A Browsing Interface for S-expressions

Kei Yuasa
Tokyo Information and Communications Research Laboratory
Matsushita Electric Industrial Co. Ltd.
3-10-1 Higashimita Tama-ku Kawasaki 214 Japan

Abstract

This paper presents a new system for Lisp object inspecting. The author has implemented a system called "List Window" which provides Lisp programmers with a method for inspecting S-expressions on a screen. This system was designed in order to help Lisp programming, especially debugging.

When developing the List Window system, it was necessary to output a big list into a given sized terminal with proper indentations. The system sets a focal point in a list and truncates elements which are far from the focus. Main parts of this paper describe this method of truncating branches of a list.

1 Introduction

Lisp is an "environment providing" language where a programmer develops, executes and debugs programs. He inputs and runs his functions in a "toplevel loop". When an error occurs in execution, he debugs the program in an "error handler". The toplevel loop and the error handler provide the same `read-eval-print` loop. However, the programmer's behavior in the error handler is stereotyped. He usually types similar kinds of function calls repeatedly. List Window has been implemented to solve this problem. It assists object inspection which is performed frequently in the error handler and accelerates program debugging.

The List Window system was first developed on UtiLisp[6] based on a character display. Afterward, it was ported to CLX, a Common Lisp[2, 3] with X-window interface. The new implementation utilizes a mouse and a window as I/O devices. An internal algorithm for printing a list has been improved in the second implementation.

This paper describes the designs of our systems, mainly the second implementation. Section 2 considers problems in typical procedures in debugging and a method of displaying a big list on a terminal. Section 3 describes the interface of List Window. And Section 4 devotes to the implementation of the system.

2 Operations in Lisp

This section analyses the typical behavior of a Lisp programmer during debugging and explains problems of conventional interface for Lisp systems.

A Lisp programmer interacts through Lisp's toplevel loop, a simple `read-eval-print` loop. There, he defines functions, sets up variables and other environments. Definitions written in an external file are loaded by calling a file function, `load`. Afterward he executes one of his defined functions with some arguments.

When a program causes an error, the system enters an error handler which presents another `read-eval-print` loop. This error handler preserves the execution stack and the environments at the time the

error happened. The programmer looks for the cause of the error using this handler.

2.1 Object Inspection during Debugging

What a programmer does in the error handler is just inspecting objects. In making and executing the programs, he changes the status of Lisp objects in the heap area. Getting an error means that the status of the object differs from what was expected. The purpose of debugging is to find the cause of the trouble which may be a mistake in the function definition or in another environment. Anyway, a clue to the error rests in the stack or in the heap area.

All non-garbage objects are connected by many kinds of pointers so that they are traceable from the execution stack or from interned symbols. "Referring functions" are used to trace these pointers. These functions are `symbol-value`, `symbol-function`, `symbol-property` (for symbols), `aref` (for arrays), `car`, `cdr` etc. (for lists).

As the object inspection consists of series of invocations of pointer tracing, the referring functions are called repeatedly in the error handler. Besides, arguments to the functions are usually the same. Or a result is used as the next function's argument. As a result, the programmer suffers from typing similar function calls repeatedly.

2.2 Inspection of Lists

Printing a big list is another problem in Lisp interactions. Some lists are too big to output all the elements on a screen. When a screen runs out of lines, parts of a list are scrolled out. Besides, a Lisp programmer sometimes wants a list to be pretty printed so that he is able to grasp its structure. However, pretty printing requires much more lines since it inserts a lot of newlines and indentations. If he wants elements which have been scrolled out, he should invoke some referring functions to extract a smaller sublist.

Most conventional Lisp systems have abbreviative printing. When these systems print out a big list, some elements of it are abbreviated. A list can be "big" in two dimensions: "depth" which is the number of CAR pointers and "length" which is the number of CDR pointers. Global variables `*print-level*` and `*print-length*` control the abbreviation of elements for printing. When the depth of an element exceeds `*print-level*`, the printer abbreviates this node and replaces it with some simple symbol.

Though a user can freely set values to these two variables, it is not practical to change these dynamically during object inspection. Besides, even if a list is truncated properly to be printed on a single screen, some list referring functions are needed to get the abbreviated elements.

