Masinter & vanMelle on CommonLisp, circa Dec 1981.

Subject: Report on Common Lisp to the Interlisp Community

Introduction

What is Common Lisp?

Common Lisp is a proposal for a new dialect of Lisp, intended be implemented on a broad class of machines. Common Lisp is an attempt to bring together the various implementors of the descendents of the MacLisp dialect of Lisp. The Common Lisp effort is apparently an outgrowth of the concern that there was a lack of coordination and concerted effort among the implementors of the descendents of MacLisp. Common Lisp's principal designers come from the implementors of Lisp Machine Lisp (at MIT and Symbolics), Spice Lisp (at CMU), NIL (both a VAX implementation at MIT and the S-1 implementation at Lawrence Livermore), and MacLisp (at MIT, of course.) Also contributing, but not strongly represented, are Standard Lisp (at Utah) and Interlisp (at PARC, BBN and ISI.)

Programs written entirely in Common Lisp are supposed to be readily run on any machine supporting Common Lisp. The language is being designed with the intent first of creating a good language, and only secondarily to be as compatible as practical with existing dialects (principally Lisp Machine Lisp.)

Common Lisp attempts to draw on the many years of experience with existing Lisp dialects; it tries to avoid constructs that are obviously machine-dependent, yet include constructs that have the possibility of efficient implementation on specific machines.

Thus, there are several different goals of Common Lisp: (1) to profit from previous experience in Lisp systems, (2) to make efficient use of new and existing hardware, and (3) to be compatible with existing dialects. These various interests pull in different directions on the design of the language. The desire to have a core language within which programs are portable urges that the language be functionally complete, so that you can write a wide range of programs in it. The desire to implement the language on a wide range of machines urges that the language be compact, and not have features that are tricky to implement. The goal of making the language good or "right" is at odds with keeping compatibility with existing languages that suffer various blemishes. Allowing implementations freedom to perform certain operations more efficiently encourages vagueness in the specification, but that vagueness makes the language weaker and can be a rich source of bugs.

These conflicting goals make consensus in the Common Lisp design difficult. The design of the language is not yet nearly complete. The amount of convergence and the number of resolved issued is impressive, but there are many difficult issues ahead.

No machine yet implements Common Lisp. The specification of Common Lisp is complex enough (and different enough from any existing dialect) that we believe that a full implementation will involve a significant amount of work once the design is stable. There is often a large amount of optimism in the time allowed for future implementations of Lisp dialects that has yet to be confirmed by any experience we know of.

The November Common Lisp Meeting

On Nov 23-24, 1981, Bill van Melle, as Interlisp representative and proxy holder for Larry Masinter attended a meeting of the various implementors of Common Lisp. The meeting was held at Symbolics in Cambridge. Also in attendance were Guy Steele (CMU), Dick Gabriel (Stanford), Jon L White (MIT), George Carrette (MIT), David Moon (Symbolics), Howard Cannon (Symbolics), Daniel Weinreb (Symbolics), Alan Bawden (MIT), Rod Brooks (MIT), and Richard Bryan (MIT).

The meeting was called to discuss some of the issues that had arisen in putting together the Common Lisp specification. The intention was to resolve as many as possible of those issues to the satisfaction of the people concerned. This report concerns both the outcome of that meeting and the Common Lisp effort in general.

There was no formal voting procedure or official mechanism to determine the relative "voting strengths" of those present. As much as possible, the group tried to resolve issues by acclaim. Lone dissenters were generally ignored, unless they were especially vehement. If there remained substantial disagreement over an issue even after discussion, the issue was deferred (though usually by that point the choices on the issue had at least been narrowed).

Informally, the Lisp Machine representatives tended to have the greatest clout, and one of their principal concerns was that Common Lisp be as compatible as possible with Lisp Machine Lisp, even if that meant retaining blemishes in the language (e.g. bad order of args to putprop, push, etc).

The interest of the Interlisp community.

The Interlisp community's interest in this discussion is not immediately obvious. None of the various possible relations with Common Lisp will be simple: it will be difficult to make Interlisp programs mechanically transportable to Common Lisp, to make Common Lisp programs mechanically transportable to Interlisp, or to implement a Common Lisp given an existing Interlisp.

