wouldn't promise that in under two years.

ALAN: [108.2] Why aren't all the `vx@` functions spelled out like "`vmismatch-of-char`", etc.? Why "`@`",

for crying out loud?

GLS: The idea behind this series of functions was to cut off the trend of `mismatch`, `vector-mismatch`, `string-mismatch`, `integers-mod-7-mismatch`, etc. Instead of extending the data-type dimension of the name space indefinitely, the "`@`" series terminates the dimension by allowing one to write the data type as an argument. The character "`@`" was intended to be reminiscent of the now-infamous and much-reviled "`#(@`*type* `...)`" syntax.

RMS: [170.8] Why is there a need for `vref@`? If you want to check the type at run time, why not just check the type *of the vector*? Why ask for it to be specified redundantly as an extra argument? Is this intended as a sort of declaration of the argument type for compiler optimization? That would be better done as something independent of the function that the argument is being passed to.

GLS: This was indeed intended primarily as a compiler declaration, and I certainly agree that it is terrible.

SEF: Rename `v`*x* to be `vector-`*x*, and `v`*x*`@` to be `typed-`*x*.

ALAN: [The following remarks are spread over all the pages of the chapter on vectors, presumably in reaction to the type-specific sequence functions.] Smlfqzzpt... !!@#?;$%@!!! Argh!!!! ... FOO...? ~~~~ ---- ----- ----- ----- ---MUMBLE!

ALAN: [108.3] I greatly disagree [that having a large number of names is more perspicuous than passing flags to a smaller number of functions]! I would much rather look up a single function in a manual and learn what all the various flags mean than concatenating parts of names this way. I think this naming scheme thing in *awful*!

ALAN: [113.2] There are *too many* of these [`remove` and friends]! (1) Why do we need to have `mumble`, `mumq`, and `mum`? Surely a compiler can easily grok `(mum #'eq ...)` jusr as easily as `(mumq ...)`. (2) How about `mumble-from-end` being the case of *count* < 0?

GLS: Well, instead of a *count*, the sequence functions have *start*/*end* pairs. One could consistently allow the case *start* > *end* to mean the reverse-direction form of the operation. However, the obvious interpretation, and the one that preserves the greatest number of algebraic identities without special-casing, is that

```
(op string ... start end)
   <=> (op (reverse (substring string end start)) ...)
```

but that isn't what we mean by `search-from-end`: we expect to find `"cd"` within `"abcdefg"`, not `"dc"`. Also, this convention might make it more difficult to open-code things. Finally, it is hard to say "the whole sequence, backwards" as simply as you can say "the whole sequence, forwards".

SEF: Reduce the number of sequence functions. For the `remove` series, for example, get rid of `rem` and `rem-if-not`; they can be expressed in terms of `rem-if`.

SEF: The `-if-not` functions would not be needed as much if we had `neq`, `neql`, `nequal`, `nequalp`, and `n=`(or something).

MOON: These families of functions which differ in the controlling predicate (e.g., `remove`, `remq`, `rem`) are totally out of hand! The predicate should be an argument (possibly optional), and when an implementation can improve efficiency by recognizing certain predicates and calling a special-case function or generating

open-code, it should be the business of the compiler to do so, not the programmer. Even the ones of these which exist now in the Lisp machine should be phased out. The name-abbreviation convention really loses, because it is not sufficiently consistent. If the predicate is optional the default should be `eql`, consistently everywhere. (Motivation: `eq` is substantially less useful, while `equal` or `equalp` would impose a hidden efficiency cost that doesn't seem justified by functionality, especially since it is debatable and application-dependent whether `equal` or `eql` is more "natural". Unfortunately a lot of the good names are taken now for functions that use `equal` as the predicate; this might be a reason to make the predicate default to be `equal` rather than `eql`.)

GLS: Well, if we wanted to go whole-hog, we could make even old friends like `member` use `eql` instead of `equal`; but that would probably be too confusing and too incompatible. From a language-design point of view I agree with the principle, however.

MOON: The `-if`, `-if-not` ones (which involve a one-argument predicate in place of a two-argument predicate) are a little harder to deal with, but something needs to be done about them, too. A little creative use of optional arguments would help a great deal. Or, lambda the ultimate conserver of system function names.

SEF: I'm sympathetic to this, but see no way to win other than the current groups-of-five proposal. Certainly it would be better to give all the good names to `eql`, with the ability to supply any other predicate as an optional argument, but this is a major incompatibility with current LISP that got off on the wrong foot with `equal` as the basic default. Do we dare take the plunge? As for `-if` and `-if-not`, writing all those lambdas is a pain. Any good ideas for a shorthand?

GLS: One problem with having an optional predicate is that many of the sequence functions which need an optional predicate also need optional indices, and it's completely unclear in which order to state them. Actually, you could get rid of lots of `&optional` arguments to all those sequence functions if one were willing to write separate calls to `subseq` or `substring` or whatever; it's just that a good compiler would have to analyze those nested calls and figure out what's going on, and even that wouldn't save you if you had to close-code the routines for which `substring` was providing the argument. The fact is that having all those routines take optional *start/end* arguments does improve performance as well as convenience.

ALAN: [140] Isn't there some way to get declarations to cover this explosion of non-generic functions? This is *silly*.

MOON: Many of these functions are ones which are very easy to write, and even easier if you have `loop`. Users are probably more likely to write them themselves, either as functions or as explicit iterations at the point of call, than to wade through mountains of documentation to find out that they already exist. If there were fewer of them users might be more likely to use them.

GLS: Actually, I get pretty tired of writing the same trivial loops over and over again. One of the few advantages of the 600 names is that they provide a regular nomenclature for the common kinds of loop (and, admittedly, for many uncommon kinds as well).

SEF: I have gradually come to believe that a function that does something obvious is a good thing to supply, even if it is easy to write yourself. This makes other people's code more perspicuous.

ALAN: [139] BLEAGH
[140] BARF
[141] FOO
[144] YUCK
[145] Mumble..
[156] FOO
[157] ...
[158] **sigh**

GLS: Now here's an idea, if you don't mind moving the whole language in the direction of functional programming style. Let us use `position` as an example. Define also a functional operator `fposition`; this takes various arguments and returns a function. Specifically:

```
(defun fposition (pred &optional (item () item-p))
  #'(lambda (sequence &optional (start 0) (end (length sequence)))
      (do ((j start (+ j 1)))
          ((= j end) ())
        (when (if item-p
                  (funcall pred item (elt sequence j))
                  (funcall pred (elt sequence j)))
          (return j)))))
```

This would allow one to write:

```
((fposition #'equal x) s 0 7)  for  (position x s 0 7)
((fposition #'eq x) s 0 7)  for  (posq x s 0 7)
((fposition #'eql 4.5) s)  for  (pos #'eql 4.5 s)
((fposition #'numberp) s)  for  (pos-if #'numberp s)
((fposition (fnot #'numberp)) s)  for  (pos-if-not #'numberp s)
   where
   (defun fnot (fn) #'(lambda (&rest x) (not (apply fn x))))
```

This is, of course, an institutionalized currying. The point is that for each level of currying you get another place to have optional arguments. Also, one might usefully do things like `(map (fposition #'atom) s)`. If a compiler is capable of open-coding and simplifying these functional operators, or of recognizing useful and common patterns, then this might be acceptable. (An implementation might provide the six hundred functions after all, and the compiler could rewrite calls to `fposition` exactly as shown above.)

GLS: Perhaps slightly more general than the above would be:

```
(defun fposition (pred &rest args)
  #'(lambda (sequence &optional (start 0) (end (length sequence)))
      (do ((j start (+ j 1)))
          ((= j end) ())
        (when (funcall* pred (elt sequence j) args)
          (return j)))))
```

The difference is that `pred` can take any number of arguments, of which the first comes from the sequence. This upsets the order of arguments: `((fposition #'> 3) x)` then finds an element of `x` which is greater than 3, rather than an element which 3 is greater than. But suppose that one then defines `(position x s start end)` to be `((fposition #'eql x) s start end)`. Then

```
((fposition #'position set) sequence start end)
```

means roughly the same as

```
(string-search-set set string start end)
```

GLS: Just because the above functional-style hack *can* be done, doesn't mean it *should* be done, or that it isn't difficult. I just want to put forth the possibility as a way of cutting down on the number of names. I'm not at all rabid about it.

** **Alternatives:**

A. Status quo: have six hundred (probably eventually a thousand) sequence functions, with names organized by concatenation along four orthogonal and therefore multiplying dimensions: data type, operation, direction, and predicate.

For example:

```
(remove-from-end 'a '(a b a c a d) 1) => (a b a c d)
(list-cnt-if #'numberp '(1 a b 4 5) 1 4) => 2
(string-search-from-end "na" "banana") => 4
(string-pos #'(lambda (c) (posq c "AEIUOaeiuo")) "host") => 1
```

[578 functions total.]

B. Make all sequence functions generic. Remove the data-type dimension. Name sequence functions within the Cartesian product of the operation, direction, and predicate dimensions.

For example:

```
(remove-from-end 'a '(a b a c a d) 1) => (a b a c d)
(cnt-if #'numberp '(1 a b 4 5) 1 4) => 2
(search-from-end "na" "banana") => 4
(pos #'(lambda (c) (posq c "AEIUOaeiuo")) "host") => 1
```

[93 functions total.]

C. Make all sequence functions generic. Remove both the data-type and predicate dimensions. Name sequence functions within the Cartesian product of the operation and direction dimensions. Let operations somehow take an optional predicate argument. Use lambda-expressions to build appropriate predicates as needed.

For example:

```
(remove-from-end #'(lambda (x) (eq x 'a)) '(a b a c a d) 1)
   => (a b a c d)
(count #'numberp '(1 a b 4 5) 1 4) => 2
(search-from-end #'char-equal "na" "banana") => 4
(pos #'(lambda (c) (posq c "AEIUOaeiuo")) "host") => 1
```

[37 functions total.]

D. Make all sequence functions generic. Remove both the data-type and direction dimensions. Name sequence functions within the Cartesian product of the operation and predicate dimensions. Let the case *start* < *end* mean the reverse direction (`-from-end`).

For example:

```
(remove 'a '(a b a c a d) 1 6 0) => (a b a c d)
(let ((z '(a b a c a d)))
  (remove 'a z 1 (length z) 0)) => (a b a c d)
(cnt-if #'numberp '(1 a b 4 5) 1 4) => 2
(search "na" "banana" 6 0) => 4  ;(did it match "na" or "an"??)
(pos #'(lambda (c) (posq c "AEIUOaeiuo")) "host") => 1
```

[53 functions total.]

E. Make all sequence functions generic. Remove all dimensions but the operator name. Let operations somehow take an optional predicate argument. Use lambda-expressions to build appropriate predicates as needed. Let the case *start* < *end* mean the reverse direction (`-from-end`).

For example:

```
(remove #'(lambda (x) (eq x 'a)) '(a b a c a d) 1 6 0)
   => (a b a c d)
(count #'numberp '(1 a b 4 5) 1 4) => 2
(search #'char-equal "na" "banana" 6 0) => 4
   ;(did it match "na" or "an"??)
(pos #'(lambda (c) (posq c "AEIUOaeiuo")) "host") => 1
```

[29 functions total.]

F. Make all sequence functions generic. Remove all dimensions but the operator name and direction. Let the plain operator name mean that the predicate is `eql`. Let there be a functional version, with a name beginning with "`f`", to build the other versions of the operation; also use lambda-expressions to build appropriate predicates as needed.

For example:

```
(remove-from-end 'a '(a b a c a d) 1) => (a b a c d)
((fcount #'numberp) '(1 a b 4 5) 1 4) => 2
((fsearch-from-end #'char-equal) "na" "banana") => 4
((fposition #'posq "AEIUOaeiuo") "host") => 1
```

[74 functions total.]

G. Make all sequence functions generic. Remove all dimensions but the operator name. Let the plain operator name mean that the predicate is `eql`. Let there be a functional version, with a name beginning with "`f`", to build the other versions of the operation; also use lambda-expressions to build appropriate predicates as needed. Let the case *start* < *end* mean the reverse direction (`-from-end`).

For example:

```
(remove 'a '(a b a c a d) 1 6 0) => (a b a c d)
((fcount #'numberp) '(1 a b 4 5) 1 4) => 2
((fsearch #'char-equal) "na" "banana" 6 0) => 4
((fposition #'posq "AEIUOaeiuo") "host") => 1
```

[58 functions total.]

H. Make all sequence functions generic. Remove all dimensions but the operator name and direction. Really go whole-hog and have only a functional version; use lambda-expressions to build appropriate predicates as needed.

For example:

```
((fremove-from-end #'eq 'a) '(a b a c a d) 1) => (a b a c d)
((fcount #'numberp) '(1 a b 4 5) 1 4) => 2
((fsearch-from-end #'char-equal) "na" "banana") => 4
((fposition #'(lambda (c)
                ((fposition #'char= c) "AEIUOaeiuo")))
 "host") => 1
```

[37 functions total.]

I. Make all sequence functions generic. Remove all dimensions but the operator name. Really go whole-hog and have only a functional version; use lambda-expressions to build appropriate predicates as needed. Let the case *start* < *end* mean the reverse direction (`-from-end`).

For example:

```
((fremove #'eq 'a) '(a b a c a d) 1 6 0) => (a b a c d)
((fcount #'numberp) '(1 a b 4 5) 1 4) => 2
((fsearch #'char-equal) "na" "banana" 6 0) => 4.
((fposition #'(lambda (c)
                ((fposition #'char= c) "AEIUOaeiuo")))
 "host") => 1
```

[29 functions total.]

**** Responses**

MOON: X	RMS: FG!	SEF: X!!!!	GINDER: X!
DLW: X!!	HIC: X	WLS: G!	CHIRON: F!!
ALAN: E!	GLS: F!!	RPG: I!!!	

HIC: This is a tuffy. It's gonna take quite a bit more discussion.

WLS: G (or E or D, at worst). I'm sure that it can be made to compile efficiently.

SEF: [Several paragraphs by SEF follow.] I have finally hit on a proposal for all the sequence functions that I like, or at least that I can tolerate without becoming nauseous. Basically, the idea is to use separate functions for each of the operations, but to indicate non-standard options with keywords. I would propose that the default in every case (meaning what is hung on the good name) is generic, forward, and with an `eql` predicate. (That last is a departure from tradition, but maybe the time has come.)

`member`, for instance, takes two required arguments plus keywords. With no keywords, it uses an `eql` test.

Keywords defined in this series are as follows:

`:from-end`	Self-explanatory, but only allowed for those operations where it makes sense.
`:test` *pred*	Takes a two-place test function to be used instead of `eql`. Thus, the old `member` uses `:test #'equal`.
`:test-not` *pred*	Same, but inverts the test predicate.
`:if` *pred*	Same, but a one-place predicate.
`:if-not` *pred*	One-place predicate, inverted.
`:input-type` *type*	Optional declaration, for efficiency. Takes any type-specifier that is a sequence.
`:output-type` *type*	Tells `map`, `concat`, etc. what to produce (default is `:list`).
`:start`, `:end`	Might or might not want to do these with keywords, but I like the idea.

It looks to me as if this gets us down to about 20 to 30 functions. The compiler can still wring out as much

efficiency as it knows how, and can just close-code the cases where it cannot be clever. The calls will be fairly nice-looking and easy to read.

If this is adopted, there is still some thinking to be done about compatibility (do we really want to change `member/memq`?), but I think our worst problems will be over.

MOON: It isn't obvious which of `:test` and `:if` is which. Call `:test` "`:compare`" instead?

RPG: About Scott's sequence reduction proposal: as I understand it you would write

```
(member x l :test #'rpgequal :input 'vector)
```

? It certainly seems nice to have fewer of these functions, but this doesn't seem too "lispy", as RMS would say. I would consider it, though.

GLS: Well, maybe it would be

```
(member x l ':test #'rpgequal ':input 'vector)
```

, but you have the general idea. Some people have asked whether it should indeed be

```
(member ':test #'rpgequal ':item x ':input 'vector ':sequence l)
```

but that seems a bit extreme to me. (Actually, keywords might be more flavorful if they were self-evaluating *and* identified by a *trailing* colon. Thus:

```
(member test: #'rpgequal item: x input: 'vector sequence: l)
```

Oh, well.)

RMS: I think that using keywords for sequence functions may be okay, but some of them want an indefinite number of data arguments already, and I don't see a way of allowing them to do both. Consider `mapcar`. For some sequence functions, there is no meaning for an indefinite number of arguments, but those that can do so ought to be able to.

SEF: I am beginning to believe that self-evaluation would be a very good thing for keywords; maybe I was too quick to dismiss DILL's scheme to set them all to themselves, but we do need some mechanism to prevent them from being set to something else. Not so sure about trailing colons: sort of cute, but maybe hard to parse, and they look wrong for singleton keywords.

I would definitely oppose any attempt to go with `:item` and `:sequence` keywords. We don't want to go overboard with this. The nice thing about my original proposal is that the thing you do most often looks really clean and making a few mods only adds a little bit of hair.

As for being "lispy", it seems to me that RMS uses this term to mean that position and levels of nesting of LISP structure are better ways of indicating important semantic differences than the use of keywords or the like. I agree only up to the point where one has to start counting parentheses and positions to understand code; human-readability is important and I have only rather limited ability to grok the difference between `((a (b c)) foo)` and `((a (b c) foo))`. So at some point I tend to give up and go for non-lispy keywords.

GLS: Well, it seems to me that Lispiness is in the eye of the beholder. Arrays, defstructs, and even macros don't seem very Lispy to someone raised on LISP 1.5. Now, SMALLTALK implements its keywords by rearranging them into some canonical order (say alphabetical), concatenating them onto the function name, and calling that. (Or maybe this was PLASMA.) So

```
(member x l :test #'rpgequal :input 'vector)
```

is roughly the same as

```
(member%input%test x l 'vector #'rpgequal)
```

So there really are six hundred functions, but they don't really have to be documented that way. (This is of course not the only way to implement keywords.)

SEF: [Several paragraphs by SEF follow.] One possible solution to RMS's objection is to enumerate a finite set of keywords that each function recognizes. The `&rest` argument is scanned until one of these is seen. Up to that point, everything is assumed to be an argument; once the first legal keyword is seen, the rest of the `&rest` argument is keywords and their associated values. This means, of course, that the keyword symbols cannot be used as normal arguments to these functions. This does not bother me, but I can see that some would find it unaesthetic to gobble up these symbols.

A second possible solution would be to set things up so that all keywords end up in a special package and they all get set (perhaps permanently) to themselves. This has the advantage that you would type `(foo :key1 bar :key2 bletch)` rather than `(foo ':key1 bar ':key2 bletch)`. Now, if keyword-taking functions are set up as macros that translate the calling form into some arbitrarily ugly internal form, it would be easy to flag whether the user had typed `:foo`, meaning use `:foo` as a keyword, or `':foo`, meaning use the symbol `:foo` as a normal argument. The latter would be used very rarely, except in implementing new sequence functions from old ones, but this gives us an escape valve.

Finally, we might set up some special syntax meaning either "this thing is a keyword" or, perhaps, "this thing looks like a keyword but use it as a normal argument". Then

```
(foo arg1 arg2 ... argn key1 value key2 key3 ...)
```

can be turned into some ugly internal form like

```
(foo list-of-keywords-and-values arg1 arg2 ... argN)
```

in an unambiguous way. Come to think of it, we could make the user stuff the keywords into a list himself, but that looks uglier to me than spreading it all out on one level.

I would expect that the external forms that the users see would turn into something very much like what GLS suggests: the commonly-used combinations would be spelled out and the less-used ones would pass in, say, the test as a parameter. The internal forms can be arbitrarily ugly, but the external forms should look as clean as we can make them.

RPG: My preference, all things being equal, is to opt for Lispy stuff. I approve of SEF's suggestion and would support it.

SEF: [Several paragraphs by SEF follow.] I have thought a little bit more about RMS's objection that keywords and the use of an indefinite number of regular arguments won't mix. I observe that in all of the cases now defined in the COMMON LISP manual, the functions that take multiple arguments take multiple *sequence* arguments. Since a keyword is not a sequence and hence not a legal normal input, we win: if the argument is a symbol, it must be a keyword, else error. I guess one could imagine sequence functions that would want both multiple arguments where a symbol is legal and keywords, but right now none are anticipated and the advantages of the new scheme might be worth giving up this option for the future.