The problem of the abbreviative printing is that the depth and the length of a node is always calculated from the top of the list. Therefore, elements on deep or distant leaves are always invisible. To get them, a programmer has to call a series of functions to access sublists. Another problem of inspecting a list in the procedure is that there is no way of going back to previous lists. As CAR and CDR pointers are one directed, a user cannot get the root of a list from its leaves. If he wants to go back to intermediate cells, he has to extract it from the top of the list.

2.3 Requirements

Our new system has been developed to solve the problems in object inspection which are described in the above subsections. In order to offer a good environment for a Lisp programmer, the following parts are required:

- Referring functions are called in a simple way
- A list should be pretty printed on a single screen
- Abbreviation of list elements should be determined from a user's point of view

3 Interface of List Window

A new interface, List Window, has been developed as a solution to the problems of object references and list printing mentioned in the last section. It allows a programmer to inspect all the elements in one list on a screen.

List Window is invoked by a new function `list-window`. A text window is opened and an inspected list is printed on it (Figure 1). The list is pretty printed with indentations and some elements are abbreviated when this list is too big to be printed within the window boundaries.

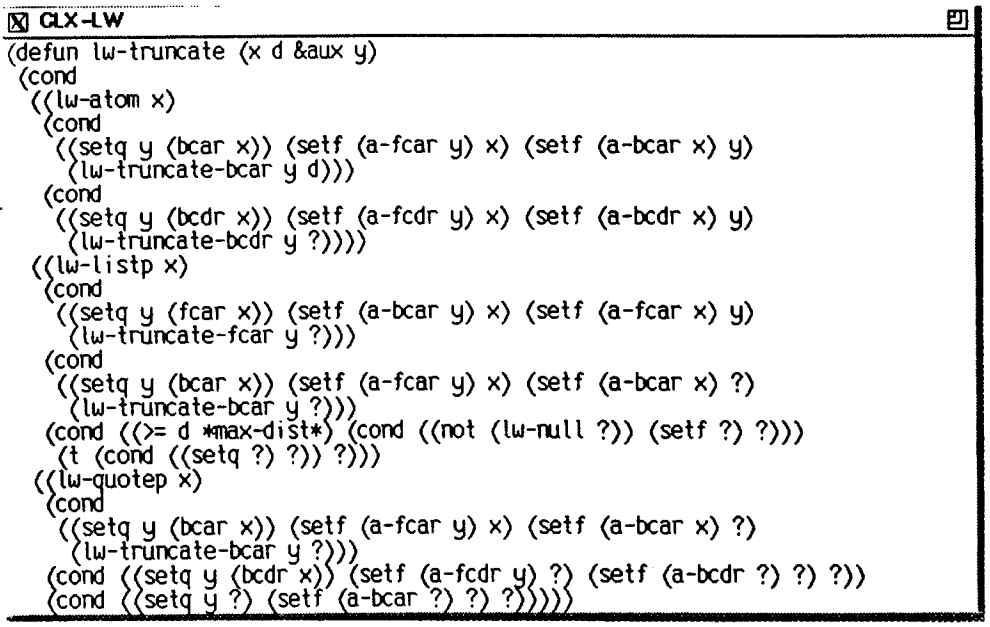


Figure 1: A List Displayed on List Window

3.1 Moving Focus

At the beginning, neighbors of a list's top are displayed and far nodes are invisible. This is a situation where the focus, which is the user's view point, is located on the top. The focus and its neighbors are printed prior to distant elements. Question marks represent ellipsis symbols which are substitutions for truncated distant elements.

A user can move the focus anywhere in the inspected list. The only thing he needs is to click the left mouse button on an element where he wants to move the focus. Distances from the new focus are recalculated. Then, the neighbors of the new focus become visible as shown in Figure 2. In the figure, the focus is on the atom "t" on the 15th line. Repetition of the focus movements enables inspection of all the elements in the list. Note that no explicit invocation of list referring functions are needed to extract invisible elements.