Thus, for the most part, we cast votes in favor of making the language a good language, seeking completeness and perspicuity. It ought to be the case that two programmers speaking different Lisp dialects should be able to look at each other's programs and have a good chance of following them without resorting to careful reading of a thick manual. This argues strongly in favor of keeping the set of standard functions small, yet allowing them to express the most common programming needs. There were places where we voted against blatant incompatibilities with Interlisp, but not always successfully (we agreed that at best, porting programs between Interlisp and Common Lisp could be done with mechanical assistance). There were other occasions where we suggested constructs taken from Interlisp to help resolve some difficulty, or fill in a gap in the language definition.

The rest of this report explains some of the major issues, especially those where compatibility with Interlisp seems problematic.

1. Symbols NIL and T.

These symbols ("litatoms" in Interlisp parlance) have since the beginning been "special" in most Lisp implementations; e.g. you can't bind or set them (or you can, producing strange and wondrous effects), you can't put properties on NIL. The implementors of the language NIL decided to remove this blemish by making NIL and T be ordinary symbols. The value "false" is still the empty list, but it is spelled (), and is not a symbol; the value "true" is a special "truth" constant, not T, spelled #t. [# is a readmacro used to read many sorts of special constructs.]

Most other implementors who expressed any opinion view this change as anywhere from silly to absolutely unacceptable. The original Common Lisp specification attempted to compromise on this issue by allowing but not requiring that NIL and () be different entities, and that (symbolp ()) be allowed to be true. That introduces an undesirable vagueness, something we had been trying to fight elsewhere. And it is not a particularly grand compromise: in order for programs to be transportable between implementations, and in particular for a program written using NIL as false to be readable by an implementation where NIL and () are different, it is necessary at least that the empty list always PRINT as (); furthermore, programmers must not pass around the symbol T unquoted.

An impressive amount of time was spent arguing about this issue at the November meeting. Many people find () and 'T an aesthetic abomination, and were aggravated that we were wasting so much time on this comparatively petty issue. There is little hope that Common Lisp will eradicate all warts from the face of the language, and this does not seem to be a wart that is worth removing (David Moon summed it up nicely by saying that NIL and T are not warts, but rather birthmarks). It was thought that this change would cause much distress to the MacLisp, Lisp Machine Lisp and Interlisp communities for little gain; the NIL representatives claim a user community as well (not independently substantiated), but admitted that users have not been flocking to use

NIL and T as variables.

The issue was left unresolved.

2. Case preservation in pnames.

In MacLisp and friends, READ canonicalizes atom names to all uppercase (except where explicitly quoted). Interlisp preserves case (although for terminal input you can be in "raise" mode, where lower-case is read as upper-case). The effect in MacLisp is that case is largely ignored--you type your program in whatever case you want, but ultimately all symbols are uppercase. Since MacLisp users typically edit their programs on files using text editors, the case of their input is in some sense automatically preserved. In Interlisp, user programs are edited inside Lisp and prettyprinted back out to files, so it is important that case is preserved (assuming you ever want to be able to use lowercase).

It is difficult, if not outright impossible, to have it both ways (have symbols read in correctly independent of the case in which they are typed, yet have the case of symbols you care about preserved), but this may not really be a significant issue. Except for use of symbols (rather than strings) in printed text, and for those users who define distinct functions and/or variables that differ only in case, it appears that a program in a case-preserving language will still run fine in a non-case preserving dialect, and that a program in a non-case preserving dialect will run in a case-preserving one if the program is first prettyprinted out to a file (so that all the symbols are in the same case).

3. Sequence Functions.

Common Lisp has the notion of a "sequence", something that has elements that can be enumerated in an obvious linear fashion. Standard sequences are lists, strings and vectors (one-dimensional arrays). There are operations that are useful on any sequence, independent of its representation. For example, determining the length of the sequence, extracting/replacing the nth element, sorting the sequence. Common Lisp proposes introducing functions that operate on sequences. Many are old favorites from the list world, e.g. LENGTH, REMOVE, REVERSE, extended now to other kinds of sequences. There are two major issues here.

3a. Type-specific sequence functions.

