

# Lisp Machines and the Analysis of Their High-Level Language Computer Architecture

Dominic Dabish and Evan Bradley

School of Engineering and Computer Science

Oakland University, 2200 N Squirrel Rd, Rochester, MI 48309

{dadabish, edbradley}@oakland.edu

**Abstract**—The high-level capabilities that Lisp affords programmers makes it very useful in certain applications, but also causes it to run slow on most computers, which were designed for different types of executions. To account for the incompatibility between Lisp and the hardware on which it is executed, computers called Lisp machines have been designed specifically to run Lisp programs by designing the processor hardware to accommodate the language features Lisp possesses. Achieving greater efficiency in running Lisp was accomplished using various methods, both in the hardware and in how data would be represented in the software. Hardware alterations largely consist of changes to the circuitry in the processor itself as well as the number and purpose of processors in the machine. Software changes mostly correspond to the representation of lists, the main data structure in Lisp. Lisp machines were largely deprecated in the 1990s, giving way to traditional computers which were fast enough to run Lisp with reasonable speed. Despite this, Lisp machines remain a popular subject for study and discussion amongst those who are familiar with Lisp.

**Keywords**—Lisp, Lisp Machine, computer engineering, vintage computing, computer organization.

## I. INTRODUCTION

The history of Lisp begins with Alonzo Church's introduction of lambda calculus in the 1930s, where Church attempted to reconcile mathematics and computation. Lisp takes many notational cues from lambda calculus, and in a sense serves as a practical implementation of the ideas Church presented. Lisp machines were created as a response to the decreasing costs and increasing complexity of computers in the 1970s, and were rendered virtually defunct in the 1990s by the same conditions that had created them.

### A. Early history of Lisp

John McCarthy created Lisp while working to create a new computer language to replace languages like the Information Processing Language (IPL) and Fortran, which he saw as cumbersome and insufficient for programming artificial intelligence (AI) [stoyan]. He began work on what was then called lisp LISP, or the LISt Processing language, in 1958 while

working at the Massachusetts Institute of Technology (MIT), and published a Communications of ACM paper in April of 1960 that detailed the basic workings of the language he constructed [stoyan, mccarthy]. By April 1958, Steve Russell had implemented some Lisp functionality in machine language on the IBM 704, much to the surprise of McCarthy, who believed his work to be entirely theoretical before seeing the implementation. Development of Lisp began to occur in a more concrete fashion after the actual implementation of Lisp code on the IBM 704. McCarthy and his team then began to focus on creating a usable version of Lisp in addition to creating something that works in theory [stoyan].

In the late 1960s, Jon L. White published a paper describing MACLISP, a variation of Lisp that ran on the Digital Equipment Corporation's PDP-6 [stoyan]. This was followed by the creation numerous other Lisp dialects by individuals and groups who were not directly related to McCarthy. Many of these were combined into Common Lisp in the 1980s [Wikipedia]. One of these dialects was Scheme, which, along with Common Lisp, continues to be one of the two major Lisp dialects to date. It was created in 1975 by Guy L. Steele and Gerald Jay Sussman at the MIT AI Lab, and on whole is much simpler than Common Lisp.

### B. History of Lisp machines

Lisp machines were some of the first single-user workstations, which at the time were an exception to the common timesharing systems where multiple users would connect to a single machine through terminal stations. This change came in part due to Lisp's nature of consuming the entirety of the computer's resources, which made its use difficult on multiple-user systems [andromeda]. Two main companies emerged from the increasing demand for Lisp Machines: Symbolics and Lisp Machines, Incorporated. Larger companies began to produce Lisp Machines as well, including Xerox and Texas Instruments.

These machines were often seen as valuable for not just their ability to run Lisp, but for the advanced features they offered in terms of graphical user interfaces (GUIs) and programmability due to the flexibility that Lisp afforded programmers. Despite their advances in usability, they were often behind in terms of hardware due to their niche audience, they sold to a much smaller market than general-purpose machines, which quickly raced past Lisp machines in terms of hardware capability

---

This paper was submitted for review on March 25, 2016. It was produced with help from Oakland University's subscription to publication sources with information on Lisp and Lisp machines.

