NIL - A Perspective

by Jon L White
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 01239
May 17, 1979

NIL is acronymic for "New Implementation of Lisp" (or possibly "Nil Is Lisp"). It is intended to be a modernization of the programming language LISP suitable in design for implementation on any of the current generation of large-address-space, low-cost computers; and maximally upward-compatible with MACLISP, the dialect of LISP developed at the M.I.T Laboratory for Computer Science, and Artificial Intelligence Laboratory. Additionally, NIL attempts to be as compatible as is possible with the LISP Machine dialect ("LISPM", also an upward-compatible extension of MACLISP - see [Weinreb 1979]), given the restraining condition that NIL be easily implementable on hardware such as the Digital Equipment VAX 11/780, the experimental S-1 developed at Stanford University under contract from the ONR, and some of the host of new 32-bit micros starting to appear now.

GOALS

To summarize the goals of the NIL project:

1) To properly utilize the architecture of current (and to some degree future) generation computers for the benefit of large systems built on top of LISP, and especially for MACSYMA; to design and use hardware instructions "tailor-made" for LISP, to achieve speed rather than economy of memory usage, and to expose in the language itself more of the capabilities of typical hardware (arithmetic options, operating-system interfaces, character-string processing instructions, etc.)
2) To make available a large-address space to the LISP programmer, of about 64M Q's (M = Mega, or a factor of 2^{26}; and a "Q" is a "pointer"-sized quantum of datum with 2 Q's stored in a cons cell); to provide, where possible, the capability for the NIL programmer to build sharable systems with sharable sub-segments.
3) To fix many of the historic, but annoying, little problems of the LISP language, while still retaining an essentially upward-compatible outlook; to implement some of the more modern features of programming languages such as "co-routines" and "closures", but probably not "spaghetti stacks", and an object-oriented sub-world of "actors" (see [Hewitt 1979]) which are similar to SIMULA's classes (see [Dahl 1968]); to provide good translators between NIL, MACLISP, LISPM, and possibly a future version of INTERLISP.

4) To build NIL in NIL; thus the implementation will be maximally machine and operating-system independent. We want a "virtual machine" design such that export to new machines will be quite easy; modulo (i) file system and I/O, (ii) operating system interface, and (iii) minor questions of efficiency, the same code, interpreter, support routines, and compiler should be applicable to all NIL incarnations.

The LISP Machine design depends essentially on having a large (8K of 48-bit words), flexible, user-writeable micro-code facility in the host computer - a criterion met only by the home-grown design of the CONS processors (see [Knight 1974], also [Greenblatt 1979]); the VAX has an optional user-writeable micro-code, and although not extensive enough to support the LISPM design, it may be able to provide a speed-up of a factor of two or more for ordinary NIL programs, as well as providing a hardware "hook" for some of the more esoteric research questions in programming languages (such as the "Baker GC", see [Baker 1977]). The S-1 has a more-fully designed micro-code, with more micro-store available, and much support for NIL, especially in regards to various kinds of arithmetic, will no doubt be put into its architecture. We will likely have to take the "micros" without hope of altering the order codes.

Previous incarnations of LISP were limited to 20K Q's in the case of LISP 1.5 (implemented on the IBM 704 and 7090 series computers, see [McCarthy 1960] and in the case of variants of it implemented on the IBM 360. The various PDP-10 LISPs, (MACLISP [Moon 1974], INTERLISP [Teitelman 1978], LISP 1.6, [Quam 1972] and ILISP [Bobrow 1972]) were generally limited to about 1/5M Q's, but lack of sufficient memory meant that 64K Q's was a more practical limit. Attempts to "overlay" programs (such as the INTERLISP "shadow space" concept, and similar ideas researched for MACSYMA's benefit in MACLISP) have not been spectacularly successful, primarily because the overlaying has been only for "compiled code" instruction bodies rather than for the attendant data structures; it is primarily the growth of the size of data structures that makes a larger address space necessary, although there are a few problems (especially in algebraic manipulation) that seem to exercise a large number of subroutine packages rapidly and thus require a large working-set of memory for code bodies too.

