MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 421 September 1977

FAST ARITHMETIC IN MACLISP

by

Guy Lewis Steele Jr. *

Abstract:
MacLISP provides a compiler which produces numerical code competitive in speed with some FORTRAN implementations and yet compatible with the rest of the MacLISP system. All numerical programs can be run under the MacLISP interpreter. Additional declarations to the compiler specify type information which allows the generation of optimized numerical code which generally does not require the garbage collection of temporary numerical results. Array accesses are almost as fast as in FORTRAN, and permit the use of dynamically allocated arrays of varying dimensions. Here we discuss the implementation decisions regarding user interface, data representations, and interfacing conventions which allow the generation of fast numerical LISP code.

This paper was presented at the MACSYMA Users Conference, Berkeley, California, July 1977.

Keywords: numerical code, optimization, compilers, data representations, dynamic allocation, LISP, MacLISP, FORTRAN

This report describes research done at the Laboratory for Computer Science (formerly Project MAC) and at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This work was supported, in part, by the United States Energy Research and Development Administration under Contract Number E(11-1)-3070 and by the National Aeronautics and Space Administration under Grant NSG 1323. Support for the Artificial Intelligence Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

* NSF Fellow

Introduction

For several years now MacLISP has supported a compiler which produces extremely good numerical code. Measurements made by Fateman indicate that the generated code is competitive with FORTRAN. [Fateman] Expressing such numerical code does not require the use of special numerical language embedded within LISP, in the manner that some higher-level languages allow the user to write machine code in the middle of a program. Rather, all numerical programs are completely compatible with the MacLISP interpreter. The compiler processes the interpreter definitions along with additional numerical declarations. These declarations are not required; omitting them merely results in slower compiled code. For convenience, special numeric functions are provided which carry implicit declared type information (such as + and +$ for integer and floating point addition, as opposed to PLUS), but the user need not use them to get optimized numerical code.

Changes to the MacLISP Language

The primary change to the MacLISP language, as seen by the user, was the creation of numerical declarations for use by the compiler. A general compiler declaration mechanism was already a part of the language, so adding the numerical declarations was not difficult. This mechanism involves writing a MacLISP expression beginning with the word DECLARE and followed by various declarations. Declarations may be global or local. Global declarations are written by themselves in a file, and affect all following functions; local declarations are written within the text of a MacLISP function, and affect only the scope of the construct they are written within.

The simplest new declarations are statements of the types of variables. Recall that MacLISP has three basic numeric types: fixnum, flonum, and bignum. These are (respectively) single-precision integers, single-precision floating-point numbers, and arbitrary-precision integers. Only the first two types can be operated on directly by hardware instructions, and so they are the only types of interest to the compiler. An example of a variable declaration:

```
(DECLARE (FIXNUM I J K)          ;single-precision integers
         (FLONUM A B FOO ZAP)    ;single-precision reals
         (NOTYPE SNURF QUUX))    ;no specific type
```

If a variable is always numeric but sometimes may hold bignums, it must be declared NOTYPE. The default assumption is that a variable is NOTYPE (that is, may contain any MacLISP data object); NOTYPE declarations are primarily useful to undo previous numeric declarations.

The types of the arguments and returned values of functions may be similarly declared:

```
(DECLARE (FLONUM (CUBE-ROOT FLONUM)
                 (INTEGER-POWER-OF-REAL FLONUM FIXNUM))
         (FIXNUM (FIBONACCI FIXNUM)
                 (LENGTH-OF-LIST NOTYPE))
         (NOTYPE (BETWEEN-ZERO-AND-ONE-PREDICATE FLONUM)))
```

This declaration specifies that CUBE-ROOT takes a FLONUM argument and delivers

a FLONUM result, that INTEGER-POWER-OF-REAL takes a FLONUM and a FIXNUM and delivers a FLONUM, and so on. The types of the arguments could also be specified by using a local declaration:

```
(DECLARE (FLONUM (CUBE-ROOT)))  ;global declaration

(DEFUN CUBE-ROOT (X)
       (DECLARE (FLONUM X))     ;local declaration
       (EXPT X .333333333))
```

The result type must be specified by a global declaration, however, and declaring the argument types globally also can help the compiler to produce better code for functions which call the declared function.

