# Programming on the Amiga: Cambridge Lisp 68000

*By Daniel Zigmond*

*For a computer to become truly successful, it needs both well-designed hardware and quality software. The Amiga clearly has the former, but many skeptics still hold that it is lacking the latter. Cambridge Lisp provides us with some potent ammunition against this claim.*

Although Lisp was one of the first programming languages, it has only recently received much attention from the personal computer world. The recent publicity about Lisp has both helped and hurt the language. Articles about Lisp have certainly encouraged many new people to learn it, but in some cases, they have also spread misconceptions about the language.

One of the most common misconceptions is that Lisp is difficult to learn. On the contrary, Lisp's interactive programming style makes it among the easiest languages to master. In fact, Logo has its roots in Lisp, and its phenomenal success among young people and hobbyists has been derived largely from features inherited from Lisp. Lisp is already used in introductory computer courses at schools such as Carnegie-Mellon University and MIT. Even non-computer science majors at these universities have found the language easy to learn and enjoyable to use.

The second most harmful myth about Lisp is that it is useful only for obscure applications in artificial intelligence research. At one time this was largely true, but because of the power and flexibility of modern versions of Lisp, it is currently used to write everything from games to word processors to operating systems. Lisp is still used in artificial intelligence, but it is also the language of choice for virtually all other applications.

## *Introduction to Cambridge Lisp*

Lisp is an interactive language. This means programming in Lisp is much like having a conversation with the computer. The user types some Lisp code, the computer reads it, processes it, prints a response and waits for the user to type some more. The processing of the code is called *evaluation* and is often abbreviated *eval*. Thus, this conversational cycle is often called the *read-eval-print* loop and is the heart of all Lisp systems.

To start Lisp on the Amiga, simply type LISP. You should see something like:

```
Cambridge Lisp 68000 entered in about 380 Kbytes
Store image was made at 18:15:35 on 23-Apr-85
Lisp version-Vex X   image size = 79856 bytes
Started at 15:10:41 on 10-Jul-85 after 27.00
                                    ·55.2% store used
Input:
```

For the time being, you can ignore everything except Input:. Input: is Lisp's way of telling you that it is ready to begin a conversation. We can ask Lisp to do some simple addition by typing

```
Input: (plus 3 4)
```

to which Lisp will respond with

```
Value: 7
```

and then

```
Input:
```

to signal that it is ready for us to type something else.

There are a few things you should notice about the above conversation. First, you can see that Value: is always typed before the computer's part of our conversation. Second, we used a somewhat strange notation in Lisp. The word *plus* was typed instead of a plus sign, and we put it before the numbers instead of between them. Third, we put our expression within a pair of parentheses.

Of course, addition is not the only thing Lisp can do for us. Some other words we can use in its place are *difference, quotient, remainder,* and *times.* These all use the same format as plus. For example:

```
Input: (difference 150 1)
Value: 149
Input:
```

More complicated problems can be solved by combining Lisp expressions.

```
Input: (times 3 (plus 2 1))
Value: 9
Input:
```

One of the reasons for Lisp's success is that it is not limited to these sorts of mathematical problems. In fact, Lisp stands for *list processing*, which is considered Lisp's most powerful feature. In Lisp, a list is any sequence of data in parentheses, and list processing is simply the manipulation of lists. Our first expression, (plus 3 4), is a list of three elements: plus, 3 and 4. All three of these elements are called *atoms*, because they cannot be broken down into any simpler form. Atoms that are words like difference, times, or even more creative ones like cindy or schoolhouse, are called *symbols*. The special symbols like plus that we can use to tell the computer how to handle data are called *functions*. Lists can contain any kind of data, including other lists. In the above example, there are again three elements, but this time they are the symbol times, the number 3 and the list (plus 2 1).