Extension of the variety of data types is one of the first things that LISP implementers want to do. NIL will use a "pointer" of at least 32 bits as its basic datum (compared to the 18 bits of MACLISP on the PDP-10), with at least 5 bits of the pointer used as a (primitive) type encoding. Some of the encodings will indicate immediate data, that is the remaining bits are not an "address" of stored data, but rather the sum total of the datum is in the type encoding and bits; other pointers will be the adjunction of the type code bits along with an address of some stored, structured datum. In the class of "immediate" data: [1i] FIXNUM - by clever encoding of the type bits (zeros in the two low order bits), we expect to get a 30-bit fixnum, and generally be able to use the standard fixed-point instructions supplied with the various machines, [2i] CHARACTER - by abstracting the notion of a "character", we avoid the problems about conversion between one character encoding and another (e.g. 7-bit ASCII, 8-bit ASCII, EBCDIC, S-1, etc), [3i] SMALL-FLONUM - an

abbreviated version of a floating-point number, suitable for many applications, with an exponent field of 7 or fewer bits, and a mantissa field of 20 or fewer, [4i] CONSTANT - certain distinguished constants with a fixed interpretation (for which symbols are not appropriate) and [5i] NULL - a distinguished code for the nullist, [6i] plus several internal kinds of markers deemed necessary to the smooth operation of the whole system. The encodings which represent "address" cover the remainder: [1s] PAIR - or "cons" cell, the main workhorse of lisp (but the predicate TYPEP will still call this class LIST), since two Q's are stored in a PAIR, then at least 64 bits are required, [2s] SYMBOL - another basic of all lisps, requiring 3 Q's to hold pointers to "pname", "property" list, and function-value cell linkages, [3s] FLONUM - of a size adequate for general scientific computation, which on most machines means "double precision", [4s] VECTOR - an indexable, no-overhead, 1-dimensional array of Q's for general use (perhaps many applications using LISTs now will use VECTORs in the future), [5s] STRING - an indexable string of "characters", which will be packed one character per machine "byte" (requiring an architecture with at least 7 bits per byte), [6s] BITS - an indexable sequence of bits (packed, of course, 1 bit per bit), [7s] SUBR - it seems prudent to be able to distinguish a pointer to a piece of executable code, [8s] EXTEND - a "vector" like structure with one extra Q of overhead which points to a "class" descriptor which is itself implemented as an EXTEND), a general user-accessible way to extend the data types and structures, especially suited for an ACTOR-like (or SIMULA-like), object-oriented programming, and providing the base over which certain important system concepts will be implemented (such as BIGNUM, BIGFLOAT, CLOSURE, STREAM, etc.); in particular the concept of ARRAY will be embedded in the EXTEND and arrays will be composed out of the more primitive data types VECTOR and BITS, with 1, 2, or 3 dimensions (and possibly higher), and it will be possible for two arrays to "overlap" in their data part in the manner of FORTRAN arrays.

As an afterthought, a "VM" (virtual machine) description of INTERLISP was produced [Moore 1976], ostensibly to help with the export of INTERLISP to machines other than the PDP-10. Unfortunately, a primary PDP-10 bias in the original design, the accretion of "features", and a heavy emphasis on I/O makes this document unwieldy.

Another attempt to "virtualize" the INTERLISP world, BYTELISP [Deutsch 1978], seems more fruitful, but does depend upon a "tailor-made" order code requiring about 1.5K of 32-bit micro-instruction storage on a cooperating computer (currently the XEROX "Alto" and "Dorado" computers). Some success has been achieved by the group at University of Utah working on Standard LISP in obtaining a flexible implementation which can be exported to other architectures easily (see [Griss 1978]). The M.I.T. LISP Machine is indeed a full operating system, for a single-user machine interacting with a network to other machines, written entirely in LISP, but here again the uniqueness and size of the micro-code is essential to the success. A straightforward transfer of the LISPM design to, say, a VAX (which has much less of a more limiting micro-store) might lead to a slow-down by a factor of 4 to 10, but a more accommodating approach might prove practical; indeed, NIL is such an "accommodating" approach.

