THE USE OP LISP FOR COMPUTER-AIDED DESIGN SYSTEMS

by

Neal M. Holts & William J. Rasdorf

DRC-12-07-82

April, 1982

THE USE OF LISP FOR COMPUTER-AIDED DESIGN SYSTEMS[1]

Neal M. Holtz[2]
William J. Rasdorf[3]

ABSTRACT

It is suggested that the LISP programming language has many advantages and few disadvantages for use in developing complex engineering application programs. This paper attempts to explore that possibility by briefly explaining a few of the concepts of the language, by discussing some of the criticisms of it and by mentioning some new developments. In summary it is suggested that LISP may currently be the best programming language to use for the development of computer-aided design (CAD) systems.

1. INTRODUCTION

It is becoming increasingly recognized in the engineering profession that FORTRAN does not satisfy all engineering programming needs. At the same time, it is obvious that there are far too many other languages, many of which have at one time or another been claimed to be the solution to all programming problems. Advocates of PL/1, the Algols, and Pascal in particular have been heard from, and it is presumed that advocates of Ada will soon be appearing. While all of these languages represent important advances over their predecessors, it may be the case that a single static programming language will never be completely adequate, and that what is needed is a language that can evolve in response to changing needs.

LISP is one of the oldest programming languages - one that has survived and flourished in a community of very sophisticated programmers. It has done this despite the absence of two factors that contribute greatly to the survival of other languages like FORTRAN: no major vendor currently promotes LISP, and up to now the investment in existing software has not been very important to the community of LISP users. LISP has flourished precisely because it does many things very well and because it can evolve as new needs are recognized

If those who develop engineering programs consider LISP at all, they probably do not regard it with much favor. Such an attitude may have been justified in the past, but recent

---

[1] Submitted to the Journal of the Technical Councils. ASCE (Technical Council on Computer Practices).

[2] Department of Civil Engineering, Carleton University, Ottawa, Canada K1S 5B6.

[3] Department of Civil Engineering. Carnegie-Mellon University. Pittsburgh. Pennsylvania 15213.

developments *are* a cause for re-examination. This we will attempt to do.

Most of the issues raised in this discussion are not new; they are very well known in the computer science community. Many of the ideas presented here owe a great deal to Sandewall, whose excellent papers on LISP programming styles [14, 15] should be read by those who are interested in pursuing this discussion further.

1.1. Warning

Not all LISPs are created equal; many incompatible dialects exist and many of the available systems are minimal and will not properly support all of the programs and environments mentioned below. Our purpose is to describe the desirable features common to several good LISP systems, among them MACLISP, UCI-LISP, and INTERLISP. The description that follows is based on the one known as "MACLISP" [13], a dialect slightly superior for engineering computations. Not all the facilities of MACLISP are described - only those that are most common.

2. THE LANGUAGE

LISP is nominally a LISt Processing language that is based on some work on nonnumeric computation published by John McCarthy in 1960 [11]. The first LISP system was implemented then at M.I.T. and another version was later described in the "LISP 1.5 Programmer's Manual" by McCarthy [12]. It is interesting to note that as one of the oldest programming languages FORTRAN and ALGOL predate it by only one or two years) LISP is now the language of choice among most researchers in artificial intelligence and in other fields requiring predominantly symbolic computation.

2.1. Data Types

An essential part of describing a programming language is to describe the types of data provided as primitives by the language, the kinds of operations provided for use on that data, and the ability of the programmer to define new data types and operations.

LISP contains four types of "atomic" data and several ways of constructing data aggregates. Atomic data (or "atoms") are integers, floating point numbers, strings and "literal atoms". Strings are any sequence of characters enclosed in double quotes. Literal atoms, also called identifiers, are sequences of characters that cannot be interpreted as numbers, strings or punctuation (punctuation being the the characters [ ] ( ) , ' ; and space). Valid atoms are:

-17	integer
373856	integer
-23.78E-15	floating point
"A String"	string
BEAM-LENGTH	identifier

Facilities exist for extracting single characters or groups of characters from atoms and for combining them with others to form new atoms. Standard arithmetic functions are defined for numerical operations.