One remaining problem is those operations that would want to coerce symbols to strings, and that therefore would want to allow multiple symbol arguments plus keywords. I have never been fond of this coercion anyway, but I know that the Lisp Machine LISP folks like it. I guess this coercion would be allowed only if the `:input-string` keyword is present. In this case, the rule would have to be "Look at each argument. If it is a legal keyword, treat it as such, else treat it as an input symbol." If you want to sneak `:from-end` in as a real input and not a keyword, you can easily enough do a `get-pname` to make it a string and not depend on the coercion. My preference would still be to flush this coercion, but it seems to me that we can handle it if necessary.

So the real issue seems to be whether we are willing to forego the future use of sequence function with both non-sequence `&rest` arguments and keywords.

RMS: [Several paragraphs by RMS follow.] Note that calls with keywords in Lisp Machine LISP are specifically *not* translated into any other ugly form. In fact, the function that takes `&key` arguments can also take a `&rest` argument which gets the full list of keywords and values. I have written very many functions which use both. So keywords must actually be exactly what they look like to the user.

Do you know about `&key`?

```
(defun foo (a b &rest r &key x &optional y (z t zflag) ((:u uvar)))
  ...)
```

takes positional `a` and `b`, followed by keyword arguments, of which `x` is mandatory and `y`, `z`, and `u` are optional. In addition, `r` gets a list of the keywords and values. `&allow-other-keys` can also appear in the list, which says that an unrecognized keyword and its associated value should both be ignored rather than an error.

This appears to overcome all the deficiencies with past keyword argument schemes, because the rest argument can be passed on to another function with `lexpr-funcall`.

Your idea of having keywords be self-evaluating is a win. Perhaps any symbol interned in the keyword package should be set to itself immediately. Or, perhaps `:foo` should be a notation for the pname of `foo`, using GLS's suggestion about keyword comparison using interned pnames.

SEF: I have heard many rumors of `&key` and seen a few partial descriptions like the one above, but I have never seen the full documentation of this keyword scheme or an explanation of how it works internally. Is such a description available anywhere? This is something that we will really have to look at carefully for COMMON LISP, since keywords are playing a larger role, but we really can't saddle all of the COMMON LISP implementations with anything too complex at function call time; not all have the microcode to hack such things efficiently.

GLS: `&key` is not documented in [Weinreb 81b] or [Weinreb 81a]. I have determined that the documentation has been edited into the document source file "`LMMAN;FD.EVA >`" `@ MIT-AI`, but apparently this has not yet been publicly distributed.

RMS: [Several paragraphs by RMS follow.] It is best not to be extremist about the direction. For some functions, it is better to allow *end*Kstart. For some, it is better to provide another name for the reverse version. I suspect that the number of functions for which reverse direction is really important is small enough that it is

okay to handle each one on its own merits.

If *end*<*start* is used, *start*=`t` can mean "the whole thing, backwards".

I don't believe that optional arguments can be used in a decent way to take care of various predicates. The functional scheme is the right idea. Internally, the system can contain the functions now called `rem` and `rem-if`, and expand functional forms into calls to those, when appropriate. This would avoid consing a closure. But the user would not notice this.

This means that the names `mem`, `rem`, etc., should go away. However, while normally the "simple" or "basic" name should be the `eql` version, traditional names such as `memq` and `member` probably should not change. They should be regarded as names for compatibility only, and some other name should be chosen as the "basic" function which uses `eql`, even if `memq` is also changed to use `eql` and therefore will be the same function. Unless you prefer to think of "`memq`" as a word in its own right, just as "xerox" is.

A little-known Lisp Machine LISP feature is that `#'(curry-before < 3)` is a function which returns `t` for an argument greater than 3. (`curry-after` also exists.) However, it would be impossible to make

```
(rem-if #'(curry-before eq x) y)
```

work efficiently without having a function equivalent to `rem` to use at run time.

GLS: Regarding *start*>*end*, I was wedged about how hard it was to say "the whole thing, backwards". Obviously you just let `()` mean "the length of the string" for *start* just as for *end*. Then *start*=`0`, *end*=`()` means the whole thing forwards and *start*=`()`, *end*=`0` means the whole thing backwards. (This is somewhat reminiscent of the use of `()` to mean "infinity" in range types such as (`integer 0 ()`).) Note too that one doesn't need `reverse` any more! One can say simply (`subseq x () 0`). Yech.

ALAN: [Several paragraphs by ALAN follow.] Some general remarks:

(1) We have put a lot of issues off by saying "well, given declarations and other type information (or given some trivial optimizers), the compiler can recover the efficiency we lost when we added this generality". I long for the days when a MACLISP programmer could "feel the bits between his toes". Perhaps those days are gone forever, or perhaps it's just that the bits have a different feel here in the future and I haven't gotten used to them yet. Still, I wonder at just how simple a COMMON LISP compiler can be.

(2) The number of inconsistencies that COMMON LISP compatibility will introduce into Lisp Machine LISP bothers me a bit. I am worried about antagonizing too many of our Lisp Machine LISP users. I really wish that COMMON LISP tried to be more of a common subset of LISP that we all agreed to implement, and less of an integrated language with a philosophy of its own. We keep considering things like changing the definition of `member`. Well, that is not going to make too many Lisp Machine LISP users happy. In fact, I very much doubt we could get away with that and live. I sympathize with those people who were afraid that COMMON LISP was being designed behind their backs (you remember the flap through the mails a few months back). If a lot of the proposed changes are actually adopted into COMMON LISP, then it seems to me that it might well be in some people's best interests that they resist COMMON LISP on the Lisp Machine. That would definitely put a monkey wrench in COMMON LISP's goal of compatibility.

(3) To narrow down to a specific instance of the above complaint (and one that I have a more personal interest in): we are proposing several incompatible changes to what `defstruct` does. Well, the more I think about

this the more unhappy/annoyed I get. You see, I feel like I *just finished* straightening all this mess out and now you propose that we start mucking around with it again. Now since the very first draft that I saw of this thing, we have moved a lot closer to what I would call a reasonable set of changes to `defstruct`, so you needn't worry that I will give up and go away. Still, I would be a lot happier if I didn't have to defend *any* incompatible changes to my users. (There is also the added annoyance that I am currently trying to help the NIL people bring my `defstruct` up, and they have their desires about what various features should do/mean in NIL...)

(4) Doesn't the general drift to generic functions run the risk of damaging the error checking in the language? The situation isn't all that bad for numeric types where the exact flavor of number you are counting with usually doesn't really matter. But with all our functions generalized to handle *anything* that might reasonably be considered a sequence (for example), handing in something of an unexpected datatype might be more likely to cause bizarre behavior than an error. (ZWEI's most common error message seems to be something to the effect that some function got a `nil` instead of an array. Well, if it was only going to do some sequence-like operations on that array...) If LISP had typed variables we wouldn't run this risk, but we don't.

(5) I am still opposed to the entire sequence function thing. I have answered many of the sequence function issues from the point of view of: well, if we must have them, then this is least objectionable choice. But I think we shouldn't have them. I am sure that there will be more discussion of this issue later, so I won't waste my breath here.

GINDER: I like SEF's proposal.

MOON: I can't vote for any of the alternatives right now, although I would lean towards B. To unbundle the issues:

- The data types go.
- The direction stays (*endKstart* doesn't make it).
- The name-abbreviation goes.
- Leave the functional programming to Backus for now.
- I'm not sure yet what I think is the right mix of options for the predicate (i.e., the various `-if`, `-if-not` things) versus writing a function. I probably want the predicate to be a required argument in many of these functions.

DLW: Several points:

- I wouldn't mind flushing `string-search` of ZETALISP [Lisp Machine LISP] in favor of telling users to write `(search (string ...) ...)`. The latter is more tasteful in the presence of generic treatment of sequences.
- I think the type prefixes have *got* to go. A generalization of `fixnum-identity` is better, even if we can't find a non-ugly one. Let the user insert something like this when he needs it, namely in a few inner loops and probably nowhere else. The `typed-length` idea of SEF is okay too, and may be easier on compilers.
- Changing `member` to use `equal` is too incompatible. It's too bad, but I think we're stuck.
- I agree with GLS about not wanting to write the same trivial loop over and over. MOON is wrong.

However, the manual *must* have a quick reference that lets you find the one you want quickly. Please put in a single page that contains (brief) descriptions of each family: what is `reduce`, what is `position`, what is `scan`, etc.

- In the `((fposition ...) ...)` examples, where did the `funcall` go? What is lurking in the myterious Chapter 18? ("The shadow knows!" (Also the `shadq`, `shad`, `shad-if`, `shad-if-not`, ...))
- The functional-style hack is elegant but very hard to read.
- Of the alternatives, I like B the most. I'd accept a SEF-modified version of B in which `typed-`*xxx* existed for all *xxx*, or a generalization of `fixnum-identity`, or even (easy for *me* to say) forgetting about type declarations altogether. However, I could be swayed pretty easily.

CLH: [GLS: refer to DLW's fifth point above.] You say that any list that is not a special form or a macro is taken to be a function call. It might be nice to specify the exact semantics here. E.g., is it legal to do

```
((lambda (x) (add1 x)) 1)
((get foo 'interp-ftn) x)
(current-ftn x) ;where current-ftn is a variable that evaluates
                ;to the function, or even to an expression which
                ;will evaluate to the function, etc.
```

What happens if you `apply` an `fexpr`, a `macro`, etc? I now believe that the right thing to do is to say that `apply` is the expert in interpreting `lambda`'s (and `subr`'s), but that it is `eval`'s problem to do funny binding. Thus `apply` would bind the arguments directly to the formals in the lambda, whatever kind of function is involved. The issue is as follows: Suppose `foo` is an `fexpr`. In R/UCI LISP the equivalent of `(foo a b c)` is `(apply 'foo '(a b c))`. In my opinion, an `fexpr` takes only one argument, namely a list of its arguments, and the right thing to do is `(apply 'foo '((a b c)))`. Similarly, I think doing `apply` on a `macro` should return the macro expansion, i.e., it should bind the (single) argument supplied to apply to the (single) formal in the `macro`, and evaluate the `lambda`. Our users want to be able to do `apply` without knowing whether the thing being applied is an `fexpr`, `expr`, or `macro`, and to have `(apply 'foo` *args*`)` be equivalent to `(foo` *args*`)`. Even though this is an ill defined task (because of the problem of whether the arguments should be evaluated or not), it is what R/UCI LISP tries to do. I am convinced that this is a mistake, and that any attempt to "make `apply` do the right thing with `fexpr`'s" is doomed to failure.

DM: I think having everybody be generic is very appealing. It would (well, I guess that should be "is", not "would") be nice to have some way of declaring to the compiler what type you expect so you don't pay any extra for the generic when you neither need it nor can afford it. I'm a little worried that all the overhead of generic functions for *everything* will add up and degrade performance on conventional machine substantially. I don't have any real feel for this, just anxiety.

DILL: Generic sequences should not go into the language until there is a general agreement that they are good and a general desire to use them. We are following this principle with other constructs in the language. We should get on with the task of getting the rest of the language (that people will actually use) designed and implemented.

ALAN: I just realized something flavorful about SEF's sequence function proposal. If the keywords `:start`, `:end`, and `:count` are all supported then you can get everyone's favorite method for specifying subsequence:

```
(member x list ':start 2 ':end 8)
(member x list ':start 2 ':count 6)
(member x list ':end 8 ':count 6)       ;(!)
```

We don't need to hurt anybody's feelings this way!

GLS: At last! An application for my Ph.D. thesis work! Seriously, the effects of conflicting specifications would have to be defined.

***** Issue 122: Use *index/count* pair instead of *start/end*?**

GJC: [109.1] [The following several paragraphs are all by GJC.] You mention on page [109] of the draft COMMON LISP manual that the overriding consideration for the choice of [*object, start, end*] over [*object, index, count*] was an informal poll. Is this really true? [GLS: yes.] I've had a good look at the Lisp Machine LISP string code, to tell what it really was that made people say that it was more convenient to have [*object, start, end*], and it looks to me that it is based on a non-recursive concept of how to get a tiny bit more efficiency without generating the required mapping idioms to do the job. I claim it's not the use of the functions that Lisp Machine LISP people were talking about; it was the implementation of them in Lisp Machine LISP.

In LISP, arrays have always been indexed [0, $N-1$], so people should be used to saying

```
(do ((j 0 (1+ j)))
    ((= j n))
  ... (elt object j) ...)
```

Take any code like this and it's a trivial transformation to

```
(do ((j 0 (1+ j)))
    ((= j n))
  ... (elt object (+ index j)) ...)
```

It can be done quite automatically; I have done it many times in implementing and using string operations in VAX NIL.

However, it is less trivial to transform to

```
(do ((j start (1+ j)))
    ((= j end))
  ... (elt object j) ...)
```

because any arithmetic one does on `j` will then be incorrect. You lose the very nice characteristics of the 0, 1, 2, 3, 4, 5, 6, ..., $N-1$ sequence. [GLS: such as?]

The result of the transformation

```
(do ((j 0 (1+ j)))
    ((= j n))
  ... (elt object-1 j) ... (elt object-2 j) ...)
```

to

```
(do ((j 0 (1+ j)))
    ((= j n))
  ... (elt object-1 (+ index-1 j)) ... (elt object-2 (+ index-2 j)) )
```

is considerably simpler than the multiple-stepping `do` one would need using the [*start*, *end*] theory.

What do you gain by this increased complexity? A bit of efficiency, for you save doing the addition `(+ index-1 j)`. (It's obvious what an optimizing compiler can do for you here.) [GLS: Actually, it's not obvious: such optimizations are often defeated by the necessities of pointer conventions necessary to the integrity of the storage system and garbage collector.]

Okay, so much for implementation; what about use? What can I say; compare the Lisp Machine LISP graphics code with the description of the Smalltalk graphics code in the August [1981] *Byte* magazine. The BITBLIT used in Smalltalk takes [*index*, *dimension*] arguments, not [*start*, *end*] as the lispmachine BITBLIT. You can see how the Smalltalk people developed a good theory of mapping in order to make the implementations easier and more efficient at the same time. All Lisp Machine LISP ends up with is the efficiency.

(End of remarks by GJC.)

GLS: I'm truly sorry, but after reading it three times, I still have no idea what you are talking about, so I can't make a sensible reply.

** **Suggestion**: Change the subsequence indication convention from a *start/end* pair to an *index/count* pair?

N

** Responses

MOON: N	RMS: N!	SEF: N!	GINDER: N!
DLW: N!!!	HIC: Y!!		CHIRON: N!!!
ALAN: N!	GLS: N!!	RPG: N!!	DILL: N

RMS: If someone can really show that *index/count* is more convenient, then I would say go along with it, but I don't see anything in what GJC wrote to support this claim.

WLS: If yes, then a negative count should indicate reverse direction.

*** **Issue 123: Reorder arguments to `replace`?**

ALAN: [110.0] Perhaps `replace` would be more useful with the arguments *source-start* and *target-end* in the other order?

** **Suggestion**: Permute the arguments to `replace` (and, for consistency, all other functions which take similar arguments).

** Responses

MOON: Y		SEF: N!	GINDER: Y!!!
DLW: N			CHIRON: N!
	GLS: N	RPG: N	

DLW: Allow ( ) for *target-end* and for *source-end* to mean the same as the length, so that you can pass one and not the other no matter how the arguments are ordered.

***** Issue 124: An *end* argument of ( ) is the same as unsupplied?**

MOON: In `replace` and all others that take a double *start/end* argument pair, it is necessary that ( ) as an *end* be the same as omitting it.

GLS: This is of course so that you can specify both *start* arguments and let one or both *end* arguments default to the length of the sequence. This is the case even if the argument permutation suggested above is adopted. For consistency, perhaps this should work even if only a single *start/end* pair is needed.

** Suggestion: For all functions which take a pair of indices *start/end*, an explicit argument of ( ) is equivalent to not supplying the argument.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!!	HIC: Y	WLS: Y!	CHIRON: N!	
ALAN: Y!	GLS: Y!!	RPG: Y		

***** Issue 125: Define what `replace` does for overlapping regions?**

ALAN: [110.4] [On `replace` having an undefined effect if the source and target regions overlap within the same sequence:] Since we are going to all this trouble, why not define it to do the useful thing? (BLT in the downward (non-replicating) case.)

GLS: It would be nice for compilers to be able to open-code native machine string-move instructions for the `replace` operation. Unfortunately there is no agreement among machines as to how the overlap case is handled. This is why I suggest that it be undefined in COMMON LISP.

** Suggestion: Define `replace` of overlapping regions to be meaningful.

N

** Responses

MOON: N		SEF: N!!	GINDER: N!	DM: N!
DLW: N!!	HIC: Y		CHIRON: Y	
ALAN: N!	GLS: N!!	RPG: N!		

***** Issue 126: Arrange for result of `nreverse` to be `eq` to its argument?**

ALAN: [110.7] [On the result of `nreverse` not being `eq` to the argument:] We could define `nreverse` to always return the same *cons* in the `list-nreverse` case.

GLS: While this can be done for `nreverse`, it cannot be done for `delete` and other destructive operators. I think it might produce only confusion to tell users that destructive operators produce unique results in some cases but not others.

**** Suggestion:** Define the result of `neverse` to be `eq` to its argument. N

**** Responses**

MOON: N		SEF: N!!	GINDER: N!	DM: N!
DLW: N!!	HIC: N		CHIRON: N!	
ALAN: N!	GLS: N!	RPG: N!!		

DM: All the n*xxx* functions should be preceded by a warning to use them solely for value, never for effect. The implementor should have the option of implementing them however he sees fit.

***** Issue 127: Rename `concat` to be `concatenate`?**

MOON: [110.8] Rename `concat` to be `concatenate`. Avoid useless abbreviation; it only makes names harder to remember. Abbreviations and neologisms make the language harder to remember, and should only be used for the most frequently-used words, which every programmer will remember (e.g., abbreviating `sequence` to `seq` is okay as long as it is done consistently everywhere). This is not used so frequently that it needs to be shorter.

DLW: I agree. Abbreviations are insidious.

SEF: Well, the thought here was that `concat` would be heavily used and that `concatenate` is a bit on the long side. Kind of like "Massachusetts Avenue".

GLS: Well, do we want to use up English words or not? PL/I calls this operation `CONCAT`. UNIX calls concatenation of files `cat` (although it is more often used with one argument as a synonym for "`type`" or "`print`" than to perform actual concatenation!). In MULTICS MACLISP string concatenation was called `catenate`. In LISP 1.5, `conc` was the *n*-argument version of `nconc`, which took only two arguments. It seems we have a spectrum to choose from: `cat`, `conc`, `concat`, `catenate`, `concatenate`.

**** Alternatives:**

A. Rename `concat` to be `cat`.

B. Rename `concat` to be `conc`.

C. Status quo: call it `concat`.

D. Rename `concat` to be `catenate`.

E. Rename `concat` to be `concatenate`.

F. Emphasize that `concat` always copies its arguments: call it `copycat`.

** Responses

MOON: D	RMS: F!	SEF: C!	GINDER: E!	DM: E!
DLW: C	HIC: E	WLS: C!	CHIRON: F	
ALAN: C!	GLS: C!	RPG: C		

MOON: According to the nearest dictionary, "catenate" and "concatenate" are synonymous.

*** Issue 128: May `concat` take arguments of mixed type?

ALAN: [110.9] [The type of the result of `concat` may depend on the implementation.] Thus making it to some extent worthless in the mixed-modes case, I suspect!

SEF: NO MIXED TYPES TO `concat`!!!

MOON: [111.0] I am dubious about `(concat) => ()`; is `()` better than `#()` or `""` or `#""` or ...?

GLS: Well, somehow it seemed the most obvious choice.

** Suggestion: Forbid arguments of mixed type to `concat`.

Y?

** Responses

MOON: X		SEF: Y!!!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!!!	CHIRON: Y	
ALAN: Y!	GLS: N	RPG: Y!		

MOON: This issue is much more difficult than these comments would indicate. I can't vote on this now. Forbidding mixed types when the types are only slightly (or "technically") different, e.g., `(vector t)` versus `(vector fixnum)` would be a major pain to users, I suspect.