New Semantics for Programming in LISP

Although SYMBOLs will continue to have property lists, the "value" and "functional" properties will not be stored there, but instead special cells will be made available for each purpose (as the need arises); thus one can still use a symbol as a function name and as a program variable with no interaction between the two. Once consequence of this design is that "function cells" may be lambda bound just as easily as the "value cell"; as of May 1979, we expect to use the same names for "function cell" primitives as are on the LISPM - FBOUNDP, FSYMEVAL, FSET and so on, but currently there is no firm decision as to how to signify the lambda-binding of such cells. Actually, only SUBR addresses will be installed in the function cell of a symbol, and where there is an EXPR definition being associated, there will be a "mediating" SUBR constructed up on the fly; the purpose of this is for fast subroutine-to-subroutine linkage, and when there is a link to an EXPR, a trap will be made automatically back to the interpreter. A new scheme has been worked out for the integration of "local" bindings of variables and "special" (or "dynamic") bindings, such that EXPR code being interpreted will have exactly the same semantics for variables as does compiled code. The key to this new scheme is the use of separate cells for the "local" and "special" bindings, and a strategy of using dynamic information to determine the lexical scope of variables (a similar scheme was independently worked out for the MACSYMA language [Genesereth, private communication, 1978]). Local variables which are not captured by a CLOSURE will merely become stack slots in the compiled version (as now happens in MACLISP), but those which are so captured will retain a structure in the compiled environment isomorphic to, but distinct from, the special value cell arrangement; the advantage of using "local" rather than "special" variables lies not only in the shielding from accidental identification to free variables of the same name, but also in running speed in that their run-time locations may be "dead-reckoned" on the stack (or in the closure) and no SPECBIND, or lambda binding, phase is necessary for them. Thus the notion of a FLEXURE (Functional LEXical closURE) will fill virtually all the purposes to which FUNARGs have generally been put. A paper will be published describing this idea.

The notion of a lambda-list has been extended, compatibly with LISPM to include the keywords &OPTIONAL, &REST, and &AUX; the effect on function definition is that some argument variables are declared "optional", and will will be bound to default values in the case of a call which does not provide the corresponding argument. The &REST argument variable causes all remaining arguments (not bound to the "required" or "optional" argument variables) to be collected into a VECTOR (which will be stack-allocated so as not to cause consing). The &AUX variable feature is merely another way to get some (possibly initialized) "prog" variables. Examples:

```
(DEFUN PRINT (X &OPTIONAL
                (STREAM (COND (MACLISP-P OUTFILES)
                              ('T  STANDARD-OUTPUT)))
                NO-BLANKP)
      (TERPRI STREAM)
      (PRIN1 X STREAM)
      (AND (NOT NO-BLANKP) (PRINC '| | STREAM)))
```

This sample definitions of PRINT shows two kinds of specifications for optional variables - STREAM, if not provided by the caller, is computed by the COND which tests the free variable MACLISP-P and then accessing one of two other variables; NO-BLANKP if not provided by the caller, is merely set to null.

```
(DEFUN NCONC (X &REST W)
    (COND ((NULL W) X)
          ((DO ((I 0 (1+ I))
                (LEN (VECTOR-LENGTH W))
                (TAIL X))
               ((>= I LEN)  X)
             (SETQ TAIL (LAST TAIL))
             (RPLACD TAIL (VREF W I))))))
```

where VREF is the element-selector function for VECTORs. This code corresponds to the MACLISP function NCONC which admits arbitrarily many arguments, and (destructively) concatenates then together.

A few, new, "special form" functions extend the programming syntax. CASEQ is like the INTERLISP SELECTQ; CATCH, CATCHALL, and CATCH-BARRIER provide convenient ways to unwind the stack of a computation to a prior point; CLOSURE is a limited FUNARG device, in which the variables "captured" by the closure must be explicitly pointed out; EVAL-WHEN advises the read-eval-print loop as to the context under which the enclosed forms should be evaluated (ordinary "eval"ing, during compilation of a file, or during loading of the compiled image file); FLEXURE is a CLOSURE over all lexically-apparent variables, with options to delete and add specifically mentioned variables; PSETQ is a "parallel" version of SETQ, meaning that all the forms are evaluated before any bindings are done; THROW unwinds a stack to a correspondent CATCH; TYPECASEQ is like CASEQ, but dispatches on the symbolic name of the data type rather than the datum itself (potentially, the compiler will be able to optimize this); UNWIND-PROTECT will cause a specified "clean-up" action to happen if, for any reason, there is an unwind of the stack through the point at which it was set.

A conceptual union of the data types LIST, VECTOR, STRING, and BITS will be called a *SEQUENCE*. We note that there are four elementary kinds of operations to be performed on sequences: SELECTing (extracting either a single element, or a subsequence), REPLACEing (updating either a single element, or a subsequence), SEARCHing (to return the index of the first occurrence of a specified element or subsequence), and COMPAREing (to return the lowest index whereat two sequences differ). When SEARCHing and COMPAREing are being considered on "Q" type sequences (LISTs

and VECTORs), the question also arises as to what it means to be equivalent; allowing for EQUAL, EQ, and a user-supplied equivalence predicate, we derive the following table of names for these generic functions:

Operation	Element-key	Subsequence-key
Select	ELT	SUBSEQ
Replace	SETELT	REPLACE
Search	POSITION	SEARCH
Compare	. . .	MISMATCH

Some standard LISP functions, originally defined on LISTs, will be generalized in an obvious way for the other "sequence"-like data types: GET and ASSOC (and some of its variants) will be generalized for VECTORs; and LENGTH, APPEND, REVERSE, and NREVERSE will be generalized for all sequences.

