Lisp and Symbolic Computation **4**,4 (March 1992),371-398.

Critique of DIN Kernel Lisp Definition Version 1.2

Henry G. Baker

Nimble Computer Corporation,[1] *16231 Meadow Ridge Way, Encino, CA 91436*
(818) 986-1436 (818) 986-1360 (FAX)

Abstract

A critique of DIN Kernel Lisp is presented which argues for greater emphasis on implementation efficiency and language cleanliness, and a greater emphasis on *parallel* and *persistent* Lisp environments. Specific recommendations include standardizing the S-expression rather than the character form of a program, using lexical scoping and shadowing to enhance subsystem modularity, relying on macros and compiler-macros for more pleasant syntax and greater modularity, requiring immutable/functional bindings, strings, vectors and lists; using object-oriented capabilities to build basic capabilities—e.g., generic arithmetic, streams and pathnames, relying on `defstruct` instead of `defclass`, and standardizing on `defmethod` for all function definitions. A virtual/synthetic class mechanism is presented to solve certain technical problems analogous to those solved by the "virtual function" mechanism of C++. Finally, we recommend the inclusion of *futures* as DKLisp's fundamental mechanism for the introduction of multiple parallel threads of computation.

1. Introduction

The last ten years have seen both a boom and and bust for the Lisp language. From its inception in 1958 until its commercialization in 1980, Lisp was content to live a life of academic obscurity, servicing the needs of Artificial Intelligence, theorem proving and symbolic algebra researchers. In this role, Lisp could explore many new ideas and quickly absorb new ideas from the programming language "mainstream". However, when AI became a business, Lisp was drawn to the money like a moth to the flame. Lisp quickly sought to become "industrial strength", and ANSI Common Lisp was the result. Unfortunately, "industrial" quickly became "rust belt", as the newly standardized Common Lisp could not quickly grow and adapt to the new, more "open" environment.

In Lisp's quest for industrial respect, it abandoned its traditional customers. Symbolic algebra developers (e.g., *Maple*, *Mathematica*) forsook Lisp for the more portable and efficient C. Logicians gave up Lisp for Prolog and ML. Vision researchers, seeking the highest performance from parallel computers, dropped Lisp in favor of C, Fortran (!) and various fringe parallel languages. AI companies were forced to move their expert systems into C, C++, Ada and Cobol (!) because of the large size, poor performance and poor integration of Common Lisp systems. Academicians teaching undergraduates chose Scheme and ML for their small size, portability and clean semantics.

Lisp now commands a smaller market share of the overall computer software field than it did ten years ago, before the rapid rise of commercial Lisp! If the current standardization effort is to avoid "rearranging the deck chairs on the Titanic", *ISO/DIN must develop a brand new language which addresses the deficiencies in Common Lisp*, particularly those which have driven most of its traditional customers away. Its goal of backwards compatibility is not reasonable, since existing Common Lisp programs are unlikely to be converted to DKLisp, and backwards compatibility loads Lisp down with too much useless baggage. The goal of standardizing only non-controversial capabilities is also not reasonable, since the only reason why controversy surrounds certain capabilities is that people need them desparately! *All progress is a process of creative destruction, wherein the hardest, but most important, decisions involve what must be pruned in order to free up*

[1]The opinions here expressed are those of the author, and do not represent the policies of Nimble Computer Corporation.

the resources needed for new, more vigorous growth. Many of Common Lisp's current woes can be traced to the ANSI Committee's shirking of this responsibility; the ISO Committee must not follow in their footsteps.

The Lisp community must listen to the marketplace, which has shouted *small*, *small*, *small* and *efficient*, *efficient*, *efficient* for two decades. The triumph of Pascal over PL/I and Algol-68, C over Pascal and Ada, ML and Scheme over Common Lisp, Smalltalk and C++ over Common Lisp/CLOS, and RISC over CISC, all reinforce this theme. By far the largest single Lisp community is the *AutoLISP*® installed base of over 300,000 customers; AutoLISP is an interpreted, dynamically-scoped Lisp written in C that runs in *32K bytes* on a 640K IBM PC. AutoLISP has fixnums and floats, immutable strings and immutable (!) list cells. Its "virtual memory" system goes beyond autoloading, by swapping source code to disk. AutoLISP is both the command language and extension language for the *AutoCAD* mechanical drawing system of Autodesk, Inc., and the success of this product can be attributed, in part, to the elegance and flexibility offered by AutoLisp.

2. Goals of DKLisp

Every venture must have goals to define itself and to inspire its followers. If DKLisp is a language worth standardizing, then it should be possible to characterize the strengths of the language that make it attractive to a potential user or potential vendor. DKLisp has competitors, most prominently ANSI Common Lisp and IEEE Scheme, but logic languages and functional languages are growing quickly in popularity.

If DKLisp is going to be successful in the crowded world of computer languages, it must have some distinctive competence not otherwise available. Having been in the position of recommending a computer language as the "command and extension language" for several commercial products, we carefully evaluated both Common Lisp and Scheme and found both to be seriously deficient; Common Lisp was too large and too difficult to master by non-wizards, even while it lacked (standardized) multiple processes, and Scheme's lack of a type system, object-oriented programming features, macros, EVAL, and multiple processes made it too inefficient and weak. We would hope that DKLisp would become an alternative to Common Lisp and Scheme, so that we could wholeheartedly recommend a Lisp dialect to the non-wizard, non-academic community.

Presumably, a *kernel* language is a *minimal* language on top of which a larger language can be built. *Since its primary function is being an implementation language for function libraries and a target language for macros*, a Kernel language need not be particularly "user-friendly" or provide a lot of flexibility in its interface. Although it is desirable, a Kernel language does not have to be a strict *subset* of the full language, since some of its features may not be visible in the full language. A Kernel language should be simpler and more efficient to implement than the full language, however, to avoid an *abstraction inversion*,[2] in which a simpler notion is defined in terms of more complex notions.

We therefore recommend the following as desiderata for DKLisp:

1. *Efficiency*. The language should be capable of efficient execution on a wide variety of existing and future architectures. It should be efficient enough to support a tower of abstraction levels without toppling. Sheer efficiency can compensate for a number of deficiencies, because a user can trade execution time for luxuries. It should not be necessary to write numerically-intensive subsystems in another language—i.e., it should be possible to achieve execution speeds for DKLisp on fixnums, characters, single-precision and double-precision floating point numbers and vector references which is within 20% of the speed of optimized C code. It should be possible to tune an application without massive rewrites of the code, and without requiring massive use of Common Lisp's "#+/#-".

---

[2]This term comes from critiques of the Ada programming language *rendezvous* mechanism.

2. *Size*. The language *definition* itself should be kept to 50 pages, and the language itself should be capable of running effectively within the 640K limit on MSDOS machines.[3] These limitations guarantee the ability to build, maintain and port the language with a relatively small staff. So long as DKLisp does not waste these 50 pages on heavy syntax and datatyping, the language can have enormous power.

3. *Parallelism*. The language should be capable of effective execution on SIMD and MIMD parallel architectures, supporting both the shared-memory and non-shared-memory (message-passing) paradigms.

4. *Persistence*. The language should be capable of seamless extension with a model of persistence along the lines of object-oriented databases [Kim89].

5. *Abstract Data Types and Object-Oriented Programming*. DKLisp should not only offer the user a mechanism for building abstract data types and using object-oriented programming techniques, but should also utilize these capabilities itself. The ultimate goal is to make most of the functionality of ANSI Common Lisp—although probably not its syntax—available through libraries of abstract data types and subroutines, rather than having these capabilities "built in" [Gabriel91].