Arrays may also be declared globally to the compiler. MacLISP arrays come in three types, which are essentially FIXNUM, FLONUM, and NOTYPE. (There are other types also, but these do not concern us here.) The ARRAY* declaration takes a subdeclaration specifying the array type; the subdeclaration in turn specifies the names of arrays and their dimensions. An example:

```
(DECLARE (ARRAY* (FIXNUM TUPLE 1 TABLE 2)
                 (FLONUM VECTOR 1 MATRIX 2)))
```

This declares TUPLE and VECTOR to be one-dimensional arrays, and TABLE and MATRIX to be a two-dimensional arrays. (MacLISP arrays may have up to five dimensions.) If the values of the dimensions are also known ahead of time, a slightly different form may be used:

```
(DECLARE (ARRAY* (FIXNUM (TUPLE 43) (TABLE 3 5))
                 (FLONUM (VECTOR 3) (MATRIX ? 17))))
```

This declares TUPLE to be of length 43, TABLE to be 3 by 5, and MATRIX to have 17 columns and an unknown number of rows. Note that "?" can be used to denote an unknown dimension value; even partial dimension information can help the compiler to optimize array accesses.

The user can write arithmetic code using the traditional names PLUS, DIFFERENCE, TIMES, and QUOTIENT; these functions work on any kinds of numbers, even bignums, and admit mixed-mode arithmetic. In the presence of type declarations, the compiler may be able to deduce that the arguments are always flonums, for example, and produce hardware instructions for floating-point arithmetic. The user can also use the FIXSW and FLOSW declarations to tell the compiler that such "generic" arithmetic will always involve only fixnums or only flonums.

As a convenience to the user, however, several versions of the common arithmetic functions are provided:

generic	fixnum only	flonum only
PLUS	+	+$
DIFFERENCE	-	-$
TIMES	*	*$
QUOTIENT	//	//$
REMAINDER	\	
GCD	\\	
GREATERP	>	>
LESSP	<	<

EQUAL	=	=
EXPT	^	^$ (fixnum exponent)

(The division functions are written as "//" instead of "/" because "/" is a MacLISP escape character.) The functions in the last two columns are completely equivalent to those in the first column, except that they convey additional type information about their arguments and results. (An exception is that the fixnum-only functions do not check for overflow, so in a situation where, for example, 100000000 and 100000000 were multiplied together, TIMES would produce a bignum, whereas * would overflow and produce a not-very-meaningful fixnum. The flonum-only functions do not check for overflow either, whereas the generic functions give an error for overflow, and either an error or zero for underflow.)

Changes to the MacLISP Implementation

In order that the arithmetic machine instructions might be used directly on MacLISP numeric data objects, it was necessary to modify MacLISP to use a uniform representation for fixnums and flonums. Before the fast-arithmetic scheme was implemented, MacLISP, like many other LISP systems, used two representations for single-precision integers. One represented the integer as a pointer to a machine word containing the value, in the same manner as floating-point numbers were represented. The other encoded the value into the pointer itself, using pointer values which were otherwise worthless because they pointed at code instead of data objects. The motivation behind the earlier dual representation was to avoid allocating storage for small integer values, which are frequently used. (InterLISP has for several years "open-compiled" arithmetic functions as single machine instructions. [Teitelman] Unfortunately, it still has a dual representation for integers; as a result, before adding two numbers it must call a routine which determines at run-time the representation of each number and converts each into a full machine word representation. Compiled InterLISP code also calls a similar routine for floating-point numbers, not because of multiple representations, but in order to perform error-checking as completely as the interpreter does. This run-time checking destroys any advantage gained by open-compiling the arithmetic instructions.)

The pointer encoding was removed from MacLISP for the fast-arithmetic scheme, and all numbers are now uniformly encoded as pointers to full machine words which contain the machine representations of the values. In order to avoid allocating storage for frequently used small integers, there are several hundred words of memory containing consecutive small integer values, and small integers are created by making a pointer to one of these standard locations, rather than allocating a new word for each use of a small integer. (MacLISP does not allow the words used to contain numbers to be modified in the way InterLISP allows using the SETN primitive [Teitelman], so there is no difficulty in sharing such words. In fact, these small integer locations are even shared among all the MacLISP processes in the time-sharing system by making them read-only.)

While arithmetic on bignums cannot be compiled as standard arithmetic machine instructions, their representation has been chosen to permit sign tests to be open-compiled. A bignum is a pointer to a word which has the sign of the bignum in the sign bit (and in fact the entire left half), and a pointer to a list of fixnums (which represent the magnitude) in the right half. Thus all numbers are pointers to words which contain the sign of the

number in the sign bit, and such functions as MINUSP can always be compiled as single machine instructions.