Arithmetic extensions, in addition to "bignum" (arbitrarily large precision integer arithmetic), will allow experimentation with concepts such as extended precision floating point ("bigfloat"), complex numbers, intervals, and infinities. Ordinary fixnum arithmetic will be open-compilable without the need for "pdl numbers" as in MACLISP, but open-compilation for flonums will require some similar such development (see [Steele 1977]), which will be postponed possibly for two or more years. For the VAX implementation, there is the prospect of cooperating with the very fine FORTRAN system provided under the VMS operating system; there would be a very smooth interface between compiled LISP programs and compiled FORTRAN programs, and this would lessen the need for very fast arithmetic capabilities in NIL, at least initially (in fact, this is supposed to be a selling point - the standardized function-calling protocol of the VAX architecture permits the output of several different language compilers to be easily loaded and interfaced in the same object environment).

Object-oriented programming will be supported with a version of "message passing" semantics; as of May 1979, this idea has been tested in simulation, but its final syntax has not been settled. Nevertheless, there appears to be good reason to believe that the "message sending" and "method dispatching" will be quite fast under this design, and will be a very practical tool, especially for user-instigated data extensions like "modes" (see [Barton 1978]). Every object (in fact, of every kind of data type in NIL) will have a class-descriptor which delineates the structure of that class of objects; the interesting and new thing about this extension is that users themselves will be able to generate new classes of objects by defining new class-descriptors, and every class-descriptor will have systemic components amounting to

(1) The SYMBOLic name of the class, which the ordinary LISP function TYPEP will return for objects in that class;
(2) a set of "super-classes", of which the given class is considered a sub-class,
(3) a table of "methods", which would be selected on the basis of the "key" word of a message sent to an object in that class;
(4) a fast-subroutine-link for the method dispatcher, so that it is possible to specify non-standard dispatch techniques to some classes without necessarily losing speed;

(5) a "road map" to the structure of the elements in the objects of this class (many "structure" macro packages do essentially this much);

(6) plus several others not fully worked out now.

One interesting feature here is that of "inheritance", in that if a given class has no method to handle a given message, then such a method is automatically sought in its "super-classes". A certain economy is evident hereby, in that if both ODDNUMBERS and EVENNUMBERS are sub-classes of NUMBERS, there need be no duplication of codes within the sub-classes which are applicable to all NUMBERS; but perhaps even more important is the modularity of coding style. The NIL proposal for "classes" is that methods may be dynamically added and removed from a class - most other such systems require all methods to be lexically present in the class definition (at class defining time).

For example, if X holds an object from a class POLY which has an ADD-TERM method, and if that method is essentially to call the function Poly+term, then a request like

```
(SEND-MESSAGE X 'ADD-TERM  (LAST-TERM Z))
```

might appear to be similar to an ordinary function call

```
(Poly+term X (LAST-TERM Z))
```

but a few major differences appear: X will be directly discernible as a POLY, by virtue of it being in a data type *extension* (hence no need to store polynomials as lists with a special interpretation); the code for how to deal with POLYs is modularized around the class definition for POLY; the method for ADD-TERM in POLY might be (dynamically) removed, and thus the SEND-MESSAGE might fail, or might defer to a super-class of POLY; different methods, whether for the same class or for different classes, could wind up calling the same piece of code. Also, it has been noted that one's style of coding seems to change when he thinks of X as an object which can be "told" to do things, rather than merely as some nameless list which could be fed as an argument to various functions.

Implementation Status

A version of the NIL system, written in NIL, is near completion, and runs under MACLISP by the aid of a set of MACLISP macros. This system will form the basis of an initial NIL system on various host computers, simply by tailoring the assembler and loader to know about a specific computer; the expectation is that the compiler will produce (at one level at least) machine-independent code to run under the NIL virtual machine. Such a virtual machine is close to trial stages now on the VAX, and consists of the following layout:

Subroutines and/or micro-code entries, linked by a table QLINK

A table of link cells for value and function cells called SLINK

A "static" heap, or storage area, which would seldom (if ever) be GC'd

A "BLINTZ" vector, to aid the interpreter in recognizing the lexical scope of local variables.