We can build lists with a function appropriately called *list*. To make a list of two numbers we just type:

```
Input: (list 42 149)
Value: (42 149)
Input:
```

Taking lists apart is just as easy. We use two functions called, for antiquated reasons, *car* and *cdr.*

```
Input: (car (list 42 149 305 7))
Value: 42
Input: (cdr (list 42 149 305 7))
Value: (149 305 7)
Input:
```

As you can see, car returns the first element of the list and cdr returns everything else. To compare two lists we can use the function *equal.* If the lists look the same, we get the value *t*; if not, we get *nil.*

```
Input: (equal (list 1 2) (list 1 2))
Value: t
Input: (equal (list 1 2) (list 2 1))
Value: nil
Input:
```

Functions like equal that return only t or nil are usually called *predicates.*

Another important feature of Lisp is the ability to customize the language by writing new functions. For example, to write a function that returns the second element of a list, we need only type:

```
Input: (de second (x) (car (cdr x)))
Value: second
Input: (second (list 1 2 3))
Value: 2
Input:
```

Using *de* to write our own functions is called *function definition.* We use the atom *x* to mean whatever piece of data comes after the symbol *second* in an expression. When we type (second (list 1 2 3)), Lisp evaluates (list 1 2 3) and substitutes the value (1 2 3) for x in our definition. It then evaluates the expression (car (cdr (1 2 3))), which returns 2, and prints the value.

All of this is only intended to give you a taste of Lisp. There are several good Lisp tutorials available. ►

◀ However, there are some problems with using these tutorials to learn Cambridge Lisp; these will be described later in this article.

## *Cambridge Lisp in Detail*

Cambridge Lisp for the Amiga was developed by Metacomco (201 Hoffman Ave., Monterey, CA 93940). It requires at least 512K and retails for $199.95. It is an incredibly rich Lisp implementation and includes many features that are quite rare for microcomputer Lisps. Cambridge Lisp allows for both single word and infinitely precise integers, as well as standard floating point numbers. Also, Cambridge Lisp has *rational* numbers. These numbers are of the form x/y, where x and y are integers. This allows for very accurate arithmetic operations.

To go along with all these types of numbers, Cambridge Lisp provides a large collection of mathematical functions that perform everything from division and square roots to 24-bit binary shifts, arc cosines and natural logarithms. There are many numeric predicates that cover all forms of comparison and type checking. Three functions are available to convert between types. For example:

```
Input: (rational 1 3)
Value: 1/3
Input: (float 49)
Value: 49.0
Input: (float (rational 1 3))
Value: 0.3333333
Input: (fix 32.0)
Value: 32
Input:
```

Cambridge Lisp has all the list processing functions you would want, including the usual ones like *car, cdr, cons, list, append, nconc, rplaca, rplacd* and *consp*, and some very advanced functions for dealing with lists as sets or trees, association lists, property lists, dotted pairs and circular list structures. It is one of the most complete sets of list processing functions you are likely to find in any Lisp implementation. The symbol manipulation functions are similarly diverse.

While Cambridge Lisp lacks both *do* and *let*, there are many of the standard control structures, like *cond, prog, progn* and the mapping functions, and some more powerful ones to facilitate dynamic non-local exits, conditional branching and complicated iteration. There are many functions to combine these structures and create all kinds of functions. User-defined functions can have a fixed or a variable number of arguments and can take these arguments either evaluated or not. There are also functions for both macro definition and macro expansion.

The stream- and buffer-based input/output functions are very impressive. These include many more than the usual printing and reading functions and additional functions for prettyprinting. The user also has complete control of the readtable through a large set of character functions and predicates.

Cambridge Lisp has many more functions that are extremely rare among microcomputer Lisps. It includes an interface to AmigaDOS, support for general vectors and strings, 266 different error codes and both normal and fluid variables. One of its nicest features is the degree to which Cambridge Lisp can be customized. The user can change the amount of memory Lisp uses, redefine the way Lisp handles errors, fine tune the compiler and debugger or even modify the two basic prompts. Both the *editor* and the *trace* function are included to simplify the writing of long functions, and the *compiler* can increase the speed of the final product code. For storing code, there are functions for saving entire core images as well as individual functions. Functions can be collected into modules that are loaded only when the functions are needed.