List Window selects elements to display according to the structural distances. This method differs from those of conventional window systems which display parts of text. Lisp Machine Lisp is equipped with a viewing interface in which a list is pretty printed as a flat text and viewed in a window. The window displays part of the text and slides on it to enable inspecting throughout the list. However, a list loses its structure when it is pretty printed as plain text. In the conventional window systems, "distance" means the number of lines on the text not a structural distance. Nodes structurally close to the focus may be distant in the flat text. As a result, a user may not be able to see the actual neighbors of the focus.

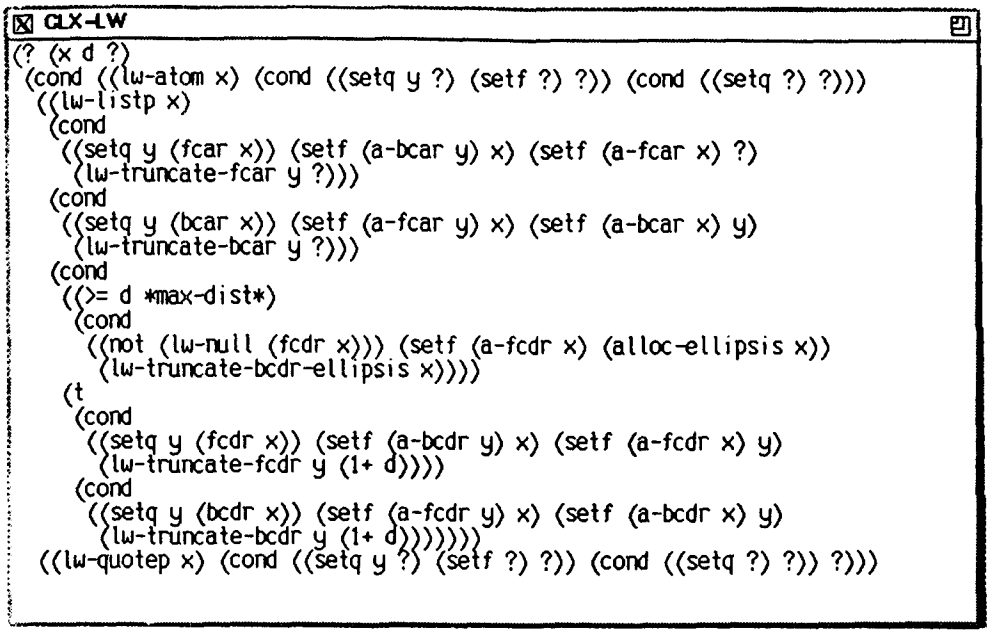


Figure 2: The Same List After Moving Focus

3.2 Pointer from Atomic Objects

List Window provides a method of tracing pointers in atoms. When a programmer wants to inspect the value of a symbol which exists in the inspected list, he simply selects this symbol with the mouse and invokes a "value" command from a mouse menu on this symbol. Then, a new list which is the symbol's value is displayed in the same window. The symbol's definition and property list can be acquired in the same way.

When a new object is displayed in the window, the system must not abandon the old list because the user may trace back to it afterwards. There is no reverse pointer from the value to the variable symbol. The List Window system holds an internal object stack to preserve inspected lists. When a user traces value or function pointers from symbols, they are pushed onto this stack. A "pop" command pops the stack and restores the previous object.

The List Window system enables simple invocations of referring functions in order to provide an inspecting interface. For lists, a new method of displaying helps tree walking in the inspected list. For pointers from atoms, the object stack holds the old objects for later restoration. Operations are all performed by simple mouse operations.

3.3 Operations on a Character Terminal

The above subsections explain operations of List Window which runs on the X-window system. The old implementation of the system runs on a character display and enables interactions using keys. Like in screen editors, the focus movements and the operations on atoms are performed by simple key presses and cursor movements.

4 Implementation

The internal implementation of List Window is discussed in this section. Given a list, the List Window system first constructs an abbreviated list which is suitable for the size of the window. Next, it displays the list on the window and then controls interactions with the user. In order to make the abbreviated

list, the system tries pretty printing ahead of actual printing. The following subsections explain these steps.

4.1 Truncating List Branches

The pretty printer in List Window is called with a list where some of the elements may be abbreviated. It not only prints out the list but also counts the number of lines needed for printing ahead. If this number exceeds the size of the display window, it is necessary to truncate more elements to shrink the list.

