Common LISP Cleanup

Larry Masinter

[P88-00041]

XEROX

System Sciences Laboratory
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Common Lisp Cleanup

Larry Masinter

10 January 1988

This paper describes some of the activities of the "cleanup" sub-committee of the ANSI X3J13 group. It describes some fundamental assumptions of our work in this sub-committee, the process we use to consider changes, and a sampler of some of the changes we are considering.

1 Assumptions.

We believed the only reasonable way to obtain a standard for Lisp was to start with a known, widely implemented description, namely "Common Lisp the Language" by Guy L. Steele Jr. (*CLtL*) and to consider individually various clarifications, changes, additions, enhancements, modifications.

Common Lisp "works". It is widely implemented. Common Lisp has a relatively small kernel of semantics, facilities for extensions, and a large library of data types and operations on them. It is a good starting point for a Lisp standard. We do not believe that it is necessary to start over to design a good Lisp standard. Even if one favors modifications to the fundamentals of the language semantics (e.g., function and variable name scoping), most of the library of data types and operations of Common Lisp will continue to form a good basis for applications and it is valuable to clarify the operation of the library.

There are three kinds of problems with CLtL we would like to resolve in creating a standard.

First, CLtL had its roots in a user's level description of a programming system; although it is wonderfully specific as user manuals go, there are numerous places that allow more implementation flexibility than is desirable in a standard. In a standard, we would expect that ambiguities to be either eliminated or else em explicitly allowed.

Secondly, CLtL was written and published without extensive prior use of some of its features; a few have turned out to be unworkable or useless and not implemented. We hope to change or remove those features.

Third, we've discovered that there are features that exist in nearly in every implementation but with different names and calling conventions or (in a very few instances) are logical extensions of the current specification. We would like to augment the standard by providing access from the standard language to those features.

2 The process.

The cleanup subcommittee considers those minor changes that are not being addressed by any of the other single-topic subcommittees of X3J13 (objects, windows and graphics, characters, iteration, compilation, validation, and a few others.)

We consider proposals both internally generated by members of the cleanup committee or are received from members of the community.

Almost all work of the committee is handled via electronic mail. We rarely meet as a group in person. Most decisions are made by consensus. When we cannot agree by consensus whether to endorse a proposal, we may present it to X3J13 for voting with the various points of view represented in the writeup; we believe our responsibility is to fully describe the issues, and allow the larger community to vote.

We attempt to analyze the costs and benefits for each proposed modification. Each proposal, considered individually, must make the language better–enough better to be worth the cost of making a change. To

this end, we require that each proposal be accompanied by an analysis of the various costs and benefits. Of course, there are conflicts in the interpretation of costs and benefits.

There are some general principles of good programming language design: programs should be easy to read; programs should be easy to write; the language should allow users to write efficient programs. These goals sometimes work against each other: for example, adding features makes the job easier for the program writer, but harder for the reader. These conflicts in desired make the design process difficult, because it is not possible to optimize all of the design goals simultaneously.

The process of explicitly addressing costs and benefits has successfully eliminated arguments based merely on personal preference. Most often, what initially seems like a matter of taste in fact has roots in a more explicit judgment of benefit over cost. By adopting an explicit format for those items, we've managed to expose the underlying principles.

We only consider "complete" proposals. While we've discussed some open problems, we generally avoid "design by committee." Rather, after some initial discussion, a single individual will produce a proposal.

2.1 Elements of a proposal.

These are the important elements of the "cleanup" proposals we develop:

Category. We attempt to distinguish between a clarification (proposal to resolve an ambiguity or case of under-specified situation in CLtL), a change (proposal to make an incompatible change to the language), and an addition (proposal to add an upward compatible feature.) The category of a proposal will likely dictate what a responsible implementor of Common Lisp would do prior to the acceptance of a new draft standard; e.g., clarifications can be adopted immediately, while incompatible changes might wait until there is an official standard or be implemented in a separate "package", at which time incompatible changes can be adopted all at once.