*** Issue 129: Who likes `reduce`?

SEF: I don't like `reduce` and friends.

GLS: I do. I think they will be useful.

** Suggestion: Flush the `reduce` family.

** Responses

MOON: N	RMS: Y!	SEF: Y!	GINDER: N!
DLW: Y!!		WLS: N!!!	
ALAN: N!	GLS: N!!	RPG: N!!	

RMS: Let people experiment with such things. Add them later if people have found them useful in practice. Perhaps some of the other new sequence functions which have no current equivalent even for lists should get the same treatment.

MOON: [111.3] Clarify the use of *start-value* and what happens if it is omitted.

MOON: [111.6] [`reduce` allows elements to be associated in any manner convenient to the implementation.] That's totally random.

GLS: It's intended to give the implementation some freedom.

***** Issue 130: Should `map` and friends be allowed to take zero sequences?**

SEF: [112.1] Require `map` to take at least one sequence. Don't use it to get infinite loops.

ALAN: [113.0] [`some` and friends perform an iteration if no sequences are provided.] I would rather they barfed. `do` is perfectly good at this sort of thing.

GLS: I always try to define the boundary cases of things if they make mathematical sense. If there are pragmatic pitfalls, however, I can be dissuaded.

**** Suggestion:** Disallow the case of `map`, etc., of zero sequences.

Y?

** Responses

MOON: Y		SEF: Y!!!		DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: N	RPG: N!		

***** Issue 131: Nice way to say do-forever**

GLS: Actually, I sometimes wish there were a nicer way to say "do forever" than to spell "forever" as "`()(())`". I find myself writing comments like:

```
(defun driver-loop ()
  (do () (())           :() (()) is cuneiform for "forever"
    ...))
```

** Suggestion: Add a special form called `do-forever`, which simply executes its body repeatedly. One can get out with a `return` or `throw`.

Y?

** Responses

MOON: N	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: N!	GLS: Y!	RPG: Y!!	DILL: Y	

WLS: Perhaps call it `repeat`.

DM: Actually, if you have `loop` around, you have a relatively clear way of saying it already. Leaving aside that this simple example can be better done other ways, I rather prefer

```
(loop do (if (this-is-stupid-p) (return)))
```

to

```
(do-forever (if (this-is-stupid-p) (return)))
```

I guess it's just that "`do-forever`" is really lying, since in practice there will always be something which terminates the loop. And the rather annoying `do` in `(loop do ...)` could probably be eliminated by taking the syntax out of `loop`, as suggested above.

*** Issue 132: Sequences of mixed type to map, and result type?

SEF: [112.1] May the sequences to `map` be of mixed type? There may be problems. Keep it simple.

GLS: I don't see the problem with allowing sequences of various types as arguments to `map`. Also, it may be stylistically useful.

```
(map #'(lambda (c b)
         (puthash c (if (zerop b) 'consonant 'vowel) letters))
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
     #"10001000100000100000100000")
```

MOON: [112.2] In `map`, it should be possible to specify the result sequence type; it should not be tied to the argument sequence types. For example, one should be able to map down a list and produce a bit-vector.

GLS: One simple way is simply to define `map` always to produce, say, a list, and to write `(to-vector (map ...))` if you want a vector out. This wastes time in the interpreter, but it's simple, and the compiler can interpret this cleverly.

** Alternatives:

A. Require all sequence arguments to `map` to be of the same type, and require the result to be a sequence of the same type.

B. Let the sequence arguments to `map` be of different types. Require the result to be a sequence of the same type as, say, the first sequence argument.

C. Require all sequence arguments to `map` to be of the same type, and let `map` take an argument

specifying the result type.

D. Let the sequence arguments to `map` be of different types, and let `map` take an argument specifying the result type.

E. Let the sequence arguments to `map` be of different types, and let `map` always return a list. Use a coercion function to change the result to some other sequenc type if desired.

** Responses

MOON: D	RMS: E!	SEF: D!	GINDER: D!	DM: A!
DLW: E!!	HIC: E	WLS: B!	CHIRON: E!	
ALAN: D!	GLS: E	RPG: E!!		

WLS: B (or E), but B seems to satisfy most non-exotic uses. B, D, and E would produce equivalent code anyway.

MOON: There isn't a function which makes a sequence given the length and the type.

ALAN: [112.4] [`map` is renamed `map1`.] Fine, `map` is essentially useless and unused currently.

*** Issue 133: Get rid of `scan-over` and friends?

??? Query: [115.8] I am not excited at all over these names [`scan-over` and friends]. In NIL these were called `skip`, `skpq`, `skp`, and so on; Fahlman and others have objected to those names. One idea is to can them and just use `pos`:

```
(scanq x s) <=> (pos #'(lambda (a b) (not (eq a b))) x s)
```

Any other suggestions?

DLW: [Regarding `skip`, `skpq`, and `skp`:] Yecch!!

SEF: Flush these.

** Suggestion: Flush the `scan-over` series.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y	GINDER: Y!
	HIC: Y	WLS: Y!	CHIRON: Y
ALAN: Y!	GLS: Y	RPG: Y	

*** Issue 134: Eliminate `sortslot` by adding optional argument to `sort`?

MOON: For `sort` and `merge`, it might be better to make *key-function* an optional argument [and presumably flush `sortslot` and `mergeslot`].

GLS: Actually, we could then flush `sortcar` too, writing instead (`sort` *sequence pred* `#'car`).

** Suggestion: Flush `sortcar` and `sortslot`. Let `sort` take an optional third argument, which defaults to the identity function, and which is applied to each element of the *sequence* to extract the sort key from the element.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y!	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

RMS: Perhaps keep `sortcar` for compatibility.

*** Issue 135: Have `stable-sort` as well as `sort`?

MOON: Should also have `stable-sort` and friends, which guarantees to use a stable sorting algorithm.

GLS: The concept of `stable-merge` is also meaningful; however, it's not difficult to guarantee that the ordinary `merge` is stable.

** Suggestion: Add the function `stable-sort`.

Y

** Responses

MOON: Y	RMS: N!!!	SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y!		CHIRON: Y!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

RMS: Sorting should always be stable.

GLS: One of the all-time favorite sorts, Quicksort, is not stable.

*** Issue 136: Should there be only a destructive merge?

SEF: Make `merge` be destructive, and flush `nmerge`.

GLS: There may be uses for a non-destructive merge.

** Suggestion: Flush `nmerge`, and make `merge` destructive.

N?

** Responses

MOON: N	RMS: Y!	SEF: Y	GINDER: N!	DM: Y!
DLW: N	HIC: N	WLS: N!		
ALAN: N!	GLS: N	RPG: N!	DILL: N	

RMS: I vote yes, unless a non-destructive merge is really better than copying the list and doing a destructive merge.

GLS: Well, a non-destructive merge might be defined to save consing by sharing the tail of whichever list didn't run out first. In some applications (especially merging short lists into a long one) this might be faster.

*** Issue 137: What does merging do?

ALAN: [120.4] [For `merge`,] must *sequence1* and *sequence2* be sorted? What happens if they aren't?

DLW: I don't think I understand what it is to merge. What would be the result if one or both input sequences were unsorted?

** Suggestion: Add the following clarification to the documentation of `merge`:

> The result of merging two sequences *x* and *y* is a new sequence *z* such that the length of *z* is the sum of the lengths of *x* and *y*, and *z* contains the all the elements of *x* and *y*. If *x1* and *x2* are two elements of *x*, and *x1* precedes *x2* in *x*, then *x1* precedes *x2* in *z*; similarly for elements of *y*. In other words, *z* is an *interleaving* of *x* and *y*.
>
> Moreover, if *x* and *y* were correctly sorted according to the *predicate*, then *z* will also be correctly sorted. If *x* or *y* is not so sorted, then *z* will not be sorted, but will nevertheless be an interleaving of *x* and *y*.

Y

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

Chapter 13

Manipulating List Structure

13.1. Conses

13.2. Lists

***** Issue 138: Treatment of dotted lists by list and sequence operations**

MOON: Specify what happens with dotted lists and be consistent everywhere.

ALAN: [109.5] Well mumble [that `length` and `list-length` behave differently given a vector]. In any case the text should make clearer what is going on here.

MOON: [125] `list-length` of a vector should be an error. All such functions should barf if a *list* argument is not `listp`.

DLW: Yes! This "dotted list" business should not be usurping all of LISP's error-checking!

DLW: [110.5] Is `reverse` of a dotted list the same as `list-reverse` of it?

DLW: [127.4] For `reverse`, too, don't consider random objects to be 0-length dotted lists. `(list-reverse 'a)` should be an error. And so should `(reverse 'a)`.

MOON: Define what to do about dotted lists. I suggest that list functions stop on `atom` (rather than `null`), but barf if their initial argument is not `listp` (rather than treating any atom as the empty list), and sequence functions use `atom` (rather than `null`) for lists, but barf if their argument is not some kind of sequence (i.e., a number or a symbol is not accepted as a zero-length list). (The draft manual hints in this direction but is not explicit enough.)

GLS: This idea is fine on the surface, and is what MACLISP does in many places, but it worries me from a formal point of view because it raises certain anomalies, an example of which is that `append` is no longer associative. The question is whether dotted lists are really to be considered legal or not to sequence and list functions. I would say not. The reason is that otherwise one gets anomalies in the case of, say, lists which are dotted by a vector. If we decide that dotted lists are not legal, then as a matter of robustness (to preserve the integrity of the storage system), implementations would be *very* strongly encouraged to use `atom` checks instead of `null` checks to avoid running off the end of a list; would be *moderately* encouraged to barf at atoms as arguments; and would be *mildly* encouraged to barf at dotted lists as arguments. (The

encouragement increases in direct proportion to danger and in inverse proportion to probable overhead.)

** Suggestion: Define sequence-type operations officially not to accept dotted lists; however, encourage implementors to use `atom` checks rather than `null` checks for robustness. Also encourage as much error-checking as the implementor can comfortably provide.

Y?

** Responses

MOON: Y	RMS: N!	SEF: N!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: N!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!		

SEF: I'm with MOON here.

RMS: A non-list should be allowed instead of a list if and only if nobody is going to look for its components, for example, in the last argument of `append`. Aside from that, it is reasonable to barf on atomic arguments. Whether to use a `null` or `atom` check is reasonable to leave up to the implementation. On microcoded machines it might as well be `atom`, but on conventional ones I don't think the cost is worth it.

*** Issue 139: Add `setnth` function?

DLW: [125.5] Add `setnth` here after `nth`.

** Suggestion: Add `setnth`.

Y

** Responses

MOON: N		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!		

MOON: Use `setf`.

*** Issue 140: Let `make-list` take keyword arguments

MOON: [126.5] Make `make-list` compatible with Lisp Machine LISP (takes keyword arguments).

DLW: Yes!

GLS: This would make it like `make-array` and other such functions.

** Suggestion: Let `make-list` take keyword arguments.

Y

** Responses

MOON: Y		SEF: Y!	GINDER: Y!
DLW: Y!!	HIC: Y!		CHIRON: Y!
ALAN: Y!	GLS: Y	RPG: Y!	

*** Issue 141: Treatment of circular structures

ALAN: [127.2] The warning in `copytree` about shared substructure applies to `copylist` and `copyalist` too, right?

MOON: It isn't that circularities aren't preserved; they cause it to lose.

** Suggestion: Clarify documentation to state that shared structures will be duplicated and circularities will cause the functions never to terminate.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!!		

*** Issue 142: Is `revappend` generic?

MOON: Is `revappend` a list function or a sequence function?

GLS: I stuck it in by analogy to `nreconc`, which in turn exists primarily because it is more efficient than `nconc` of `nreverse` and has to exist anyway as part of the implementation of `nreverse`. Also, you usually need to reverse this thing you are appending because you built it up backwards, which occurs with lists because of cons, but doesn't typically occur with vectors. I'd say keep it a list function; if not, rename it `revconcat`.

** Suggestion: Let `revappend` operate only on lists.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!		

***** Issue 143: Rename `nconc` to be `list-nconc`?**

MOON: [128.0] In Lisp Machine LISP there are `nconc` operations possible on sequences other than lists. Should this keep the name `nconc` (rather than `list-nconc`)? It's okay with me if it does.

**** Suggestion:** For compatibility and historical reasons, retain the name `nconc`.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y	RPG: Y!		

DM: `nconc` is an historical abomination. Surely `nappend` would be better? Actually, I don't like n*xxx* for destructive functions either. It would be far nicer to have something mnemonic (or is n*xxx* mnemonic? I always assumed it came from the `conc/nconc` of LISP 1.5, and I certainly see nothing menmonic there). Things like `dreverse`, or `reverseip` (for reverse-in-place), or even `reversewoc` (reverse-without-copying). Perhaps they should even have uncomfortably long names, to discourage indiscriminate use?

***** Issue 144: Second argument form to `pop`?**

MOON: [128.9] Lisp Machine LISP `pop` allows an optional second argument, which is where to store the *car*. If this has been decided to be flushed, put a compatibility note.

GLS: Indeed the Lisp Machine LISP code implements it, but it is not documented in [Weinreb 81a], nor is it even mentioned in `AI:LMMAN;MACROS >`. However, it sounds okay to me. My one reservation is that the information flow is "backwards" from that of a `setq`, and that `push` and `pop` fail to have the push-down-list argument in the same position. The Lisp Machine LISP code causes the value of a two-argument `pop` to be the value of a `setf` form, which in turn is undefined (see [Weinreb 81a]). Perhaps it is more perspicuous simply to write the `setf` yourself?

**** Suggestion:** Allow `pop` to take an optional second argument form, which if present is `setf`'d to the popped *car*.

**** Responses**

MOON: N	RMS: N!!!	SEF: N!!!		DM: N!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: N!	GLS: N!!	RPG: N!!		

***** Issue 145: Optional second argument to `butlast` and `nbutlast`**

MOON: `butlast` and `nbutlast` should take an optional second argument *n* defaulting to 1, the number of trailing elements to excise.

**** Suggestion:** Let `butlast` and `nbutlast` take an optional second argument as above.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!		RPG: Y!!		

***** Issue 146: Do we really want `firstn`, `lastn`, and `ldiff`?**

??? Query: [129.7] Do we really want `firstn`, `lastn`, and `ldiff`, given the existence of `sublist`?

MOON: `sublist` is orthogonal to `ldiff`. Note that `firstn` and `lastn` behave diffrently from `sublist` if you ask for more elements than there are. I don't think they are important, however.

DLW: RMS threw these in because he saw them in the INTERLISP manual and he thought they might be good. I believe I once saw a use of `ldiff` (it was by RMS).

**** Suggestion:** Get rid of them?

Y?

**** Responses**

MOON: N	RMS: N!	SEF: Y!!		DM: Y!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y		

RMS: I put these into Lisp Machine LISP when I ran into uses for them (in the compiler). The functions will continue to be there, whether their names are global or not. So they might as well be available. If I found them useful, so will other people. I don't really know to what extent other functions you are introducing will do the same job.

13.3. Alteration of List Structure

13.4. Substitution of Expressions

***** Issue 147: Substitution functions**

MOON: Allow the user to supply a predicate to `subst`. I suggest making it an optional argument which uniformly defaults either to `eq` or to `eql`. Then flush `substq` and `nsubstq`. In NIL, `subst` is a sequence operation, I think. There doesn't seem to be any predefined function for substitution in sequences in COMMON LISP.

MOON: There should also be a version of `subst` that copies only what it needs to (like `subst` [GLS: did you mean `sublis`?]). Some Lisp Machine LISP programs define this for themselves.

GLS: Besides `subst`, which operates on trees, there should be some function to perform substitution in sequences.

**** Suggestion:** Add a generic sequence-substitution function. Resolve the matter of predicates to `subst` in a manner compatible with the sequence functions. Provide a version of `subst` on trees that copies only what it must.

**** Responses** **Y**

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: X!	GLS: Y!	RPG: Y!!		

ALAN: I like the idea of a version of `subst` that copies only what it must. I am against a generic sequence substitution function.

13.5. Using Lists as Sets

***** Issue 148: A cross of `push` and `adjoin`**

MOON: There should be a `push` version of `adjoin` (sometimes called `push*` or `add2l`, but these are lousy names).

**** Suggestion:** Add a macro `adjoinf`, such that

```
(adjoinf x y) ==> (setf y (adjoin x y))
```

**** Responses** **Y**

MOON: Y		SEF: Y!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: X!	GLS: Y	RPG: Y!		

ALAN: We do want something like this, but I don't like the name `adjoinf`.

***** Issue 149: A function to eliminate duplicates**

MOON: There needs to be a function which "converts a list to a set", i.e., eliminates duplicates from a list. Takes a comparison predicate and comes in destructive and non-destructive versions. Probably does not need to work on non-list sequences.

SEF: Yeah, I guess so. I've written this a few times.

GLS: Probably should be a generic sequence operation.

** Alternatives:

A. Add a function `eliminate-duplicates` to non-destructively eliminate duplicate entries from a sequence, and `neliminate-duplicates` to do this destructively. (Somehow the equality predicate must be specifiable.)

B. Same, but call them `to-set` and `nto-set`.

C. Same, but call them `remove-duplicates` and `delete-duplicates`.

D. Same, but call them `remtwins` and `deltwins`.

C?

** Responses

MOON: C	RMS: C!	SEF: C!	GINDER: A!	DM: B!
DLW: C!!	HIC: C	WLS: C!	CHIRON: C!!	
ALAN: X!	GLS: C!	RPG: A!!	DILL: C	

ALAN: Yes, there should be such functions; no, they should not be generic sequence functions. I resist the move to make sets out of any sequence. Representing sets using lists using `equal`, `eq`, or `=` is something I have done often enough to want in the language, but I have *never* wanted to use an array. This is not to say that that might be a bad idea, just that if I wanted it I wouldn't mind doing the engineering myself.

ALAN: [135.4] [The names `unionq` and `uniteq` are used inconsistently for the same thing.] I would prefer if they were *all* rooted from "`union`" (somehow).

GLS: I would hope that the outcome of the generic sequence issue might clarify these naming conventions.

***** Issue 150: Let "sets" be sequences of any form?**

GLS: Should `union`, `intersection`, and so on be generic sequence operations?

** Suggestion: Let the set operations be generic sequence operations.

** Responses

MOON: N		SEF: N!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!!!	CHIRON: Y!	
ALAN: N!	GLS: Y!!	RPG: Y!	DILL: N	

*** Issue 151: Are destructive functions also guaranteed to be non-consing?

MOON: Are the "n" (destructive) versions of things guaranteed not to cons? This isn't discussed.

SEF: That's the whole point, right? May as well come out and say it.

** Suggestion: Explicitly require and document this property.

** Responses

MOON: X		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: N!	GLS: Y!!	RPG: Y!!	DILL: Y	

MOON: In systems with cdr-coding it is impossible to avoid some consing in some cases. Document that the intention is to build new structure by destroying and recycling the old, rather than the consing, but the precise details are implementation-dependent.

13.6. List-Specific Sequence Operations

MOON: Is `list-map` identical to `mapcar`?

GLS: I think so.

13.7. Association Lists

ALAN: [142.1] [It is permissible to let ( ) be an element of an a-list in place of a pair.] Why? Explain.

GLS: I think that this is so that you can splice out an a-list pair without really splicing; you can just `rplaca` the pair to be ( ). This might be important if several a-lists share the same tail; splicing a pair out of one a-list might fail to remove the pair from a-lists it shares with. I think the MACLISP compiler uses this trick.

*** Issue 152: Flush the function acons?

RMS: [142.2] The function `acons` is equivalent to two calls to `cons`, so I don't think it needs a name of its own.

SEF: I disagree. The use of `acons` indicates the user's intent.

** Suggestion: Retain `acons`.

Y

** Responses

MOON: N		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!!	RPG: Y!		

ALAN: [146.7] [`sassoc` and `sassq` have been omitted.] Who would notice them in this forest?

13.8. Hash Tables

ALAN: [147.8] [The arguments to `gethash` and friends are not consistent with `get` and friends.] Not necessarily true! You can think of the hashtable as an *indicator* that can be used to get properties from arbitrary objects. With property lists the *object* must be a *symbol*. With hashtables the *indicator* must be a *hashtable*, a much more reasonable restriction, really.