In order to preserve the uniformity of the function-calling interface, it was decided that all arguments to functions must be valid MacLISP data objects. On the other hand, it is not desirable to have to "number cons" out of free storage, with the garbage collection overhead that implies, in order to pass numbers to functions. The solution used was to introduce two extra pushdown lists (stacks) called the fixnum and flonum pdls. The storage in these pdls appear to have fixnum or flonum data type, but they are allocated as stacks rather than as garbage-collected heaps. These stacks can be used to hold temporary numerical values and the values of PROG variables which have been declared to be numeric, but they can also be used to allocate pseudo-data objects compatible with MacLISP's standard number representation. A pointer to a fixnum pdl location is indistinguishable from an ordinary fixnum for most purposes; it is a pointer to a full machine word containing the numeric value. A typical code sequence resulting from compiling (FOO (+ A 5)) is:

```
;assume accumulator 1 has the pointer value of A in it
        MOVE 7,(1)      ;get the machine word for A into accumulator 7
        ADDI 7,5        ;add 5 to the machine word
        PUSH FXP,7      ;push resulting word into fixnum pdl
        MOVEI 1,(FXP)   ;copy fxp pointer into argument accumulator 1
        CALL 1,FOO      ;call foo
        SUB FXP,[1,,1]  ;remove pushed word from fixnum pdl
```

To the function FOO the pointer passed in accumulator 1 has the precise format of a MacLISP integer: a pointer to a machine word containing the integer value. Note that the value of the variable A may itself have been such a "pdl number"; the MOVE instruction would move the machine word value into accumulator 7 whether it was a pdl number or an ordinary fixnum.

One of the difficulties of using stack-allocated numbers is that they have a definite lifetime; on return from the function they are passed to, they are de-allocated and no longer exist. By the time they are de-allocated, there must be no more pointers to that word accessible to the user program, or else subsequent references might see a wrong value because the pdl word was re-allocated for some other purpose.

To overcome this difficulty the notion of safety was developed. A copy of a pointer is safe if it can be guaranteed that the copy will become inaccessible before what it points to is de-allocated if the pointer in fact points to a pdl number. Alternatively, a use for a pointer is safe if that use doesn't require a safe pointer. The fast-arithmetic compiler does some complex analysis to determine what situations are safe. Some standard conventions for safety:

[1] A pointer in a global (special) variable may have an indefinite lifetime, and so putting a pointer in a global variable is unsafe. It follows that such a variable may not contain a pointer to a pdl number, since we cannot guarantee such a pointer to be safe. Consequently, any pointer actually obtained from a global variable is safe.

[2] Consing a pointer into a list cell (or using RPLACA to put a pointer into an existing list call) is similarly unsafe. Pointers actually occurring in list structure must therefore be guaranteed safe.

[3] It is not possible to return a pdl number as the value of a function, because there would be no return to the code to de-allocate it. Therefore returning a pointer from a function is unsafe, and all pointers actually returned from functions are safe.

[4] Passing a pointer as an argument to a function is safe; therefore pdl numbers (unsafe pointers) may be passed as arguments to functions. All function arguments are thus potentially unsafe. They may be passed on down to other called functions, but may not be returned or otherwise used as if they were safe.
[5] Pdl numbers may be pointed to by ordinary compiled local variables. Such local variables may or may not have unsafe values, depending on where the values came from. The compiler must guarantee that when the value of a local variable is used either the value is safe or the use is safe.

Suppose we wrote a function such as:

```
(DEFUN ZAP (A) (CONS A 'FOO))
```

We are putting the argument A into a list cell (an unsafe use), but the argument A is also (potentially) unsafe. In this situation the compiled code must create a safe copy of the unsafe pointer. The compiled code therefore uses a routine PDLNMK ("pdl number make") which checks for a pdl number and makes a copy by doing a number cons if necessary. That is, if the pointer given to PDLNMK is already safe, it is returned as is; but if it is unsafe, a safe copy is made with the same value. The compiled code for ZAP would look like this:

```
MOVEI 2,'FOO     ;put constant "foo" in accumulator 2
JSP T,PDLNMK     ;make sure accumulator 1 has a safe pointer
JCALL 2,CONS     ;call CONS
```

If A is not a pdl number, PDLNMK does nothing; but if it is, PDLNMK replaces the pointer in accumulator 1 with a freshly allocated fixnum with the same value as the pdl number. In this way a safe value will be passed to the CONS function. (The convention about function arguments being potentially unsafe has an exception in CONS, so that CONS itself need not always perform PDLNMK on its arguments. The compiler knows about this exception, and guarantees that anyone who calls CONS will provide safe arguments. In practice, arguments passed to CONS often can be guaranteed safe by compile-time analysis, and it saves time not to have CONS use PDLNMK.)