Problem Description. Suprisingly, requiring an explicit description of the problem, indpendent of some design for its solution, has been most helpful in avoiding unnecessary changes to the language. Why does anything need to happen at all? ("If it ain't broke, don't fix it.")

Proposal. We expect proposals to separate out the description of the proposal from the arguments for it. Again, this simple measure helps lend a great deal of objectivity to the process. While our proposals are expected to be precise and accompanied by test cases and examples, we generally look to the editor and editorial committee of X3J13 to actually produce the specification that reflects the proposal.

Current practice. What do current Common Lisp implementations do? Current implementations (at least implementation that attempt to be faithful to CLtL) often give us good indication of what implementors believed CLtL to mean. We put some faith that they attempted a reasonable interpretation. In cases where no implementation exactly matches CLtL, this is a good indication that CLtL's description is flawed.

Costs. Every change or resolution costs something to adopt: to implementors (What does it take to change a non-conforming implementation to a conforming one? Does this affect some, one, many functions? Is public-domain code available?), and to current users (to convert existing user code or tools.) We hope to give a measure of whether the changes required for each are small or large. In general, we take cost to users more seriously than cost to implementors.

Benefits. Benefits are diverse: our goals are to improve the portability of the language, its performance, the cleanliness of its semantics, and its aesthetics. Our major concern is with allowing important applications of Common Lisp to work well across many different implementations.

3 Open issues.

This section lists most of the proposals currently being considered, in an extremely abbreviated form. Many of these issues have *not* been adopted and many *will not be*; they are listed here to give an idea of the kinds of issues the cleanup committee is currently addressing. New issues–especially that has affected the portability of any current program–are welcome.

Reminder: these proposals have not been accepted; they are not part of any standard. This is a list of issues considered, not of issues passed. While some have been endorsed by X3J13, others are likely to be rejected.

Clarifications. These are cases where CLtL does not specify behavior; usually these are also cases where implementations actually differ and users have found code non-portable.

APPEND-DOTTED: `(APPEND '(A . B) '(E F))` is not an error, returns `(A E F)`.

COLON-NUMBER: `:123` is an error and not a keyword symbol.

DECLARATION-SCOPE: Specify more precisely the scope of declarations.

COMPILER-WARNING-BREAK: `*BREAK-ON-WARNINGS*` applies to compiler warnings too.

FLET-IMPLICIT-BLOCK: The bodies of functions defined in `FLET`, `MACROLET` and `LABELS` have implicit `BLOCK`s wrapped around them in the same manner as `DEFUN` and `DEFMACRO` bodies.

DEFVAR-DOCUMENTATION: In `(DEFVAR x y documentation)`, documentation is not evaluated.

DEFVAR-INITIALIZATION: In `(DEFVAR x y)`, y is evaluated when `DEFVAR` is.

PATHNAME-STREAM: Pathnames can be obtained only from streams opened with pathnames (or synonym streams.

PUSH-EVALUATION-ORDER: `(PUSH (A) (CAR (B)))` evaluates`(A)` before `(B)`.

REMF-DESTRUCTION-UNSPECIFIED: What side effects are allowed/possible by REMF, NREVERSE etc.?

SETF-METHOD-FOR-SYMBOLS: Examples in CLtL of SETF-method for symbols and LDB are inconsistent with left-to-right evaluation order. Change them.

SHADOW-ALREADY-PRESENT: What does SHADOW do if symbol is already there?

SHARPSIGN-PLUS-MINUS-PACKAGE: Elements of `*FEATURES*` are in the keyword package.

SHARPSIGN-PLUS-MINUS-NUMBER: Numeric `*FEATURES*` are not allowed.

UNWIND-PROTECT-NON-LOCAL-EXIT: Exit from cleanup clause of `UNWIND-PROTECT` overrides any unwind in progress.

ADJUST-ARRAY-DISPLACEMENT: More precise rules for interaction of `ADJUST-ARRAY` and displaced arrays.

DO-SYMBOLS-DUPLICATES: Can `DO-SYMBOLS` visit a symbol more than once?