GLS: I'll remove the remark from the COMMON LISP manual if you will remove it from the Lisp Machine LISP manual.

13.8.1. Hashing on EQ

MOON: [148] No colons on the keywords for make-hash-table?

GLS: This was a typo.

*** Issue 153: Rehash threshold for hash arrays

MOON: The `rehash-threshold` option is not meaningful for some hash algorithms.

GLS: True. What algorithm does Lisp Machine LISP use? Maybe this parameter can be generalized to cover more hash methods.

** Suggestion: Retain the parameter for now.

Y

** Responses

MOON: Y		SEF: Y	
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y
		RPG: Y!	

*** Issue 154: Add the function `swaphash`?

MOON: Add the function `swaphash` as in Lisp Machine LISP.

GLS: This function is like `puthash`, but returns the old associated value, if any, and a flag saying whether there was in fact an old value. Should we also have `swapq` and `swapf` to match `setq` and `setf`? How about `swapprop` (in England they would spell it `swopprop`)?

** Alternatives:

A. Add `swaphash`.

B. Add `swaphash`, `swapq`, `swapf`, and `swapprop`.

C. Add `swaphash`, `swapq`, `swapf`, `swapprop`, `rswapa`, `rswapd`, `aswap`, `vswap`, and everything else similar.

D. Status quo: add no swapping operations.

** Responses

MOON: X	RMS: N!	SEF: D!	GINDER: D!	DM: D!
DLW: B!!	HIC: A	WLS: D	CHIRON: D	
ALAN: A!	GLS: B	RPG: B!		

GLS: RMS really did vote "no"; probably that should be interpreted as "D".

MOON: Add `swaphash` and `swapf`.

13.8.2. Hashing on EQUAL

*** Issue 155: Have only one set of hash functions?

MOON: Don't perpetuate this mistake! Have one set of functions, and let them behave according to the type of table given.

GLS: If I interpret this remark correctly, this would mean having two `make-` functions, but only one set of access functions?

** Suggestion: Flush `gethash-equal`, `puthash-equal`, `remhash-equal`, `maphash-equal`, and `clrhash-equal`. Let `gethash`, `puthash`, `remhash`, `maphash`, and `clrhash` operate on any kind of hash array.

Y

** Responses

MOON: Y	RMS: Y!!!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y!	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y	DILL: Y	

MOON: Obviously there should not be a function `make-equal-hash-table`, and `make-hash-table` takes a keyword for what comparison function to use. Defining a new type of hash table requires supplying at least a comparison function and a matching hashing function, and may require or desire differences of implementation. Certainly `eq`, `eql`, `equal`, and `equalp` are all reasonable types of hash tables.

13.8.3. Primitive Hash Function

*** Issue 156: Result of `sxhash`

MOON: In Lisp Machine LISP, `sxhash` is now guaranteed to return a non-negative integer.

** Suggestion: Require COMMON LISP implementations of `sxhash` to return a positive fixnum?

Y?

** Responses

MOON: N		SEF: Y!	
DLW: Y!	HIC: Y		CHIRON: Y
ALAN: N!	GLS: Y!	RPG: Y	

MOON: `sxhash` is only non-negative because we don't have `mod` yet. Better to fix the problem at its root.

Chapter 14

Strings

DLW: It is okay to abandon the automatic coercion of single characters into strings in the string processing functions.

MOON: I'm not sure I agree, but don't feel strongly about it. I should think more about this.

***** Issue 157: Strings and fill pointers**

DLW: COMMON LISP apparently only supports *fixed* strings and not *varying* strings (in the PL/I sense); that is, they can't have fill pointers. This greatly detracts from their utility!

MOON: There is no provision for strings with fill-pointers. They have been found to be extremely useful. Don't be misled by the name. Fill-pointers are not just for filling; they are really for what PL/I calls "varying". This is important.

DLW: [179] Fill-pointers aren't just for filling; they're also for *varying* strings and 1D arrays.

RMS: I think that all vectors should have fill pointers. About half the use of fill pointers in Lisp Machine LISP is for strings. The other half could use COMMON LISP arrays, but the string half cannot. So it may be okay for ordinary vectors to have no fill pointers, but strings need them. Since it is somewhat ugly for one type of vector to have a fill pointer and not all vectors, it is best for all to have them. I'm sure it will be useful on the other kinds of vectors anyway.

GLS: It would seem a pity to impose the overhead of a fill pointer on all kinds of vectors, even strings, which don't need them (such as print names).

SEF: Better to just allow an array type with string data vector and get the fill pointer that way. I still want to keep vectors simple.

GLS: I propose the following model. Vectors are simple 1D objects, and come in various specialized forms (such as strings and bit-vectors). For any specialized vector type, there is a corresponding array type. Arrays are more complex than vectors, containing leaders, fill pointers, and a vector of domension data (just as vectors contain scalar dimension data), and allowing the sharing of data with other arrays (displacing). Now, we could add fill pointers to vectors, but instead I propose to allow arrays to be used as sequences. Thus the string operations would accept strings, symbols, and also character arrays. If the `#A` syntax is retained, then a

varying string could be printed in a manner different from a simple string, such as `#1A"Foobar"` instead of `"foobar"`; however, some switch to `print` could cause such arrays to print as ordinary strings instead. I would also suggest that a facility be added to destructively shrink vectors, which would allow one to allocate a large vector, fill it partway, and then shrink it to fit the data.

** Alternatives:

A. Require all vectors, as well as arrays, to have fill pointers.

B. Let arrays be sequences too, and use them when a vector with a fill pointer (or a leader, or sharing) is needed.

B?

** Responses

MOON: A	RMS: B!!!	SEF: B!!!	GINDER: B!	DM: B!
DLW: B!!	HIC: A!	WLS: B!	CHIRON: B	
ALAN: B!!!	GLS: B!!!	RPG: B!		

*** Issue 158: Control of case dependency in string operations

MOON: Decide what to do about alphabetic-case-dependency in string and character comparison. The draft COMMON LISP specification is incompatible with Lisp Machine LISP.

GLS: Lisp Machine LISP currently has an interesting problem. The manual, last time I looked at it, said there is a global switch `alphabetic-case-affects-string-comparison` to control case dependency, but if you ever use it everything in the world will break because everyone depends on *not* depending on case. I think the comment by RMS on `read-preserve-delimiters` is relevant here. At any given function-call site, you always know whether or not you want case dependency, so a global variable is an inappropriate way to do it. In any case, it ought to be easy to get either behavior.

** Suggestion: Do something about this compatible with the resolution of treatment of predicates in generic sequence operations.

Y?

** Responses

MOON: Y	RMS: N!	SEF: Y!!!	
DLW: Y!!	HIC: X	WLS: Y!	CHIRON: N
ALAN: Y!	GLS: Y!	RPG: Y!	

RMS: It would be a good idea, but I'm afraid that calls to `string-equal` are buried inside too many things to be handled that way. That is, the cases where one might want to set this flag will include ones where you call `A` which calls `B` which calls `C` which calls `string-equal`, and `A` and `C` are old-fashioned names for which there are no modern equivalents that ask you to specify the predicate, and you wouldn't want to write those things yourself because it would mean duplicating lots of code. Perhaps this fear can be proved groundless, but look into it carefully.

MOON: The Lisp Machine LISP manual [Weinreb 81a] says you may bind `alphabetic-case-affects-string-comparison` but not `setq` it, not that you may not use it. The reason it's a special variable rather than an argument is precisely because of the problem of making the string functions take a whole bunch of different optional arguments. Also note that it is sometimes passed down through several intermediate functions to a string primitive; the most obvious example is `assoc` but I'm sure there are others involving more levels. The suggestion is reasonable.

14.1. String Access and Modification

14.2. String Comparison

***** Issue 159: Have *start/end* arguments for `string<` and friends?**

MOON: `string<` and friends should have *start/end* arguments like `string=`.

GLS: I left this out because there is an ambiguity, or at least an asymmetry, as to whether the returned index is into the first or second string. This doesn't arise if no *start/end* arguments are permitted. However, `mismatch` has the same problem, and so we could resolve this in a similar way.

** Suggestion: Let `string<` take a pair of *start/end* arguments, and let the result be as for `mismatch`: an index into the *first* string argument.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y	

14.3. String Construction and Manipulation

***** Issue 160: Keyword arguments for `make-string`**

MOON: `make-string` should take keyword arguments as `make-list` [in Lisp Machine LISP] and `make-array`.

** Suggestion: Let `make-string` take keyword arguments as `make-array` does.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!
ALAN: Y!	GLS: Y!	RPG: Y!	

*** Issue 161: Generalization of `string-repeat`

GLS: Let `string-repeat` take extra optional arguments defaulting to "" which are interpolated; thus

```
(string-repeat "Foo" 3 ", " "(" ")") => "(Foo, Foo, Foo)"
```

A rough definition of this would be:

```
(defun string-repeat (string count
                      &optional (between "") (before "") (after ""))
  (let ((step (+ (length string) (length between)))
        (nbetween (max 0 (- count 2))))
    (let ((result (make-string (+ (length before)
                                  (length after)
                                  (* (length between) nbetween)
                                  (* (length string) count)))))
      (replace result before)
      (do ((k (length before) (+ k step))
           (j nbetween (- j 1)))
          ((zerop j)
           (replace result string k)
           (replace result after (+ k (length string))))
        (replace result string k)
        (replace result between (+ k (length string))))
      result)))
```

Maybe this should be generic? (Unfortunately, the name `repeat` probably ought not to be used for anything but something analogous to a PASCAL-style **repeat-until** statement.)

** Alternatives:

A. Generalize `string-repeat` as above.

B. Generalize it and make it a generic sequence function.

C. Don't generalize it, but make it generic.

D. Status quo: leave it alone.

** Responses

MOON: X	RMS: C!	SEF: D!	GINDER: A!	DM: D!
DLW: A	HIC: B	WLS: A!	CHIRON: B	
ALAN: A!	GLS: B	RPG: A!	DILL: A	

RMS: However, I suspect that `string-repeat` should not even exist, because the right thing is to use a **do** inside `with-output-to-string`.

GLS: I have to agree with RMS on that one. I came close to asking to flush it. However, a generic version might be very useful for setting up patterned arrays. Maybe that's silly, though.

MOON: I'd prefer to see `string-repeat` flushed, but could also live with C or D.

***** Issue 162: Should `string-upcase` and friends be required to copy the argument?**

MOON: [154.4] `string-upcase` and friends should be allowed to return the argument itself if no character needs to be changed.

**** Suggestion:** Allow `string-upcase` to return the argument itself if appropriate.

Y?

**** Responses**

MOON: Y		SEF: Y!!		DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: Y	RPG: N!	DILL: N!!!	

14.4. Type Conversions on Strings

14.5. Sequence Functions on Strings

***** Issue 163: Is `string-reduce` really useful?**

??? Query: [156.5] Should the `reduce` functions be omitted as useless, or retained for symmetry?

DLW: Retain, I guess. But I agree that these functions shouldn't exist explicitly.

**** Suggestion:** Defer this until the generic sequence business is resolved.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y	RPG: Y!!		

***** Issue 164: Add the `string-...-set` series of functions from Lisp Machine LISP?**

MOON: COMMON LISP doesn't provide the `string-search-...-set` series?

DLW: This series is really useful: `(string-search-set '(#\space #\tab) x)`.

GLS: You can build it from generic sequence functions, but maybe they should be included anyway.

** Suggestion: Defer this until the generic sequence business is resolved.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y	RPG: Y!!	DILL: Y	

Chapter 15

Vectors

15.1. Creating Vectors

***** Issue 165: Keyword arguments for `make-vector`?**

MOON: `make-vector` should take keyword arguments.

** Suggestion: Let `make-vector` take keyword arguments as for `make-array` (and as proposed for `make-list`).

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!
ALAN: Y!	GLS: Y	RPG: Y!	

15.2. Functions on General Vectors (Vectors of LISP Objects)

***** Issue 166: Argument order for `aref`, `vref`, `setelt`, and others**

MOON: [108.6] Unfortunate argument order for `setelt`.

DLW: Foo. Argument order completely incompatible with `aset`. This would be confusing.

MOON: I am unhappy about using the name `vset` for a function whose arguments are not in the same order as for `aset`. This is certain to cause bugs forever.

SEF: But so is having them both be wrong.

GLS: Nearly every other "update" operator in the language takes the new value last, `putprop` being the outstanding exception. But consider `rplaca`, `setplist`, MACLISP's `store`, `setq`, and `setf`. It is `aset` that is inconsistent. (It is of course that way because it must take a variable number of array subscripts, which are placed last so as to make up a `&rest` argument.)

** Alternatives:

A. Status quo: leave `aset` alone, and let everything else be parallel to `rplaca` (new value is the last argument).

B. Leave `rplaca` and friends alone, but change the new sequence operations such as `setelt` and `vref` to have the new value be the first argument, like `aref`.

C. Revamp the language so that *everything* (even `setq`) takes the new value first, in order to accommodate `aset`.

D. Change `aset` to be compatible with `rplaca`, taking the new value last.

** Responses

MOON: X	RMS: A!!!	SEF: A!!!	GINDER: D!	DM: D!
DLW: B!!	HIC: D	WLS: BD!	CHIRON: D	
ALAN: A!	GLS: A!	RPG: D!!	DILL: D	

RMS: Changes to these things would be painful, and they are not necessary for cleaning up the language. Just tell users to use `setf`. They should not use `aset` or `vset` or `rplaca` directly.

MOON: Leave `setelt` the way it is, by analogy with `setq`, but change `vset` to be compatible with `aset` or else change its name.

15.3. Functions on Bit-Vectors

*** Issue 167: May sequence operations call a predicate on a non-element?

MOON: [167.3] [Actually, one should be more careful, to avoid calling *predicate* at all if the bit-vector is empty...] This seems like an unreasonable restriction on the implementation.

GLS: The issue here is that certain vector operations can be implemented cleverly by knowing that the element domain is small; one can then invoke a user-given function or predicate once on each element of the domain, make up a table of results, and use the table to drive a fast (possiblky hardware) implementation of the operation. Should such an implementation be forced not to call the predicate on elements not actually in the vector?

** Suggestion: Allow such operations to be called on non-elements, just as an implementation is permitted to cache values by assuming them to be free of side effects.

Y?

** Responses

MOON: N!!!		SEF: Y	GINDER: Y!	DM: Y!
DLW: Y	HIC: Y	WLS: Y!		
ALAN: N!	GLS: Y!	RPG: N!	DILL: Y	

??? Query: [169.4] These functions [`bit-sort` and friends] are incredibly useless, but have an efficient (linear-time) implementation!

ALAN: Aww... come on!!!

15.4. Functions on Vectors of Explicitly Specified Type

Chapter 16

Arrays

16.1. Array Creation

MOON: Not all array types are distinguishable by their element types. Lisp Machine LISP will accept other things here in addition to data types.

***** Issue 168: Grammar of keywords for `make-array`**

MOON: `:initial-value` or something would be more grammatical than `:initial` (for `make-array`).

GLS: I think so too, but it's kind of long.

**** Suggestion:** Rename the `make-array` keyword `:initial` to `:initial-value`. **Y**

**** Responses**

MOON: Y
SEF: Y!
GINDER: Y!
DLW: Y!
HIC: Y
CHIRON: Y
RPG: Y!

DLW: [176.9] [Both Lisp Machine LISP and FORTRAN store arrays in column-major order.] We will fix Lisp Machine LISP, at least on the L-machine. FORTRAN will have to fend for itself.

***** Issue 169: Second result value from `make-array`**

> **??? Query:** [176.9] From the Lisp Machine LISP manual: "`make-array` returns the newly-created array, and also returns, as a second value, the number of words allocated in the process of creating the array, i.e. the `%structure-total-size` of the array."

SEF: Flush second value.

MOON: Ignore this; it is a historical artifact of something like the cold-load generator.

**** Suggestion:** Remove mention of this from the COMMON LISP manual.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!!		

16.2. Array Access

16.3. Array Information

DLW: `array-active-length` just returns the length of the array if there is no fill pointer.

GLS: Will fix documentation.

*** Issue 170: Argument types for `array-in-bounds-p`

MOON: Surely `array-in-bounds-p` signals an error if the *subscripts* are not integers (fixnums in most implementation, actually).

GLS: Well, the Lisp Machine LISP manual [Weinreb 81a] is not clear either on this point. However, I am willing to go along with this.

** Alternatives:

A. `array-in-bounds-p` signals an error rather than returning ( ) if any argument is not an integer.

B. `array-in-bounds-p` signals an error rather than returning ( ) if any argument is not a fixnum. (This is probably less portable.)

C. Status quo: *any* unsuitability of arguments simply causes ( ) to be returned.

A?

** Responses

MOON: X		SEF: A!!	GINDER: C!	DM: C!
DLW: A!!	HIC: A	WLS: C	CHIRON: A	
ALAN: A!	GLS: A!	RPG: A!		

MOON: Certainly `array-in-bounds-p` is not some substitute for `errset`, by which you tell if applying this array to these arguments would signal any sort of error. Like most functions, `array-in-bounds-p` should rely on whatever error checking the implementation provides for type-checking its arguments, and should not have to do it explicitly. My comment was supposed to mean that `array-in-bounds-p` may not be relied upon to return ( ) if you give it non-numbers as arguments, and is allowed to err if the implementation type checks arithmetic operations at run time. What should be made clear is that `array-in-bounds-p` isn't allowed to get the wrong answer if the integers you give it happen to be bignums, even if

the implementation does not allow bignums as array subscripts, which is probably true of all implementations.

16.4. Array Leaders

16.5. Fill Pointers

***** Issue 171: Should array fill pointers live in the leader?**

> ??? Query: [179.3] The following comes from Lisp Machine LISP, and is somewhat of a crock. Should this be retained for compatability? (If so, fill pointers should be initialized to ( ), not the array-length.)
>
> "By convention, the fill pointer is kept in element number 0 of the array's leader. We say that an array *has a fill pointer* if the array has a leader of non-zero length and element number 0 of the leader is an integer. Normally there is no fill pointer."
>
> It would be nice if fill pointers and named structures did not interact so randomly with the leader. (Then again, what's a leader for?)

DLW: Exactly.

MOON: I used to think that it was a crock that the fill pointer and the named-structure symbol for an array were in the leader. But I now believe that I was wrong. The leader is for all the things that don't go in the main part of the array because they don't fit into the "space" defined by the array dimensions. It makes no sense to make a leader-leader which is where things go that you don't want to put in the leader. The numerical indices into the leader should not be a concern of the user; the leader should not be used like a 1-dimensional array (or vector); rather it should be used as a structure. The problem with this was that the user always saw these mysterious system usages of some of his structure elements, and had to know to `defstruct` things in the right order. Now that there are slot-options in defstruct, this can be handled very neatly. When using the `array-leader` type, defstruct never automatically allocates anything to one of the system-defined array-leader entries by default; it starts allocating the user's structure slots after those entries. However, if a `defstruct` slot has the `:fill-pointer` slot-option, it goes into the fill-pointer slot (regardless of the order it is written in the defstruct slot list). Similar slot-options can exist for named-structure-symbol and any other future system usages of the array leader. A non-issue is "What if new system array leader usages are defined? Doesn't that invalidate old (compiled) defstructs?" This is a non-issue because each system usage of an array leader slot should be turned on by some bit in the array (as named-structure is) so that they aren't there in arrays that aren't using them, and if the system gets a new feature all arrays that already exist don't suddenly invoke that feature and break. Note well: this bit doesn't just enable the use of an array-leader slot, rather it enables a behavior of the array, which incidentally ascribes a meaning to an array-leader slot, as well as the other changes it makes to the behavior of the array. The current Lisp Machine LISP feature that non-fixnums in the fill-pointer are ignored rather than signalling a data-type error can be flushed. I guess arrays with leaders should always have fill pointers, rather than having a bit which enables the fill pointer, although I could go either way on this.

SEF: Well, the difference between Lisp Machine LISP arrays and SPICE LISP arrays is coming up here. For SPICE LISP it is much nicer to let the fill pointer live in the array header, and leave the whole leader for

optional use by the user.