"Generic" sequence functions may incur significant overhead over type-specific sequence functions; and generic functions "catch fewer errors", e.g., where you think you're handing around a list but really it's a string. The former is really only significant for a few primitive operations (elt, length) that can be cleverly open-coded; and it's not clear the latter is much worse than the situation that exists already with numbers. (All of the Common Lisp numeric functions are "generic" in that they accept numbers of any type (integers, floating point numbers or ratios, sometimes complexes) and behave like type-specific functions of the type suggested by their arguments.)

An earlier proposal of Common Lisp had the notion of type-specific functions in addition to each generic function, e.g., list-length, string-length and vector-length in addition to length. This was rejected at the November meeting; we opted instead for a means of declaring the types of arguments in those cases where there is efficiency or documentation concern. The preferred choice for declaration was a type declaration wrapper around designated arguments, transparent to the interpreter (except for optional error checking) but useable by the compiler. The THE construct from the Interlisp DECL package was tentatively considered for this.

3b. Variations of common sequence functions.

The issue here is how to specify in a convenient way the various forms of the same basic operation. For example, in traditional Lisp, there are some functions that differ only in whether they test elements of a list using EQ or EQUAL. (E.g., MEMB and MEMBER in Interlisp.) Common Lisp wants you to be able to write something that means "remove from y all elements that satisfy condition p", where the condition might mean "EQ to z", "EQUAL to z", or some arbitrary condition, without having to make the common cases ugly or awkward (the first two cases are MacLisp's REMOVE and REMQ functions, I believe).

An earlier Common Lisp draft had a profusion of separate functions for each of these. This was rejected at the November meeting, but no alternative was agreed upon. The major contenders were

(1) some keyword based scheme, e.g., (member x lst :test (function superequal)), or (remove lst :if-not (function baz)); and somewhat less enthusiastically,

(2) a functional programming style, where each sequence operator has a functional operator that produces a function to do the desired thing, e.g. the two previous examples would be ((fmember (function superequal) x) lst) and ((fremove (function (lambda (x) (not (baz x)))) lst).

As an adjunct to (1), it was also recommended that LOOP (see below) be used to handle the less common cases.

4. Iterative constructs

Some of the MacLisp dialects have a construct called LOOP, which is similar to Interlisp's CLISP iterative constructs. There are some objections to it (it is too complicated; code-walking programs have to parse it, and the translation into a PROG with GO's is ugly to analyze), but most people seem to think it is a win, or at least is more desirable than having lots of different functions to do special kinds of iterations. It's not clear that the more violent objections are based on anything more than lack of familiarity.

It was proposed that we wait for the Lisp Machine people to submit a revised spec for LOOP that counters some of the objections; if the result is satisfactory, several functions may fall by the wayside.

4. Multiple values.

Lisp Machine Lisp allows functions to return multiple values. The idea is that this allows a function to return more than one result to its caller without having to create a list containing the results, or resorting to the unaesthetic method of setting freely some variables that are bound in the caller. In order to use the multiple results, the caller encloses the call in a construct that in some way spreads the results; the principal methods are a "multiple-value setq" form and a "multiple-value bind" form. If the caller just uses the result of a function as a single value, it gets the first of the multiple values and the rest are discarded. Thus, most of the time, you need never know that a function returns multiple values; this means that system functions can return possibly useful extra values that the caller can use or not as it chooses. For example, QUOTIENT returns the remainder as the second value, since the remainder is frequently computed anyway as a side effect of the division.

Common Lisp currently requires the support of the multiple values feature. Unfortunately, the use of multiple values severely corrupts the semantics of Lisp, and nobody has yet figured out how to "do multiple values right". The major corruption is that, as currently designed, you can't bind a variable to multiple values. This means it is difficult to transmit multiple values in a transparent fashion, making it a tricky, if not impossible business to "break" or "advise" a function that returns multiple values. It also inhibits some compiler optimizations; e.g. it is no longer true that you can transform the sequence (SETQ X (FOO)), (RETURN X) into (RETURN (FOO)), because the latter may return multiple values while the former never does. It complicates an implementation's call/return mechanism to accommodate an infrequent case, resulting in somewhat arbitrary restrictions on how multiple values are handled by certain forms. If FOO returns multiple values, then should (PROG1 (FOO) --)? What about (OR (FOO) (FIE))? (AND (FOO) (FIE))? Requiring that these forms transmit multiple values somewhat cramps the implementation in order to satisfy a very infrequent case, yet not requiring this leaves gaping inconsistencies in the handling of multiple values.

