RSD-TR-9-85

COMPARISONS BETWEEN ADA AND LISP

Paul Bhugra

Trevor N. Mudge

October 1985

Robot Systems Division

Center for Research on Integrated Manufacturing

College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109 USA

COMPARISONS BETWEEN ADA AND LISP[1]

Paul Bhugra

Trevor N. Mudge

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

October 1985

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

[1]This work was supported by a grant from the General Dynamics Land Systems Division under contract number DEY-600483.

TABLE OF CONTENTS

1. Introduction 2

2. History 2

3. Basics of Lisp 4

4. Major Dialects of Lisp 6

5. Lisp with Ada 8

6. Lisp for AI applications 8

6.1. Ada for AI applications 9

6.2. Possible Interface 11

6.3. Object-oriented Design with Zetalisp and Ada 17

6.3.1. Zetalisp 19

6.3.2. Ada 21

6.4. Storage Management: 22

7. Conclusion: 25

Abstract

This paper addresses the feasibility of using Ada for programming AI systems, and developing an interface between Lisp and Ada. The object-oriented design methodologies supported by the two languages are also contrasted.

1. Introduction

Lisp is a list processing language that was introduced by John McCarthy during the early 1960's. Unlike Fortran and Algol which are algebraic programming languages, Lisp is a language designed for symbolic manipulation. It has gained unprecedented acceptance from the artificial intelligence (AI) community, at least in the United States, for programming AI applications.

The Ada language has been in existence a short time in comparison to Lisp. It was developed in the early 1980's at the initiative of the United States Department of Defense for programming real time embedded systems. Ada is based primarily on Pascal, but incorporates concepts from many different languages.

This paper addresses the feasibility of using Ada for programming AI systems, and developing an interface between Lisp and Ada. The object-oriented design methodologies supported by the two languages are also contrasted. We begin with the history and an overview of Lisp followed by a summary of its major dialects.

2. History

Lisp has its roots in the Information Processing Language (IPL), which was the first list processing language and was introduced by Newell, Simon, and Shaw in 1956. The first version of Lisp, Lisp I, provided a unique style of programming

which emphasized the use of functional composition and recursion over the traditional imperative constructs. By 1965, Lisp I had become Lisp 1.5 and contained constructs not present in the earlier version. The most controversial of these constructs was the PROG feature, which made it possible to write ALGOL-like programs in Lisp.

In the early 1970's, Lisp 1.5 gave birth to two major dialects of Lisp, BBN-LISP (1974) and Maclisp (1974). BBN_LISP later became Interlisp and is currently used on the XEROX machines. Maclisp had numerous descendants including Lisp 1.6, UCILISP, Franz Lisp, and NIL. Stanford University adopted an early version of Maclisp and called it Lisp 1.6, that later became UCILISP at the University of California, Irvine and Standard Lisp at the University of Utah. Franz Lisp was developed at the University of California, Berkeley for VAX-UNIX and NIL was developed at MIT for VAX-VMS.

In 1981, a committee was formed to standardize the various versions of Lisp. The goals of the standardization process included portability, commonality, consistency, compatibility, and efficiency [Ste84]. This led to the specification of Common Lisp, which is primarily a descendant of Maclisp and a subset of Zetalisp. Whether Common Lisp achieves its goals remains to be seen since Standard Lisp and Portable Standard Lisp (PSL) had similar objectives. The introduction of Common Lisp did slow the evolution of Lisp to a certain extent, but new dialects are still being proposed as research in the areas multiprocessing and strong data typing progresses.

3. Basics of Lisp

The basic data structures in Lisp are lists and atoms. Atoms are fundamental entities in Lisp while lists may consist of atoms or other lists. A Lisp program contains a set of function definitions followed by the application of these functions to specific data structures.

Some of the major features of Lisp include [McC60]:

1) Computing with symbolic expressions (S-exprs) rather than numbers.
2) Representation of S-exprs by list structure in memory.
3) Use of selector and constructor operations expressed as functions.
4) Use of conditional expressions for branching.
5) Representation of program and data in the form of S-exprs.