An important data aggregate is the *list.* A list is represented syntactically by a T, followed by any number of s-expressions that are the elements of the list followed by a ")". The term *s-expression* is short for symbolic expression and is a generic term meaning any valid LISP data item. Valid lists are:

```
()                          the empty list, often written as NIL
(A)                         a list of one element
(12 3 4)                    a list of four elements
(PLUS (TIMES 13.7 A) B)     a list of three elements,
                              the second of which is itself
                              a list of three elements
```

The size and complexity of lists is limited only by the amount of available memory. Primitive operations on lists include: adding a new element to the front of an existing list; extracting the first element; and extracting the list obtained by removing the first element These are sufficient for all list manipulations, but in practice many more, such as appending one list to another, are provided

There is a second data structure that does not have a special external syntactic representation - the *array.* Arrays can have an arbitrary number of dimensions, with arbitrary integer upper and lower bounds on each. The value of each array element can be any s-expression. Array operations include: creating new arrays; setting the value of any element; and retrieving the value of any element

The extensibility inherent in LISP allows one to create new methods of forming aggregates of data, and to have these appear exactly as if they were primitives in the language. For example, combining the macro facility explained below with an array allows the programmer to create data structures with named fields. This is very similar to the record facility of Pascal; it is not really a primitive in LISP systems, but it gives the appearance of being so.

All of the above result in data structures that can be stored as the values of variables. There is another method of storing and retrieving data, somewhat akin to the record facility - the property list A property list is a list of attribute-value pairs stored independently for each variable. Property lists are completely dynamic and may be added to or deleted

from at will. Each attribute or "property" has a name and any value can be associated with that name. The functions "GET and "PUTPROP" are used to retrieve and store values using property lists. For example, if "P" has the value "LENGTH' and T has the value "BEAM-17", then (GET I P) will return the value stored under the property "LENGTH" of identifier "BEAM-17". Property lists may be thought of as associative memory. In combination with APPLY and unnamed functions they allow 3 very powerful programming style; this is known as data directed programming and is discussed further below.

2.2. Variables

LISP identifiers serve much the same function as variables in more traditional programming languages. While it is reasonable in most programming languages, FORTRAN in particular, to think of variables as names for fixed locations in memory, this is not a helpful concept in LISP. LISP variables are more akin to the variables of algebra - they are just temporary holders of data values. Unlike most programming languages LISP does not restrict variables by type. Values of both variables and properties can consist of any of the valid data types described above, or they may be arbitrary expressions of large and complex data structures.

Scope and extent are two properties of variables that differentiate programming languages. The *extent* of a variable is its lifetime - that portion of the program during which the variable exists. LISP supports both local and global extent In local extent the variable is created upon entering a *range* and is destroyed upon leaving it A range is a program segment that requests the binding of values to variables. A function is one example. Its formal parameters are variables that are created and bound to the values of the arguments when execution of the function begins, and are destroyed when the function returns. Alternatively, LISP also supports global extent by allowing new variables to be created If the expression (SET 'A 10) is evaluated in an environment where "A" is unbound, "A" is created as a new variable with global extent This is in direct contrast to FORTRAN where variable extent is always global meaning that storage is allocated at compile time, and variables are created when the program is loaded and last until its end

The *scope* of a variable is that portion of the program text in which all uses of its name have the same meaning. In LISP the scope of all variables is dynamic; their binding in one environment is local to that environment but is global to all other environments that it subsequently calls. As a result, the determination of variable scope can only be made during program execution. FORTRAN variables, on the other hand, have static scope and can be used only in each subroutine in which they are declared (COMMON is a special statement that allows locations but not names to be shared).

2.3. Evaluation of Expressions

LISP is an applicative or functional language. This means that all computation takes place through the application of functions to arguments; in LISP this takes the form of evaluating s-expressions.

Numbers and strings evaluate to themselves; identifiers evaluate to their current value. A list of the form

(<fn> <arg1> <arg2> ... <argn>)

is a function call whose meaning is as follows. <fn> may be a the name of a function, a function definition itself (explained below), a reserved word, or the name of a macro. <arg1> through <argn> are s-expressions and are arguments to the function.

If <fn> is a function name or a function definition, then the evaluation procedure is the intuitively obvious one; first all of the arguments are evaluated, then the function is applied to the values of those arguments. For example, if "A" has the value of 5, then the expression:

(TIMES (MINUS A 2) 4)

evaluates to 12.