** Suggestion: Put the fill pointer in the leader, as in Lisp Machine LISP.

** Responses

MOON: Y		SEF: N!!!	GINDER: N!!!	DM: Y!
DLW: Y!!!	HIC: Y!!!		CHIRON: N!	
ALAN: X!	GLS: Y	RPG: Y!		

ALAN: It might not be necessary to specify where the fill-pointer lives at all. If functions are provided to store and retrieve the fill pointer, and if array leaders are only accessed through `defstruct`, then it doesn't make any difference where they really live. If they are really in the leader, then `defstruct` knows how to avoid them unless the user uses a slot option indicating he wants the fill pointer. If they are not in the leader, then `defstruct` knows how to define accessors for the fill-pointer correctly.

GLS: Is `defstruct` really the only way array leaders are used??

SEF: This should be left to the implementor's discretion. Putting the fill pointer in the leader is inefficient in SPICE LISP, and putting it elsewhere is ugly in Lisp Machine LISP.

*** Issue 172: Shall `array-push` and `array-pop` accept multidimensional arrays?

MOON: [179.8] The Lisp Machine LISP documentation is in error; `array-push` will accept a multi-dimensional array. However, the multi-dimensional case is essentially useless. Currently `array-pop` *does* require a one-dimensional array.

** Suggestion: Let them both accept multidimensional arrays.

** Responses

MOON: N		SEF: Y!!	GINDER: Y!
DLW: N!!	HIC: N	WLS: Y!	CHIRON: Y!!
ALAN: N!	GLS: Y!	RPG: Y	

*** Issue 173: Should `array-push` and `array-pop` be uninterrruptible?

MOON: I don't care whether `array-push` remains uninterruptible.

GLS: Might be hard for some implementations to implement uninterruptibility efficiently.

** Suggestion: Do not require them to be uninterruptible.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: N!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!!	RPG: Y!		

16.6. Changing the Size of an Array

*** Issue 174: Growing displaced arrays

MOON: [180.6] [If the *array* used to share with other arrays, then after an `adjust-array-size` operation it may or may not continue to be shared with other arrays.] But what does this mean? And does it have anything to do with displacing?

MOON: What is a "shared" array?

DLW: Since the keyword is "`:displaced-to`", why not just call them displaced arrays, rather than using "displaced", "indirect", and "shared" interchangeably?

MOON: [In Lisp Machine LISP it is possible for the array returned by `adjust-array-size` not to be `eq` to the argument array.] If this is not reflected in the COMMON LISP definition, then the implementation would be forced into a particular implementation of arrays (separate headers and data). Note that in any case, in Lisp Machine LISP, the old array will continue to work, since it will be forwarded to the new array if there is one. Make this explicit also.

GLS: Before, I thought that displacedness and sharedness (the way SPICE LISP arrays are implemented) might have slightly different properties. I think I was confused, and everything works out fine. I'll clean up the wording.

** Suggestion: Clarify the documentation, and make precise the interactions caused by growing displaced arrays.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y!	RPG: Y!!		

***** Issue 175: Compatible (but ugly) extension of `array-grow`**

MOON: The way `array-grow` takes an "optional" *after* a "rest" argument is a crock. I would rather you changed it incompatibly to add an extra (required) argument.

RMS: For a few functions, I saw argument lists that try in effect to have a mandatory argument following a `&rest` argument. I think those argument orderings are very bad, and that an incompatible change to the order of arguments is better.

GLS: I think it is important to be able to specify the initial value of newly created array components.

**** Suggestion:** Revise argument order to `array-grow` to put the dimensions at the end.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y	DILL: Y	

Chapter 17

Structures

17.1. Introduction to Structures

MOON: [184.6] Define what one-argument `typep` does for structures.

17.2. How to Use Defstruct

***** Issue 176: `defstruct` slot-option keyword format**

DLW: [185.7] [Regarding *slot-options*] Wouldn't alternating keywords and values be more obvious, that is, analogous to other things?

GLS: Well, we have the problem that half the keywords don't take arguments, just as with `open`.

**** Suggestion:** Change the format to use alternating keywords and values. Instead of "`:invisible :read-only`" one would write "`:invisible t :read-only t`".

**** Responses**

		SEF: N!	GINDER: N!	DM: N!
DLW: Y!!		WLS: Y!	CHIRON: N!!	
ALAN: X!	GLS: N!	RPG: Y!		

ALAN: I guess I am in favor of alternating keywords and values despite the fact that it is inconsistent with the `defstruct` header format, and despite the fact that it requires you to always give a value. I would like to clarify that the scheme I favor works like this:

```
(defstruct (space-ship :conc-name)
  (captain 'kirk)
  (weight :type :fixnum)
  (crew :init '() :type :list))
```

That is, we special-case the common case of just giving the `:init` keyword, by allowing you to omit it. If you want to use any other slot options, then you have to say `:init` to control initializations. (Alternatively, the presence or absence of the initialization could be determined from the parity of the length of the options list.)

17.3. Using the Automatically Defined Macros

17.3.1. Constructor Macros

17.3.2. Alterant Macros

17.4. `defstruct` Slot-Options

17.5. Options to `defstruct`

***** Issue 177: Default for `defstruct` `:conc-name` option**

JONL: [188.5] `:conc-name` should be the default. The only reason it isn't now on ALAN's `defstruct` is backwards compatibility. So most usages provide the key word, but the "right" thing should be done for COMMON LISP.

GLS: In that case you would need to let (`:conc-name ()`) mean that no name concatenation is to be done.

** Suggestion: Make `:conc-name` be the default situation for `defstruct`.

Y?

** Responses

MOON: Y		SEF: Y		DM: Y!
DLW: N!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: N!	GLS: Y	RPG: Y!	DILL: N!!!	

DILL: The default `:conc-name` should be nothing at all, under the assumption that a new structure will be encapsulated in its own package. So the `foo` structure's operations will automatically become "`foo:gazorch`", etc.

***** Issue 178: `defstruct` structure types which are not data types**

MOON: [188.9] There are `defstruct` structure types which are not data types. `tree` is the least esoteric example.

GLS: But are these esoteric types really useful enough to have in COMMON LISP?

** Suggestion: Have other `defstruct` structure types such as `tree`.

** Responses

MOON: Y	RMS: Y!	SEF: N!!	
DLW: Y	HIC: Y		CHIRON: N!
ALAN: Y!	GLS: N	RPG: Y!	DILL: N

RMS: I don't know that you need to have `tree` in particular, but certainly don't decide to rule out all such things.

*** Issue 179: The default `defstruct` structure type

MOON: [188.9] [the type defaults to `:vector`...] Change this to "the type defaults to an implementation-chosen default type, such as `vector`".

GLS: Hm, maybe that's okay, since in the default case it should be invisible to the naive user. (For example, `typep` won't give it away.) I'm not sure what is the point of this freedom, though. Does Lisp Machine LISP want to use an array? But are not vectors to be implemented as arrays?

** Suggestion: Allow the `defstruct` default type to be implementation-dependent. **Y?**

** Responses

MOON: Y		SEF: N!!	GINDER: Y!	DM: Y!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: N	RPG: Y!		

*** Issue 180: Default for named/unnamed `defstruct` option

MOON: [189] Possibly it would be better for the default to be `:unnamed` *if* a `:type` option was specified. I'm unsure.

SEF: The default should always be `:named` unless forbidden by the choice of `:type` (rather than letting the default depend on the type, that is, letting the default be arbitrary if the type permits either)?

** Alternatives:

A. Let default be `:named` unless a `:type` option was specified.

B. Let default be `:named` unless a `:type` option incompatible with `:named` is specified.

C. Status quo: the default for namedness depends on the specified `:type` option, and is `:named` if no `:type` is specified.

** Responses

MOON: A!		SEF: B!!	GINDER: B!
DLW: B!	HIC: C	WLS: C!	CHIRON: B!
ALAN: X!	GLS: B!	RPG: C!	DILL: B!!!

ALAN: I think that A is least objectionable to me. However, I would prefer that the real status quo be maintained. That is, I would prefer that default *always* be unnamed. I am still against the `:unnamed` option, and I note that if we decide on option A than `:unnamed` becomes `:unnecessary` (and should therefore be flushed).

ALAN: [189.4] [On the option `:type` (array *type*):] O.K.!

***** Issue 181: May specialized arrays be named?**

MOON: the restriction to `:unnamed` for specialized arrays is bogus. What are array leaders for?

DLW: We have lots of specialized arrays that are named!

GLS: I was just trying to keep crap out of the leader. Oh, well.

** Suggestion: Allow specialized array `defstruct` structures to be `:named`.

Y

** Responses

MOON: Y		SEF: Y		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

***** Issue 182: The `defstruct` type `:fixnum`**

??? Query: [189.8] Is this [type `:fixnum`] really necessary? Or can this be determined at `defstruct` expansion time from the byte field information?

DLW: The latter.

MOON: Flush. But what happened to byte fields?

** Suggestion: Flush the `:type :fixnum` option.

Y

** Responses

MOON: Y		SEF: Y!!		DM: Y!
DLW: Y!!		WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y	

DLW: [191.3] Explain what `:include` does with respect to two-argument `typep`. That is, every `astronaut` is also a `person`.

GLS: I thought it was explained there. I'll make it even more clear.

RPG: [November] About `defstruct`, in the `:include` stuff, if you define an `astronaut` to `:include person`, can you say `(person-name loser)` and `(astronaut-name loser)` to get the same things? So, the `:include` option would cause the included accessor functions as well as new ones?

GLS: [November] My understanding is that in Lisp Machine LISP, if `astronaut` includes `person` then `person-name` works but `astronaut-name` does not. Would you rather it be otherwise?

RPG: [November] It seems to me that that might be a useful feature for novices, because when someone hears that there is a record for "astronaut" that has all the stuff a person has, the temptation is to say `(astronaut-weight ...)`.

*** Issue 183: Consistency of `:read-only` and `:invisible` properties

MOON: [191.6] [If a slot is invisible or read-only in the included structure, then it must also be so in the including structure.] This seems like a silly restriction.

GLS: I think that consistency of `:read-only` options is necessary for consistency. Invisibleness is less necessary. However, if consistency is enforced, then the type descriptions can be shared (a minor reason).

** Suggestion: Eliminate the consistency restriction.

** Responses

MOON: Y		SEF: N		DM: N!
DLW: N!	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: Y!	GLS: N!	RPG: N!!		

ALAN: [192.2] [An error is signalled if `defstruct` and user `make-array` specifications conflict.] Now that old-style `make-array` has been flushed I will do this.

MOON: This was accidentally left over from before the `:make-array` option was keyword-oriented.

*** Issue 184: The `:size-variable` option to `defstruct`

??? Query: [192.7] The name of this option in Lisp Machine LISP, `size-symbol`, indicates a confusion between "symbol" and "variable" (which tends to pervade Lisp Machine LISP). A symbol can represent or implement a variable, but also a function, a macro, etc.

ALAN: Well, okay. But it is a rather useless option anyway (you *needed* it for grouped arrays!).

** Alternatives:

A. Call it `:size-symbol`.

B. Call it `:size-variable`.

C. Flush it (and `:size-macro` as well).

C?

** Responses

MOON: X		SEF: C!		DM: C!
DLW: B!	HIC: C	WLS: C!	CHIRON: B!!!	
ALAN: C!	GLS: C!	RPG: B	DILL: C	

MOON: It is not useless; you need it for other things besides grouped arrays. Keep `:size-variable`; flush `:size-macro`.

*** Issue 185: How to print structures

ALAN: [192.8] [On the `:print-function` option:] O.K.!

MOON: If the default way of printing a structure is going to be to dump its guts, there should be an option which automatically writes a print function which prints something concise (for example, #<, structure name, optionally contents of some slot, probably some numeric identifier such as the memory address, >).

GLS: How about letting `(:print-function t)` be the same as the default case (spill guts), and `(:print-function ())` means to use #<....> print syntax?

** Suggestion: Use a ( ) argument to the `:print-function` option to get #<...> syntax.

Y?

** Responses

MOON: X		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: X	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

MOON: `t` and ( ) are not mnemonic at all here. Also there should be a way to say what slots appear in the #< syntax.

GLS: But it's so Lispy!

CHIRON: Almost, but have the default be reversed. You want to *not* print the guts out by default.

***** Issue 186: Add `inspect` and `describe` to the COMMON LISP core**

MOON: I think `describe` should be part of the core system. It is especially useful with structures. ALAN has a `describe` and `inspect` for MACLISP which he finds extremely convenient. [DLW: Me, too, when I use MACLISP!] They're pretty simple, too.

**** Suggestion:** Add a simple `describe` and `inspect` facility.

Y

**** Responses**

MOON: Y		SEF: Y!!	GINDER: Y!
DLW: Y!	HIC: Y!		CHIRON: Y!!!
ALAN: Y!	GLS: Y	RPG: Y!!!	DILL: Y

***** Issue 187: `callable-accessors` option to `defstruct`**

ALAN: [193.3] [On the `:callable-accessors` option:] You could flush this option really; it is mostly for MACLISP `defstruct` where it defaults to `()` because it is expensive there.

GLS: Possibly it might be expensive elsewhere too. Should the option still be available to the user?

MOON: How is it not compatible with Lisp Machine LISP?

DLW: I see. Difference is minor and okay.

**** Suggestion:** Flush the `:callable-accessors` accesors option, and always have the access functions be functions.

Y

**** Responses**

MOON: Y		SEF: Y	GINDER: Y!	DM: N!
DLW: Y!		WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: N!!	DILL: Y	

17.6. By-position Constructor Macros

***** Issue 188: By-position constructor functions?**

ALAN: Well, why don't we just make these *be* functions then, okay?

GLS: Makes sense to me. You can just use the spec almost directly as the lambda-list. And that way you

could map them.

** Suggestion: Make by-position `defstruct` construcor macros really be functions.

Y

** Responses

MOON: Y		SEF: Y	GINDER: Y!	DM: N!
DLW: Y	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y	

17.7. The `si:defstruct-description` Structure

*** Issue 189: `si:defstruct-description` description

ALAN: Flush this section! It is only for `defstruct-define-type` that this really needed to be explained.

SEF: Flush this, or move to red pages.

** Suggestion: Flush this section.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y	

MOON: The extensibility features for defstruct needn't go in the COMMON LISP specification now, but they do need to be standardized across all implementations that have them. I imagine the code that implements `defstruct` will be identical for all implementations, hopefully, so this won't be a problem.

*** Issue 190: Flush the `default-pointer` option

??? Query: [195.6] Ought to flush the `:default-pointer` option? Also, it would be nicer to rename `ppss` to `byte-specifier`; `ppss` has representation-dependent connotations. Finally, may need a `data-type` slot.

** Suggestion: Flush it.

** Responses

MOON: Y	RMS: N!	SEF: Y!!	DM: Y!
DLW: N!!			
ALAN: N!	GLS: Y!!	RPG: Y!	

RMS: Default pointers are sometimes convenient, especially in programs where only the default pointer is ever

used.

MOON: Put byte specifiers back in.

Chapter 18

EVAL

Chapter 19

Input/Output

19.1. Printed Representation of LISP Objects

19.1.1. What the `read` Function Accepts

DLW: [200.7] [A macro-character function may or may not return a LISP object.] Does this mean returning zero values, or what?

GLS: Yes; I'll clarify this in the text.

DM: [November] I'm somehow made very uncomfortable by this chapter. Everything seems rather kludgey, sort of "Well, we'll make it as similar to what we're used to as possible". I'd rathers ee a fresh start, prefereably with less overloading of the ugly character #. Try to figure out what is needed, allocate "anonymous" characters for various roles, and then actually map the real characters to these anonymous guys in as visually pleasing a manner as possible.

The use of ";" and ":" tends to make life a little more awkward for folks putting ALGOL-like syntax on top of LISP (see above for why anyone in their right mind would actually want to do it). It obviously can be done, but it's more pleasant if ";" and ":" aren't used indiscriminately in the LISP. This may be rather a hopeless cause. The same reasoning leads to the restriction that most non-alphabetic characters shouldn't be included in symbols un-escaped (so that the ALGOL-like front end can use the same format for lexemes, but allow `x+y` to be `(+ x y)` instead of a symbol). It really is convenient to be able to type `*foovar` instead of `\*foovar`. I'm probably just being dense.

CLH: [November] General comment on # syntax: In general I find MACLISP ugly to look at, at least in the manuals. (Possibly that is because manuals have a relatively high density of odd things.) The colons were bad enough. But now all these # syntaxes are muddying it even more. It is beginning to look like TECO code. `#(#\* #\;)`, etc. I am not sure quite what to do about it, but have you considered adopting other notations, e.g., using more special characters, or using words spelled out for at least some constructs? `<...>` would be obvious for vectors. And why not do like ELISP and use atoms for your characters? That would let you get rid of the #\ syntax for character objects.

***** Issue 191: Non-token-terminating macro characters**

MOON: The reader syntax here does not allow for characters which are macro characters but do not break symbols (being treated as consituents inside symbols). If this is an intentional omission, put a compatibility note. [I would prefer to] bring them back. And make # one.

SEF: This seems to add a fair amount of extra hair just to make some additional ugly tokens available. Worth it? I doubt it.

GLS: I think Lisp Machine LISP uses # in the name of two functions: `array-#-dims` and `set-syntax-#-macro-character`. Are these worth the extra conceptual complexity? (I would like to keep the COMMON LISP reader specification as simple as possible.)

**** Suggestion:** Add macro characters that don't activate in the middle of a symbol?

N?

**** Responses**

MOON: Y!!!		SEF: N!!	GINDER: N!	DM: N!
DLW: N	HIC: Y!	WLS: N!!!	CHIRON: N!	
ALAN: N!	GLS: N!	RPG: Y!!	DILL: Y	

RMS: I think that the function names with # in them should be changed: `array-#-dims` becomes `array-rank`, and flush `set-syntax-#-macro-character` because you can make # dispatch through an advertized vector of functions.

DILL: Putting these in the reader is no extra trouble; in the SPICE LISP reader, the character attributes for "number/symbol terminator" and for "macro character" are independent. As a language issue, I don't care at all.

MOON: I'm not convinced by the motivation for flushing single-character objects, but admittedly it's not important since a user can easily put them back.

***** Issue 192: Is ":" a symbol constituent character?**

MOON: [201.7] ":" is not a constituent if there are packages.

GLS: Well, one can look at it as a macro character, but then it is a very funny kind that can look behind it, which I hoped to avoid. The intent here was to punt the issue: the token recognizer can just treat colons as constituents, and then the guy who distinguishes numbers from symbols can also take colons into account. It's not necessary to make the reader be one-pass with limited lookahead; that is partly what made the MACLISP reader so hairy (for example, parsing a token as if it were a symbol, a decimal bignum, and an octal bignum all in parallel).

**** Suggestion:** Treat colon as a symbol constituent for COMMON LISP readtable purposes.

Y?

** Responses

MOON: X		SEF: Y!!	GINDER: Y!	DM: X!
DLW: Y!	HIC: N	WLS: Y!	CHIRON: Y	
ALAN: X!	GLS: Y	RPG: Y!	DILL: Y!!!	

ALAN: I am afraid to vote for this because I don't fully understand what the ramifications of it are. Specifically, I am worried that you are going to rule out the construct `si:(...)`. I am against that. If, on the other hand, you just want to say that ":" is just like "%" as far as *tokenizing*, then that is okay.

MOON: Shouldn't colon be a separate syntax class, so that you can turn it on and off, move it around, and so forth?

*** Issue 193: Why does <rubout> have *ignored* syntax?

MOON: Why does <rubout> have *ignored* syntax? For ASR33 compatibility?

GLS: Well, that doesn't hurt, but mostly I just wanted to have the property that the user could set up a character syntax by copying the syntax from some standard character. This meant that *some* standard character had to have *ignored* syntax, and I chose <rubout> pretty much because none of the others could be, and the ASR33 cultural history clinched it.

** Suggestion: Let <rubout> retain *ignored* syntax.

Y?

** Responses

MOON: N		SEF: Y!!	GINDER: Y!	DM: N!
DLW: N!!	HIC: X	WLS: Y	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!		

MOON: Wouldn't it be better to give the syntax classes names rather than putting in a kludge?