6. *Programming in the Large*. DKLisp should *not* follow the ridiculous Unix "make" model of system-building based on large numbers of ASCII files linked together with chewing gum and baling wire. The Ada library model (especially with Ada-9X nested libraries) is far more Lisp-like, and could be almost trivially done in a persistent DKLisp, since DKLisp does not have the type complexity of Ada.[4]

7. *Clean Semantics*. Language implementors and users should not have to become "language lawyers", nor should it be necessary for ACM *Lisp Pointers* to institute a "Dear John" (McCarthy) column[5] to explain the intricacies of obscure portions of DKLisp. It should be possible to write a simple metacircular interpreter in a few pages, as in McCarthy's original Lisp.

3. Applying these goals to DKLisp.

Obtaining *efficient execution* from DKLisp requires minimizing decisions and operations which must be performed at run-time. This means that 1) the number and complexity of control operations be minimized; 2) the number and datatypes of arguments to primitive functions be fixed; 3) the datatype of function results be fixed; 4) the selection of generic function methods be done at "compile" (i.e., macro-expansion) time; 5) copying and (heap) consing be minimized; 6) opportunities for compiler/runtime optimizations be maximized; and 7) portable, transparent access to traditional control and data primitives be provided—e.g., goto's, double-precision multiply-divide, character manipulations and floating point operations. Meeting the efficiency requirement will require a set of non-generic arithmetic functions for at least the 3 datatypes: `fixnum`, `single-float` and `double-float`, which functions can be utilized to efficiently implement generic arithmetic.

Preserving a *small definition* for DKLisp requires that the standard ruthlessly eliminate non-primitive notions to focus on simple, efficient, modular primitives from which the rest of a Lisp system can be easily and efficiently constructed. In other words, DKLisp primitives must be chosen not for their ability keep a user program small, but for their ability to keep an implementation small and fast.

---

[3]These size restrictions are not due to any love for the Intel 8088 architecture or MSDOS, but because a language of this size is also compatible with the short attention span of non-wizards.

[4]For example, Fortran-90 seems to have picked up some of the better Ada module features.

[5]Analogous to ACM *Ada Letters*' "Dear Ada" column.

Providing for *parallelism* in DKLisp means far more than specifying forking and synchronization primitives. The basic data structures and binding models must also be compatible with parallel execution. Since imperative constructs—most notably assignments—greatly interfere with the potential for clean and efficient parallel execution, these constructs will have to be severely curtailed in DKLisp. Furthermore, DKLisp will have to indicate for every binding whether it is permanent (immutable) or temporary (mutable).

Providing for *persistence* in DKLisp means providing for 1) execution environments whose lifetimes may be measured in months and years, and 2) execution environments some of whose data structures may be shared among several users. Some implications of this requirement are the same as for parallelism, but others include a system of first-class types/classes (i.e., `type` is a type).

Providing for *abstract data types* and *object-oriented programming* requires the ability to extend types and polymorphically extend function definitions.

Providing for *programming in the large* requires the ability to define and debug subsystems and then transitively compose them into larger and larger systems, without requiring massive changes in the components being incorporated. Traditional compiled languages—e.g., Ada—make extensive use of a lambda-calculus-like naming visibility scheme, wherein "alpha-renaming" assures that semantics are independent of the spelling of identifiers, and wherein "lexical shadowing" assures the referential transparency of subsystems. Common Lisp has not yet capitalized on the full power of lexical scoping to achieve this goal.

Providing for *clean semantics* is the hardest requirement of all. Most languages take a "divide and conquer" approach to simultaneously meeting a number of different requirements; this *ad hoc* approach uses a completely different mechanism for each problem to be solved. ANSI Common Lisp is the biggest offender of this requirement since PL/I [Radin78].[6] It has at least 12 kinds of dynamic binding—static variables, dynamic variables, function/macro names, block names, catch tags, go tags, unwind-protect forms, keyword association lists, property lists, hash tables, pathname/file bindings and method/function bindings—as well as a number of kinds of "static" binding—package, defconstant, defun/defmacro, readtable macro, etc. Common Lisp has not taken enough advantage of the ability of lexical scoping to singlehandedly finesse many of these binding problems.

4. Specific Comments on DKLisp Definition Version 1.2

Primitive Control Constructs

The DKLisp standard document wastes valuable space on trivialities. If certain concepts are not *primitive*—i.e., if they can be easily defined in terms of other concepts—then these non-primitive definitions should be relegated to an appendix. The best examples are `let`, `let*`, `null`, `and`, `or`, `cond`, `case`, `progn`, `do` and `do*`. Since these are all trivially defined by macros in terms of function calling, `lambda` and `if`, DKLisp need only give a macro definition for these forms in an appendix, along with words to the effect that any implementation must produce the same results as if they had utilized these macro definitions. Since we are talking about the language *definition*, and not the language *implementation*, an implementation is free to recognize and specially handle non-primitive capabilities. However, all of the above constructs, with the possible exception of `case`, have macro-expansions which can be compiled efficiently without having to be specially recognized. Getting rid of these non-primitive concepts reduces the DKLisp standard by 5.4%.

Boolean and Character Data Types

Some modern languages—e.g., Ada—define the Boolean type and the character type by means of enumeration. In other words, these types are defined by exhaustively listing their members, which

[6]The Ada language has far fewer *ad hoc* features than a cursory perusal would indicate.

types are guaranteed by the semantics of the language to be disjoint from all other types. Other languages—e.g., C—define the Boolean type and the character type as subranges of the integers, where the overlap of the types is at least tolerated, if not intentional. Common Lisp has no separate Boolean type, and primitively defines characters as a separate type. One could define the Common Lisp characters non-primitively as an abstract data type whose representation utilized a subrange of the fixnums; if Common Lisp were then required to offer specialized vector representations for this subrange of the fixnums, and if the type system were efficient enough, this scheme would work. In spite of its elegance, we have not recommended this course for DKLisp because it is not clear that the amount of work involved in such a definition might not be greater than simply making characters primitive. On the other hand, such a scheme would generalize to non-ASCII character sets (e.g., ANSI-C "multibyte characters") much more easily than the current Common Lisp scheme.

We recommend that DKLisp abandon the traditional Lisp practise of unifying Boolean "false" with the empty list `NIL`, and unifying "true" with anything non-`NIL`. In other words, we recommend that DKLisp take up Scheme's original suggestion of using `#t` and `#f` to mean Boolean "true" and "false" respectively. There are several good reasons for this. First, the practise of primitive predicates returning arbitrary values encourages the user to do the same, putting a needless load on the garbage collector to recycle this garbage. Second, the use of specialized Boolean values allows the compiler to do more optimization, since it has more detailed information about the possible value types that may occur at run-time [Baker90TI]. Third, the use of specialized Boolean values allows for more error-checking at run-time, since random values will not be misinterpreted as true. Finally, specialized Boolean values make a program more readable, since the reader doesn't have to perform mental flow analysis to determine whether a `NIL` means an empty list or a "false". To summarize, we recommend the addition of a new DKLisp type `boolean`, whose only members are the objects `#f` and `#t`; these objects are not Lisp symbols, so they do not have, e.g., property lists. We further recommend that "`()`" (= `NIL`), the empty list, *not* be a DKLisp symbol. With these changes, DKLisp can decommission the symbol `t` back to civilian (i.e., non-special, non-constant) status.