An S-expr is either:

1) An atom
   A) Numeric
      Ex: 5
   B) Literal
      Ex: ABC55

2) A dotted pair of atoms
   Ex: (A . B)

3) A dotted pair of S-exprs
   Ex: ((A . B) . C)

The dotted notation used in (2) and (3) above can become unreadable when many elements exist. Hence, list notation is often used to make certain S-exprs

more readable. However, only dotted pairs that end in NIL may be represented as lists since NIL is used as a terminator for lists. For example, (A . B) can not be represented as a list but (A . (B . NIL)) can be represented as a list by writing (A B). The rules for transforming S-exprs from dot notation to list notation and vice versa can be found in [McC60].

The various dialects of Lisp are supersets of what some people call "Pure Lisp" [Glo82]. Pure Lisp consists of [Glo82]:

1) Functions CAR, CDR, and CONS

   CAR takes a list as input and returns the first element in the list.

   CDR takes a list as input and returns a list containing all of the elements except the first element of the list.

   CONS takes a list and an atom as input and adds the atom to the front of the list.

2) Predicates EQ and ATOM

   EQ returns T only if its arguments are represented by the same memory cells.

   ATOM returns T if its argument is an atom.

3) Control structures: COND, recursion, function composition, and functional parameters.

   COND allows conditional expressions to be implemented.
   Support for recursion and functional parameters allows functions to call themselves and functions to be passed as parameters.

4) Data structure: S-expressions

5) Label and Lambda constructs.

   Lambda allows new functions to be defined by the user, while
   Label allows names to be associated with the functions.

A simple function to append one list, Y, onto another list, X, can be written as follows:

```
(defun APPEND (X Y)
  (cond ((eq nil X) Y )
        (T (cons (car X) (append (cdr X) Y)))
```

4. Major Dialects of Lisp

Most dialects of Lisp support Pure Lisp as defined in the previous section. They differ primarily in the environments, data types, scope rules, binding rules, and data typing rules they provide. Most Lisp systems offer an excellent, integrated development environment to the programmer. The environment generally consists of an interpreter, debugger, multiple windows, and other features specific to each manufacturer. Data types traditionally restricted to procedural languages, such as bitstrings and arrays, are now supported by many dialects of Lisp. Some dialects use dynamic scope rules while others use lexical scope rules. Dynamic scope rules imply the scope of a variable depends on program execution, whereas with lexical scope rules, the scope of a variable can be determined at compile time. The binding used may be either shallow or deep depending on the dialect. If deep binding is used, the nonlocal reference environment of a procedure instance is the same as the environment in force when the procedure was created. If shallow binding is used, the nonlocal reference environment of a procedure instance is the one in force when the procedure is invoked. Conventional languages such as Algol and Pascal use static scope rules combined with deep binding while many Lisp dialects use dynamic scope with shallow binding.

Maclisp is one of the few dialects of Lisp which can be used efficiently for numerical computation. It also supports shallow binding and uses a partitioned memory. By storing different types of data in different parts of memory, Maclisp can quickly determine the data type from a data address. Interlisp uses deep binding and cdr-coding to store lists efficiently in memory [section 5.5]. Franz Lisp is written primarily in the C language, and is known for its ability to interface to C, Pascal, and Fortran subprograms. Zetalisp provides constructs for data abstraction [section 5.4] and also uses cdr-coding for storing lists. Sill is a dialect of lisp designed primarily for vision applications. It attempts to increase execution speed by taking advantage of representational information thru the use of strong data typing. Common Lisp is the most recent of the dialects. It supports lexical scope rules and various data types including complex numbers, arrays, and vectors.

Fundamental differences exist between the Lisp dialects described above and the Lisp language defined by McCarthy [McC60]. Lisp continues to incorporate concepts and constructs commonly found in procedural languages such as Algol-60. For instance, some dialects of Lisp support iterative constructs such as a 'Do-loop' as well as lexical scope rules. It appears newer dialects of Lisp will continue to incorporate ideas from procedural languages unless Common Lisp becomes a standard and no further dialects of Lisp are proposed.

5. Lisp with Ada

Lisp has traditionally been the language of choice for developing AI software. We describe some features of Lisp which make it suitable for AI applications [section 5.1] and try to determine whether the Ada language can be used for such applications [section 5.2]. In section 5.3, possible interfaces between Ada and Lisp are outlined. Support for a structured design methodology is often an important criterion when selecting an implementation language. The support of an object-oriented design methodology for developing AI software in a dialect of Lisp, Zetalisp, as well as in Ada is reviewed in section 5.4. Finally in section 5.5, we discuss a crucial aspect of any AI software, storage management.