If <fn> is a reserved word, then the evaluation depends entirely on the definition of that word; the arguments may or may not be evaluated For instance the reserved word "SETQ", the assignment operator, is used to assign values to variables. The expression:

(SETQ A (PLUS 5 8))

results in "A" being assigned the value 13. Note that the first argument is not evaluated (but the second one is), ahd then contrast that with the following example. If the value of "B" is "C" (another identifier) then:

(SET B 8)

results in the identifier "C" being assigned the value 8! SET is a function, therefore both of its arguments were evaluated before it was invoked

If <fn> is the name of a macro, then all arguments are passed unevaluated to the body of the macro. The macro returns a new s-expression which is itself evaluated The result of this evaluation is the value of the original s-expression. Macros are a good way of making extensions to the language, particularly in defining new control structures. Because interpreted programs are represented by lists, the built-in list manipulation primitives can be used to construct new programs or expressions which can then be evaluated or compiled exactly like other programs or expressions.

Because data and programs have the same syntax, a mechanism must be introduced to distinguish between constant data and expressions that are to be evaluated The reserved

word QUOTE is used for this. The s-expression:

```
(QUOTE <s-expr>)   or   '<s-expr>
```

when evaluated, returns the unevaluated <s-expr> as its value.

2.4. Function Definition

A function definition is a list of three or more elements. The first element is the reserved word "LAMBDA" and the second is a list of the identifiers that are the formal parameters of the function being defined. Following these are one or more s-expressions. When the function is applied to its arguments, the s-expressions are evaluated sequentially, with the values of the formal parameter identifiers being temporarily set to the corresponding arguments. After the last expression is evaluated, the old values of the formal parameter identifiers are restored and the value of that last expression is returned as the value of the function. For example:

```
(LAMBDA (X) (TIMES X X))
```

is the definition of a function that squares its single numeric argument.

Three ways exist for applying a function to its arguments, i.e., to invoke or execute the function. The least common way is to do

```
((LAMBDA (X) (TIMES X X)) 73.5)
```

to compute the square of 73.5. The most common way is to associate the function definition with a name (using DE, discussed below), say "SQR", and to write

```
(SQR 73.5)
```

to have the same effect.

A third and potentially very powerful method is to leave the function unnamed and to have its definition either computable or stored in a data structure and accessible by a mechanism that is more meaningful and flexible than a simple name. This involves use of a built-in function called "FUNCALL" of one or more arguments:

```
(FUNCALL <fn> <arg1> ... <argn>)
```

where <fn> evaluates to a function name or definition and <arg1> through <argn> are the arguments to be passed to the function. For example:

```
(FUNCALL '(LAMBDA (X) (TIMES X X)) 73.5)
```

will return the square of 73.5. Note that "FUNCALL" is the name of a function, therefore all of its arguments are normally evaluated before it is invoked. It then applies the given function to its second and succeeding arguments. Here the first argument evaluates to a function definition which is then applied to the argument 73.5. It is important to realize that any computation could have appeared in either of the two argument positions and the result would have been taken as a function definition and the arguments to the function. Contrast this with most programming languages, where the only mechanism to choose

between alternate functions to apply is to explicitly enumerate all possibilities at compile time.

Function definitions, because they *are* simply lists, can be constructed or otherwise manipulated exactly as any other data structure. Thus the distinction between program and data becomes much less clear than in traditional programming languages.

2.5. Program Control

LISP supports a range of language constructs that allow for flexible program control. The simplest and most direct form of supported control is sequential execution of expressions, where control flows from one expression directly to the next one.

Conditional execution in LISP is controlled by the "COND" function. Its form is:

```
(COND (conditionl  exprll  ...  exprln)
      (condition2  expr21  ...  expr2n)
                    .
                    .
                    .
      (conditionn  exprnl  ...  exprnn))
```

and its associated rules of evaluation are somewhat like the Pascal case statement A COND is evaluated as follows. Conditionl is evaluated first If it is *TRUE* then expressions expr 11 through expr 1 n are executed and the result of the last expression is returned as the result of COND. No further evaluation takes place. However, if conditionl is *FALSE* then each succeeding condition is evaluated until one evaluating to *TRUE* is encountered. That condition's statements are then executed as described above. If no conditions evaluate to *TRUE* then the value of the COND is NIL

Repetition is achieved in LISP by either iteration or recursion. The iteration function is a very general DO construct which allows for completely arbitrary initializing, incrementing and testing of any number of variables. Special functions known as mapping functions are provided for iterating over elements of lists and arrays.