Primitive Data Types

The primitive numerical functions on integers include `minusp`, `zerop`, binary `+`, unary `-`, binary `*`, unary `/`, binary `floor` ("returning" 2 values—discussed later), and binary `logand`. The non-primitive numerical functions include `plusp`, `=`, `>=`, `<=`, `>`, `<`, unary `+`, binary `-`, binary `/`, `min`, `max`, `abs`, `mod`, `gcd`, `lcm`, `oddp`, `evenp`. The rational numbers are easy non-primitive applications of object-oriented programming to the integers; the complex numbers are non-primitive applications of object-oriented programming to the floats and rationals.[7] The non-primitive character and string functions include `char<`, `char>`, `string<`, `string>`, `string-upcase`, `string-downcase`, `string-less-p`, `string-greater-p`. Separate character-ordering predicates are redundant, since this ordering is defined in terms of the injective function `char-int`. Strings themselves are non-primitive, since they are vectors of characters. The only primitive string concepts are: 1) specialized vectors of characters exist, and 2) the double-quote notation maps into them.[8] We reiterate that non-primitive does not imply a function call, because an implementation is free to compile non-primitives into a single machine instruction, if they wish, so long as the function has not been declared `notinline`. Eliminating these non-primitive concepts reduces the DKLisp standard by 8%.

---

[7]"Die ganzen Zahlen hat der liebe Gott gemacht, alles andere is Menschenwerk" ("Integers are God's creation, all else is the work of Man") — Leopold Kronecker.

[8]The external mapping is non-primitive and depends upon the primitive concept of `readtable`.

Fixnums and Integers

The primitive operations of `fixnums` must be slightly different from those of the integers, to avoid overflow problems. Thus, binary `<` and `=` are primitive instead of `minusp` and `zerop`, and binary `-` is primitive rather than unary `-`.

We recommend that arbitrary precision integers *not* be primitive in DKLisp.[9] There are two reasons for this: 1) many primitive uses of integers—e.g., `do` loops—involve only small integers—i.e., `fixnums`—and must be extremely efficient; and 2) DKLisp implementations in which large integers—i.e., `bignums`—are programmed in DKLisp itself will be far more portable. If the DKLisp standard follows the rest of our suggestions, then a DKLisp implementation of `bignums` will be as efficient as one written in C. The primitive operations required for `fixnums` can be found in any microprocessor machine language manual—add-with-carry, subtract-with-carry, double-shift, double-multiply, double-divide, etc. Since many of these operations produce two `fixnum` results, and since we agree with DKLisp's policy not to support Common Lisp's "multiple values", we must provide an efficient mechanism for DKLisp to accept these values.

We recommend the following programming style for these multiple-result primitives.

```
(fixnum-add-with-carry (the fixnum x) (the fixnum y) (the fixnum c)
  #'(lambda (sum-low sum-high)
       (declare (fixnum sum-low sum-high))
       << do something with low and high parts >>)) =>
<< returns (single) value returned by inner lambda expression. >>

(defun fixnum-add-with-carry (x y c f)
  (declare (fixnum x y c) (ftype f (fixnum fixnum) t))
  ;;; Has these semantics, but is implemented far more efficiently.
  (let ((n (1+ most-positive-fixnum)))
    (funcall f (mod (+ x y c) n) (floor (+ x y c) n))))

(fixnum-divide-with-remainder
  (the fixnum dividend-low) (the fixnum dividend-high) (the fixnum divisor)
  ;;; A decent compiler can figure out the fixnum typing itself.
  #'(lambda (quotient remainder)
       (declare (fixnum quotient remainder))
       << utilize quotient and remainder here >>)) =>
<< returns (single) value returned by inner lambda expression >>

(defun fixnum-divide-with-remainder (dividend-low dividend-high divisor f)
  (declare (fixnum dividend-low dividend-high divisor)
           (ftype f (fixnum fixnum) t))
  ;;; Has these semantics, but is much more efficient.
  (let ((dividend (+ (* dividend-high (1+ most-positive-fixnum))
                     dividend-low)))
    (funcall f (floor dividend divisor) (mod dividend divisor))))
```

Since a DKLisp compiler can recognize `fixnum-divide-with-remainder` as a primitive, it can open code not only this primitive, but also the (usually constant) lambda-expression given as its 4th argument,[10] resulting in nearly optimal machine code (optimal code requires `dividend-low` and `dividend-high` to reside in adjacent machine registers). By providing for efficient `fixnum` operations, as well as efficient `simple-vectors` of `fixnums`, a DKLisp `bignum` implementation should pay no penalty relative to a "native" `bignum` implementation. For example, the Symbolics

[9]"Die ganzen Fixnumen hat der liebe McCarthy gemacht, alles andere ist Hackerenwerk"—Baker.

[10]Ada compilers routinely open-code "generic" subprograms like these.

3600 efficiently implemented `bignums` in Lisp in a like fashion, although it utilized hardware multiple values instead of our open-coded closures.

Floats

Floats of several precisions and their operations must be primitive, since IEEE floating point datatypes are provided by most modern hardware, and since there are no simple algebraic relationships among the various IEEE functions. DKLisp's use of the term "real" instead of the more accurate "float" is embarrassing and should not be tolerated in a Lisp standard, because it advertises to the whole world the Lisp community's ignorance of mathematics. We strongly recommend that DKLisp eliminate dynamic *floating point contagion* in favor of a statically computable coercion policy. Floating point contagion is almost never what a system builder wants, and the mere possibility of such contagion eliminates a number of effective optimizations. Similarly, complex contagion—usually the result of inverse irrational functions such as `sqrt` and `asin`—produces similar uncertainty as to the result datatype. It would be better to define `sqrt` and `asin` to *always* produce a complex number, and then the programmer can coerce it back to a `float` himself by throwing away its imaginary part when it is identically zero. Of course, this policy requires that `sqrt` of a non-negative number have an imaginary part which is identically zero, and that `asin` of a number in the range $[-\pi,\pi]$ have an imaginary part which is identically zero; this policy is as easily arranged as the current Common Lisp contagion policy.

The default external representation is unreasonably inefficient for the transmission of floating point numbers, because it requires that floating point numbers be converted to decimal and reconverted when read back in. While there exist routines [Clinger90] [Steele90] to perform these conversions without the loss of even one bit, they require great subtlety and the conversions cannot be as efficient (i.e., $O(n\bullet \log n)$ versus $O(n)$) as when a binary or hexadecimal external representation is used. We therefore recommend that DKLisp define a non-decimal external representation for floating-point numbers,[11] so that efficient, lossless external representations can be used.

Functional/Immutable CONS cells

We recommend that DKLisp provide for only immutable `CONS` cells; i.e., `RPLACA` and `RPLACD`, along with all of the "`NXXX`" destructive list functions, should be eliminated from Lisp. Since eliminating these ancient capabilities from Lisp is certain to be controversial, we have given a thorough discussion of the issues in a 30-page paper "Equal Rights for Functional Objects" [Baker93ER], whose arguments are briefly summarized here.

A major issue is engineering pragmatism: if most `CONS` cells are functional, and if the mutability of `CONS` interferes with their optimization, then the majority of `CONS` cells are paying for the capabilities of the few. The efficiency of `CONS` cells is paramount because they are intimately involved with the implementation of Lisp itself—Lisp programs are represented by `CONS` cells, and Lisp `&REST` arguments are lists made of `CONS` cells. The mutability of `CONS` cells exhibits itself in the treatment of source code expressions like `(QUOTE (a b c))`. Since `CONS` cells are mutable, the compiler cannot assume that the value of this expression is truly constant, and cannot conclude that `CAR` of the expression will always be the symbol `a`.