Elements distant from the focus are truncated first. The distance between each node and the focus is the number of CDR pointers. CAR pointers are neglected though conventional Lisp printers provide the two variables, `*print-level*` and `*print-length*` for CAR and CDR respectively. The reason is that it is difficult to control the two independent variables automatically. The CDR pointer is used for distance calculation because most lists connect their elements with CDR pointers.

Our system holds a threshold value `*max-dist*` and truncates nodes whose distances are bigger than it. If the list is proved printable within the window, the pretty printer prints it. Otherwise, it decrements this `*max-dist*` and invokes a truncator.

Given a threshold and a focus position, the truncator abbreviates elements far from the focus. As described before, the focus may be located anywhere. When it is on a leaf, the top of the list may be truncated. Four examples of truncating are shown here. In each example, the threshold value is two and the underlined symbol means the focal point.

In (1), the symbol `c` has a distance 2 and `d` has 3. Thus, the list `(d e f)` will be abbreviated. As a result, (1) becomes (2).

(a b c d e f) (1)

(a b c ?) (2)

When a focus is located on a middle element of the list, both ends are abbreviated. In (3), the first two should be truncated. By replacing CAR and CDR of the top cell, we get (4)

(a b c d e f g h i) (3)

(? c d e f g ?) (4)

CAR pointers are never counted as distances. In (5), `f` and `g` have distances 1 and 2 respectively.

(a b c d e (f (g h) i)) (5)

(? c d e (f (g ?) ?)) (6)

When a focus is located on a very deep element, a lot of elements are abbreviated. The list (7) is shrunk to (8).

(a b c (d e f (g h i j k l m) n) o) (7)

(? (? (? h i j k l ?) ?) ?) (8)

Ellipsis symbols in outer parentheses are not rather important when the focus is on the middle symbol `j`. These ellipsis symbols cause lots of newlines and indentations when this list is pretty printed. In these cases, outer ellipsis symbols are abbreviated. After all, (7) becomes (9). Like in (9), if there are series of

left parentheses from the top and the first atom is an ellipsis symbol, outer lists might have abbreviated elements.

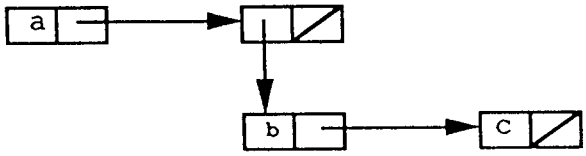
```
(((? h i j k l ?)))                                    (9)
```

4.2 Doubly Linked List

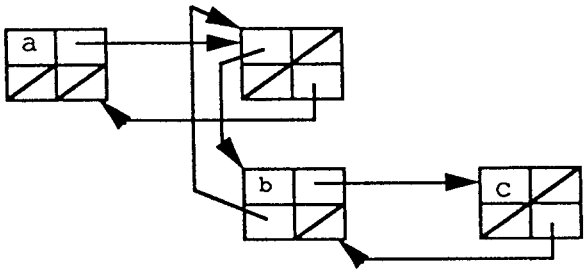
In the first implementation of List Window, the system used two working lists to make an abbreviated tree. One is a distance list and the other is an attribute list, both having the same structure as the original inspected list. The former holds the distances from the focus of each atomic element and the latter holds the number of characters of the corresponding atom which are used in pretty printing.

In this implementation, making the distance list causes a bottle neck. Since a focus is not always located at the top, the system should know the focus location before giving a distance to each node. Thus, searching for the focus needs extra time. This is because a list has only pointers pointing in one direction. As the focus may be located anywhere in a list, it is necessary that we can trace pointers in both directions.

The second implementation of List Window adopts a doubly linked list. Figure 3 shows the structures of this doubly linked list. The original list and its doubly linked form are shown in (i) and (ii) respectively. Each node has four pointers, normal CAR and CDR and additional backward pointers of them. Atomic objects are represented as structures with backward pointers.