6. Lisp for AI applications

To understand why Lisp is widely used for AI programs, we will first attempt to characterize such programs. AI programs tend to have a short development cycle and rarely involve teams of programmers. Often, the design and implementation phase occur concurrently. When describing AI programming, phrases such as "programming under uncertainity" [She83], "exploratory programming" [She83], and "algorithms in flux" [SMS80] are common.

Lisp is ideal for exploratory programming since Lisp, as specified by McCarthy, is a typeless language with dynamic scope and dynamic binding. The lack of explicit typing and static binding allows the programmer to delay design decisions until run time, and in a sense, explore different alternatives easily.

AI applications, in particular expert systems, require extensive pattern matching facilities. Pattern matchers can be implemented efficiently in Lisp since Lisp was designed specifically for symbolic processing.

Massive storage is necessary to explore alternative solutions to a problem. Consequently, most Lisp implementations provide sophisticated storage management facilities to avoid memory depletion [section 5.5].

The identical representation of program and data allow lisp programs to modify themselves. This feature may be appropriate for applications involving learning but is inappropriate from a software engineering point of view.

6.1. Ada for AI applications

Some of the above features, which are good for exploratory programming are actually detrimental for programming large software systems. For instance, the lack of strong data typing combined with dynamic scope rules can make it very difficult to integrate and debug a complex software system. Alternatively, Ada provides strong data typing, static scope rules and a number of other features which support structured programming of large software systems. Thus, if we could use Ada for AI applications, we could use structured programming techniques for developing AI software.

Ada can be used for many AI applications and, in addition, allows structured programming techniques to be used for implementing such programs. There is at least one dialect of Lisp, Zetalisp, which attempts to support certain

software engineering concepts such as data abstraction and object-oriented programming. However, as we will see in section 5.4, there are differences between the object-oriented design methodology supported by Zetalisp and that supported by Ada.

Although Ada does not incorporate certain features of Lisp such as functional parameters and the identical representation of program and data, Ada can still be used for a number of AI applications. For example, an expert system can be viewed as an interpreter for a higher level language defined by the system designer. We can implement an expert system in Ada simply by writing an interpreter which maps rules onto Ada data structures. A possible data structure for list cells in Ada is specified in the next section.

Ada's ability to efficiently execute numerical computations can be very useful when writing expert systems involving uncertainty. In such a system, probabilities are associated with rules and extensive calculations may be required to determine the validity of a deduction.

An AI application implemented in Ada will probably require more lines of code than its Lisp counterpart, but it will be more readable and require less maintenance. For instance, a system developed by Intellimac, Inc. for translating English text to French required only three per cent of the total labor hours for maintenance [Nae85]. The same system implemented in Lisp would probably have required a larger maintenance phase and a smaller design phase.

In light of the arguments for both Lisp and Ada for AI applications, perhaps the best approach is to take advantage of features of both languages at different points in the design cycle. This can be achieved by using Lisp for developing prototype systems and using Ada for implementing the final system: the exploratory programming features of Lisp can then be used for developing prototypes and the structured programming aspects of Ada can be used for the final implementation. It would also be useful to have an interface between Ada and Lisp for algorithms which are awkward to express in Ada.

6.2. Possible Interface

An interface between Ada and Lisp can be achieved but it would include certain disadvantages as well as advantages. The advantages of such an interface would be:

1) The ability to use Lisp to implement artificial intelligence techniques and to use Ada for time-critical activities.

2) Reduced application development time; we could select which language to use, either Lisp or Ada, depending on the given problem.

3) The ability to use current Lisp programs with Ada programs

The disadvantages of such an interface would include:

1) The loss of important features of each language depending on the interface used. For instance, if we chose a translator as the interface, we would lose certain features of each language. A Lisp to Ada translator would lose the following features of Lisp: typelessness, late binding, and the identical representation of data and program. If we used an Ada to Lisp translator, we may lose the strong typing of Ada depending on the complexity of our translator.

2) Performance degradation.

The interface between Ada and Lisp can be implemented in different ways. We list four possible methods along with our comments on each method.