19.1.2. Sharp-Sign Abbreviations

*** Issue 194: Should the case of a letter after # matter?

MOON: [204.6] [(To be precise, *# does* distinguish case...] Delete this note; don't encourage bad style.

GLS: Do you mean that the implementation should force case independence? It's not a bad idea.

** Suggestion: However a # macro is set up, defining either an upper case or lower case version also defines the other case to do the same thing.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: N!!	DILL: Y!	

*** Issue 195: Syntax of characters

ALAN: [13.8] Not `#/Rubout`? [As opposed to the specified `#\Rubout`.]

ALAN: [206.5] [Lisp Machine LISP has recently changed to allow `#/` to handle both syntaxes.] INCORRECT! In Lisp Machine LISP `#/` *only* gobbles *one* character. `#\` was extended to allow `#\Control-Meta-/"`. I'm not convinced that what you propose here is unambiguous. (Maybe it's just hard.)

MOON: Explain how `#\`*char* and `#\`*name* are distinguished; I assume it's by whether the second following character is a constituent (skipping ignored characters).

GLS: I thought it was obvious, and charmingly simple, and was impressed that the Lisp Machine LISP people had invented it! Maybe I read too much into the notes I saw. Anyway, the rule for interpreting `#\` is trivial: `untyi` the "\" character, and then parse a symbol. Loop: If this symbol is the name of a character, then that is the character. Otherwise, if the symbol has a one-character pname, then that pname character is the character. Otherwise, if there is a hyphen in the pname, break the symbol into the parts before and after the first hyphen; treat the first part as the name of a bit, and iterate from Loop on the second part. Otherwise error.

** Suggestion: Use the syntax for `#\` described above by GLS.

Y

** Responses

MOON: X	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!!	
ALAN: Y!	GLS: Y!!	RPG: Y!	DILL: Y	

MOON: GLS's rule doesn't work because `#\rubout` and `#\Rubout` are different. With the Lisp Machine LISP reader, the new `#\` syntax is a trivial and straightforward piece of BNF (actually one piece for named characters and one piece for quoted characters).

GLS: Right you are. You can patch my rule by making character-name lookup case-insensitive.

***** Issue 196: Why have ignored characters?**

MOON: Why have ignored characters at all?

GLS: Primarily so that implementations can define ASCII 0 (NUL) to have that syntax. Some host enviroments (for example, SAIL) require ignoring of null characters.

**** Suggestion:** Retain *ignored* syntax? **Y?**

**** Responses**

MOON: N		SEF: Y!!		DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!!!		

MOON: Surely discarding of nulls for WAITS must be done at a low level in the I/O stream, not in the readtable, or everything that does input with `tyi` or `readline` will become non-portable.

GLS: Hm. I think you may be right here.

***** Issue 197: Terrible proposed syntax for arrays, vectors, and structures!!!**

ALAN: I would much prefer if we didn't have to commit to the printed representation of some of these types. #S, #A, and #( ) in particular seem worthless. (You might convince me in the case of #( ), but arrays and structures?) Why not let this be up to the implementation? It doesn't help portability or anything like that that I can see.

ALAN: [207.1] Bletch: "`#foo(@bar...`".

MOON: What an awful crock? Isn't there a better way?

GLS: I'm sure there must be!

JONL: How about *#@type*`(...)` rather than `#(@`*type* `...)`? Somehow the latter violates a sense that bracketing things should match and be fully containing; thus the former announces the type before the bracketer occurs.

RMS: The syntax for vectors, arrays, complex numbers, and structures can be made more uniform and more Lispy. Just make it `#(`*type* `...` *data* `...)` for all of them. Thus, `#(V 1 2 3)` for a vector, `#(C 1 5)` for a complex number, `#(BIT 1 0 1)` as an alternate syntax for bit-vectors, `#(FOO (A B) C)` for the structure `foo`, and `#(A (3 4)` *...12 things...*`)` for an array. This has the additional advantage that EMACS can parse them all correctly. Parsing `#C(1 2)` as one object is nearly impossible to implement. It should be legal to spell out the names `vector`, `array`, `complex` as well as to use `V`, `A` and `C`. This syntax is no more complicated than the NIL syntax in any case except the vector, for which two characters are added.

SEF: Not bad, but it does screw up the nice syntax for the most common case, vectors of type `t`. Still, this might get around some of the hair about arrays and `@`.

ALAN: [207.5] Mumble! This [#A] is *useless* unless you are using arrays for structures, but you should be using vectors then. I vote not to specify how arrays print. ("#<ARRAY ...>" is okay.)

HIC: #A is awful!

GLS: Of course it is. I was hoping someone would suggest something much better.

ALAN: [209.4] Flush #S for the same reason you should flush #A.

GLS: Well, what is proposed in the draft COMMON LISP manual isn't great. *However*, I think it is very important that vectors, arrays, and structures have first-class citizenship in the language if they are to become more widely used and more useful. This includes a readable print syntax. One advantage is that it provides a portable and standard way to send such objects via network or tape to some other implementation. Now perhaps we want alternative print methods, such as #<...> syntax, and we may not even want #A or #S to be the normal default print method. But such standard methods must exist.

MOON: #A provides no notation for leaders, and is otherwise non-extensible.

GLS: Fine; let us extend it.

** Alternatives:

A. Flush #A and #S syntax.

B. Keep #A and #S syntax as defined.

C. Keep #A and #S, but clean them up, flushing the stupid "@" characters, making them EMACS-parsable, and extending them to handle array leaders.

C?

** Responses

MOON: X	RMS: C!!!	SEF: AC!!	GINDER: C!	DM: X!
DLW: C!!	HIC: C	WLS: C!	CHIRON: C!!	
ALAN: A!	GLS: C!	RPG: C!!!	DILL: C!!!	

RMS: Don't forget to change #(...) and #C also. Arrays should not by default always print as #(A...), especially if they are large, but it should be available.

MOON: Use "#.", that's what it's there for.

ALAN: [209.2] See! You forgot #B. I *knew* it was unused and unknown!

MOON: It sounds like #S works regardless of the name of the constructor-macro. If so, say so more explicitly. If not, also say so!

GLS: This is an awkwardness in the documentation; I'll fix it.

***** Issue 198: Circular list syntax**

??? Query: [209.6] Resolve when the labels get reset [for #: and ## circular-list syntax].

SEF: Add a warning about #: and ##: Do not use!

ALAN: I have seen a *new* package proposal (by MMCM) that has a proposed use for #:. I prefer #= anyway for this. Is there really a need for circular list printing? Or is this just for reading? What about forward references?

GLS: I think it would be useful to have a circular-list printer. INTERLISP has found it useful. I think I have a way to layer it on top of the standard printer, and even to make it interact properly with a pretty-printer.

MOON: Lisp Machine LISP will be using *name#:name* as a package syntax. Doesn't conflict since one syntax is inside a token and the other is [left-]delimited.

GLS: Yech!

ALAN: Lisp Machine LISP ## was flushed some time ago.

MOON: Lisp Machine LISP ## has been flushed long ago.

** Suggestion: Retain circular-list syntax. Resolve the scope issue somehow. **Y**

** Responses

MOON: Y	RMS: Y!	SEF: Y	GINDER: Y!	DM: Y!
DLW: N!	HIC: Y	WLS: Y!	CHIRON: N	
ALAN: X!	GLS: Y!!	RPG: Y!!!	DILL: Y	

ALAN: I have no objections to circular list printing. I do object to using up #: for that purpose. It is *clear* that whatever #: eventually should mean (if anything) is something package-related. I much prefer that #= be gobbled down for this purpose.

***** Issue 199: Testing of `features`**

MOON: [210.1] `features` continues the campaign to use up all English words for system use. Call it something else. GJC (George J. Carrette) and RWK (Robert W. Kerns) have a proposed feature-test system which looks like a step in the right direction although there are some small problems with it. Documentation on the current state can be found in `MC:VAXMAX;FEATUR DOC`. I haven't finished looking at this but I think it is probably the right thing.

GLS: Here follows a slightly edited copy of the file `MC:VAXMAX;FEATUR DOC`.

Macros and functions for supporting conditionalized programs and cross compilation.

A <feature-spec> is a form which is one of the following:

symbol	- feature name
`(or` *spec1 spec2* `...)`	- spec disjunction

`(and` *spec1 spec2 ...*`)`	- spec conjunction
`(not` *spec*`)`	- spec negation
`(name` *spec*`)`	- same as `(featurep 'spec 'name)`

The function `featurep` evaluates a <feature-spec> in the context of the <feature-set-name> which is its second argument, defaulty the value of `target-features`. A <feature-set-name> is defined with `def-feature-set`.

Note: The read-macro "#+" calls the function `featurep`.

Convenient special forms to call in conditionalizing macros:

```
(when-feature
  (featurespec1 . clause1)
  (featurespec2 . clause2)
  ...)
```

Executes the first clause which corresponds to a feature match. A *featurespec* of `t` here means always.

```
(when-features
   (featurespec1 . clause1)
   (featurespec2 . clause2)
   ...)
```

Executes all clauses which corresponds to a feature match. A *featurespec* of `t` here means always.

Summary of user interface functions for feature sets:

```
(featurep exp &optional feature-set-name)
(nofeaturep exp &optional feature-set-name)
(set-up-feature-set name target-name
        features nofeatures query-mode)
(indirect-feature-set from-name to-name)
(set-feature exp &optional feature-set-name)
        ;This makes (featurep exp) => t
(set-nofeature exp &optional feature-set-name)
(set-feature-unknown exp &optional feature-set-name)
```

Note: If a feature *exp* is unknown then what happens to `(featurep` *exp*`)` depends on the *query-mode* of the feature-set in question, which may be either `:error`, `:query`, `t`, or `()`.

```
(set-feature-query-mode feature-set-name mode)
(copy-feature-set from-feature-set-name to-feature-set-name)
```

`def-feature-set` is a convenience macro; all it does is parse its arguments to set up a call to `set-up-feature-set`. Here are some examples of its use:

```
(def-feature-set maclisp
  :features   (maclisp for-maclisp hunk sendi bignum
                       fasload paging roman)
  :nofeatures (nil for-nil lispm for-lispm franz vax
                   unix vms))

(def-feature-set lispm
  :features   (lispm for-lispm bignum paging string)
  :nofeatures (maclisp for-maclisp nil for-nil franz sfa
                       noldmsg vector vax unix vms multics
                       pdp10 its tops-20 tops-10))
```

System interface notes:

The variable `target-features` is globally bound to the name `local`. It is the features to assume objects read are for. `local` is a feature set name which indirects to either `maclisp`, `lispm`, or whatever. A cross-compiler would bind `target-features` to the system targeted for. Also, the evaluator would make sure to set `target-features` to `local` before doing a macroexpansion.

Example usage:

```
(eval-when (eval compile)
  (load (when-feature ((local ITS) '((kip) hhead ))
                      ((local VMS) '((rlb h) hhead)))))

(defmacro frobcall (f &rest l)
  (when-feature
     (maclisp '(subrcall nil ,f ,@l))
     (nil     '(normalized-subroutine-call ,f ,@l))
     (lispm   '(funcall ,f ,@l))
     (t       '(funcall ,f ,@l))))

;; Generate call to a general procedure
(if (when-feature ((local Maclisp)
                   (get (car form) 'fsubr))
                  (T ()))
    (barf "Calling a Maclisp FSUBR ~s" form))

(when-feature
   ((local Maclisp) (get f 'expr))
   ((local NIL)
    (interpreter-lambda-definition f))
   (t ()))
```

(End of the copy of the file `MC:VAXMAX;FEATUR DOC`.)

** **Suggestion:** Adopt the mechanism detailed in the `VAXMAX` file.

** Responses

MOON: X	RMS: N!!!	SEF: N!	
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y!!!
		RPG: Y!!!	DILL: Y

RMS: It is a real loss how some much of what is present in the rest of the system is duplicated in feature-test reader prefixes (or `format`, or `loop`, or any other special-purpose feature). Things such as `and`, `or`, and `not` should not be duplicated. Instead, provide a form to test for a feature, and let people combine them with the LISP functions `and`, `or`, and `not` (or anything else) and use the result as the thing to test.

MOON: Use this but clean it up extensively. Defer until someone creates a cleaned-up proposal.

SEF: It seems a bit too complex for me. Unless problems can be pointed out, I favor a simple `features` list. (I don't care if that is its name.)

19.1.3. The Readtable

***** Issue 200: Defining character syntaxes by copying**

DLW: [212.9] [It is more intuitive for the user simply to copy some standard character...] I agree.

MOON: But non-extensible if *new* syntaxes are made; there are application programs which do so.

** Suggestion: Provide a more extensible definition system? **Y?**

** Responses

MOON: Y!		SEF: N		DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y	
ALAN: Y!	GLS: N	RPG: Y!!!	DILL: Y	

***** Issue 201: Dispatch macro character setup**

ALAN: [214.3] `set-dispatch-macro-character` is pretty random.

RMS: [214.3] Dispatch macros seem to be unnecessary. An ordinary macro which looks in a vector to find the function to call would do the job just as well. To enable the user to alter subcases of the built-in # macro, just advertise the name of the vector it uses. This is just as simple to use in practice, and avoids adding any conceptual complexity.

SEF: Well, it bothers me to have a function whose only purpose is to define the # macro. I'm not sure it's as simple as RMS thinks, however.

** Suggestion: Flush `set-dispatch-macro-character`?

** Responses

MOON: Y	RMS: Y!	SEF: N		DM: Y!
DLW: Y				
ALAN: Y!	GLS: N	RPG: N!!	DILL: Y	

MOON: This assumes the `#` `characters` are the same in all readtables that have `#` `at` all. This may be okay, but should be thought about. (Of course one can always write a new # macro that duplicates the code in the old one except for the variable name).

ALAN: If the "performance problems" of general LISP readers refers to the Lisp Machine LISP reader, it is dead wrong. I have nothing against simple readers, but the performance problems one experiences when

calling (`read`) in Lisp Machine LISP have little to do with the regular-expression tokenizer.

MOON: Sounds like misinformation.

GLS: I was somewhat wedged. My apologies.

19.1.4. What the `print` Function Produces

***** Issue 202: Must everything print?**

ALAN: [19.8] Must *everything* print???

GLS: Yes.

MOON: I don't think structures and arrays ought to spill their guts by default (any more than symbols print their property lists).

DLW: I agree. This would be quite unwieldy in the default case.

HIC: Third.

ALAN: I vote not to specify how arrays print. ("`#<ARRAY ...>`" is okay.)

SEF: For the human interface, I tend to agree with this. I suppose it is useful, though, for dumping things and reading them back in, if that can be guaranteed to work in all cases. Maybe a switch associated with the stream (in the spirit of `prinlevel` and `prinlength`) to indicate if such gut-spilling is desirable.

MOON: RG [Richard Greenblatt] suggests that slashification be generalized. There are at least four useful levels: no slashification (`princ`), slashing of special characters (`prin1`), printing of special objects so that they read in as something `equal` (use `#.`<LISP form> rather than a `#<...>` syntax), and use of anaphora syntax so that circular and reentrant lists read in as a tree with the same structure (this is not the default because it is expensive). Perhaps one of the print functions should take an optional argument which is a slashification keyword, and also the slashification mode that gets passed down to print handlers would be a keyword rather than just `t` or `()`. This seems like a good idea to me. I'm not sure if the possible use of a "grinding" top-level printer is also a slashification mode or related.

** Suggestion: Provide some kind of convenient handle on multiple printing methods.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y	WLS: Y!	CHIRON: Y!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y	

19.2. Input Functions

19.2.1. Input from ASCII Streams

***** Issue 203: Allowing `t` as a synonym for `terminal-io`**

MOON: [215.0] [Allowing `t` as a stream to mean the value of `terminal-io` provides some semblance of MACLISP compatibility.] Hardly anything else here is compatible with MACLISP; why bother? (Admittedly Lisp Machine LISP does it too.)

**** Suggestion:** Retain specification that `t` as a stream means the value of the global variable `terminal-io`.

Y

**** Responses**

MOON: N	RMS: Y!	SEF: Y!		DM: Y!
DLW: Y!	HIC: Y		CHIRON: N!	
ALAN: Y!	GLS: Y!	RPG: Y		

SEF: Rename *eof-option* to be *eof-value* everywhere, to make it clear that it is always evaluated, and produces a simple value to be returned at EOF [as contrasted with a COBOL sort of `AT END` clause which is executed iff EOF is encountered].

***** Issue 204: Global variable `read-preserve-delimiters`**

RMS: [215.7] The distinction controlled by the `read-preserve-delimiters` variable is a necessary one, but I don't think that a special variable is the best way to make it. Here is the reason: suppose we have a recursive call to `read` (such as for a macro character); how often will this call to `read` want to use the same value of `read-preserve-delimiters` that was used by the outer call to `read`? I think the answer is: *never*. Each place a call to `read` appears in the code, either it should always preserve delimiters or never preserve them. To specify such a choice, either an explicit optional argument or a choice of function names is best. In Lisp Machine LISP, there are `read` and `read-for-top-level`. `read` preserves delimiters; the other doesn't. I think `read` should ignore the delimiters and the other function should preserve them. `recursive-read`? `read-subexpression`?

SEF: Excellent suggestion.

**** Suggestion:** Eliminate the variable `read-preserve-delimiters`. Specify that `read` does *not* preserve delimiters (more precisely, discards a *whitespace* character which delimited a token). Have a new function which does a `read` operation preserving delimiters.

Y?

** Responses

MOON: N	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!!	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y	

MOON: RMS's last sentence is incorrect; `read` discards delimiters by default. It was put in as a variable rather than as a separate function by agreement with JONL, who was going to put the variable into MACLISP and didn't want another function. Since JONL never held up his end of the bargain, I don't particularly care if the variable is flushed and replaced by a separate function. Probably no one uses the variable anyway.

MOON: [Several paragraphs by MOON follow.] RMS's argument that each place that calls `read` knows whether or not it wants to preserve delimiters is specious. Consider the single-quote macro; when it calls `read` to read the quoted expression, it doesn't know or care whether a white-space delimiter character after the expression should be preserved; this is entirely dependent on the caller of the outer call to `read`. This applies to any reader macro that consists of a prefix followed by an S-expression.

So I think that the `read-preserve-delimiters` special variable should remain.

I remember now that this variable was put in when we fixed `read` to work right instead of wrong; the variable was put in so that things could be changed back to the old way in case anyone cared. At the same time MACLISP was supposed to get the variable so that it could change to the new way if desired. It might be acceptable for COMMON LISP to specify that `read` never preserves delimiters, and not have a variable, if there was an `ratom`-type function for applications like the UNIX pathname example in the draft COMMON LISP manual (a bogus example if I ever saw one; what if there is a ' mark in the pathname?)

*** Issue 205: Useless *eof-option* to `read-delimited-list`

MOON: [216.6] I don't think an *eof-option* argument to `read-delimited-list` makes sense, since EOF will *always* be in the middle of an object.

GLS: I believe you are right. However, might not the error handler decide to do something with the *eof-option*? It depends on the details of the error interface for this error, I suppose.

** Suggestion: Eliminate the *eof-option* argument to `read-delimited-list`.

Y?

** Responses

MOON: Y		SEF: N	GINDER: Y!
DLW: Y	HIC: Y		CHIRON: Y
	GLS: Y	RPG: N!	

***** Issue 206: Eliminate `read-delimited-list`?**

RMS: `read-delimited-list` sounds trivial to write using `tyipeek`, so I think it is not worth having as a function name.

SEF: Leave it in. It makes code clearer, and lots of people don't like to hack low-level I/O, since so much strangeness seems to be going on.

RMS: I am not strongly against `read-delimited-list`.

**** Suggestion:** Eliminate `read-delimited-list`?

N

**** Responses**

MOON: N		SEF: N!	GINDER: N!
DLW: N	HIC: N		CHIRON: N
ALAN: N!	GLS: N!!	RPG: N!!	

***** Issue 207: Can one `untyi` anything, or many things?**

??? Query: [217.6] This from the Lisp Machine Lisp Manual: "Note that you are only allowed to `untyi` one character before doing a `tyi`, and you aren't allowed to `untyi` a different character than [*sic*] the last character you read from the stream. Some streams implement `untyi` by saving the character, while others implement it by backing up the pointer to a buffer." Opinions? Current trend is to opt for generality rather than hacks.