No real decision was reached. We argued against multiple values until someone has figured out how to do them right, but this was not a popular sentiment. It was proposed that there be a construct that allows you to bind a variable to the (conceptual) multiple-value vector returned from a function, so that a program can at least pass values thru itself without great hassle. There is still a fair amount of design to go on here.

6. Arithmetic

There is a great deal of interest that Common Lisp support a full range of arithmetic, with the goal of supporting in Lisp both number crunching and algebraic manipulation. (This appears to be a major goal of the S-1 implementation, for example.) In addition to conventional integers and floating-point numbers, Common Lisp provides arbitrary-precision integers (bignums), ratios (i.e., fractions consisting of two integers, possibly big themselves) and complex numbers (pairs of floating-point numbers).

We did not argue against requiring these in the Common Lisp standard, because their presence seemed necessary if there were in fact going to be Common Lisp algebraic manipulation systems.

There were some questions on how wide a range of functions to supply (e.g. should there be trigonometric functions that work in degrees?), and how much to shield the user from "optional" types, such as complex. For example, what should (SQRT -1) return--a complex number or an error?

7. Characters.

Common Lisp includes an elaborate specification for how characters are handled. It tries to encompass varying character sets (e.g. to let Common Lisp work on machines using EBCDIC), keyboards with multiple shifts (e.g. control, meta, super, hyper,...), and multiple fonts. Characters are (possibly) distinguished as a separate datatype (not integer character codes or litatoms), and a large collection of functions is supplied to avoid ever having to do arithmetic on characters (e.g., alphanumericp, upper-casep). Those functions also serve the purpose of allowing various character sets. Shifts and fonts are accommodated as extra numerical attributes of a character, with functions to test/extract/set those attributes.

Unfortunately, there is a lot of confusion here. Shift bits such as Meta- or Greek- are relevant to keyboard input (on some machines/keyboards), but make no sense in files. Similarly, keystrokes don't intrinsically have fonts. Identifying fonts by a small set of numbers is a confining notion; our experience in Interlisp-D is that it is better to have "change font" on a stream with ability for full font specification (family, size, boldness, etc); font numbers only lead to confusion between systems with different "font-sets". The implementation may still have font numbers at some level, but it is not so good at the user level. There was little discussion of this issue at the November meeting.

The Common Lisp specification also tries to define a common "standard" character set, which is basically all ASCII printing characters, plus some format characters. The discussion on format characters contains some of the same confusion mentioned above. For example, one person argued strenuously that <rubout> be a standard character, and then people argued on about whether <rubout> could appear in files. (He wanted a character such that he could write (eq (tyi) #\Rubout) in order to write a simple input routine that any Common Lisp could run.)

8. Name conflicts.

There are some hopeless incompatibilities with Interlisp of a syntactic nature, part of why a mechanical pre-processor will be important in moving programs between Interlisp and Common Lisp. For example, PUTPROP and PUSH do not take their "new value" argument last, as is the convention most everywhere else. Indexing functions (NTH, ELT) are zero-based instead of one-based. SELECTQ specifies the default differently. The function LISTP means "a cons or NIL" (this being the definition of "list" in Comon Lisp), and there is a separate function CONSP that does what Interlisp's LISTP does.

9. Is a "property list" a list in prop value format?

Some implementations want freedom in representing a symbol's "property list". E.g. Standard Lisp wants to use an a-list. Discussion proceeded on how to properly abstract the property list, and conjectures on to what extent programmers rely on the format of the property list. This hasn't been resolved yet.

10. Macros vs Special Forms vs functions.

Common Lisp currently prohibits special forms (NLAMBDA's in Interlisp) in APPLY and FUNCALL (= APPLY*). The reasoning is that some of these may be implemented via macros rather than NLAMBDA functions which explicitly evaluate their arguments.

11. NIL vs optional.

There are some places in the Common Lisp design where a null argument is treated differently than an omitted argument. We have requested that these be avoided, on the grounds of obvious confusion.

-- Larry and Bill

----- End Forwarded Messages -----