To call an Ada subprogram from Lisp:

i) Compile the Ada subprogram with an Ada compiler and either link the resulting object module with compiled Lisp code or load the object module into the Lisp interpreter.

The simplest way of implementing this method would be to write a Lisp interpreter in Ada and compile it with a validated Ada compiler. Other Ada programs could then interface with this Lisp interpreter, provided they were compiled using the same compiler.

An example of a similar system is the Franz Lisp interface with the C programming language. It is possible to write C programs, compile them using the C compiler, and then load them into the Franz Lisp interpreter. This is possible because the majority of the interpreter is written in C. This provides a clean interface between Lisp and C, simply by using the C parameter passing conventions. Of course, there are restrictions about what types of data structures the C program can use. It is the responsibility of the programmer to ensure that the C program returns values that can be recognized by Lisp, such as lists, integers, etc.

ii) Translate Ada source code to Lisp source code and then execute compiled Lisp code.

This would not be useful for real time embedded systems since Lisp does not provide primitives for real time control.

To call a Lisp function from an Ada program:

i) Compile the Lisp function using a Lisp compiler and link resulting object module with Ada object module.

This would be a reasonable interface but it would require

significant modifications to the Lisp compiler to match the parameter passing conventions of Ada. Verdix attempted this interface by modifying the back end of its compiler but was unable to get the interface to work.

ii) Translate Lisp code to Ada and execute compiled Ada code.

This is the best approach for interfacing Lisp and Ada since it retains the real time features of Ada and allows the symbolic processing features of Lisp to be incorporated into Ada. Furthermore, it does not require any modifications to either compiler.

Both Intellimac, Inc. and Systems Research Laboratories use this approach in their Ada/Lisp systems. These systems take a Lisp function as input and translate it into Ada code. The Ada source code is then compiled and executed. The exact implementation that Intellimac and SRL use is unknown to us, but we can think of at least one possible implementation of such a system which we will describe at this point.

To construct a Lisp to Ada translator, two major problems have to be addressed. First, a data structure has to be defined to which we can map Lisp atoms and lists. Second, the Lisp primitives have to be implemented in Ada and made visible to the translated function.

Both the data structure and the Lisp primitives could be specified within an Ada package. [1] The data structure could be specified by using a variant record, where each value of the variant corresponds to a particular data structure in Lisp. For instance, we would need to support, as a minimum, a representation for an atom and a list. The Lisp primitives could be handled by including in the package, the specification and implementation of various Lisp primitives. Each translated function would have access to the basic data structure and the

[1] An Ada package is a tool which allows us to encapsulate design decisions.

Lisp primitives by including this package within its scope. The specification of an Ada package containing the data structure and the primitives car, cdr, cons, eq, and atom is shown on the next page.

```
with TEXT_IO; use TEXT_IO;
package LISP is

type STYPE is (UNDEF,LIST,INT,SYMBOL,STRING_T );

type LCELL(STAG: STYPE) is private;

type LCELL_ACCESS is access LCELL;

function CAR (ITEM1:  LCELL_ACCESS) return
        LCELL_ACCESS ;

function CDR (ITEM1:  LCELL_ACCESS) return
        LCELL_ACCESS ;

function CONS (ITEM1:  LCELL_ACCESS; ITEM2: LCELL_ACCESS) return
        LCELL_ACCESS ;

function EQ (ITEM1:  LCELL_ACCESS; ITEM2: LCELL_ACCESS) return
        BOOLEAN ;

function ATOM (ITEM1:  LCELL_ACCESS) return
        BOOLEAN ;

private                 -- define private data structure

 type LCELL (STAG: STYPE) is record
  case STAG is

    when UNDEF => NULL;

    when LIST => CAR: LCELL_ACCESS;
                 CDR: LCELL_ACCESS;

    when INT => INTVAL: INTEGER;

    when SYMBOL => PLIST: LCELL_ACCESS;
                   VALUE: LCELL_ACCESS;
                   PNAME: LCELL_ACCESS;
                     DEF: LCELL_ACCESS;

    when STRING_T => TEXT: string(1..256);

  end case;
 end record;

end LISP;
```

An example will help to clarify the use of this Lisp to Ada translator.

Let us assume we would like to call the append function of section 3, which appends one list onto another, from our Ada program.