D. Dabish and E. Bradley are Computer Science students at Oakland University in Rochester, MI, 48309.

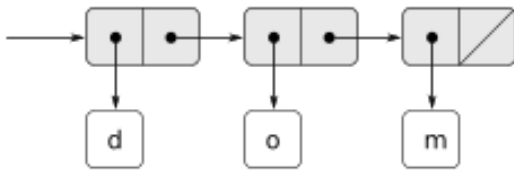


Fig. 1. Box-and-pointer diagram representing an example of a sequence chain. This particular chain contains the sequence of the elements d, o, m. The empty box represents nil.

[withington]. Eventually general-purpose computers became faster at running Lisp despite the hardware incompatibilities, and in combination with the slow development of AI in the early 1990s, most Lisp Machine vendors either left the market or went bankrupt.

## II. THE BASICS OF LISP

### A. Lists and functions

Lisp programs is constructed using functions that invoke each other. After the nest of functions are evaluated, it is typical of Lisp programs to evaluate to one final value.

All expression in Lisps are written in forms. One of the most common forms includes an application of function. For example, the function call in common mathematics  $g(x)$  is represented in lisp as `(g x)`. In this example,  $g$  is the name of the function and the argument(s) follow. Furthermore,  $x$  is the only argument in this example.

Numbers and some other basic expressions such as booleans qualify as /self-evaluating forms/, which means that those expressions evaluate to the same values.

Lisp is a language that was built in mind for data abstraction. Data abstraction can isolate a compound data object and its usage from the fabrics of how primitive data objects are constructed. The reason that data abstraction is necessary is to provide foundation to a program in such a way that tasks can be performed without having information about the data beforehand. This is different from a concrete representation of data, which can be defined independently from the program which will handle said concrete data.

In Lisp, `car` and `cdr` are primitive operations that can be performed on lists and other data objects. They are used to extract from lists and manipulate them. The `car` operation extracts the first element from the list, and the `cdr` operation extracts the rest of the list. For example, consider a list with three elements:

```
'(d o m)
```

The following expression involving the `car` and `cdr` operations on said list evaluate to the following:

```
(car '(d o m))
```

evaluates to `d`, and

```
(cdr '(d o m))
```

evaluates to `'(o m)`.

These are classified as procedures operated on list objects. However, this is not the same case as pairs. When given the following pair:

```
(cons D d)
```

and operate `car` and `cdr` on the pair, however, the evaluation would return to `D` and `d`, respectively.

Pairs are useful when building a sequence in lisp. A sequence is defined as an ordered collection of data objects. Sequences are one of the fundamental building blocks to perform recursive and other useful procedures in Lisp. As shown in the Figure 1, the sequence d, o, m is represented as a chain of pairs.

```
(cons d (cons o (cons m nil)))
```

There are also procedures that can perform operations on lists. For example, the procedure `*reverse*` take a list as an argument and returns a list with the same elements, but in reverse order. In this example, the s-value `(reverse '(d o m))` evaluates to `'(m o d)`.

### B. Environments

Environments in lisp is a specific kind of datatype that allows one to define a set of variables that are bound to specific values. The actual datatype for the environment can be implemented in a variety of ways. Using a data-centric approach, we can introduce the environment datatype as follows:

```
Env :=
(empty-env) | (extend-env var val Env)
```

Each datatype includes constructors and observers, which Lisp enables the ability to define and construct. An environment is comparable with a dictionary, in the fact that, when using an extractor, the function must search for the variable (word) and, once found, will then be able to extract the value (definition). Environments in Lisp share some of the fundamental properties as the logic units in Lisp machines do.

### C. Recursion

Lisp conventional programming emphasizes on recursion for its uses in environments and lists. Recursive functions in lisp are functions that refer to themselves in the function-body. Typically, the recursive call argument differs from each step of when the function recurs. A classic example in Lisp is the factorial function. As shown below, the factorial function makes use of a decreasing integer with a base case that enables the program to terminate.

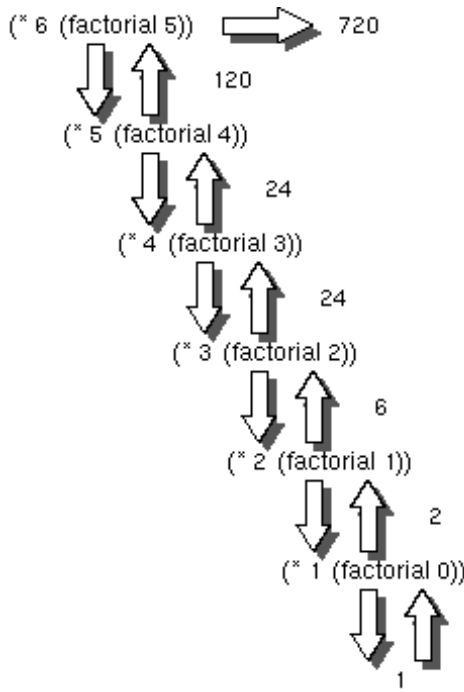


Fig. 2. Diagram of complete recursive evaluation of (factorial 5)

```
(defun factorial (n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))
```