A normal heap, and an alternate heap, so that the GC strategy may be either "stop and copy", or the Baker GC (see [Baker 1977]).

A REGPDL, or regular push-down stack, for normal recursive, computations
A SPECPDL, which is pushed synchronously with the REGPDL, but which holds the variable-binding information.

Of additional benefit for MACSYMA is the ability to store a body of position-independent code as a MODULE (in the file-system, as output from the compiler), and to link these modules into a running job in a way that "shares" maximally with everyone else linking to that MODULE.

The method of implementing co-routines is not fully worked out yet, but it likely will follow the some of design of the LISPM's "stack groups", and of the SL5 programming language (see [Griswold 1978]).

It is expected that there will be a pilot version running on the VAX by early this Fall.

REFERENCES

[Baker 1977]
Baker, H. G., Jr.; List Processing in Real Time on a Serial Computer; *Comm. ACM 21*, 4 (April 1978), Pp. 280-293.

[Barton 1978]
Barton, D. R.; "Guide to the MODE Package for SCA" Memorandum to Mathlab Group, M.I.T.; June 26, 1978.

[Bobrow 1972]
Bobrow, R. J., Burton R. R., and Lewis D.; "Technical Report No. 21", University of California at Irvine, Information and Computer Science Dept., October 1972.

[Dahl 1968]
Dahl, O., and Nygaard K.; "Class and Subclass Declarations"; in Simulation Programming Languages, N. Buxton (Ed.), North Holland, 1968, Pp. 158-174.

[Deutsch 1978]
Deutsch, L. P.; "Inside INTERLISP: Two Implementations", Unpublished paper dated November 26, 1978; Xerox Palo Alto Research Center. See also A Lisp Machine with Very Compact Programs, *Proc. 3rd IJCAI*, 1973

[Greenblatt 1979]
Greenblatt, R., Knight, T., Holloway, J., and Moon, D.; The LISP Machine; (submitted to the IEEE Transactions on Computers)
See also: Bawden, A., et. al. "LISP Machine Progress Report", AI Memo No. 444, Artificial Intelligence Lab, M.I.T.

[Griss 1978]
Griss, M. L., and Hearn, A. C.; "A Portable LISP Compiler", (in preparation, at the Computer Science Dept., University of Utah, Salt Lake City, UT 84112)

[Griswold 1978]
Hanson, D. R. and Griswold R. E.; "The SL5 Procedure Mechanism", *Comm. ACM 21*, 5 (May 1978), Pp. 392-400.

[Hewitt 1979]
Hewitt, C., Attardi, G., Lieberman, H.; "Specifying and Proving Properties of Guardians for Distributed Systems"; *Proc. International Symposium on Semantics of Concurrent Computation*, Evian les Bains, France, July 1979. See also AI Working Paper #172, M.I.T. Artificial Intelligence Laboratory,

[Knight 1974]
Knight, T.; "The CONS Machine"; AI Lab Working Paper #80, [currently out of print]
Additional information is "on line" at the M.I.T. Artificial Intelligence Lab computer.

[McCarthy 1960]
McCarthy, J., et. al.; LISP 1.5 Programmer's Manual,
The M.I.T. Press, Cambridge MA 02139

[Moon 1974]
Moon, D. A.; MACLISP Reference Manual,
Project MAC - M.I.T., Cambridge MA 02139, April 1974
[currently out of print]

[Moore 1976]
Moore, J. S.; The INTERLISP Virtual Machine Specification, CSL-76-5
Xerox Corporation, Palo Alto Research Center, Palo Alto CA, 1976

[Quam 1972]
Quam, L. H., and Diffie, W.; Stanford LISP 1.6 Manual,
AI Operating Note 28.7, Stanford University, 1972

[Steele 1977]
Steele, G. L. Jr.; "Fast Arithmetic in MACLISP",
Proc. 1977 MACSYMA Users' Conference, Pp. 215-224.

[Teitelman 1978]
Teitelman, W.; INTERLISP Reference Manual, Oct 1978,
Xerox Corporation, Palo Alto Research Center, Palo Alto CA.

[Weinreb 1979]
Weinreb, D., and Moon D. A.; LISP Machine Manual,
M.I.T. Artificial Intelligence Laboratory, Jan 1979.