The append function was written in Lisp as follows:

```
(defun APPEND (X Y)
  (cond ((eq nil X) Y )
        (T (cons (car X) (append (cdr X) Y)))))
```

The translator would accept the above function as input and output the following Ada function:

```
with TEXT_IO; use TEXT_IO;
with LISP; use LISP;
function APPEND (ITEM1: LCELL_ACCESS; ITEM2:LCELL_ACCESS) return
        LCELL_ACCESS is
begin
  if  CAR(ITEM1) = null THEN
    return ITEM2;
  else
    return (CONS(CAR(ITEM1), APPEND(CDR(ITEM1), ITEM2)));
  end if;
end APPEND;
```

We could then call this append function from our Ada program by writing

```
append(X, Y)
```

where X and Y are both defined to be of type LCELL defined earlier.

Intellimac, Inc. ran benchmarks with their Lisp to Ada translator to determine the efficiency of their translator. They concluded the test function they used, which was highly recursive, ran 200 times slower if it was coded in Lisp

(interpreted) than if it was coded in Ada. If the Lisp function was translated to Ada using their translator, it ran 29 times slower than the original Ada implementation. The inefficiency of the translated Lisp function is partially due to the additional code (80KB) necessary to support the Lisp run time system. In an embedded system, we might be able to reduce the size of this by including only the essential parts of the run time system.

6.3. Object-oriented Design with Zetalisp and Ada

The design methodology used in designing a complex software system can have a major influence on its reliability. It is desirable to use a methodology which promotes system flexibility and reliability. Conventional design, or top-down design, consists of decomposing the high level function of the system into subfunctions, where each subfunction corresponds to a major processing step. The idea being to embody each major processing step within a module. This leads to designs that may be structured from a procedural point of view but not necessarily from the point of view of data. Consequently, data may be accessed by many different modules, thus making the system highly inflexible. For instance, if we needed to change the implementation of a particular data structure, we would have to modify all the modules which accessed this structure directly. Furthermore, system reliability is reduced since the probability of one of these modules corrupting the data structure is higher.

A better approach to system design, one which places an emphasis on data

structure design as well as procedural design, is the object-oriented design methodology. Object-oriented design attempts to embody design decisions in modules and to restrict this knowledge to as few modules as possible. The criterion used to decompose a program into modules using object-oriented design is information hiding[Pa72b]. This design methodology can be summarized as follows [Boo82]:

1) Define the problem
2) Develop an informal strategy
3) Formalize strategy
    a) Identify objects and their corresponding operations
    b) Define the interfaces
    c) Implement operations

Related to information hiding is the concept of data abstraction, which may be realized by an abstract data type. Our purpose for using data abstraction is to abstract the details of data objects. Liskov [Lis74] defines an abstract data type as:

> a class of objects which is completely characterized by the operations which may be performed on those objects.

An example often used for an abstract data type is a stack. One possible implementation of a stack is an array of elements with the operations pop and push defined on objects of type stack. The actual implementation of the stack could be embodied in a module, while operations pop and push could be speci-

fied in the module interface. By hiding the representation of the stack within a module, it is possible to change the implementation of the stack simply by modifying this module.

The only operations available to users of objects of type stack are those defined by its interface, namely pop and push. The stronger this interface is enforced, the more difficult it is for a user to access information regarding the implementation of the stack and possibly modifying it. As a result, programs designed using an object-oriented design philosophy are more reliable and flexible, provided the object interface is enforced. We will now consider the implementation of an abstract data type in both Zetalisp and in Ada.

6.3.1. Zetalisp

An abstract data type is implemented in Zetalisp by defining a convention for calling functions and by using a language feature called Flavors. The function calling convention allows operations to be performed on objects by defining a way of sending messages to objects. A message is sent to an object by calling the object as a function with the name of the message as the first parameter and the arguments of the message as the rest of the parameters. No new construct is added to Lisp for sending messages--only a convention is defined. However the language feature, Flavors, has been added to define abstract data types and instantiate objects of a particular data type.