Notice that one consequence of the use of PDLNMK is that two numbers which are apparently EQ (i.e. the same pointer) may not be if the compiled code has to make a copy. For example, consider this code:

```
(DEFUN LOSE (X)
       (SETQ G X)
       (EQ X G))
```

The result of the EQ test could be NIL, even though the global variable G apparently is assigned the same pointer as was passed to LOSE as an argument. If an unsafe pointer is passed to LOSE, G will receive a safe copy of that value, which will not be the same pointer, and so the EQ test will fail. (This is another reason why MacLISP does not have a SETN primitive; since the compiler can make copies of a number without warning, conceivably SETN might modify one copy of a number but not the other, with anomalous results.)

Recall that one unsafe use of a pointer is returning it as the value of a function. We would like for numeric code not to ever have to "number cons", but we cannot return a pdl number from a function. The solution to this dilemma is to allow numeric-valued functions to have two entry points. One is the standard MacLISP entry point, and is compatible with the standard

MacLISP calling sequence; calling the function there will produce a MacLISP pointer value, which will involve a number cons if the value is in fact numeric. The other is a special entry point which is non-standard, and can only be used by compiled code which knows that the called function is numeric-valued. Entering a numeric function there will deliver a machine word in accumulator 7 instead of the standard pointer in accumulator 1.

In order to use this special calling sequence, both the called function and the calling function must be compiled with declarations specifying that the called function is numeric-valued. The compiler will then compile the called function to have two entry points, and the calling function to use the non-standard numeric entry point.

The entry points are actually implemented as two consecutive locations at the beginning of the function. The first is the standard entry point; it merely pushes the address of a special routine FIX1 (or FLOAT1, for a flonum-valued function) onto the stack, and then falls into the non-standard entry point. The function then always produces a machine number in accumulator 7. If the function is called at the numeric entry point, it will deliver its value as a machine word. If called at the standard entry point, then on delivering the machine word it will "return" to FIX1, which performs a "number cons" on the machine word, producing a normal fixnum (or FLOAT1, which produces a flonum), and then returns to the caller.

As an example, here are two functions with appropriate declarations:

```
(DECLARE (FLONUM (DISC FLONUM FLONUM FLONUM)))

(DEFUN DISC (A B C)
       (-$ (*$ B B) (*$ 4.0 A C)))

(DEFUN QUAD (A B C)
       (PROG (D)
             (DECLARE (FLONUM D))
             (SETQ D (DISC A B C))
             (COND ((MINUSP D) (RETURN (ERROR)))
                   (T (RETURN (//$ (-$ (SQRT D) B)
                                   (*$ A 2.0)))))))
```

The code produced would look like this:

```
DISC:   PUSH P,[FLOAT1] ;for normal entry, push address of FLOAT1
        MOVE 7,(2)      ;numeric entry point; get machine word for B
        FMPR 7,7        ;floating multiply B by itself
        MOVSI 10,(4.0)  ;get 4.0 in accumulator 10
        FMPR 10,(1)     ;floating multiply by A
        FMPR 10,(3)     ;floating multiply by C
        FSBR 7,10       ;floating subtract ac 10 from ac 7
        POPJ P,         ;machine word result is in ac 7
```