## *Caveats*

All of these features make for one of the most advanced Lisp systems I have ever seen. However, Cambridge Lisp makes little attempt to be usable by a novice programmer. Although one can certainly learn Lisp ▶

# The History of Lisp

Lisp is one of the oldest programming languages, second only to Fortran. It was invented by John McCarthy in the late 1950's. Lisp was quickly implemented on many machines, and virtually all versions of Lisp were compatible with each other. The first version of Lisp was known as Lisp 1.5 and was the basis for practically all other dialects.

During the next decade, however, several distinct families of Lisp began to appear and were soon available on a number of computers. There was no guarantee that a program written with a particular version of Lisp would run on another version. Furthermore, in cases where programs could be transported from one Lisp to another, the programs often yielded completely different results. These problems not only existed between computers, but between versions of Lisp on the same computer! Programmers had to keep track of which Lisp they had used and take detailed notes on exactly what the program was supposed to do.

In 1966, a standard Lisp was finally proposed by a group at the University of Utah. The main purpose of this standard was to allow the REDUCE computer algebra program to be used on many computers. This became Standard Lisp.

Standard Lisp underwent many changes until the final specifications were published in 1979. Cambridge Lisp is based on this final standard, although it also borrows some important features from two more modern dialects, called Portable Standard Lisp and Common Lisp. Both of these are widely used in computer science today. This combination makes Cambridge Lisp powerful and flexible, as well as compatible with Lisps on many other computers. ■

◄ using this software, for several reasons, it's not as easy as it should be.

First, Cambridge Lisp was originally written for large mainframes to provide an environment for computer algebra research. This means that the system is not tailored for the Amiga and is missing some of the niceties that Amiga users might expect. While this does not detract from the system as a powerful software development tool, it does make it less friendly than it could be.

Second, the documentation leaves something to be desired. As a reference manual, it is fair, but it makes no effort either to teach Lisp or to give references to good tutorials. The index is poor and the descriptions of individual functions make far too much use of cross referencing, forcing the reader to constantly flip back and forth between sections.

Finally, Cambridge Lisp is not compatible with many other dialects of Lisp. It is quite different from the most popular dialect, Common Lisp, and thus cannot be used with the many Common Lisp-oriented tutorials (see reference 4). It is instead based on Standard Lisp, and is therefore somewhat compatible with Portable Standard Lisp. Although some notes on compatibility are given, transporting large amounts of code from other dialects would be difficult.

### *Summary*

Despite these few faults, I like Cambridge Lisp very much. For experienced users, it is among the finest versions of Lisp available, and patient novices will find their time and effort well rewarded. Cambridge Lisp will certainly help popularize the Amiga as an advanced programming tool as well as increase the diversity of Amiga software by giving serious developers an innovative environment in which to work.■

# References


1. Marti, J.B. et al. *Standard Lisp Report*, SIGPLAN Notices 14, 10 (October 1979), 48–68.

2. Metacomco (1985). *Cambridge Lisp 68000 Manual*, Trenchstar Limited, Bristol, United Kingdom.

3. Steele, Guy L. (1984). *Common Lisp: the Language*, Digital Press, Billerica, Massachusetts.

4. The Utah Symbolic Computation Group (1982). *The Portable Standard Lisp Manual*, Computer Science, University of Utah, Salt Lake City, Utah.

5. Winston, Patrick Henry and Berthold Klaus Paul Horn (1984). *LISP*, Second Edition, Adison Wesley Publishing Company, Reading, Massachusetts.



*Address all author correspondence to Daniel Zigmond, Carnegie-Mellon University, Computer Science Dept., Schenley Park, Pittsburgh, PA 15213.*



*Special thanks go to Dr. Scott Fahlman for his help in the preparation of this article.*