An abstract data type definition for a ship can be defined in Zetalisp as

follows [Sym84]:

```
(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity 5.0)
                 (y-velocity 7.0)
                  mass)
                 ()
                  : gettable-instance-variables
                  : settable-instance-variables
                  : initable-instance-variables)
```

This defines the abstract data type ship, and automatically generates methods which allow the instance variables of objects of type ship to be accessed, set, and initialized. A method is a function which handles a specific message to an object while an instance variable is analogous to a record component in Pascal. We can define additional methods by using the defmethod construct. For example, a method to calculate the direction of a ship would look like:

```
(defmethod (ship:direction) ()
  (atan y-velocity x-velocity))
```

We could create an instance of type ship simply by typing:

```
(make-instance 'ship)
```

We could access the x-position of an object of type ship, say ship5, by using the following form:

```
(send ship5 ':set-x-position 15.0)
```

It is important to note that no attempt is made in Zetalisp to prevent the user from using other functions to circumvent the object-oriented style of programming. If the user does not abide by the rules of object-oriented programming he is said to be "in error".

6.3.2. Ada

An abstract data type may be implemented in Ada by using a package. As stated earlier, an Ada package is a tool which allows us to enforce our abstractions. For instance, we can implement the ship data type of the previous section in Ada as follows:

```
package SHIP_PKG is
  type SHIP_REC is limited private;
  type SHIP is access SHIP_REC;
  procedure GET-X-POSITION (OBJECT: in SHIP; X_POS: out INTEGER);
  procedure GET-Y-POSITION (OBJECT: in SHIP; Y_POS: out INTEGER);
  procedure GET-X-VELOCITY (OBJECT: in SHIP; X_VEL: out INTEGER);
  procedure GET-Y-VELOCITY (OBJECT: in SHIP; Y_VEL: out INTEGER);
  procedure GET-MASS   (OBJECT: in SHIP; SHIP_MASS: out INTEGER);
  procedure SET-X-POSITION (OBJECT: in SHIP; X_POS: in INTEGER);
  procedure SET-Y-POSITION (OBJECT: in SHIP; Y_POS: in INTEGER);
  procedure SET-X-VELOCITY (OBJECT: in SHIP; X_VEL: in INTEGER);
  procedure SET-Y-VELOCITY (OBJECT: in SHIP; Y_VEL: in INTEGER);
  procedure SET-MASS   (OBJECT: in SHIP; SHIP_MASS: in INTEGER);

  private
    type SHIP_REC is record
      X_POS: INTEGER;
      Y_POS: INTEGER;
      X_VEL: INTEGER;
      Y_VEL: INTEGER;
      MASS : INTEGER;
    end record;
end SHIP_PKG;
```

A program can instantiate objects of type ship by including the above package within its scope and using the 'new' operator. A critical difference between the Ada and Zetalisp implementations of ship is the level of enforcement each provides for our abstraction. By declaring ship to be limited private, we can restrict the operations a user can perform on objects of type

ship to be the ones we have defined.[2] In the Zetalisp version, however, the user is free to use functions which are not explicitly defined for our abstract data type. By using functions other than the ones we have explicitly defined, the user may be able to determine the implementation of an object of type ship and possibly modify it.

Symbolics, Inc., which uses Zetalisp on its Lisp machines, makes no attempt to forbid constructs which allow a caller to determine the implementation of a particular data structure. In fact, Symbolics claims their implementation of abstract data types aids in the debugging process. However, for programming large software systems, we feel it is essential to enforce abstractions to reduce the probability of error.

6.4. Storage Management:

Artificial intelligence applications require a very large physical memory irrespective of the programming language used. We can reduce the amount of memory required by choosing an efficient representation of lists in memory and by reclaiming inaccessible list cells at run time.

Conventional Lisp systems use list cells to store lists in memory. A list cell is a memory cell with two halves; the left half points to the car of the list while the right half points to the cdr of the list. This method of storage is inefficient since both halves of the list cell have to be capable of storing complete

[2] It may be possible to determine the implementation of the object using UNCHECKED PROGRAMMING, but the Ada Reference Manual strongly discourages the use of UNCHECKED PROGRAMMING.

pointers. An alternative representation of lists, called cdr-coding, is used by Symbolics and other Lisp machine manufacturers to reduce storage requirements. Cdr-coding can achieve as much as a 50% reduction in memory usage over the conventional method of list representation. It takes advantage of the linear ordering of memory and allocates a two-bit code for the cdr instead of storing a full pointer for the cdr. Each list cell thus contains a pointer, p, for the car of the list and a two-bit code for the cdr of the list. The code is interpreted as follows [Bak78]:

00: NORMAL; Car of node is p; Cdr is in next cell.