The intimate relationship between `CONS` cells and argument lists shows itself in the treatment of the following Common Lisp expression:

```
(let ((x '(a b c))) (eq x (apply #'(lambda (&rest foo) foo) x)))
```

If the expression is true, then the function `apply`'ed can side-effect `apply`'s last argument, while if it is false—i.e., either `apply` or `&rest` (or both!) copies the list—then much copying is required, even when there are no side-effects. If `apply`/`&rest` always copies, then this is a source of considerable inefficiency, while if `apply`/`&rest` does not copy, then the compiler must generate

[11] Ada [Ada83] defines a syntax for floating point numbers that utilizes an arbitrary radix.

code for the worst case scenario. Immutable list cells allow sharing without side-effects, providing the best of both worlds.

In a parallel Lisp system, the relative costs of mutable `CONS` cells versus immutable `CONS` cells are much greater than in a serial system. Every `CONS` cell becomes a shared database which must be protected by a synchronization protocol. Even seemingly insignificant violations of strict access sequentiality in a shared-memory machine[12] can exhibit bugs which are highly obscure and almost impossible to find. A "remote procedure call" in Lisp which attempts to preserve Lisp semantics must either simulate a cache coherency protocol, or face the cost of laboriously and sequentially querying for the `CAR` and `CDR` of every `CONS` on an argument list.

Preserving `CONS` coherency in a *persistent* Lisp system involves the same problems as those in a parallel Lisp system—one can never copy an entire list in one "transaction", but must query each list element in turn.

The simplest and most direct solution to all of these problems is to make all `CONS` cells immutable; those wanting traditional side-effectable cells can "roll their own" utilizing `defstruct`. Then, Lisp source code is constant, quoted constants are constant, list sequences are constant, argument lists are constant, etc. Since we have already argued for immutable strings (the default) and immutable arrays (at the programmer's option), many common argument "lists" involving numbers and strings can be legitimately shared. This constancy means that lists no longer need to be constantly copied to avoid the small chance of a side-effect, but need be copied only when it is convenient. The constancy of immutable lists means that additional representation tricks can be played—e.g., a list can be represented by a number of adjacent cells—i.e., a vector—without any worry about that `RPLACD` may spoil the scheme.

Immutable `CONS` cells cannot be used to create circular lists without other kinds of objects intervening—e.g., a mutable vector, and therefore `PRINT` can be somewhat simplified. The predicate `EQL` becomes equivalent on immutable list cells to the predicate `EQUAL`, which never was well-defined on mutable list cells. Similarly, `EQL` and `EQUAL` are equivalent on all immutable objects, which implies that `EQUAL` is redundant and can be eliminated.

We therefore have the following definition of `EQL` (see [Baker93ER] for more details):

```
(defun eql (x y)
  (and (eql (type-of x) (type-of y))
       (cond ((atom x) (eq x y))           ; symbols & "immediate" values
             ((mutablep x) (eq x y))        ; mutablep(x) iff mutablep(y)
             (t (every #'(lambda (component)
                          (eql (funcall component x) (funcall component y)))
                          (components (type-of x)))))))
```

Vectors and Sequences

We recommend that only *simple* (i.e., non-filled, non-adjustable, non-displaced) vectors of the primitive data types be defined by DKLisp. In other words, we recommend that DKLisp define homogeneous simple vectors of `characters`, `fixnums`, and `floats` of each provided precision, in addition to non-homogeneous type `t` vectors. Note that we recommend against providing primitive `simple-bit-vectors` as they can be built efficiently from `character` or `fixnum` vectors [Baker90BV], and we go further than Common Lisp in requiring `fixnum` and `float` vectors.

The more sophisticated kinds of vectors and arrays in Common Lisp can be easily and efficiently implemented by means of object-oriented programming and generic functions, so long and the rest of our recommendations are followed. For example, `aref` becomes a generic function which dispatches on the type of its first argument to discover the number of dimensions which must be supplied; this dispatch requires that `array-rank-0`, `vector` (= `array-rank-1`), `array-rank-2`,

---

[12]E.g., in the newest generation of workstations.

`array-rank-3`, etc., all be subtypes of `array`.[13] The syntax of many of the vector and array functions of Common Lisp must change, because they do not conform to standard generic function mechanisms or meet other important criteria.[14]

We strongly recommend that DKLisp provide for both *mutable* and *immutable* vectors, which is an extension of ANSI Common Lisp.[15] Mutable vectors are the traditional Common Lisp vectors, while immutable vectors are new to DKLisp.[16] Immutable vectors of characters are most useful as constant strings—e.g., the strings which appear as `symbol-names`. Since constant strings may be referenced without having to be copied, they can be used in a large number of places where safety would otherwise require copying—e.g., `symbol-name`, `subseq`. Since the vast majority of strings in Common Lisp programs are never side-effected, the trade-off between safety and efficiency has been drawn in the wrong place, thus slowing down all but a small fraction of the applications of strings.[17] We further recommend that character strings which are read in using double quotes ("") be classified as constant vectors of characters. This policy solves the problem of non-constant constants, and eliminates any worry about whether an implementation optimizes the space for its character strings by sharing. In an analogy to `fixnums` and `integers`, "short" functional DKLisp strings of perhaps 1-4 characters can even be an "immediate" datatype, in which case individual characters no longer need be primitive, as they are simply immediate strings of length 1.

Immutable vectors of non-characters can be extremely useful as lookup tables, which can be passed *in toto* without worrying about the callee side-effecting the table. Since it is impossible to incrementally create constant vectors, they are normally created by coercion from a (non-constant) sequence.

While immutable strings and vectors are a convenience in serial systems, they are a necessity in parallel systems. This is because almost all parallelism is lost unless one can somehow prove that there is no conflict, and these assurances in a totally mutable environment are almost impossible to achieve.

The default external representation for vectors in Common Lisp is unreasonably inefficient, because it requires that the entire vector be read in first into a temporary form of indeterminate length, which must then be copied to the final object once its length is known. Worse still, the default sequence for this temporary form is list structure, whose small cells are relatively expensive to garbage collect. DKLisp should define a syntax which optionally includes the vector length at the very beginning. If this length is supplied on input, then the final vector can be immediately allocated and filled in without any additional garbage collector overhead. If DKLisp follows our recommendation by providing immutable strings, then it must supply a primitive function to return the next n characters read from a stream as an immutable string—e.g., using Unix/ANSI C `fread` [ANSI-C].

There are several problems with DKLisp sequences. It is not clear whether `subseq` applied to a list can return a portion of the list, or must always create a new list; for immutable lists this distinction would not matter. Similarly, for immutable vectors, it would be acceptable to return a displaced vector, rather than a copy of the original. The syntax of `concatenate` in DKLisp (and Common Lisp) does not conform to the requirements for a CLOS generic function, because CLOS generic

---

[13]Common Lisp's syntax allowing arrays of arbitrary rank is elegant but impractical. Common Lisp requires the support of ranks 0-7, but it is a very rare program that utilizes ranks 0,4,5,6,7.

[14]E.g., Common Lisp screwed up in its definition of `:initial-contents`, in that a multidimensional array cannot be copied by the straight-forward expression:

```
(make-array (array-dimensions x) :element-type (array-element-type x) :initial-contents x).
```

[15]ANSI-C provides the `const` modifier to indicate immutability of (a portion of) a variable.

[16]MacLisp [Moon74] provided a function `purcopy` to coerce vectors to immutability.

[17]See the Gabriel "browse" benchmark [Gabriel85] for a typical example of excess string consing.

functions can dispatch on an object using `eql`, but not on a type specifier, which would require a `subtypep` dispatch. The syntax must therefore be changed, but the solution is trivial—`concatenate` is a generic function of exactly two arguments, which is initially supplied with methods for the simpler kinds of vectors and lists.