(i) original form (a (b c))



(ii) Doubly Linked Form of (i)

Figure 3: Doubly Linked List

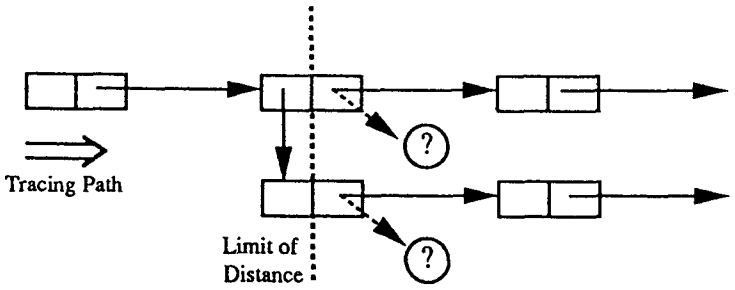
The doubly linked structure helps truncating of branches. Figure 4 shows the basic three methods of abbreviations using doubly linked lists. In the figures, dotted lines are new pointers after truncating some pointers. The system traces CAR and CDR pointers back and forth from the focus counting distances from it. When a CDR pointer is traced, the distance is incremented. And if it exceeds the limit, the node should be truncated. Figure 4-(i) is the simplest form of truncating forward pointers, which is the same case as (1) in subsection 4.1. The truncator traces pointers from left to right. If the distance of a traced node reaches the threshold, its CDR is substituted for an ellipsis.

Figure 4-(ii) explains truncating of backward pointers, exemplifying expression (3) in 4.1. If the distance of a traced node reaches the threshold, the top of the list is searched first. Then the CAR of the

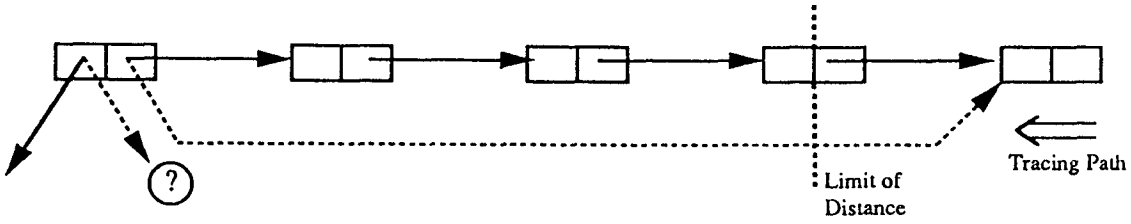
top is substituted into an ellipsis and the CDR skips the abbreviated elements.

Figure 4-(iii) explains the case (7) in 4.1. If the distance of a traced node reaches the threshold, CAR's and CDR's of parent tops are replaced as shown.

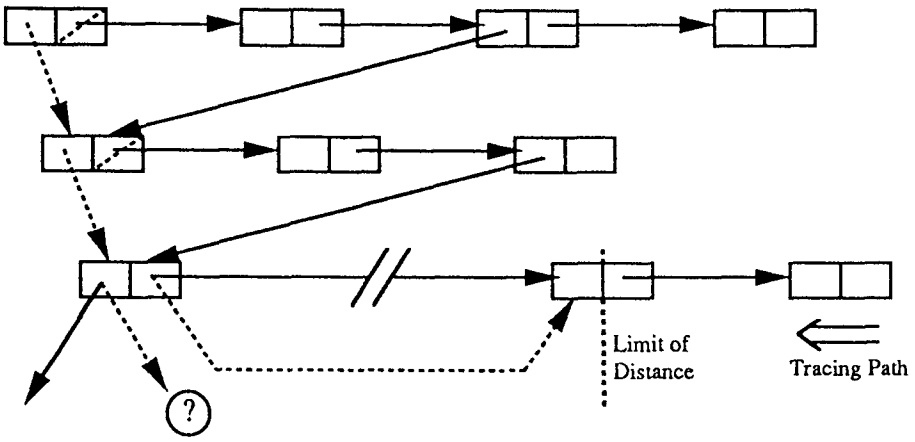
Each cell of a doubly linked list is represented as an array in the system. A cell for a cons cell holds backward CAR and CDR pointers as well as forward pointers. Besides, it has copies of these four pointers for pointer linking in an abbreviated list. This is because the original form of the list should not be lost when the pointers are rewritten when truncated. After all, the array is at least four times as large as an original cons cell node. For an atomic object, an array needs extra slots for backward pointers, attributes of the atom and so on.