FORMAT-UPARROW-SCOPE: What is the scope of `^` in `FORMAT` loops?

FUNCTION-ARGUMENT-TYPE-SEMANTICS: What are the semantics of argument types in `FUNCTION` declarations?

LISP-SYMBOL-REDEFINITION: It is illegal (non-portable) to redefine or shadow with `FLET`, `MACROLET`, `LABELS` functions defined in CLtL.

PRINC-CHARACTER: `(PRINC #\C)` prints C.

IMPORT-SETF-SYMBOL-PACKAGE: IMPORT has no effect on a symbol's home package.

FORMAT-OP-C: (format t "~C" '#\C) prints C.

DISASSEMBLE-SIDE-EFFECT: DISASSEMBLE has no side-effects.

Additions. These are some new features we are considering adding to Common Lisp.

AREF-1D: Add (ROW-MAJOR-AREF X N) to reference n-th row-major element of an array.

FORMAT-COMMA-INTERVAL: Add option to FORMAT to specify intervals other than 3 between digit delimiter character.

GET-SETF-METHOD-ENVIRONMENT: Add an environment argument to GET-SETF-METHOD.

KEYWORD-ARGUMENT-NAME-PACKAGE: Allow arbitrary symbols as keyword-argument tokens.

SEQUENCE-FUNCTIONS-EXCLUDE-ARRAYS: Allow some sequence functions to work on multi-dimensional arrays.

ASSOC-RASSOC-IF-KEY: Add :KEY argument to ASSOC-IF.

REDUCE-ARGUMENT-EXTRACTION: Add :KEY argument to REDUCE.

SETF-FUNCTION-VS-MACRO: Allow (DEFUN (SETF FN) (VALUE ARGUMENTS ...) ...) to define what (SETF (FN ...) value) means.

PATHNAME-SUBDIRECTORY-LIST: Recommend that directory component of a pathname be a list in a hierarchical file system.

FORMAT-ATSIGN-COLON: ~@: and ~:@ are the same in format directives.

STREAM-CLASS-ACCESS: Add functions for finding out stream class, accessing parts.

STRUCTURE-DEFINITION-ACCESS: Add functions for getting at DEFSTRUCT accessor slots.

TRACE-FUNCTION-ONLY: Enhance TRACE arguments.

Changes. These proposals are generally incompatible changes to the language, although in some cases they are consistent with CLtL.

: Require STREAM, PACKAGE, PATHNAME, READTABLE, RANDOM-STATE be disjoint types.

PATHNAME-SYMBOL: Symbols do not automatically coerce to pathnames, e.g., (LOAD 'FILE) is not allowed.

DECLARE-MACROS: (LET (A B) (MY-DECLARE) ...) Disallow macros expanding into declarations.

FUNCTION-TYPE: Change the FUNCTION type to be distinct from any other type, and change FUNCTIONP to accept only FUNCTION objects (not symbols or lists that begin with LAMBDA.). Require the FUNCTION always returns a FUNCTION object. Disallow automatic coercion from symbols to functions by FUNCALL, APPLY, etc.

DEFMACRO-BODY-LEXICAL-ENVIRONMENT: Allow DEFMACRO and DEFTYPE bodies to have a lexical environment.

4 Open problems.

There are a number of areas that are currently not being addressed, either by the cleanup committee or by other current committee's of X3J13. We would like to see some attempts at standardizing on those areas that are most concern to users attempting to write portable code.

One is that with the acceptance of a reasonable signal system, the hard work of specifying the signals which may be raised by various error conditions remain. We would like to make a part of the standard interface to functions the classes of errors they can invoke under various circumstances.

No work is currently going on in the area of specifying reference to multi-programming features, although many implementations of Common Lisp include them.

There are several problems in the language specification for which we have no good proposals. Most of these involve interactions of the many functions in the Common Lisp library, e.g., how **EQUAL** behaves on **DEFSTRUCT** objects. We believe these items need to be standardized, but have not found a reasonable design.