LISP also supports recursion, i.e. it allows functions to recursively call themselves. This is possible because LISP allows the necessary dynamic creation of variables. Because FORTRAN supports only statically declared variables bound at compile time, recursion is not possible.

3. PROGRAMMING ENVIRONMENT

A programming environment is that set of tools provided to aid in the task of program development In traditional programming languages like FORTRAN, these tools are editors, compilers, linkers, debuggers, etc.. Unfortunately these are usually not well integrated LISP systems often have a well integrated set of development tools.

3.1. System Environment

Program development and execution take place in an interactive environment and are controlled by the read-evaluate-print loop of the LISP system. That is, the user types an s-expression, the interpreter reads it, evaluates it, and prints the result The expression is often intended to produce lasting side-effects, such as defining a function or assigning a value. Alternatively, it may request execution of other defined functions.

In most programming systems it is difficult to develop a single function and test it independently of others; yet that is theoretically the best way to develop large programs. LISP very naturally allows this methodology. Because programs and data have the same form the interactive system automatically provides I/O for data structures, something very few other languages do. Additionally, interpreted programs allow very flexible error handling. These features lead to a tremendous improvement in program development methodology.

In LISP it is very natural to code a function for a particular task and to test it immediately. This can be done even if that function calls other functions not yet defined One can simply type in the required data, invoke the function, and examine the result If a problem or error occurs such that the evaluation of the function cannot continue, execution is suspended It is possible at the point of suspension to examine or modify data, to edit the program to correct the error, or to perform any other computation including defining new functions and debugging them. Execution can then be resumed from the point of suspension or it can be restarted

The net effect of this is that one has complete and integrated control over the entire execution process. It provides a programming environment that is superior to that of modern time sharing systems and interactive debuggers. Although this type of environment is not inherently limited to LISP systems, it is much easier to implement in LISP and is therefore more likely to exist there.

3.2. System Defined Extensions

LISP is an extensible language; it enables programmers to write extensions (in LISP) to the programming environment Such extensions can greatly aid the program development process. They may appear to an application programmer to be built-in features of the language but they are extensions to it provided with the system

A useful feature that is often implemented in LISP is a built-in help facility. Such a facility enables one to quickly browse through a tree structured documentation file to obtain information about any of the system concepts or functions. The help system can also be used by programmers to provide on-line documentation for their programs; this is discussed further in the next section

Another useful feature is a history facility. Recall that the LISP system reads an expression typed by the user, evaluates it and prints the result Some systems can remember previously typed expressions. It is then very easy to recall these and have them re-evaluated or perhaps modified and re-evaluated. This is a great advantage when debugging that is not limited to LISP but is easier to provide there.

Some LISP systems have an editor that knows about the structure of programs and data (most editors know only about single characters). The modification of programs is greatly eased when one can communicate with the editor in terms of language constructs rather than characters. Data structures can also be changed with the same editor used for programs.

3.3. Programmer Defined Extensions

One of the authors (Holtz) developed the initial version of a design specification processing program in LISP. While the programming environment provided by the system was very powerful and lead to much faster program development than would have been possible in any other language, the author was easily able to provide useful extensions to that environment

We have already mentioned the system defined help facility to extract documentation for system concepts and functions. The author constructed a similar on-line documentation for his program This proved invaluable when it was necessary to recall the details of some of the more arcane functions. The total program contained 250 to 300 functions, few of which were more than 20 lines long. It was possible to forget exactly what some of them did and being able to quickly recall a few sentences of on-line documentation was extremely helpful. This capability is an improvement over reading comments in a program listing because one does not have to look for the appropriate spot in the listing, and

because the system can automatically maintain cross-references to other functions.