(i) Forward Pointer



(ii) Backward Pointer



(iii) Elliminate Outer Ellipsis

Figure 4: Truncating List Nodes

Though the doubly linked list needs extra storage, it is not necessary to consume new cells when an abbreviated list is created. When a focus is moved to other location and an abbreviated list is recalculated, the system never generates garbage. Another advantage of the doubly linked list is that

scanning of pointers in both direction is possible. The time spent for tracing the list is proportional to the size of the list regardless of the location of the focus.

4.3 Output into Window

After making an abbreviated tree, List Window displays it in a window. The system has an internal vector called "screen attribute". Each element of this vector is associated with a character location of the window. When List Window prints out a character, it reserves information about each node in the vector. It contains the pointer to a corresponding node in the doubly linked list and the type of the node. This information is used when a user interacts with List Window. When the focus is moved, the new location of the focus is picked up from the vector using the coordinates of the mouse cursor. When operations are performed on atoms, its operand is picked up from the mouse cursor location.

5 Evaluation

This section evaluates our new system. It is difficult to compare performances of user interface systems. In this section, we consider the cost of operations needed for extracting some element in an inspected list.

The list in Figure 1 is a definition of `lw-truncate`, one of the functions of List Window. The list includes 284 cons cells and 178 non-null atoms. It takes 45 lines to pretty print this list on an 80-column terminal. The atom "d" in the 21st line in Figure 2 is the most distant element from the top of the list. There are 19 CDR pointers between them and the atom appears in the 31st line when pretty printed.

When this list is inspected with List Window using 80 columns and 24 rows, the element "d" is invisible as in figure 1. However, only one mouse click is needed to let it be displayed on the screen(figure 2). Once the element is displayed, another mouse click moves the focus on it. After all, it is possible to set the focus to this element by clicking the mouse button twice.

In a usual toplevel loop, the operation is not so simple. In order to extract the element, it is necessary to trace 19 CDR pointers and 7 CAR pointers. And if the user traces a wrong pointer, he is not able to retrieve because the Lisp pointers are one-directional.

In List Window, the focus and its neighbors are visible so that the user recognizes the list structure easily. For example, in figure 1, it is obvious that the first cond in the second line has three conditional branches: `(lw-atom x)`, `(lw-listp x)`, `(lw-quotep x)`. When the list is viewed in a conventional window system with a scroll bar, these predicates do not appear simultaneously in a 24 row terminal.

6 Summary

The system List Window has been developed in order to ease procedures in Lisp debugging. The purpose was to provide users with an interface for browsing Lisp objects.

First, List Window is introduced as a new method of displaying a list on a limited region of a screen. All elements are traceable by simple movements of the focus. Second, the system enables tracing pointers from atomic objects by controlling an object stack. Thus, the user can reach any objects connected from the first object. Operations are either mouse actions or key presses. Explicit invocations of referring functions are no longer necessary.

7 Acknowledgement

The studies of List Window were pursued at the University of Tokyo as a part of the doctoral research of the author. The author thanks Professor Eiiti Wada for his supervision and member of his laboratory for their helpful advice.

Discussions with Dr. Atsushi Maeda of Keio University contributed to improve the algorithm of truncating list elements in the second implementation of the system. Thanks also go to Dr. Katsura Kawakami and Mr. George Michelitsch of Matsushita Electric Industry for their advices on writing this paper.

References

[1] George W. Furnas. *Generalized Fisheye Views*, Human Factors in Computing Systems, CHI '89 Conference Proceedings, ACM April 1986

[2] Guy Steele. *Common Lisp the Language* Digital Press, 1984.

[3] Symbolics. *Program Development Utilities*, in Symbolics Manuals, Symbolics, 1986.

[4] Kei Yuasa. *Process Monitoring Interface for Multiprocess Lisp*, Doctoral Thesis, Faculty of Engineering, University of Tokyo, December 1988

[5] Kei Yuasa. *Character Terminal Interface for S-expression* Reprints of WGSYM, 90-SYM-54, IPS Japan, Mar. 1990

[6] Wada Laboratory. *UtiLisp Manual Revision 2.0*, Department of Mathematical Engineering, University of Tokyo, Jan. 1988