Lambda and let bindings

We recommend that variables bound with `lambda` and `let`/`let*` utilize the typing syntax of `defmethod`, since it is more obvious and less ugly than the `declare` syntax of Common Lisp. Furthermore, it becomes clear that a function can only be called with arguments of the specified types. In essence, `lambda` should be an anonymous generic function with a single method, and `let`/`let*` should be defined in terms of `lambda`. Of course, if `let`/`let*` creates only immutable bindings, then there is no need to specify a type/class for those variables, since the type/class is obvious from the bound value. Note that `defun` thus becomes a synonym for `defmethod`.

```
(defun factorial ((n integer))
  ;;; This is so much prettier and obvious than (declare (integer n)).
  (if (zerop n) 1 (* n (factorial (1- n)))))
```

We strongly agree that the function declaration `defun` always produce an immutable binding. Lisp has other mechanisms for handling function *variables*, and the uncertainty surrounding the constancy of the function name binding inhibits many optimizations [Gabriel91]. Immutability of this function binding does not mean that this binding cannot be shadowed; `labels` and `flet` can both shadow a `defun`-bound function name.

We strongly recommend that the formal parameters of DKLisp lambda-expressions be immutably bound, and similarly for `let`/`let*` variables. In other words, `setq`/`setf` should not be legal on such bindings. Mutable bindings should be introduced by a different syntax—perhaps `let-mutable`. There are several reasons for this restriction. First, having mutable bindings as the default legitimizes and encourages the gratuitous use of side-effects. Second, a compiler or CASE tool must look at the entire body of the `lambda-` or `let-expression`, which requires expanding all macros, in order to determine if `setq`/`setf` is every used on the variable before it can determine the mutability of the binding and the types bound to the variable.[18] The body of the expression must then be re-expanded, since this information may affect the expansion (remember, we may have generic functions present), and such re-expansions carried to several levels can blow up the compile/macro-expansion by at least $O(n^2)$, if not $O(2^n)$. Third, a human reader of the code should not be required to scan through the code to determine whether the binding is mutable; the immediate indication of mutability is a great documentation aid. Fourth, a mutable lexical binding involved in a closure requires additional overhead; the different syntax signals that the programmer is aware of this overhead and is willing to accept it.

We believe that DKLisp should define the order of evaluation of multiple arguments to a function (and the ordering of evaluation for `let`-expressions, as well). In a language which allows side-effects, specifying this order is necessary to define the semantics of the nested composition of expressions. While Common Lisp and many other languages (Ada, C) have demurred on this important point, argument ordering is far more important in Lisp than in other languages, since virtually all imperative operations take place as side-effects of nested function calling. The usual reason for the lack of specificity for the argument evaluation ordering is to provide a compiler with more freedom for optimization. However, if DKLisp accepts our other recommendations regarding more functional binding mechanisms and data structures, then a DKLisp compiler will already have a wide range of possibilities for optimization due to the Church-Rosser properties of completely functional expressions, and should not need the additional latitude of unspecified argument

[18]At least mutability for lexical variables is decidable; mutability for "special" variables is not.

ordering. We have little preference regarding which ordering (left-to-right or right-to-left) is chosen,[19] so long as DKLisp chooses only one.

Macros

The lack of macros in DKLisp is truly distressing. The power of traditional Lisp macros is so immense and so useful, that the users and designers of other popular languages (C, C++, Ada) cannot begin to comprehend what their languages are missing without a decent macro system. In the most modern terminology, *Lisp macros are the ultimate CASE tool*. The expert Lisp programmer rarely writes any Lisp code himself, because his macros produce it all for him (see [Baker90BV] [Baker91PP]). In an industrial setting in which programs have to get written, debugged and delivered *in spite of system bugs and inefficiencies*, the macro is a miracle. Bug fixes and compiler kludges can be hidden inside macros, instead of cluttering up the code with a large number of `#ifdef's` and `#+/#-'s`. Macros can provide an entirely new user interface and language—without losing anything in efficiency.

Granted, many traditional uses of Lisp macros can be more cleanly handled by means of functions declared `inline`, without giving up any efficiency.[20] In other cases, however, it is hard to imagine a compiler smart enough to perform certain optimizations, even after inlining. Take the Common Lisp `map` sequence function, for example. This function takes a function and an arbitrary number of sequences as arguments, and produces a new sequence as a result. Unfortunately, one cannot in general determine until run-time which of the sequences are vectors and which are lists, and the lists and vectors can be intermixed. All Common Lisp implementations include a functional implementation of sequence `map`, complete with `&rest` arguments, but they either reference the sequences using `elt`—making them hopelessly inefficient on lists—or reference the sequences via functional "steppers", making them quite inefficient for both lists and vectors. We know of no type inference/constant propagation techniques which can be applied to `map` in such a way that `map` *automatically* generates optimal code when inlined. A macro (or more particularly a `compiler-macro`), on the other hand, can very simply *look* at the argument list for `map` at expansion time,[21] count the number of sequence arguments, and (most of the time) tell by looking which are vectors and which are lists. In this way, the macro can easily generate nearly optimal code for `map` in a large number of common cases.

Of course, Common Lisp systems can achieve efficiency for `map` without utilizing macros. However, the additional knowledge about how to expand `map` must be stored somewhere other than in the functional version of `map`; it can be stored either in spaghetti code in the compiler, or it can be associated with the symbol `map` in some way. The more modular approach is to associate the expertise about `map` with the symbol `map`; in this case, one might as well utilize a `compiler-macro` which can be as smart about `map` as you please, and can also be compiled, so that it runs quickly during macro-expansion.

The Lisp tradition (at least until Common Lisp), however, is to assume that what is good for the implementation is probably good for the user, since many users of Lisp are themselves in the system building business. Therefore, if a Common Lisp system achieves its own efficiency with a `compiler-macro` capability, then this capability should be offered for user functions, as well.

---

[19]We have found [Baker92CONS] that efficiency may be slightly better with a right-to-left ordering, but some may object that programs written in this style are counter-intuitive, unless operations like `reverse-funcall` are defined.

[20]Traditional Lisp macros can provide very nearly the correct semantics for function inlining when variables are dynamically-scoped, but fail miserably for lexically-scoped variables.

[21]Of course, if `map` is `apply`'ed to an argument list known to the mapping function, side-effects can produce havoc; this is one reason for our recommendation for functional/immutable lists.

Of course, it is possible to write bad macros which are not referentially transparent and have other problems, but the solution to these problems is not to throw macros away.[22] DKLisp should provide better alternatives such as inlining and hygienic macros [Kohlbecker86] rather than "throwing out the baby with the bath water".

Lisp Source code consists of S-expressions, not sequences of characters

DKLisp has wrongly focussed its standardization of Lisp source code as a *character string*, rather than as a Lisp *S-expression*; i.e., `(defun foo (x) (1+ x))` is treated as a sequence of 22 characters instead of as a Lisp *list* of 4 objects, the first two of which are *symbols*, and the last two of which are *sublists*.[23] The fact that such a list has an external representation as 22 characters is completely irrelevant, because there are many external representations mapping into the same S-expression. Given immutable CONS cells, however, there is only one S-expression representation.

By focussing on the character string, rather than the S-expression, form, DKLisp forgets the importance of the S-expression form, and the fact that most of the S-expressions compiled or executed by DKLisp will not have been touched by human hands, but will have been created by macros and other S-expression-generating functions. Furthermore, the S-expression form is far more useful for storing libraries of source code in a *persistent* DKLisp than is a character form.