Notice that DISC does no number consing at all if called at the numeric entry point. It does all arithmetic in the accumulators, and returns a machine word as its result. The code is remarkably compact, of the kind one ordinarily expects from a FORTRAN compiler.

```
QUAD:   PUSH P,1        ;save A, B, and C on the stack
        PUSH P,2        ; to preserve them across the
        PUSH P,3        ; call to DISC
```

```
        NCALL 3,DISC     ;call DISC with the same arguments
        PUSH FLP,7       ;push the result onto flonum pdl
        JUMPGE 7,G0003   ;jump if value non-negative
        MOVEI T,0
        CALL 16,ERROR    ;call the ERROR routine
        JRST G0005       ;go to G0005
G0003:  MOVEI 1,(FLP)    ;get a pointer into flonum pdl
        NCALL 1,SQRT     ;call SQRT with that pointer
        FSBR 7,@-1(P)    ;floating subtract machine value of B
        MOVE 10,@-2(P)   ;fetch machine word value of A
        FSC 10,1         ;multiply by 2.0 (using "floating scale")
        FDVR 7,10        ;divide ac 7 by ac 10
        JSP T,FLCONS     ;perform a flonum cons
G0005:  SUB P,[3,,3]     ;clean up the stacks
        SUB FLP,[1,,1]
        POPJ P,          ;return pointer value in accumulator 1
```

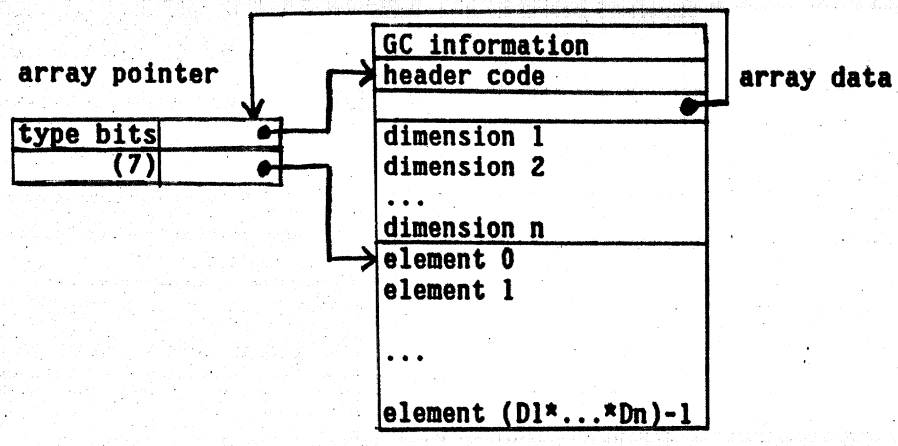
There are several points to note about QUAD:
(1) It was not declared to be numeric-valued. As a result, when returning a number it must do a number cons. Moreover, it does not have a numeric entry point.
(2) Because DISC was declared to be numeric-valued, QUAD uses NCALL instead of CALL to invoke it; NCALL enters at the numeric entry point. The result of DISC is expected in accumulator 7. Since QUAD needs to use this result to pass to SQRT, it makes a pdl number out of this machine word. In this way function values can be made into pdl numbers after all -- but by the caller rather than the called function.
(3) As an aside, the compiler makes some other neat optimizations. It uses a JUMPGE instruction for MINUSP, because the value to be tested is in an accumulator anyway. It takes advantage of the address arithmetic of the PDP-10 to fetch machine words pointed to by pointers on the stack in one instruction. It knows how to use several accumulators for arithmetic, and to arrange for the result to end up in the correct accumulator. It expresses the multiplication by 2.0 as a "floating scale" instruction, which is faster than the multiplication instruction if one operand is a floating-point power of two.

The representation of arrays in MacLISP was carefully redesigned to allow fast access to them by compiled code, again taking advantage of the powerful address arithmetic of the PDP-10. There are essentially two kinds of arrays: s-expression arrays, whose components may be any safe pointers, and numeric arrays, whose components must be all fixnum machine words or all flonum machine words.

The MacLISP ARRAY data type is a pointer to a double word (the "special array pointer") which in turn points to the array data. The reason for this is that the pointer must point to a fixed place (as all MacLISP pointers must), but the actual array data may have to be shifted around by the garbage collector to accommodate new storage requests, because arrays are not of a uniform size. When the garbage collector moves the array data, it updates the the contents of the special array pointer, but the special array pointer itself may remain in a fixed place.

In exchange for the flexibility of dynamically allocated arrays, however, one pays the price of always accessing the array data indirectly through the special array pointer. This cost is alleviated by taking advantage of addressing arithmetic. The second word of each special array pointer points to the array data, which is arranged linearly in row-major

order; this second word furthermore specifies indexing by accumulator 7.