Because of the ease with which programs can be manipulated as data, the author was able to define several tools to manipulate functions that significantly aided in the program development process. These tools included programs that would modify functions to insert timing or counting code. They also included programs to search functions looking for arbitrary patterns of expressions. This was particularly useful when the calling sequence of a function had to be changed and all functions that called it had to be located The tools also included a program that would load functions into memory only when required All of these programs were small - they occupied less than three pages of listing in total. Obviously the benefits derived from them outweighed the small investment in writing them.

4. RECENT DEVELOPMENTS

LISP is currently receiving a good deal of attention in the computer science community, mostly from researchers in artificial intelligence. It is being developed in several different directions.

One of the more interesting developments is that of constructing special purpose machines that have instruction sets and capabilities specifically tailored to the execution of LISP programs. These are the so called "LISP Machines"; among the most advanced of these is the one developed at MIT. and described by Greenblatt et al [5]. Marshall [9] reports that two similar computers are now commercially available. Other researchers are micro-coding specialized LISP instruction sets on standard computers. Among these are the efforts of Carnegie-Mellon University to base a system on a PERQ computer, and of Bolt Beranek and Newman, Inc. to use a PRIME computer. These efforts promise to result in fast and relatively inexpensive systems.

Other efforts have concentrated on developing "personal" LISP machines. Some have developed systems for the smallest microprocessor such as Z80 based systems [8]. Efforts to create a relatively portable implementation, as has been done for Pascal, are progressing [6, 18], and soon LISP systems may be as ubiquitous as Pascal and Basic.

5. CRITICISMS: VALID AND OTHERWISE

5.1. Machine Cost

Perhaps the major criticism that has been levied is its "inefficiency". This is generally taken to mean that LISP programs take too much CPU time to execute and consume too much memory.

While it is certainly true that any interpreted program will suffer in direct comparison of sheer execution speed with a compiled program, this is not the only consideratioa In fact, it is very likely that because LISP programs are normally interpreted, total machine costs can be significantly less than would be the case if all program development were done in a compiled language. A significant saving is achieved because programs do not have to be recompiled, relinked and reloaded when changes are made.

There is nothing inherent in LISP that requires interpretation - it is simply natural and easy to do. Programs can also be compiled into machine code. Effort has gone into producing compilers that will generate very efficient code. Once a program has been debugged, compilation may lead to speed-ups on the order of ten to one and space savings on the order of two to one. Several years ago it was reported [3] that the MACLISP compiler produced a purely numeric program (to find cube roots) that ran faster than the equivalent FORTRAN program

A close examination of the storage implementation of most LISP systems reveals that all atomic data are located by pointers, i.e., an identifier that has a numeric value will generally hold a pointer to that value. It is reasonable to assume that this may be expensive in terms of memory requirements. However, two things should be considered One is that this cost is important primarily when large arrays of numbers are required Most modern implementations allow one to declare that a particular array will hold only numbers; in this case the indirect pointers are not used The second thing to consider is that memory costs are decreasing more rapidly than any other computing cost; the additional storage is easily offset by the advantages gained

The rich programming environment of LISP implementations occupies a considerable amount of memory. The major drawback, however, is not memory cost but memory size. Most personal machines are currently limited to 64K. While very good implementations exist for machines of this size, 64K is not enough memory - 128K would be much better. Certainly within a few years the second generation of microprocessors will be more in evidence; these will have absolutely no problem supporting full implementations.

LISP provides automatic storage management for the programmer. This means that he can continually request storage as if he had an infinite supply because the system automatically performs garbage collection and heap storage allocation. A performance price is paid for this service but the cost can be minimized by careful programming. The assistance that this service gives the programmer is usually well worth the small increase in computation cost

5.2. Portability and Availability

The most serious problems currently facing LISP systems are lack of portability of application programs and limited availability of good implementations. The variety of incompatible dialects that exist is certainly a cause for concern. But it is encouraging to note that the language lends itself to automatic translation between dialects which somewhat alleviates the problem of portability. Unfortunately, a similar solution does not exist for its availability. Each different dialect is available on only a small number of machines. As a result a particular dialect may not be available for a given machine.

Neither portability nor availability have been of great concern to most LISP programmers because they are not usually involved in developing production programs. However, major production programs written in LISP, such as the EMACS based text editors discussed below, are becoming more and more prevalent As they do, the importance of their portability increases. Fortunately, there are ongoing attempts to create a standard [6, 10], but so far these have been viewed with distrust or disinterest by most of the LISP community. There is nothing inherent in the language that should make standardization difficult to achieve; the retardant so far has been its use by a community that views standards with malevolence.