The traditional means for specifying scope in Lisp programs is via parentheses—i.e., by means of the nesting of S-expressions themselves. Traditional Lisp systems violated "parenthesis-mediated scoping" only for very serious reasons—e.g., an early LAP (Lisp Assembly Program) format for a compiled function utilized a *sequence* of S-expressions terminated by `NIL` rather than a more obvious *list* due to space limitations. Modern Lisps have no such space limitations, and therefore they should utilize nested expressions exclusively.

DKLisp violates this nesting principle through its use of `in-package` and `end-package` to bracket sections of code which should be bracketed instead by parentheses. The Common Lisp-84 scheme for specifying packages to avoid symbol name collisions in a large system was very kludgy, as the revised report admitted.

---

[22]According to one cynical grey beard, "macros are good for me, but not for you".

[23]Common Lisp-84 was also bitten by this error—e.g., there was no S-expression equivalent for the "`#,`" notation for load-time evaluation. Sensibly, Common Lisp-90 eliminated "`#,`" in favor of the `load-time-value` special form.

No Lisp standard has yet taken advantage of lexical scoping's ability to avoid name collisions, as we do in the following code:

```
;;; This form exports global-function1 through global-function9.
(let-mutable
  ((local-variable1 <initial-value1>) ; Some load-time initialization here.
   (local-variable2 <initial-value2>)
   ...
   (local-variablen <initial-valuen>))
 (labels
  ((local-function1 <args> <body>)
   (local-function2 <args> <body>)
   ...
   (local-functionn <args> <body>))
  << Do more load-time initialization here. >>
  (defconstant global-function1 #'local-function3) ; export these functions
  (defconstant global-function2 #'local-function5)
  ...
  (defconstant global-function9 #'local-function13)
  << Do more load-time initialization here. >>))

;;; This form imports global-function1 through global-function9.
(flet
 ((local-function1 (&rest args) (apply global-function1 args))
  (local-function2 (&rest args) (apply global-function2 args))
  ...
  (local-function9 (&rest args) (apply global-function9 args))
 (declare (inline local-function1 ...))
 << whatever definitions and initializations are needed here. >>)
```

Of course, these examples are only suggestions to spur the imagination, and their syntax could be cleaned up with a few macros, but they at least exhibit the appropriate semantics. We therefore recommend that DKLisp drop the use of "packages"[24] and better utilize the power of lexical scoping to solve the name-collision problems of large-scale systems.

Object System

We are heartened to see an object system ("Gauss") as part of DKLisp. Nevertheless, we propose a number of changes, both to orthogonalize and to simplify this system.

The CLOS (Common Lisp Object System) documents never made it clear that the mechanism for defining polymorphic functions—`defgeneric`/`defmethod`—is completely independent of the mechanism for defining class structures—`defclass`. This is because the operation of generic functions is controlled entirely by the *types* of the function's arguments, and the type system is only incidentally updated during the definition of a new class by `defclass`. Thus, one can conceive of a Common Lisp subset with generic functions but no classes, as well as a subset with classes but no generic functions.

In fact, we have built [Baker91CLOS] a single-inheritance mini-CLOS with generic functions, using `defstruct` instead of `defclass` to define "things with slots". There are significant efficiency advantages to this approach, since one is not saddled with the whole "metaobject protocol" [Bobrow88] [desRiviéres90], as well as with the full complexity of `defclass`. Furthermore, since `defstruct` can already incrementally extend an existing structure using single inheritance, and since the real power of object-oriented programming resides in generic functions, we gain in simplicity, while losing almost nothing in power.

---

[24]Packages were originally an implementation kludge of MacLisp, where they were called "multiple obarrays", and their semantics never advanced beyond the kludge stage.

One reason for preferring `defstruct` to `defclass` is `defclass`'s unfortunate dependence upon `slot-value` for its implementation. Whereas `defstruct` defines slot accessor functions (which do not have to be generic functions in a single inheritance system) which can be expanded inline into a single machine instruction, `defclass` defines slot access in terms of `slot-value`, which takes a symbol as its second argument. For a compiler to compile `slot-value` as efficiently as it does `defstruct` accessors, it has to know a lot more about internal data structures of the object system—e.g., the property list of the `slot-name`, or hash tables indexed by the `slot-name`—than `defstruct` does. Thus, `defclass` implements a lower level concept—a slot accessor—in terms of a higher level concept—a complex object data structure in an *abstraction inversion*. How much easier and more straightforward for a compiler to traffic in slot-accessor *functions* themselves (readers, writers and updaters[25]), and utilize the capabilities it already has for function inlining to achieve efficiency, rather than having the compiler know all sorts of specialized knowledge about the object-oriented system's data structures.

Another reason for preferring `defstruct` to `defclass` is `defclass`'s unreasonable dependence upon assignments and side-effects to achieve an initialized instance. *There is no mechanism in CLOS for creating an initialized functional/immutable object.* The `make-xxx` function generated by `defstruct`, on the other hand, performs both allocation and initialization together as an atomic action, making trivial the construction of both mutable and immutable objects. If CLOS offered some mechanism such as *I-structures* [Arvind89] or *linear logic* [Baker92LLL] for protecting a newly allocated, uninitialized object from being referenced outside of its "nursery", then the dependence upon `initialize-instance` might be bearable, but CLOS offers no such assurances. We sympathize with the problems of getting instances initialized in a modular fashion [Baker91CLOS] [Baker91OOAda], and the solution may be to generate a number of temporary sub-instances which are immediately thrown away after the final instance has been created. While a straight-forward runtime implementation of this idea would place great demands upon a "generational" garbage collector, a combination of compiler optimizations and greater use of "lazy (stack) allocation" [Baker92CONS] for the temporary sub-instances could reduce the load on the garbage collector.[26]

CLOS generic functions depend upon a partial order of Common Lisp types, and utilize `typep` and `eql` to determine (at runtime) the applicable method. If an implementation wishes to determine method applicability at compile time, however, it must utilize the `subtypep` predicate. While an efficient decision procedure for `subtypep` exists [Baker92ST] for the full set of Common Lisp-90 type specifiers (excluding `satisfies`), it is not necessary or desirable that DKLisp allow the flexibility of Common Lisp-90 type specifiers as specializers in generic methods. In fact, DKLisp can utilize a relatively simple type specifier system which includes only class names and `eql` specializers, in which case the decision procedure for `subtypep` is much simplified.

The ability to efficiently compile CLOS generic functions depends upon the compiler's knowledge of the type system. If the compiler has complete knowledge of the type system—i.e., it is static—then the compiler can do a good job of optimizing generic function application. Common Lisp-84 `defstruct` must be used in an essentially static way, since slot modifications are performed via `setf`, which requires compile-time knowledge of the slots. Thus, `defstruct` is an ideal mechanism for statically defining the "things with slots".

There is a significant problem with `defstruct` (and also `defclass`), as discussed in both [Baker92ST] and [Baker91CLOS]. One is the inability to specify that *a structure type/class has no direct instances*, and is to be used only for extension—i.e., the creation function `make-xxx` must not be defined. This class attribute is critical to the concept of *synthetic classes*, which have one or

---

[25]In a parallel system, readers and writers are not enough; atomic updaters are also important.