SEF: Be general.

ALAN: Well *foo*! Who cares? If you want to look more than one character ahead you can buffer the characters yourself! Why make every stream in the world implement this totally hairy protocol for the benefit of virtually nobody? The simply `untyi` operation is trivial to provide and useful to `read` mainly; we don't need to hair things up here.

MOON: This is to allow freedom of implementation. `untyi` is *not* a mechanism to put input into a stream through the side door.

GLS: Well, if you are only allowed to back up one character, and to back up only the character you just read, then maybe `untyi` shouldn't have to take the character as an argument.

**** Alternatives:**

A. Permit one to `untyi` any character, and arbitrarily many characters.

B. Permit one to `untyi` only characters just read, but arbitrarily many of them.

C. Permit one to `untyi` any character, but only one.

D. Permit one to `untyi` only the character just read, only once.

E. Permit one to `untyi` only the character just read, only once, and `untyi` doesn't take the character as an argument; it just knows how to back up over the last character read.

** Responses

MOON: D!!!!	RMS: DE!	SEF: AE!!		DM: E!
DLW: D!!!	HIC: D!		CHIRON: E!!	
ALAN: D!	GLS: E	RPG: A!	DILL: B!	

RMS: D or E. E if it is not too much pain for all streams to remember the last character that they fed you (since some will not naturally do so).

*** Issue 208: Action of `inch-no-hang` at end of file

MOON: [218.6] Do `inch-no-hang` and `tyi-no-hang` error out at EOF if no *eof-option* is supplied?

** Suggestion: These functions signal an error if no *eof-option* argument is given.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y	
ALAN: Y!	GLS: Y!!	RPG: Y!		

*** Issue 209: Let `read-from-string` take *start/end* arguments?

DLW: [218.8] Shouldn't `read-from-string` take arguments *start* and *end* like every other string function?

MOON: This is my fault, but that's no excuse.

** Suggestion: Let `read-from-string` take *start/end* specifiers; also, put them the *eof-option* argument, because otherwise there is no way to specify a *start/end* pair explicitly and get the effect of omitting the *eof-option* argument.

Y

** Responses

MOON: Y		SEF: Y!!		DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!	
ALAN: Y!	GLS: Y!	RPG: Y!	DILL: Y	

***** Issue 210: New function `parse-number`**

MOON: There should also be a function `parse-number` taking arguments *string* `&optional` *start end radix* which returns a number, or ( ) if there is no number there, and a second value which is the next character position (index of delimiter). Should there be a flavor of this which errors unless the specified range is exactly a number, rather than returning ( ) or stopping at a delimiter?

**** Suggestion:** Add a new function `parse-number` as described above.

Y?

**** Responses**

MOON: Y	RMS: N!	SEF: Y!!		DM: Y!
DLW: Y!!	HIC: Y		CHIRON: X	
ALAN: Y!		RPG: Y!	DILL: N	

RMS: I'm not convinced this function is needed very much. Sticking information into a string and getting it out again by anything other than `read` is a thing needed only in the insides of badly designed network file access protocols.

CHIRON: Should have a complement to `format` called `scan` or something, similar to `printf/scanf` of C. [GLS: `scanf` in C does fixed-format input processing.]

***** Issue 211: End of string in the middle of an object**

??? Query: [218.9] In Lisp Machine Lisp, what happens if end-of-string occurs in the middle of an object?

MOON: An error, of course.

**** Suggestion:** Let hitting the end of the string behave exactly like encountering end-of-file.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!!	DILL: Y!!!	

19.2.2. Input from Binary Streams

19.2.3. Input Editing

19.3. Output Functions

19.3.1. Output to ASCII Streams

***** Issue 212: Add the `*nopoint` switch?**

MOON: Is there a `*nopoint`, or is it gone? (Seems desirable to have when back-porting.)

**** Suggestion:** Add `*nopoint` as in MACLISP.

Y

**** Responses**

MOON: Y!!!		SEF: Y!!		DM: N!
DLW: Y!!	HIC: Y		CHIRON: X	
ALAN: Y!		RPG: Y!!	DILL: N!!	

MOON: There should also be a switch, probably the same variable, which is a number such that if `base` is not equal to it, numbers print with a leading #R prefix. The default value should be `10`.

CHIRON: Extend `*nopoint` as it is currently implemented to mean "[do not] show `base`".

***** Issue 213: Result returned by `print` and friends**

HIC: I want `print` and friends to return what they print. What is the rationale for the change?

GLS: This was an attempt to be compatible with MACLISP.

MOON: The change to `print` is unnecessary and makes debugging more difficult. Also note that the reasons for doing this in MACLISP were completely bogus, since the problem with pdlnmk'ing could have very easily been solved in another way. Time efficiency is clearly not an issue when you have just called `print`; the only issue is space efficiency.

GLS: Right: the compiler could simply have arranged to compile calls to a `print` that returns its argument as a call to a print that doesn't, and then using the argument itself as the result value as if it were a simple local variable reference. This would handle pdl numbers correctly. But it is almost certainly not worth the trouble anyway, as MOON points out.

**** Suggestion:** Let `print` and friends return the object printed.

Y

**** Responses**

MOON: Y	RMS: Y!	SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y!!!		CHIRON: Y!!!	
ALAN: Y!	GLS: Y!!	RPG: Y!!	DILL: Y!!!	

*** Issue 214: Flush `string-out` and `line-out`?

MOON: `string-out` and `line-out` seem totally unnecessary.

GLS: They may be very fast in certain implementations (possibly "open-codable" as supervisor requests).

** Suggestion: Flush `string-out` and `line-out`?

** Responses

MOON: Y	RMS: Y!	SEF: Y	GINDER: N!	DM: Y!
DLW: N			CHIRON: X	
	GLS: N!!	RPG: N!	DILL: N	

RMS: Even if they are open-codable, the extra overhead of the stream call is not very bad on these operations.

CHIRON: Keep `string-out` for reasons you specified.

19.3.2. Output to Binary Streams

19.4. Formatted Output

*** Issue 215: Behavior of `format` ~P on floating-point arguments

??? Query: [223.7] Should this [~P in `format`] work for floating-point numbers also?

MOON: That is, should it use `equal` or `equalp`? Usually one says "1.0 dogs" rather than "1.0 dog". But I don't care.

GLS: I suppose you're right.

** Suggestion: Let ~P produce an "s" if the argument is floating-point.

Y

** Responses

MOON: Y		SEF: Y		DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!!	
ALAN: Y!	GLS: Y	RPG: Y!		

*** Issue 216: Floating-point output in `format`

??? Query: [224.0] Is this [~F in `format`] really what we want?

??? Query: [224.3] Is this [~E in `format`] the right thing? Study PL/I, FORTRAN.

MOON: This one is admittedly random.

??? Query: [229.5] Unfortunatly [*sic*], the ~F command as defined above isn't really flexible enough [for formatting floating-point numbers, as with `$*****2.59`]?

MOON: ~$.

ALAN: There is also ~$ which does the line-them-up-in-columns type of formatting that PL/I and FORTRAN are good at. I think ~F and ~E are more human-readable myself.

GLS: There isn't anything yet that accomplishes what the FORTRAN G format specifier does.

** Suggestion: Defer this issue for now, pending a specific proposal for improvement.

Y

** Responses

MOON: Y		SEF: Y	GINDER: Y!	DM: Y!
DLW: Y!	HIC: Y		CHIRON: Y	
ALAN: Y!	GLS: Y	RPG: Y!!		

*** Issue 217: Mnemonic for ~X in `format`

SEF: [225.6] "FORTRAN X *format*" is the best mnemonic you could come up with for ~X?

GLS: Well, why *was* "X" used for this? Was it borrowed from MULTICS `ioa`, which in turn probably borrowed from PL/I, which borrowed from FORTRAN? Actually, I would like most to let ~X mean "hexadecimal", by analogy with #X, and use ~@T to replace ~X (the @ flag meaning "relative" tab).

** Alternatives:

A. Incompatibly change ~X to mean hexadecimal printout, and introduce ~@T to mean relative tab to replace the old ~X.

B. Leave `format` alone, but remove the reference to FORTRAN X format.

C. Status quo: leave document and implementation alone.

A?

** Responses

MOON: A		SEF: A!	GINDER: A!	DM: A!
DLW: A!	HIC: C		CHIRON: A!	
ALAN: B!	GLS: A!	RPG: C!	DILL: A	

*** Issue 218: Should the `format` iteration constructs be retained?

RMS: [227.5] The hairy `format` iterations should not be part of COMMON LISP. `format` is perspicuous when only the simple format specs are used, but for things that involve iterations or conditionals it is better to use the Lisp Machine LISP formatted output functions together with LISP control structure.

SEF: I tend to agree with this. I think we let in too much of TECO here.

GLS: Well, I think there is a fuzzy line dividing not enough hair from too much, and that the line does not fall neatly between constructs. For example, if you have ~{ and ~↑, then you can almost directly transcribe any FORTRAN FORMAT statement (you need ~↑ to express the FORTRAN 77 ":" operator). I admit that ~[is too hairy, but I use ~:[and ~@[a lot.

** Suggestion: Remove some of the hairier `format` constructs.

** Responses

MOON: N	RMS: Y!	SEF: Y		DM: Y!
DLW: Y	HIC: N		CHIRON: N!	
ALAN: N!	GLS: N!	RPG: N!!	DILL: Y	

RMS: Being able to transcribe a FORTRAN FORMAT statement is not an important goal.

DM: `format` seems too big. It reads like TECO. How much of it is "really" used in Lisp Machine LISP and elsewhere right now? What isn't, flush.

*** Issue 219: What should `format` ~C (no flags) do?

ALAN: [224.5] I don't think the description of ~C (without flags) is right. I know that it now does `tyo`.

MOON: In any case, ~C with no bits simply outputs the character a la `tyo`.

DLW: Actually, I strongly hope to flush the stupid Greek letters and do exactly what is suggested here!

MOON: The Greek letters in the `format` ~C should go away and "`c-`" should be the abbreviated prefix for "control".

** Alternatives:

A. Require `format` ~C to output control bits as `c-`, `m-`, etc.

B. Require `format` ~C to simply `tyo` its argument.

C. Require `format` ~C to simply `tyo` its argument, but also require `tyo` to handle control bits by outputting `c-`, `m-`, etc.

D. Status quo: let the precise treatment be implementation-dependent.

** Responses

MOON: A	RMS: A!	SEF: A!		DM: C!
DLW: C!	HIC: B		CHIRON: A!!!	
ALAN: B!	GLS: C	RPG: D!		

ALAN: There seems to be some confusion here. The problem is that there needs to be a way to take an arbitrary character and pass it *untouched* through `format` to the stream you are outputting to. This seems to

be the kind of basic operation that `format` should be able to handle no matter what other bells and whistles it has. There is also a need for human-readable character printing, but we shouldn't let that need cut off *all* the escapes we have. I think that

```
(format t "foo~Cbar" #\return)
```

should print:

```
foo
bar
```

To this end it is silly to have `~C` prefix the characters with α or `C-` or `Control-`. Characters are for printing, so `format` should be able to do just that!

GLS: But all three of choices A, B, and C above would have that effect. The definition of standard characters implies that `#\return` does not have its Control bit set (despite the fact that its internal representation might happen to be an ASCII "control" character).

***** Issue 220: Control list in place of string for `format`**

MOON: I believe `format` control lists have been flushed, but a list means simply output the string in its *car* without looking for "~". This can be passed to programs which expect a format string and arguments, to output that string without going through `format`. (An alternative might be that "`~.`" means to output the rest of the string without looking at it.)

GLS: If you want to pass arguments for `format` and just want to pass along a string to print verbatim, why can't you just pass `"~A"` as the control string, and the string as an argument?

** Alternatives:

A. Require `format` to take a string as the control argument.

B. Permit a list whose *car* is a string, and output that string verbatim.

A?

** Responses

MOON: AB	RMS: B!!!	SEF: A!	DM: A!
DLW: A!!	HIC: B	CHIRON: A!!!	
ALAN: A!	GLS: A	RPG: A!	

RMS: I did this because there were problems with passing `"~A"` as the control argument and the real thing as the data. The problems come in connection with error signalling, where the control argument is considered "for human eyes only" and not examined except by `format`, whereas the things that follow have standardized meanings which relate to the particular error condition. So there is no freedom to pass the string to print as the argument.

MOON: The reason you can't use `"~A"` and an extra argument is because of `ferror`, which already defines precisely what each of its arguments means for particular conditions.

19.5. Querying the User

***** Issue 221: Help characters to `fquery`**

MOON: [233.1] Can `fquery` possibly take an implementation-defined HELP key, in addition to "?"?

GLS: I don't see why not; the draft simply specifies that "?" *must* give help.

** Suggestion: Specify that the effect of non-standard characters when input to `fquery` (and, more generally, other input functions) may be implementation-dependent.

Y

** Responses

MOON: Y		SEF: Y!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y!!!	
ALAN: Y!	GLS: Y	RPG: Y!		

***** Issue 222: Meaning of the term "beep"**

SEF: [233.2] Say that what it means to "beep" is implementation-dependent (might be a screen flash instead of a noise, for example).

GLS: Maybe the `:beep` option should be renamed `:attention`?

** Alternatives:

A. Rename the `:beep` keyword to be `:attention`, and clarify the documentation.

B. Retain the `:beep` keyword, but clarify the documentation.

** Responses

MOON: B		SEF: B	GINDER: A!	DM: A!
DLW: B!	HIC: A		CHIRON: A!	
ALAN: B!	GLS: A!	RPG: A!!	DILL: A	

19.6. Streams

SEF: Move this section to the front of the chapter.

19.6.1. Standard Streams

***** Issue 223: Are bidirectional streams needed?**

SEF: Why are bidirectional streams needed? Why not just have `terminal-input` and `terminal-output` instead of `terminal-io`?

**** Suggestion:** Flush bidirectional streams.

N

**** Responses**

MOON: N!!!!	RMS: N!	SEF: Y!		DM: Y!
DLW: N!!	HIC: N!!		CHIRON: N!!!	
ALAN: N!		RPG: N!!!		

RMS: Lisp Machine LISP seems to provide some support now for bidirectional streams. One should not require any implementation to offer them for any particular purpose, since some operating systems may interfere. But don't prevent people from creating them.

19.6.2. Creating New Streams

SEF: Need to have a way to construct a stream from a given function or closure.

> **??? Query:** [235.9] In Lisp Machine LISP this is called `make-syn-stream`. The documentor found it necessary to explain that "syn" meant "synonym"; it cvertainly isn't obvious. The abbreviation "syn" could be mistaken for any number of other things, such as "synchronous" or "syntactic" or "synthetic" ... Here this confusion is eliminated.

MOON: Okay. Brain damage accidentally inherited from MULTICS.

***** Issue 224: Add *start/end* arguments to `make-string-input-stream`?**

MOON: [236.6] Let `make-string-input-stream` take arguments *start* and *end*.

**** Suggestion:** Add *start/end* arguments to `make-string-input-stream`.

Y

**** Responses**

MOON: Y		SEF: Y!		DM: Y!
DLW: Y!!	HIC: Y			
ALAN: Y!	GLS: Y	RPG: Y!	DILL: Y	

*** Issue 225: Add `with-input-from-string` and `with-output-to-string`?

MOON: What happened to the structured interfaces to these (`with-input-from-string`, etc.)?

GLS: My only hesitation was the restriction in Lisp Machine LISP that only one of these be outstanding at a time. This strikes me as inelegance in the name of efficiency.

** Alternatives:

A. Provide `with-input-from-string` and friends as in Lisp Machine LISP, with the restriction that they cannot nest (also as in Lisp Machine LISP).

B. Provide `with-input-from-string` and friends as in Lisp Machine LISP, but with no nesting restriction.

C. Do not provide these constructs.

B

** Responses

MOON: B	RMS: B!	SEF: C!	GINDER: B!	DM: C!
DLW: B!!	HIC: B		CHIRON: B!!	
ALAN: B!	GLS: B!!	RPG: B!	DILL: C	

MOON: The restriction will go away when there is lexical scoping.

19.6.3. Operations on Streams

19.7. File System Interface

*** Issue 226: Should the COMMON LISP manual specify a file system interface?

MOON: This is somewhat outside the domain of a language core.

GLS: I disagree. The more commonality we can achieve, the more code is portable.

** Suggestion: Remove this section from the COMMON LISP manual.

N?

** Responses

MOON: Y	RMS: N!	SEF: N!!		DM: N!
DLW: N!	HIC: X		CHIRON: N!!!	
ALAN: N!	GLS: N!!	RPG: N!!!		

MOON: Defer this for now.

***** Issue 227: Should files be sequences?**

MOON: For some reason you forgot to make files be sequences.

**** Suggestion:** Let files be sequences, with all the rights and privileges pertaining thereto.

**** Responses**

MOON: Y	RMS: N!!!	SEF: N!!!!	GINDER: N!	DM: X!
DLW: Y	HIC: X		CHIRON: N	
ALAN: Y!	GLS: N!!!	RPG: Y!!		

RMS: I think this analogy will be very hard to carry through, and will require terrible hair in the generic sequence functions. if what you mean is that a file is a sequence of the characters of data it contains. I also think it will not be very useful for any of the ways we typically use files. To make it useful *ever*, you would need to define new kinds of self-defining structures for files to be sequences of.

MOON: Files, or streams open to files? Do sequence operations make new files?

DM: Does it really make sense to perform *all* the sequence operations on files? Do random operating systems always give the support you need for, say, `nreverse` or `nconc`? It seems a nice idea, though.

19.7.1. File Names

***** Issue 228: Pathnames and namelists**

MOON: What to do about this crockish excuse for a pathname system? It does not meet the needs of Lisp Machine LISP. Do we have to have two coexisting pathname systems, one for people to use and one for compatibility? Lisp Machine LISP could easily understand namelists (it already understands the MACLISP version of them). Likewise a `namelist` function that translated Lisp Machine LISP pathnames into namelists. The problem is then only with functions with the same name that return pathnames in Lisp Machine LISP and namelists in COMMON LISP, such as `truename`. For compatibility a user must either call `namelist` on the result of these functions, or COMMON LISP could provide a complete set of functions for reading, setting, and defaulting components of a namelist, which could work on pathnames just as well. (The set in the draft is not complete.)

GLS: I avoided the Lisp Machine LISP pathname system primarily because, pathnames being flavor instances, it was heavily message- and flavor-based, and the consensus seems to be not to put message-passing into COMMON LISP yet. A secondary reason was that I thought it awfully complicated, though admittedly what's in the draft COMMON LISP manual isn't a whole lot better. I do think that the Lisp Machine LISP pathname stuff forces even more of a fixed view of a file system on the user and host environment than the draft COMMON LISP suggestion. Can Lisp Machine LISP encompass MULTICS and UNIX smoothly, for example? I'm not sure. Perhaps the operational essence of the Lisp Machine LISP pathname stuff could be extracted and made available with a function-call instead of message-passing interface. What are the ramifications of

the fact that Lisp Machine LISP pathnames are interned? Can this get one into trouble on some odd file system?

GLS: What are in fact the needs of Lisp Machine LISP for a file system interface? Maybe this needs to be discussed more explicitly. I'm sure I do not understand all of the needs and consequences. What is in the COMMON LISP draft is simply the MACLISP namelist stuff, slightly tweaked to add an extra level of hierarchy intended to handle networking. It certainly is not inspired.

** Alternatives:

A. Adopt the Lisp Machine LISP pathname system.

B. Status quo: use namelists as specified in the draft COMMON LISP manual.

C. Adopt *both* pathnames and namelists.

D. Defer this issue for more discussion.

D

** Responses

MOON: D		SEF: B!	GINDER: D!	DM: D!
DLW: A	HIC: D!!		CHIRON: D	
ALAN: D!	GLS: D	RPG: D!!	DILL: D	

DM: Yes, COMMON LISP should specify some common file naming scheme. But if it is to be kept running on lots of bizarre machines, it should be kept very simple and stupid. Let the implementation extend it however it sees fit, but the "required" one should be very simple. This will encourage portable code, so long as the author sticks to the strictly common stuff, while allowing it to actually make sense on nearly all machines.