In this example, the program checks if the argument  $n$  is equal to 0 (the base case). If so, the function evaluates to 1 and terminates; otherwise, the function will multiply the argument by the factorial with  $n-1$  as the argument instead. This creates a chain similar to Fig. 1, and evaluates to the factorial of  $n$ .

A way that recursion is commonly used in Lisp includes list-based manipulation is shown below. In this classic example, which shows another design pattern that is possible and common in this language, is the `list-of-all?` function, which has two arguments, a `/predicate/` and a `/list/`, and evaluates to the value `#t` (true) or `#f` (false), based on whether or not the entire list contains elements that satisfy the predicate function requirements. The example below demonstrates how a recursive function manipulates a list until the list empties.

```
(define (list-of-all? predicate lst)
  (or (null? lst) (and (predicate (car lst))
    (list-of-all? predicate (cdr lst)))))
```

This example evaluates to `#t` and terminates given an empty list, which is the base case. Otherwise, it will extract the first element of the list and check the predicate against it. If the check evaluates to `#f`, then the function will evaluate completely to `#f`, and not check the rest of the list. However, if `#t`, the function will check the rest of the list, disclusing

the first element. This function will therefore terminate when an element that does not match the predicate is found, or when every element of the list has been checked on by the predicate.

Recursion, which is a main focus for the language of Lisp, has implications of its own. Being a major part of the language, these language features come with their own problems, namely when it comes to executing functions and recursive calls on large lists. In the next section, these problems will be identified clearly and addressed in the subsequent section.

### III. IDENTIFIABLE PROBLEMS IN THE EXECUTION OF LISP CODE

In A 1987 survey of Lisp machines, Andrew Pleszkun and Matthew Thazhuthaveetil concluded that there were four main problems engineers needed to address when designing a Lisp machine: function calls, environment maintenance, efficient list representation, and heap performance. There were multiple solutions to some of these problems, at the time of their survey, methods of improving these aspects in Lisp machines were seen as valuable topics for further research [pt].

#### A. Problems in Execution

Function calling is a significant feature in Lisp due to its recursive nature, and in general is expensive due to the operations that must be performed for an invocation or return. In particular is the problem of dealing with variables, which must be accessed in a way that costs a minimal amount of time, due to the high frequency with which functions are accessed. Another key component is the versatility of functions in Lisp, which in some cases can contain different types of arguments, a variable number of arguments, and can sometimes be passed as arguments themselves [pt].

Integral to the problem of function calling is that of maintaining the environment associated with each function. The current environment, which contains the set of variables within the function and their associated values, is changed every time a function is called. As a result, the environment that exists outside the body of the function must be stored during the function's execution, which is not necessarily difficult, but must be implemented in such a way that it is fast. A more complicated problem is that of accessing variables outside of the function, which are contained in other environments. This is called the `funarg` problem, which presented difficulties to McCarthy and his team even before it became a focus for Lisp machine engineers [stoyan, pt].

Lists are the primary data structure of Lisp, and as such a significant portion of execution time in a Lisp program is spent traversing these lists. Therefore, reducing the time required to traverse a list is seen as an important goal in creating a machine optimized to run Lisp. A critical component in achieving this is efficiently determining how data should be handled, which is checked at runtime due to Lisp's lack of typed variables. This is often done through tagging the type of data referred to in the list, typically through hardware solutions [pt].

Coupled tightly with the management of lists is the heap, which contains all of the referenced Lisps in a Lisp program

and holds them in memory [pt]. Garbage collection is one solution that Lisp proposes to ensure proper memory management without burdening the programmer, but efficient garbage collection is challenging, and presents a difficulty in both hardware and software implementations.