01: NIL; Car of node is p; Cdr is NIL.

10: NEXT; Car of node is p; Cdr is the next cell.

11: EXTENDED; Cell extension at p holds the car and cdr for this node.

The available memory in a Lisp system is initially linked together to form a free list. Each time a new list cell is needed, it is taken off the free list. To prevent the free list from becoming NIL, we need to be able to distinguish accessible cells from inaccessible cells (garbage cells) so that we may add the inaccessible cells onto the free list. This process of reclamation is commonly referred to as garbage collection. In early Lisp systems, garbage collection was the user's responsibility. However, in the process of deallocating unnecessary storage, the user sometimes deallocated active storage, making the program

virtually impossible to debug. To avoid such problems, most Lisp systems today provide automatic garbage collection.

Two methods often used for garbage collection are the mark/sweep method and the reference count method. The mark/sweep method works by marking all cells which are accessible by the program and then adding all unmarked cells to the free list. The cells can be marked either by using a part of each word for marking or by using a separate bit map, where each element of the bit map corresponds to a single memory cell.

In a system using reference counts, each list cell has a reference count associated with it which represents the number of pointers (references) which point to that cell. The reference count of a list cell is incremented each time a new pointer to the cell is created, and decremented when a pointer to the cell is destroyed. When the reference count of a particular list cell reaches zero, it becomes inaccessible (i.e. no pointer points to it) and is reclaimed by adding it to the free list. If a cell is reclaimed, the reference counts of all the cells it points to are decremented and possibly reclaimed in a recursive manner. A major disadvantage of reference count systems is that cyclic pointer structures can not be reclaimed.

Most Ada implementations do not provide a garbage collector as a part of their run time support but some form of garbage collection will be necessary if AI applications are to be supported. Garbage collection may either be provided

by the system or left up to the user. We believe the system should provide some type of a garbage collector which may be requested via a pragma statement. Since Ada is designed for programming real-time embedded systems, the garbage collector should provide some upper bound on its execution time and should use a minimal amount of memory. One possibility is the garbage collector outlined by Baker [Bak78].

This method of garbage collection divides the list space into two subspaces, one of which stores list cells at any one time. The garbage collector moves all accessible cells in one subspace (fromspace) to the other subspace (tospace) leaving forwarding pointers in the fromspace. After all the accessible cells have been copied, the two subspaces switch roles.

7. Conclusion:

Ada may not replace Lisp for programming AI applications, but it may be used for a significant percentage of them. The use of Ada for such applications would allow structured programming techniques to be applied to AI software. We believe Lisp is better suited for developing prototype applications than it is for developing production quality software. As a result, we advocate using Lisp for prototype systems and Ada for the final implementation. For applications which can not be easily implemented in Ada, we have outlined possible interfaces between Lisp and Ada which may be employed.

REFERENCES

[All78] Allen, J. *Anatomy of LISP,* McGraw-Hill, Inc., New York, NY, 1978.

[Bak78] Baker, H. "List Processing in Real Time on a Serial Computer," *Communications of the ACM,* Vol. 21, No. 4, April, 1978, pp. 280-294.

[Bar84] Barnes, J. *Programming in Ada,* Addison-Wesley Publishers, London, 1984.

[Bob72] Bobrow, D. "Requirements for Advanced Programming Systems for Lisp Processing," *Communications of the ACM,* Vol. 15, No. 7, July, 1972, pp. 618-27.

[BoR61] Bobrow, D. and Raphael, B. "A Comparison of List-Processing Computer Languages," *Communications of the ACM,* Vol. 7, No. 4, April 1961, pp. 231- 240.

[BoW73] Bobrow, D. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM,* Vol. 16, No. 10, October 1973, pp. 591-603.

[Boo82] Booch, G. "Object-Oriented Design," *ACM Ada Letters,* Vol. I, No. 3, March - April 1982, pp. 64-76.

[Bro84] Brooks, R. *Programming in Common Lisp,* John Wiley and Sons, 1984.