*** Issue 229: Parse termination in `parse-namestring`

MOON: [239.5] Why not leave termination of parsing in `parse-namestring` to the normal string-processing functions?

GLS: Well, it seemed like something you might often want. Also, it might be that it is hard to find the end of a file name without actually parsing it (consider system-dependent quoting characetrs, such as control-V for TOPS-20).

** Suggestion: Flush the *break-characters* argument to `parse-namestring`?

N

** Responses

		SEF: N!	
DLW: N			CHIRON: N
ALAN: N!	GLS: N!!	RPG: N!	

19.7.2. Opening and Closing Files

MOON: [240.9] There are many keyword-taking functions that work this way (keywords sometimes followed by arguments and sometimes not) in Lisp Machine LISP. The reason for this is so that you can simply pass the keywords as arguments, for example `(open` *name* `':read ':byte-size` *bs*`)`. Unfortunately this is damaged in the case of `open`, since it doesn't take a `&rest` argument. So I don't care what `open` does, but don't rule this out in general.

MOON: [241.7] [Regarding the `:echo` keyword to `open`:] Is it necessary to put echoing in in so many different places? It's an operating system function anyway.

GLS: The draft COMMON LISP manual contains bits of several theories of echoing in a confused way (sorry). However, some coherent method for controlling it should be devised.

??? Query: [241.8] In Lisp Machine LISP, `:probe` also implies `:fixnum`. Why??

MOON: It affects what the `:info` message returns. But it's accidental; would be okay to change.

MOON: [242.4] [While `with-open-file` tries to automatically close the stream on exit from the construct, for robustness it is helpful if the garbage collector can detect discarded streams and automatically close them.] That implies a lot of assumptions about the implementation.

GLS: Well, this is just an exhortation to implementors, not a requirement.

19.7.3. Renaming, Deleting, and Other Operations

MOON: Think of a better name for `faslp`.

GLS: Suggestions?

***** Issue 230: Names of file-inquiry functions**

MOON: Besides `file-creation-date` and `file-author`, what about the other fifty attributes of a file you might want to ask for?

GLS: Shall we add many functions? Doesn't seem any worse than fifty messages.

DLW: Why isn't `lengthf` called `file-length`? You didn't use `authorf`.

RMS: Functions such as `deletef`, `lengthf` should be renamed to `file-delete`, `file-length`.

** Suggestion: Add more file inquiry functions, and consistently name them `file-`*xxx*.

Y

** Responses

	RMS: Y!	SEF: N!!	GINDER: Y!	DM: X!
DLW: Y!	HIC: Y		CHIRON: Y!!	
	GLS: Y!	RPG: Y!!		

HIC: Comment about message passing: I'm really disturbed that message passing is not going into COMMON LISP. Though I suppose I'm somewhat biased, I feel that it's one of the few really useful and different ways of looking at interfaces that's been invented, and to have COMMON LISP ignore it in the core is a travesty. It means that COMMON LISP can't become a really reasonable portable subset. I'm not proposing any specific kind of inheritence scheme (Flavors, or classes), but I think that some consideration should be given to including at least the building blocks of a message passing system in the core.

DM: A general file attribute lookup function would be preferable. It's easier to extend in implementation-dependent ways, and doesn't add yet another fifty functions. Thus instead of `(file-author f)` one would have `(file-attribute f ':author)`.

SEF: How about `(file-info f ':author)`, etc. I agree that `file-`*x* is better than *x*`f`.

19.7.4. Loading Files

*** Issue 231: `load`, `fasload`, and `readfile`

??? Query: [244 6] There are several problems with the above specifications, which are essentially as in Lisp Machine LISP.

- The arguments for the three functions [`load`, `readfile`, and `fasload`] are not compatible; there is not even a subset relationship.
- The file name defaulting has been criticized as being "very M.I.T."; not all cultures prefer to maintain default file names in this way.
- There ought to be an option to print the results of each evaluation (and in the case of `fasload`, the name of each function as it is loaded). Another flavor of this option is to print a one-character blip every so often to signal progress. This is useful for debugging, for example to determine where in a load file an error is occurring.

These problems should be fixed.

ALAN: MACLISP doesn't do this [file name defaulting] any more, right? Flush it. It always annoyed me. GSB has a `load` function that takes a second argument of *keywords* specifying various things. This strikes me as best since there are an unlimited number of options that can be invented for loading.

MOON: The arguments to `load` (and `readfile` and `fasload`) should be keywords, with a special variable to contain the implementation- or user-specified default set of options. The `-internal` versions (which take an already-open stream as an argument) should exist as in Lisp Machine LISP; I don't care about the names (they could be the same functions with a keyword argument telling them their first argument is a stream, or they could check the type of the first argument (a slight incompatibility with MACLISP, which allows a stream to be used as a filename)). `qc-file` (`comfile`) should take its arguments in a compatible way. I wouldn't mind making certain incompatible changes in the defaults, e.g., whether it prints a message

by default.

MOON: Make all three functions take keyword arguments. Make file-name defaulting a mode. Make printing of results be a keyword. Printing a blip sounds monumentally useless. It wouldn't bother me to change the defaultsd about messages and so forth when they aren't overridden with keywords. There should be a special variable with a default set of keywords in it.

** Suggestion: Let `load` and friends take keywords after the first (file-name) argument. Provide a global variable which contains the defaults.

Y

** Responses

MOON: Y		SEF: Y!!	GINDER: Y!	DM: Y!
DLW: Y!!	HIC: Y		CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!		

*** Issue 232: Flush `readfile` and `fasload` functions?

RMS: Users don't need to be told about the functions `readfile` and `fasload`. They were useful once, but that was because `load` had not been defined yet. Now that there is `load`, the other two do not need to exist except as internal subroutines of `load`.

GLS: Under certain implementations, you need `readfile` or `fasload` to defeat the wrong choice by `load`. (There can be problems with not knowing whether the source or object file is the latest one.)

** Suggestion: Flush `readfile` and `fasload` functions?

** Responses

MOON: Y	RMS: Y!	SEF: Y!	GINDER: N!	DM: Y!
DLW: N	HIC: N		CHIRON: N	
ALAN: Y!	GLS: Y!	RPG: N!!	DILL: Y	

RMS: You do not need `readfile` or `fasload` just because `load` guesses wrong about whether you want the source or the fasl file. `load` does not guess unless you leave that part of the filename unspecified. If you specify the filename completely, `load` will (should) use the precise file you specify.

MOON: Re "wrong choice by `load`". `load` only chooses if the filename doesn't specify a file type, right?

GLS: RMS and MOON are right. I was wedged. Sorry.

19.7.5. Accessing Directories

Chapter 20

Errors

MOON: A lot of the stuff in this chapter reproduces Lisp Machine LISP stuff which [GLS: that?] we consider to be brain-damaged and are redesigning. I would say to leave out things marked with an *X* for now. We do understand the issues better now. [Nearly the entire chapter is *X*'d out.]

20.1. Signalling Conditions

20.2. Establishing Handlers

ALAN: [246.3] If at all possible you should think of a better description than "stack group".

20.3. Error Handlers

20.4. Signalling Errors

20.5. Break-points

20.6. Standard Condition Names

Chapter 21

The Compiler

*** Issue 233: `cl` and `disassemble`

MOON: Flush `cl`. Compiler maintainers can define it in their init files.

SEF: Well, sometimes random users want to see what things will compile well, but I basically agree that this has no place in the white pages.

SEF: If `cl` is retained, let it take an output stream as an optional second argument.

MOON: `disassemble` could compile a function first if given an interpreted function.

** Suggestion: Flush `cl`. Specify that if `disassemble` is pointed to an interpreted function it first compiles the function and then disassembles it.

Y

** Responses

MOON: Y	RMS: Y!	SEF: Y!!		DM: Y!
DLW: Y!	HIC: Y	WLS: Y!	CHIRON: Y	
ALAN: Y!	GLS: Y!	RPG: Y!		

Chapter 22

STORAG

Chapter 23

LOWLEV

Ballot for Common LISP Issues

For each issue, please mark "Y" or "N" for a single (yes/no) suggestion, or the letter for one of several alternatives. In either case, mark "X" if you agree with none of the choices, and attach commentary.

1. ____ 2. ____ 3. ____ 4. ____ 5. ____ 6. ____ 7. ____ 8. ____ 9. ____ 10. ____ 11. ____ 12. ____ 13. ____ 14. ____ 15. ____ 16. ____ 17. ____ 18. ____ 19. ____ 20. ____ 21. ____ 22. ____ 23. ____ 24. ____ 25. ____ 26. ____ 27. ____ 28. ____ 29. ____ 30. ____ 31. ____ 32. ____ 33. ____ 34. ____

35. ____ 36. ____ 37. ____ 38. ____ 39. ____ 40. ____ 41. ____ 42. ____ 43. ____ 44. ____ 45. ____ 46. ____ 47. ____ 48. ____ 49. ____ 50. ____ 51. ____ 52. ____ 53. ____ 54. ____ 55. ____ 56. ____ 57. ____ 58. ____ 59. ____ 60. ____ 61. ____ 62. ____ 63. ____ 64. ____ 65. ____ 66. ____ 67. ____ 68. ____

69. ____ 70. ____ 71. ____ 72. ____ 73. ____ 74. ____ 75. ____ 76. ____ 77. ____ 78. ____ 79. ____ 80. ____ 81. ____ 82. ____ 83. ____ 84. ____ 85. ____ 86. ____ 87. ____ 88. ____ 89. ____ 90. ____ 91. ____ 92. ____ 93. ____ 94. ____ 95. ____ 96. ____ 97. ____ 98. ____ 99. ____ 100. ____ 101. ____ 102. ____

103. ____ 104. ____ 105. ____ 106. ____ 107. ____ 108. ____ 109. ____ 110. ____ 111. ____ 112. ____ 113. ____ 114. ____ 115. ____ 116. ____ 117. ____ 118. ____ 119. ____ 120. ____ 121. ____ 122. ____ 123. ____ 124. ____ 125. ____ 126. ____ 127. ____ 128. ____ 129. ____ 130. ____ 131. ____ 132. ____ 133. ____ 134. ____ 135. ____ 136. ____

137. ____ 138. ____ 139. ____ 140. ____ 141. ____ 142. ____ 143. ____ 144. ____ 145. ____ 146. ____ 147. ____ 148. ____ 149. ____ 150. ____ 151. ____ 152. ____ 153. ____ 154. ____ 155. ____ 156. ____ 157. ____ 158. ____ 159. ____ 160. ____ 161. ____ 162. ____ 163. ____ 164. ____ 165. ____ 166. ____ 167. ____ 168. ____ 169. ____ 170. ____

171. ____ 172. ____ 173. ____ 174. ____ 175. ____ 176. ____ 177. ____ 178. ____ 179. ____ 180. ____ 181. ____ 182. ____ 183. ____ 184. ____ 185. ____ 186. ____ 187. ____ 188. ____ 189. ____ 190. ____ 191. ____ 192. ____ 193. ____ 194. ____ 195. ____ 196. ____ 197. ____ 198. ____ 199. ____ 200. ____ 201. ____ 202. ____ 203. ____ 204. ____

205. ____ 206. ____ 207. ____ 208. ____ 209. ____ 210. ____ 211. ____ 212. ____ 213. ____ 214. ____ 215. ____ 216. ____ 217. ____ 218. ____ 219. ____ 220. ____ 221. ____ 222. ____ 223. ____ 224. ____ 225. ____ 226. ____ 227. ____ 228. ____ 229. ____ 230. ____ 231. ____ 232. ____

Your name: _____

Please mail this ballot to:

Guy L. Steele Jr.
Computer Science Department
Carnegie-Mellon University
Schenley Park
Pittsburgh, Pennsylvania 15213

preferably by November 8, 1981. Thank you for your help.

References

[ANSI 76a] ANSI X3J3 Committee.
Draft Proposed American National Standard FORTRAN.
ACM SIGPLAN Notices 11(3), March, 1976.
This is a draft of the ANSI FORTRAN 77 standard. The final standard is ANSI X3.9-1978.

[ANSI 76b] *American National Standard Programming Language PL/I*
ANSI X3.53-1976 edition, American National Standards Institute, Inc., New York, New York, 1976.
This is the official ANSI standard definition of the PL/I language.

[ANSI 78] *American National Standard Programming Language FORTRAN*
ANSI X3.9-1978 edition, American National Standards Institute, Inc., New York, New York, 1978.
This is the official ANSI standard definition of the FORTRAN 77 language. It defines both the full FORTRAN language and a standard subset.

[Forkes 81] Forkes, Doug.
Complex Floor Revisited.
In *APL 81 Conference Proceedings*, pages 107-111. ACM SIGAPL, San Francisco, September, 1981.
Proceedings published as *APL Quote Quad* 12, 1 (September 1981).
Three different proposals for extending the floor function to the complex domain are compared. Each satisfies certain interesting identities, but none satisfies all.

[IBM 70] *IBM System/360 Operating System PL/I (F) Language Reference Manual*
GC28-8201-3 edition, International Business Machine Corporation, White Plains, New York, 1970.
The concepts and many features of the programming language PL/I are discussed and documented. This document is meant by IBM to be used in conjunction with the *PL/I (F) Programmer's Guide.*

[Kuki 72] Kuki, Hirondo.
Complex Gamma Function with Error Control.
Communications of the ACM 15(4):262-267, April, 1972.
An algorithm to compute the gamma function and the loggamma function of a complex variable is presented. The standard algorithm is modified in several respects to insure the continuity of the function value and to reduce accumulation of round-off errors. In addition to computation of function values, this algorithm includes an object-time estimation of round-off errors. A FORTRAN program for the algorithm appears in the algorithms section of this issue.

[McDonnell 73] McDonnell, E.E.
Complex Floor.
In *Proceedings of the APL 73 Congress*, pages 299-305. Copenhagen, Denmark, August, 1973.
An extension of the floor and ceiling functions to the domain of complex numbers is proposed. Compatibility with the real domain is preserved, as well as a number of familiar algebraic identities.

[McDonnell 81] McDonnell, E.E.
An Implementation of Complex APL.
APL Quote Quad 11(3):19-22, March, 1981.
The extension to complex numbers is described of the version of APL provided by I.P. Sharp Associates. This implementation follows the proposal by Penfield in the APL 79 Conference Proceedings, with a few minor exceptions. Among these are the notation for complex numbers; altering and renumbering of the left

arguments to the circle function; extension of the "or" and "and" functions to be "gcd" and "lcm", extended to the complex domain; and omission of the proposed extensions to the floor, ceiling, residue, and encode functions.

[Moon 74] Moon, David.
MacLISP Reference Manual, Revision 0.
M.I.T. Project MAC, Cambridge, Massachusetts, April 1974.

[Penfield 77] Penfield, Paul, Jr.
Notation for Complex "Part" Functions.
APL Quote Quad 8(1):11-13, September, 1977.
Five proposals are presented for denoting the real part, imaginary part, magnitude, and phase of complex numbers in APL. This is the first of a series of three papers on the subject of extending APL to complex numbers.

[Penfield 78a] Penfield, Paul, Jr.
Extension of APL Primitive Functions to the Complex Domain.
APL Quote Quad 8(2):36-41, March, 1978.
The possible ways are discussed of extending the primitive functions of APL to operate on complex numbers. This is the second of a series of three papers on the subject of extending APL to complex numbers.

[Penfield 78b] Penfield, Paul, Jr.
Design Choices for Complex APL.
APL Quote Quad 8(3):8-15, June, 1978.
Problems with grafting complex numbers onto the existing APL language are discussed. In particular, possible notations for complex numbers and the question of compatibility with the current "real-number" APL are considered. This is the last of a series of three papers on the subject of extending APL to complex numbers.

[Penfield 78c] Penfield, Paul, Jr.
Complex APL: Comments from the APL Community.
APL Quote Quad 9(1):6-10, September, 1978.
This paper discusses the responses of many readers to three papers by Penfield on the subject of extending APL to complex numbers.

[Penfield 79] Penfield, Paul, Jr.
Proposal for a Complex APL.
In *APL 79 Conference Proceedings*, pages 47-53. ACM SIGPLAN/STAPL, Rochester, New York, June, 1979.
Proceedings published as *APL Quote Quad* 9, 4 (June 1979).
This is a concrete proposal for extending APL to handle complex numbers. It is based on a series of papers by Penfield which appeared in *APL Quote Quad* and reader responses to those papers.

[Penfield 81] Penfield, Paul, Jr.
Principal Values and Branch Cuts in Complex APL.
In *APL 81 Conference Proceedings*, pages 248-256. ACM SIGAPL, San Francisco, September, 1981.
Proceedings published as *APL Quote Quad* 12, 1 (September 1981).
A proposal is presented for a consistent and systematic choice of principal values and branch cuts for complex exponentiation, logarithm, trigonometric, hyperbolic, and Pythagorean functions for use by APL. The branch cuts are chosen and justified according to explicitly stated criteria.

[van Wijngaarden 77]
van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C.H.; Merrtens, L.G.L.T.; and Fisker, R.G. (eds.).
Revised Report on the Algorithmic Language ALGOL 68.
SIGPLAN Notices 12(5):1-70, May, 1977.
This is a revision of the definition of the algorithmic language ALGOL 68. The working group which developed this revision has decided that it should be "the final definition of the language ALGOL 68", and the hope is expressed that it will be possible that all implementations may be brought into line with this standard.

[Weinreb 78] Weinreb, Daniel, and Moon, David.
LISP Machine Manual, Preliminary Version.
Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, November 1978.
The title of this manual wraps around from the back cover to the front in such a way that the last two words are split across covers. For this reason it is sometimes colloquially called the "Chine Nual" because that is what appears on the front cover. This edition has a black cover.
The LISP dialect which runs on the LISP Machine processor is described and documented. The dialect is a greatly altered and extended descendant of MacLISP.

[Weinreb 81a] Weinreb, Daniel, and Moon, David.
LISP Machine Manual, Fourth Edition.
Symbolics, Inc., Cambridge, Massachusetts, July 1981.
The title of this manual wraps around from the back cover to the front in such a way that the last two words are split across covers. For this reason it is sometimes colloquially called the "Chine Nual" because that is what appears on the front cover. This edition has a gray cover.
This is a revision is the language reference manual for Lisp Machine LISP, a descendant of MacLISP which runs on the Lisp Machine (also known as the CADR processor). This is the first edition printed by Symbolics, Inc.

[Weinreb 81b] Weinreb, Daniel, and Moon, David.
LISP Machine Manual, Third Edition.
Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1981.
The title of this manual wraps around from the back cover to the front in such a way that the last two words are split across covers. For this reason it is sometimes colloquially called the "Chine Nual" because that is what appears on the front cover. This edition has a bright red cover.
This is a revision is the language reference manual for Lisp Machine LISP, a descendant of MacLISP which runs on the Lisp Machine (also known as the CADR processor). In this revision first appears documentation for the LOOP macro and for flavors.

Chap	Title	Issues	Time	Number
Monday	10:00-12:00			
1	Intro	4-5	20	2
2	Data types	7-18	90	10
3	Program Str. 1	21-22	10	2
Monday	1:30-3:30			
12	Sequences	120-136	60	13
4	Predicates	23-31	25	5
5	selectq	37-42	10	6
5	iteration	43-48	20	5
5	assgnmnt	32-35	5	4
Monday	3:40-5:30			
5	mult val	55-58	40	3
5	mapping&prog	50-54	10	3
10	Complex	81	20	1
5	catch/throw	60-65	10	4
8	Declarations	69-75	20	3
10	various	83-90	10	4
Tuesday	9:30-11:30			
11	Characters	109-118	50	9
13	List Structure	138-156	45	11
10	irrational	91-95	10	4
10	type conv	99-102	5	4
10	random	106-108	5	3
9	Symbols	77-80	5	3
Tuesday	1:00-3:00			
19	File System	226-232	40	4
19	Read	191-193	20	3
19	Sharp Sign	195-199	30	5
19	Streams	203-214	20	6
19	readtable	200-201	10	2
Tuesday	3:10-5:00			
17	Structures	176-190	50	10
14	Strings	157-162	15	4
15	Vectors	166-167	5	2
16	Arrays	170-172	15	3
19	Format&beep	217-222	25	5