5.3. Readability

The thickets of parenthesis that abound in s-expressions do appear rather formidable at first glance. However, "pretty printers" that display program segments with meaningful, consistent indentation certainly assist in their readability. In addition, the use of "super parenthesis", "[" and "]", can go a very long way toward eliminating the need for counting parenthesis on input A single "]" will close all open left parenthesis back to the last "[", or to the beginning of the expression

Recall also that the parenthesis rule is the only syntactic rule that one must remember. The simplicity gained by not having to be aware of many syntax rules is certainly an advantage. On the other hand, the LISP programmer must remember or have access to documentation for a large library of standard functions and the arguments they require.

It is true that complex arithmetic expressions will be less easy to understand when written in prefix form. However a short LISP program, less than one page long, can be used to convert infix notation to prefix. Such a program can translate these expressions as they are read so that the translation is done only once. Additionally, an emperical study by Knuth [7] of FORTRAN programs found that fewer than 6% of over 250,000 statements were assignment statements containing more than one arithmetic operator. The bulk of the statements were declarations and control statements, all of which are either

nonexistent or simpler in LISP. For example, knowing that FOR-EACH is a control structure that in turn selects all objects of a given type from a collection of objects, the LISP program segment

```
(FOR-EACH  'BEAM  X  IN  FRAME-7
           (PRINT  (NAME  X)
                   (SIZE  X)
                   (LENGTH  X)
                   (WEIGHT  X)))
```

probably expresses its meaning much more clearly than the dozens of lines of code that would be necessary in FORTRAN or Pascal. The ability to easily define new control constructs can lead to a more readable program

6. ADVANTAGES FOR CAD

Computer-aided design is the application of computer technology to the methods and procedures of planning and performing the design process. As such, its concerns transcend programs whose main purpose is to perform large numerical operations such as finite element analysis. It involves a style of programs that are easy to use and that are expected to be interactive and data directed The ability of LISP to satisfy these requirements makes it an ideal candidate for use in CAD systems. Some of its specific advantages will now be discussed.

6.1. Ease of **Learning**

LISP is very likely the easiest of all programming languages to learn, particularly as a first language. A few hours of work will allow anyone to understand some very interesting programs. It Is so different from other languages such as FORTRAN or Pascal that a knowledge of them may be somewhat of a hindrance to learning LISP. Only a brief overview of the language has been provided here; for additional detail the reader is directed to descriptions by Siklossy [16] or Winston and Horn [19].

6.2. Data Directed Programming

We will present here one particular concept that is difficult or impossible to realize in most other programming languages - the concept of data directed programming It is a style of programming that can markedly reduce the complexity of some programming tasks, and should prove to be very useful in an interactive CAD environment

Data directed programming takes advantage of the existence of property lists and unnamed functions to organize programs around the data in a database (as opposed to transferring data between programs). It is a way of using data to choose which particular procedure to call in a given context but in such a fashion that new choices can be added

without modifying existing programs. It is also a way of restructuring the database by modifying data types. These concepts are perhaps best illustrated by means of examples; we will present a simplified illustration in the context of a structural design database.

Assume that the database is organized in terms of named objects. That is, every object (beam, slab, frame, floor, etc.) is given a unique name - an identifier. On the property list of each such identifier is stored all of the data that relates to that object; its type, its physical description, the objects of which it is composed, as well as its topological and geometrical relationship to other objects. The kind and amount of data stored on each property list is dependent on the objects type and is defined by the user, not the programmer. In a language other than LISP the object and its properties would most appropriately be represented by a record

Now suppose that we need to add a new attribute to a database object, say to a beam. To add the new piece of data to the record would require the addition of a new field, but the structure of records does not permit their dynamic extension. One is therefore forced to predetermine all potentially necessary data fields prior to the initialization of the data structure. The dynamic aspects of CAD are therefore not well supported in languages relying heavily on records.

LISP, on the other hand, permits the addition of new items of data to database objects. It can do so because its property lists are dynamic, permitting the addition and deletion of both properties and relationships as data requirements change. Such changes are expected to be the norm in a CAD environment and are well supported by the language.