[BuM83] Buzzard, G. and Mudge, T. "Object-based Computer Systems and the Ada Programming Language," *Technical Report,* CRL-TR-29-83, Computing Research Laboratory, The University of Michigan, Ann Arbor, MI, August 1983.

[ClG77] Clark, D. and Green, C. "An Empirical Study of List Structure in Lisp," *Communications of the ACM,* Vol. 20, No. 2, February 1977, pp. 78-87.

[Den75] Dennis, J. "An Example of Programming With Abstract Data Types," *ACM SIGPLAN Notices,* Vol. 10, No. 7, July 1975, pp. 25-9.

[DoD83] United States Department of Defense, *Reference Manual for the Ada Programming Language,* 1983.

[Glo82] Gloess, P. *Understanding LISP,* Alfred Publishing Co., Inc., Sherman Oaks, CA, 1982.

[Lin79] Linger, R. *Structured Programming,* Addison-Wesley Publishers, Reading, MA, 1979.

[LSA77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU," *Communications of the ACM,* Vol. 20, No. 8, August 1977, pp. 564-76.

[Lis75] Liskov, B. "Data Types and Program Correctness," *ACM SIGPLAN Notices,* Vol. 10, No. 7, July 1975, pp. 16-17.

[Lis74] Liskov, B. and Zilles, S. "Programming with Abstract Data Types," *ACM SIGPLAN Notices,* Vol. 9, No. 4, April 1974, pp. 50-59.

[Mc78a] McCarthy, J. "A Micro-manual for Lisp - Not the Whole Truth," *ACM SIGPLAN Notices,* Vol. 13, No. 8, August 1978, pp. 215-16.

[McC60] McCarthy, J. "Recursive Functions of Symbolic Expressions," *Communications of the ACM,* Vol. 3, No. 4, April 1960.

[Mc78b] McCarthy, J. "History of Lisp," *ACM SIGPLAN Notices,* Vol. 13, No. 8, August 1978, pp. 217-23.

[Nae85] Naedel, R. "Ada and Artificial Intelligence," *Technical Report,* Intellimac, Inc., Rockville, MD, 1985.

[Pa72a] Parnas, D. "A Technique for Software Module Specification with Examples," *Communications of the ACM,* Vol. 15, No. 5, May 1972, pp. 330-36.

[Pa72b] Parnas, D. "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM,* Vol. 15, No. 12, December 1972, pp. 1053-58.

[Ric83] Rich, E. *Artificial Intelligence,* Addison-Wesley Publishers, Reading, MA, 1983.

[Ros66] Rosen, S. *Programming Systems and Languages,* McGraw-Hill, Inc., New York, NY, 1966.

[Sam69] Sammet, J. *Programming Languages: History and Fundamentals,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1969.

[SMS80] Schwartz, R. and Melliar-Smith, P. "The Suitability of Ada for Artificial Intelligence Applications," *Technical Report,* SRI International, Menlo Park, CA, May 1980.

[She83] Sheil, B. "The Artificial Intelligence Tool Box," *Proceedings of the NYU Symposium on Artificial Intelligence and Business,* 1983.

[Sik76] Siklossy, L. *Let's Talk Lisp,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[Ste84] Steele, G. *Common LISP: The Language,* Digital Press, Maynard, MA, 1984.

[Ste82] Steele, G. "Report on the 1980 LISP Conference," *ACM SIGPLAN Notices,* Vol. 17, No. 3, March 1982, pp. 22-35.

[Sym84] Symbolics, Inc. *Documentation for Lisp Machine,* Symbolics, Inc., Cambridge, MA, 1984.

[VMG83] Volz, R., Mudge, T., and Gal, D. "Using Ada as a Robot System Programming Language," *Proceedings of the 13th International Symposium on Industrial* Robots and Robots 7 Conference, April 1983, pp. 12-42 thru 12-57.

[WiB79] Winston, P. and Brown, R. (editors) *Artificial Intelligence: An MIT Perspective,* Vol. 2, The MIT Press, Cambridge, MA, 1979.

[Win84] Winston, P. *Artificial Intelligence,* Addison-Wesley Publishers, Reading, MA, 1984.

[WiH84] Winston, P. and Horn, B. *LISP,* Addison-Wesley Publishers, Reading, MA, 1984.