[26]Other operations on immutable objects also place a greater load on the garbage collector; e.g., functional arrays use the garbage collector to perform their side-effects for them [Baker91SB].

more subclasses, but no direct instances. For example, the type/class `integer` is the synthetic union of the classes `fixnum` and `bignum`, yet `integer` has no instances of its own—every particular integer being either a `fixnum` or a `bignum`. The predicate `subtypep` must be told not only of the subset relations of `fixnum` and `bignum` to `integer`, but also of the fact that there are no other `integers`, before `subtypep` can conclude that `integer` is a subtype of the union of `fixnum` and `bignum`. *The ability to define synthetic classes in CLOS is analogous to the ability to define virtual functions in C++* [Stroustrup86], although the synthetic class concept is more elegant.

Below, we define the Common Lisp number classes utilizing synthetic classes.

```
(defsynclass number (t)      ; number has the superclass of all Lisp objects
  )                          ; but no slots[27] and no direct instances.

(defsynclass complex-rational (number) ; the exact (non-float) numbers.
  )                          ; has no slots and no direct instances.

(defclass complex-imaginary-rational (complex-rational) ; non-real rationals.
  realpart imagpart)         ; 2 slots and presumably many instances.

(defsynclass rational (complex-rational) ; real rationals.
  )                          ; has no slots and no direct instances.

(defclass ratio (rational)   ; non-integral real rationals.
  numerator denominator) ; 2 slots and presumably many instances.

(defsynclass integer (rational) ; integer real rationals.
  )                          ; has no slots and no direct instances.

(defclass bignum (integer)   ; integers of large absolute value.
  rep-vector)   ; has a representation vector and presumably many instances.

(defbuiltinclass fixnum (integer) ; integers of small absolute value.
  )       ; this definition is used only for the sake of subtypep/subclassp.
```

Our static implementation of a generic function [Baker91CLOS] translates it into a set of nested `typecase` expressions[28] which dispatch on the various required arguments. An optimizing compiler capable of good type inference will be able to eliminate some or all of the arms of the `typecase` expression utilizing a precise `subtypep` predicate. If only one arm of a `typecase` remains, then no runtime dispatch is necessary, and the generic function has been "resolved" to a single method.[29] The net result of good static type inference and a static type system (compliments of `defstruct`) is the ability to statically resolve many methods at compile-time and obtain generic function efficiency similar to C++.

Streams, Files and Pathnames

One of the attractive innovations of Common Lisp is its large number of `with-xxx` macros. These expressions initialize an object, perform some user-specified operations on the object, and then finalize the object. These "with" forms are particularly useful when dealing with streams—especially I/O streams—because the form of their nesting guarantees that an object (e.g., file stream) will always be initialized and finalized when it is used. Since the opening and closing of a file object is a dynamical action analogous to the binding of a dynamical variable, it is not clear what

[27]Synthetic classes may have slots, although we do not show an example here.

[28]`case` expressions must also be used in the presence of eql specializers.

[29]The actual scheme is somewhat more complex due to CLOS *method combinations*—e.g., `next-method-p` and `call-next-method` must be removed by inlining—but the principle is clear.

purpose is served by unpaired operations (`open`, `close`) on files. As a result, we recommend that DKLisp define only `with-open-file` and its friends, and not provide for separate creation, opening and closing operations for file objects. Not only does this simplify DKLisp, but it also eliminates a significant source of bugs (see [Baker91SP] for a similar capability in Ada).

We recommend defining `with-open-file` as a *function*, rather than a macro, as in the following code:

```
(with-open-file "/usr/foo.bar" :input
  #'(lambda (input)
       (declare (stream input))
       << read from input and do things >>)) =>
<< returns value returned from lambda-expression. >>
```

In a multithreaded environment, the side-effects of stream operations can become an embarrassment. We therefore recommend a lower-level notion of stream—perhaps called `primitive-stream`—which does not incorporate a stream position pointer. A read operation on a `primitive-stream` takes both the primitive-stream and a stream index, and "returns" (using a kind of continuation-passing described earlier under `fixnum` arithmetic) both an object and a new stream index. Under this protocol, a read-only stream has no state at all and is completely functional, while a write/update/append `primitive-stream` has only the state of the stream bits, and not a position pointer.

While we are on the subject of I/O, we should point out the abject obsolescence of global variables (e.g., `*package*`, `*print-base*`) to control the operation of `read`, `print` and `format`. In an environment with multiple read streams and multiple write streams, the use of these global variables was already a nightmare, but they are truly disastrous in a multithreaded environment. The context controlling the operation of each stream must be unique, which can only happen if the information is passed to these functions explicitly in the form of arguments, or implicitly as attributes of the I/O stream itself. The elimination of these global variables will not only simplify DKLisp, but make it a more bug-free programming environment.

The definition of DKLisp pathnames is a trivial application of `defstruct`, and should be relegated to an appendix.

Futures

We recommend that DKLisp standardize on *futures* [Baker77] as a mechanism for programmer-specified multithreaded tasking. Futures have been implemented in several different Lisp systems [Gabriel84] [Halstead85] [Allen87] [Swanson88] [Ito90], and have been shown to provide for most of the standard forms of parallelism with a simple structured form. Futures and continuations seem not to get along very well [Katz90], but we feel that the problem lies in a continuation's ability to linearize the march of time and defeat parallelism.[30] DKLisp should have no such problems with `future`, since it does not provide continuations.

5. Conclusions

We have presented some criteria on which to evaluate a modern Lisp system, and have critiqued DKLisp against these criteria. While the existing DKLisp proposal comes much closer to meeting these criteria than does ANSI Common Lisp, we find that DKLisp still requires a large amount of work. The major areas of concern involve preparing DKLisp for a clean insertion into a modern MIMD parallel architecture, whether shared-memory or not. The most controversial of these changes involve converting DKLisp into a "mostly functional" language [Knight86], so that parallel threads will not interfere with one another. We advocate the use of the *future* as the basic threading primitive.

[30]Continuations are also known to interact poorly with Scheme's `letrec`.

Lisp and Symbolic Computation **4**,4 (March 1992),371-398.

Other significant changes involve making a smaller portion of the language primitive—especially in the area of numbers—and in integrating generic functions into the basic fabric of the language. We estimate that 35-50% of the current DKLisp document can be relegated to appendices describing libraries of datatypes, functions and macros based on more primitive concepts. We find that `defstruct` provides a more efficient and convenient data structuring primitive than does `defclass` for the purposes of a single-inheritance object-oriented hierarchy.

We strongly disagree with the standardization of the external character form of a program, rather than the internal S-expression form. Since many Lisp systems are constructed largely by Lisp macros, and since persistent Lisp systems are on the horizon, the internal S-expression form is far more convenient and efficient than is a character form. Similarly, we strongly disagree with the lack of macros in DKLisp, since they are one of the single most powerful CASE tools ever invented, and productivity for building large Lisp systems would be crippled by their loss.

References

Ada83. *Reference Manual for the Ada® Programming Language.* ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Allen, D.C., *et al.* "Recent Developments in Butterfly™ Lisp". *AAAI-87*, Seattle, WA, July, 1987, 2-6.

ANSI-C. *Draft Proposed American National Standard Programming Language C.* ANSI, New York, 1988.

Arvind, and Nikhil, R.S. "I-Structures: Data Structures for Parallel Computing". *ACM TOPLAS 11*,4 (Oct. 1989), 598-632.

Autodesk, Inc. "AutoLISP® Release 10 Programmer's Reference". Publ. TD111-005.2, Dec. 1988.

Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". *CACM 21*,8 (Aug. 1978),613-641.

Baker, H.G., and Hewitt, C. "The Incremental Garbage Collection of Processes". *Proc. ACM Symp. on AI & Prog. Langs., Sigplan Not. 12*,8 (Aug. 1977),55-59.

Baker, H.G. "List Processing in Real Time on a Serial Computer". *CACM 21,4* (April 1978), 280-294.