Dynamic property lists offer an additional advantage. When a new data field is added to a Pascal record, storage is allocated for the new field in every occurrence of a record of that type. But if only some of the objects of that type need the new field (concrete beams require f'c, for example, whereas steel beams do not) then storage is wasted Property lists attach properties directly to instances of objects rather than to all objects of given type. Properties are therefore added only where they are needed

Next consider the problem of allowing the user to modify the location of an object When this happens the location of other related objects must also be modified Moving a wall, for example, necessitates that its doors and windows also move. To do this in a traditional programming language requires one to write a MOVE procedure that performs the correct operation for that type of object If a change, such as the addition of a new field to a record, is made to any portion of the data structure on which the procedure operates then the procedure itself must be changed When such changes are made it is

difficult to insure that they do not affect other parts of the data structure. With LISP the approach is considerably different One may store very specific functions on the property lists of the identifiers that name object types. These functions need only know how to move the specific type of object and how to find which other objects must be moved in turn. These unnamed functions may be stored, for example, under the property MOVE-FN. That is,

```
(GET •BEAM •MOVE-FN)
```

will return the unnamed function that knows how to move beams, and thus

```
(GET (GET 'B18 'TYPE) •MOVE-FN)
```

will return the function that knows how to move the object B18.

6.3. Program-Data Independence

Suppose it is desired to implement a MOVE program that can move any object a certain distance in the three orthogonal directions. That is, if we wish to move a frame 1.2 meters in the x-direction, we can type:

```
(MOVE 'FRAME-7 1.2 0.0 0.0)
```

One possible definition of this function is the following:

```
(DE MOVE (OBJECT X Y Z)
         (FUNCALL (GET (GET OBJECT 'TYPE) •MOVE-FN)
                  OBJECT X Y Z))
```

DE defines a function named MOVE with four parameters; an object name and three numeric displacements. The MOVE function uses GET to obtain the actual mover function that is appropriate for the type of the object and calls FUNCALL to apply that function definition to the three displacement arguments.

The advantage of this program-data independence is that when new object types are added, no changes to existing programs need be made. All that is required is that a function that knows how to move the new type of object be added to the property list of the new type identifier. The function will be invoked by MOVE as required

There is another advantage that occurs when a change is made to the kinds of data stored with an existing type of object It is very simple to find the function that needs to be modified; it is stored on the property list of the type identifier. One may be quite confident that modifying the function will not introduce bugs elsewhere; functions are completely isolated from one another. Program complexity is thus substantially reduced because of the independence in the LISP system between programs and data of different types.

Other languages such as SIMULA [1] and GLIDE2 [2] attempt to achieve the same thing through special built-in constructs such as modules and system invoked routines. Both of

these languages require special syntax and runtime support to do so and both lack complete flexibility. The LISP implementation seems to be much cleaner and easier for the CAD system programmer to use.

6.4. Extensibility

If a CAD system is to be accepted as a day to day design tool, it has to be extensible to the maximum amount possible. This means that it must be possible to add and test new functions at any time, and by any user. It must also be easy to incorporate user extensions into the basic system.

It is often stated that all programs must be fully specified before coding begins. However, there is a large class of programs for which this cannot be the case. Many CAD systems are in this class; simply not enough is known about desirable facilities to allow them to be defined a priori. Effective CAD systems are constructed from the efforts and suggestions of many users. Unless one can initially and fully specify the perfect CAD system for a given application, extensibility is imperative.

There have been very few really extensible systems developed. One of the most interesting and recent is EMACS, described by Stallman [17]. EMACS and its derivatives are nominally text editors, but they are so much more that the term is probably an injustice. Stallman describes how EMACS could only have reached its present advanced state by retaining complete and convenient extensibility. MULTICS EMACS, a commercial product of Honeywell, and ZWEI, a newer generation of EMACS, are both written in LISP. Stallman and Greenberg [4] both credit the use of LISP with the fact that these text processors are so successful, and with the fact that both are the product of small improvements by many people. Greenberg further states that people who have never programmed before have successfully written useful extensions in LISP to customize and extend the editor for their particular needs. In doing so, they program in a special purpose language embedded in LISP. In fact, this language is nothing but a set of functions and control structures that are right for the job of manipulating text; the extensibility of LISP makes these capabilities possible.