Compiled code can access a numeric array datum by calculating the linear subscript value in accumulator 7 and then using an indirect fetch through the second word of the special array pointer for the array. The linear subscript value is of course calculated as

(... (J1 * D2 + J2) * D3 + J3 ...) * Dn + Jn

where the Ni are the dimensions of the array and the Ji are the actual subscripts. For example, suppose that accumulator 1 contains a pointer to a 3 by 5 by 13 fixnum array, and that accumulators 2, 3, and 4 contain fixnum subscripts for that array. Then to fetch the desired datum this code would be used:

```
MOVE 7,(2)      ;fetch first subscript into ac 7
IMULI 7,5       ;multiply by 5 (second dimension)
ADD 7,(3)       ;add in second subscript
IMULI 7,13      ;multiply by 13 (third dimension)
ADD 7,(4)       ;add in third subscript
MOVE 7,@1(1)    ;fetch indirect through special array pointer
```

If the number of dimensions of the array has been declared to the compiler but not the values of the dimensions, the compiler arranges to fetch the dimension values at run time. This is easy because the array is arranged so that negative subscript values fetch the dimension information. (The LISP user is not supposed to use this fact, but only compiled code.) The same example for a three-dimensional array of arbitrary dimensions might look like this:

```
MOVE 10,(2)     ;fetch first subscript into ac 10
MOVNI 7,2       ;put -2 into ac 7
IMUL 10,@1(1)   ;multiply by second dimension
ADD 7,(3)       ;add in second subscript
MOVNI 7,1       ;put -1 into ac 7
IMUL 10,@1(1)   ;multiply by third dimension
ADD 10,(4)      ;add in third subscript
MOVE 7,10       ;move into ac 7 for subscripting
MOVE 7,@1(1)    ;fetch indirect through special array pointer
```

The code is a little longer than before, but will work for any three-dimensional array. In general, the compiler tries to minimize subscript

computations. If the exact dimensions are declared, or if some of the subscripts are constant, the compiler will do part or all of the subscript calculations at compile time.

For s-expression arrays, the pointer data are stored two per word, with elements having even linear subscripts in the left half of a word and the succeeding odd subscripted elements in the right half of the word. The compiler must generate code to test the parity of the linear subscript and fetch the correct half-word. Suppose that a pointer to a one-dimensional array is in accumulator 1, and a fixnum subscript is in accumulator 2. Then the following code would be generated:

```
        MOVE 7,(2)      ;fetch subscript into ac 7
        ROT 7,-1        ;divide by 2, putting remainder bit in sign
        JUMPL 7,G0006   ;jump if linear subscript was odd
        HLRZ 3,@1(1)    ;fetch pointer from left half
        JRST G0007      ;jump to G0007
G0006:  HRRZ 3,@1(1)    ;fetch pointer from right half
G0007:  ...
```

If the compiler can determine at compile time that the linear subscript will always be odd or always even, it will simplify the code and omit the JUMPL, JRST, and the unused halfword fetch.

Summary

MacLISP supports the compilation of numerical programs into code comparable to that produced by a FORTRAN compiler while maintaining complete compatibility with the rest of the MacLISP system. All numeric code will run in the MacLISP interpreter; additional information may be given to the compiler in the form of declarations to help it generate the best possible code. If such declarations are omitted, the worst that happens is that the code runs slower.

Compatibility with non-numeric functions was achieved by the judicious choice of a uniform representation for LISP numbers combined with a compatible stack-allocated representation for temporary numeric values passed between functions. The use of stack allocation reduces the need for garbage collection of numbers, while the uniformity of representation eliminates the need for most run-time representation checks. One exception to this is that the use of stack-allocated numbers must be restricted; this difficulty is kept in check by maintaining a careful interface between safe and unsafe uses, and analyzing the safety of pointers as much as possible at compile time.

While numeric functions and non-numeric functions may call each other freely, a special interface is provided for one numeric function to call another in such a way as to avoid number consing.

Arrays are stored in such a way that they may be dynamically allocated and yet accessed quickly by compiled code. This is aided by the rich address arithmetic provided by the PDP-10.

The philosophy behind the implementation is that the generality of LISP and the speed of optimized numeric code are not incompatible. All that is needed is a well-chosen, uniform representation for data objects suitable for use by hardware instructions, combined with a willingness to handle important special cases cleverly in the compiler.

References

[Fateman] Fateman, Richard J. "Reply to an Editorial." SIGSAM Bulletin 25 (March 1973), 9-11.

[Teitelman] Teitelman, Warren. InterLISP Reference Manual. Revised edition. Xerox Palo Alto Research Center (Palo Alto, 1975).