Baker, H.G. "Efficient Implementation of Bit-vector Operations in Common Lisp". *ACM Lisp Pointers 3*,2-4 (April-June 1990),8-22.

Baker, H.G. "The NIMBLE Project-An Applications Compiler for Real-Time Common Lisp". *Proc. InfoJapan'90 Conf.*, Tokyo, Japan, Oct. 1990, Info. Proc. Soc. of Japan.

Baker, H.G. "The Nimble Type Inferencer for Common Lisp-84". Subm. to *ACM TOPLAS*, 1990.

Baker, H.G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Progr.*, June 1990,218-226.

Baker, H.G. "Requiem for a Heavyweight Lisp; or, If It Ain't Baroque, Fix It". Unpubl. manuscript, June 1990.

Baker, H.G. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, H.G. "Pragmatic Parsing in Common Lisp". ACM *Lisp Pointers IV*,2 (April-June 1991),3-15.

Baker, H.G. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". ACM *Ada Letters XI*,5 (July/Aug. 1991),79-90.

Baker, H.G. "Shallow Binding Makes Functional Arrays Fast". *ACM Sigplan Not. 26*,8 (Aug. 1991),145-147.

Baker, H.G. "Object-Oriented Programming in Ada83—Genericity Rehabilitated". ACM *Ada Letters XI*,9 (Nov/Dec. 1991),116-127.

Baker, H.G. "CLOStrophobia: Its Etiology and Treatment". ACM *OOPS Messenger* 2,4 (Oct. 1991), 4-15.

Baker, H.G. "CONS Should not CONS its Arguments, or, A Lazy Alloc is a Smart Alloc". ACM *Sigplan Not. 27*,3 (Mar. 1992), 24-34.

Baker, H.G. "A Decision Procedure for Common Lisp's SUBTYPEP Predicate". *Lisp & Symb. Comp.* 5 (1992), 157-190.

Baker, H.G. "Lively Linear Lisp—'Look Ma, No Garbage!'". ACM *Sigplan Not. 27*,8 (Aug. 1992), 89-98.

Bobrow, D.G., and Kiczales, G. "The Common Lisp Object System Metaobject Kernel: A Status Report". *Proc. 1988 ACM Conf. on Lisp & Funct. Progr.*, Snowbird, UT, July1988, 309-315.

Brand, Heiner, *et al.* "An approach to the DIN Kernel Lisp Definition, Version 1.0". Subm. to SIO WG16, June, 1991.

Clark, D.W., and Green, C.C. "An Empirical Study of List Structure in LISP". *CACM 20*,2 (Feb. 1977),78-87.

Clinger, William D. "How to Read Floating Point Numbers Accurately". ACM *PLDI'90, Sigplan Not. 25*,6 (June 1990),92-101.

Cointe, Pierre. "Metaclasses are First Class: the ObjVlisp Model". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),156-167.

Cyphers, D.S., and Moon, D. "Optimizations in the Symbolics CLOS Implementation". *Proc. 3rd CLOS Users and Implementors Workshop, OOPSLA'90* (Oct. 1990),18-23.
des Riviéres, J., and Kiczales, G. "The Art of the Metaobject Protocol: A backstage look at CLOS implementations". Unpubl. man., Xerox PARC, Oct. 15, 1990, 203p.
Gabriel, R.P., and McCarthy, J. "Queue-Based Multi-Processing Lisp". *Proc. 1984 ACM Symp. on Lisp & Funct. Prog.*, (Aug. 1984),25-44.
Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA, 1985.
Gabriel, R.P. "Lisp: Good News, Bad News, How to Win Big". Unpublished memo, Lucid, Inc., Feb. 1991.
Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. McGraw-Hill, New York, 1983.
Halstead, R. "MultiLisp: A language for concurrent symbolic processing". *ACM TOPLAS 7*,4 (Oct. 1985),501-538.
Harper, R., *et al*. "Standard ML". Tech. Rept. ECS-LFCS-86-2, Comp. Sci. Dept., Edinburgh, UK, March, 1986.
IEEE-Scheme. *IEEE Standard for the Scheme Programming Language*. IEEE-1178-1990, IEEE, NY, Dec. 1990.
Ito, T., and Halstead, R.H.Jr. *Parallel Lisp: Languages and Systems*. Springer LNCS-441, 1990.
Katz, M., and Weise, D. "Continuing into the Future: On the Interaction of Futures and First-Class Continuations". *Proc. 1990 ACM Conf. on Lisp and Funct. Progr.*, Nice, France, June 1990, 176-184.
Keene, S.E. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, 1989.
Kim, W., and Lochovsky, F.H., *eds*. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989.
Knight, T. "An Architecture for Mostly Functional Languages". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986), 105-112.
Kohlbecker, Eugene E., Jr. *Syntactic Extensions in the Programming Language Lisp*. Ph.D. Thesis, TR-199, CS Dept., Indiana Univ., Aug. 1986.
Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*, 6 (June 1983),419-429.
MacLane, Saunders, and Birkhoff, Garrett. *Algebra*. Macmillan, 1967.
MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.
McAllester, D., and Zabih, R. "Boolean Classes". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),417-423.
Moon, D. *MacLisp Reference Manual, Rev. 0*. Proj. MAC, MIT, April 1974.
Moon, D. "Garbage Collection in a Large Lisp System". *ACM Symp. on Lisp and Functional Prog.*, Austin, TX, 1984, 235-246.
Novak, G.S.Jr. "Data Abstraction in GLISP". *Proc. SIGPLAN'83, Sigplan Not. 18*,6 (June 1983),170-177.
Queinnec, C., and Cointe, P. "An Open-Ended Data Representation Model for Eu_Lisp". *Proc. 1988 ACM Lisp & Funct. Progr. Conf.*, Snowbird, UT, 1988, 298-308.
Radin, George. "The Early History and Characteristics of PL/I". *ACM Sigplan History of Prog. Langs. Conf., Sigplan Not. 13*,8 (Aug. 1978),227-241.
Rees, J. and Clinger, W., *et al*. "Revised Report on the Algorithmic Language Scheme". *Sigplan Notices 21*,12 (Dec. 1986),37-79.
Steele, G.L.Jr. *Common Lisp, the Language*. Digital Press, Burlington, MA, 1984.
Steele, G.L.Jr. *Common Lisp, the Language: Second Edition*. Digital Press, Bedford, MA, 1990.
Steele, G.L.Jr., and White, J.L. "How to Print Floating-Point Numbers Accurately". *ACM PLDI'90, Sigplan Not. 25*,6 (June 1990), 112-126.
Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.
Swanson, M.R., *et al*. "An Implementation of Portable Standard Lisp on the BBN Butterfly". *Proc. 1988 ACM Conf. on Lisp and Funct. Progr.*, Snowbird, UT, July 1988, 132-141.
Swinehart, D., *et al*. "A Structural View of the Cedar Programming Environment". *ACM TOPLAS 8*,4 (Oct. 1986),419-490.
Taft, Tucker, *et al. [Ada-9X] DRAFT Mapping Document*. Ada-9X Proj. Rept., Feb. 1991.
Taft, Tucker, *et al. [Ada-9X] DRAFT Mapping Rationale Document*. Ada-9X Proj. Rept., Feb. 1991.
Wand, M. "A Semantic Prototyping System". *Proc. ACM Sigplan '84 Symp. on Compiler Constr., Sigplan Not. 19*,6 (June 1984),213-221.
Yuasa, T., and Hagiya, M. *Kyoto Common Lisp Report*. Research Inst. for Math. Sci., Kyoto U., 1985.