7. CONCLUSIONS

The preceding discussion seemed to be less concerned with CAD than with programming systems. It is the authors' belief that the choice of programming language very much influences the flavor of the product derived from that language. The programming system should provide a style of interaction and a programming environment that is a good model for software developers. At the very least, this will lead to a consistency among most

programs developed using that system.

It is also the authors' belief that good interactive CAD systems are much easier to develop when programmed in a language that is much like LISP. This belief is based in part on experience gained from coding a large program in both LISP and Pascal. More importantly, it is felt that better quality products ensue when the style provided by LISP systems is reflected in the CAD system - products that tend to be more easily tailored to the needs of individual users.

As computers are incorporated into more and more of the design process, emphasis is shifting markedly away from straight numerical computation to manipulation of complex relationships among data It is essential that programming languages be used that help reduce program complexity and that permit one to program at the level of application concepts, rather than at the level of memory locations. LISP or LISP-like languages provide perhaps our greatest opportunity for reducing program complexity to manageable proportions.

8. ACKNOWLEDGEMENT

The authors greatfully acknowledge the review of Dr. Steven Fenves of Carnegie-Mellon University. His comments and suggestions were most helpful in the preparation of this paper.

References

[I] Birtwistle, G M., Dahl, O., Myhrhaug, B., and Nygaard, K.
SIMULA BEGIN.
Van Nostrand Reinhold Company, 1979.

[2] Eastman, C, and Thornton, R.
A Report on the GLIDE2 Language Definition.
Technical Report, Computer Aided Design Group, Institute of Physical Planning, Carnegie-Mellon University, Pittsburgh, PA, March, 1979.

[3] Fateman, R.
Reply to an Editorial.
ACM SIGSAM Bulletin (25):9-11, March, 1973.

[4] Greenberg, B. S.
Prose and CONS (Multics EMACS: a Commercial Text-processing System in LISP).
In *Conference Record of the 1980 LISP Conference.* The LISP Conference, P. 0. Box 487, Redwood Estates, CA, 1980.

[5] Greenblatt, R., Knight, T., Holloway, J., and Moon, D.
The LISP Machine.
Technical Report, MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1979.

[6] Griss, M. L *
Portable Standard LISP - A Brief Overview.
Operating Note 58, Department of Computer Science, University of Utah, Salt Lake City, UT, December, 1981.

[7] Knuth, D. E.
An Emperical Study of FORTRAN Programs.
Software - Practice and Experience 1:105-133, 1971.

[8] Levitan, S. P., and Bonar, J. G
Three Microcomputer LISPs.
BYTE :388-412, September, 1981.

[9] Marshall, M.
LISP Computers go Commercial.
Electronics :89-90, November 20, 1980.

[10] Marti, J. B., Hearn, A. C, Griss, M. L, and Griss C.
Standard LISP Report.
Technical Report UUCS-78-101, Department of Computer Science, University of Utah, Salt Lake City, UT, January, 1978.

[II] McCarthy, J.
Recursive Functions of Symbolic Expressions and their Computation by Machine.
Communications of the ACM 3(4): 184-195, 1960.

[12] McCarthy, J., et al.
LISP 1.5 Programmer's Manual
The M.I.T. Press, Cambridge, MA, 1962.
Description of the IBM 7090 LISP implementation

[13] Moon, D.
MACLISP Reference Manual
1979.

[14] Sandewall, E
Machine Intelligence : Some Observations on Conceptual Programming.
Ellis Hornwood Limited Chichester, England, 1976, pages 223-265chapter 14.

[15] Sandewall, E
Programming in an Interactive Environment the "LISP' Experience.
Computing Surveys 10(11:35-71, March, 1978.

[16] Siklossy, L
Let's Talk Lisp.
Prentice-Hall, 1976.

[17] Stallman, R.
EMACS: The Extensible, Customizable Self-Documenting Display Editor.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation.*
ACM, June 8-10, 1981.

[18] White, J. L
NIL - A Perspective.
In *Proceedings of the MACSYMA User's Conference, Washington, D.C.,* pages 190-199. June, 1979.

[19] Winston, P. K, and Horn, B. K. P.
LISP.
Addison-Wesley, 1981.