### B. Solutions in Lisp Machines

In their survey, Pleszkun and Thazheuthaveetil found three different general approaches to Lisp machine design, which in turn affected how these problems were solved. The first class of machines contain processors that were not altered on a hardware level, but in the microcode. The second class contains multiple processors that each have specific functionality to assist in executing Lisp code. The third class contain multiple identical processors and function by executing multiple functions in a program concurrently. They call these distinctions Class M, S, and P machines, respectively [pt].

In solving the problem of evaluating the arguments of functions the Class M and Class S machines will typically allow for code to be run in different formats, the fastest being a compiled code that tags the arguments types and largely bypasses the evaluation process, and the slowest being interpreted code that runs them directly. The Class P machines attempt to solve this through passing each argument to a different processor for evaluation, and in the case one machine, even allowing incomplete results to be sent to encourage further parallel execution [pt].

Two primary implementations of the environment are called *deep binding* and *shallow binding*, which serve as variations on a linked list of name-value pairs of functions and referencing contexts of variable associations. Deep binding makes use of a simple linked list of name-value pairs that is searched every time the function is called, and in the worst case the entire list must be traversed. Shallow binding uses a table that holds the name-value pairs. While this allows for faster function calling, the values in any variables that are overwritten have to be stored elsewhere while the function executes.

While lists function quite well as a way of mathematically representing data, they can be computationally inefficient. Therefore, some Lisp machines have represented them in more compact forms, which the authors classify as *vector-coded* and *structure-coded*. Vector-coded representations assume that a list is linear, or that none of its elements are themselves lists. Making this the default case speeds up the evaluation of list elements, with elements that point to other objects treated as exceptions that will evaluate slower. Structure-coded representations attach tags to each list cell that identifies its place in the list. This information allows the list to be traversed quickly, and allows for fast access to individual cells. The implementations of each of these representations vary from machine to machine, as there are multiple implementations of each [pt].

Additionally, tail recursion is a common programming technique and design pattern to follow that increases speed tremendously, and decrease memory usage. Using traditional recursion, typically calling the function recursive is preceded by calculations. Consequently, the final evaluation is dependent

on the complete transaction of the recursive calls. However, when using tail recursion, the calculations are all completed prior to executing recursive calls. Tail recursion takes the evaluation of the current step before passing it into the next recursive step. It is more optimal due to the fact that the current memory allocation is then no longer required.

Lisp delegates the management of the heap to the system rather than the programmer, which necessitates the need for an algorithmic method in which heap data can be allocated when needed and deallocated when it is no longer wanted. The limited amount of memory available to systems has made the reclamation of used space the most researched aspect of heap maintenance, which is commonly referred to as *garbage collection*. Garbage collection is a two-stage process: the system must first identify memory locations that should be reclaimed, and then must simply identify them as such. Two common methods of garbage collection were present in most Lisp systems: *reference counting* and *marking*. Reference counting consists of keeping a list of the number of references for each object in the code and marking it as free space once that number goes to zero, but suffers from numerous drawbacks, such as the memory requirement of keeping such a list. Marking is more common in Lisp machines, and consists of marking each object with a one-bit field to indicate whether it should be removed. Once all available heap space has been allocated, any the memory cells containing objects marked for reclamation are overwritten.

### IV. A PROPOSED LISP-BASED PROCESSOR BY STEELE AND SUSSMAN

In March 1979, Steele and Sussman, researchers at the MIT AI Lab in the 1970s, outlined and discussed a processor that they suggested would allow for the efficient execution of Lisp programs under a dialect of Lisp they had created [ss]. While it is uncertain whether their ideas were directly implemented in an actual Lisp machine, they provide solutions to the environment maintenance and efficient list representation problems proposed by Pleszkun and Thazhuthaveetil, providing a great degree of detail in describing their implementation.

The proposed processor would run a simplified version of Lisp, called Scheme, that is a minimal Lisp dialect without many of the features of other dialects, such as Common Lisp. The representation of data structures in Lisp and how the program is traversed are of particular concern, as Lisp's code is itself a data structure. Therefore, their processor is built specifically for traversing the Lisp data structure in an efficient manner. To achieve this, they wrote a Lisp interpreter which functions recursively, operating on a small set of registers to hold pointers to a list memory system which holds the stack for the program's state information between procedure invocations. The evaluation component of the interpreter uses five global variables that simulate registers in a normal machine to hold different components of the program's state. They describe this system as a state machine implementation where the program moves between states as procedures are invoked [ss].

To reduce evaluation times for atoms and lookup times for variables, the representation of list data is altered slightly so

