MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 611 October 1981

GPRINT

A LISP Pretty Printer Providing Extensive User Format-Control Mechanisms

by

Richard C. Waters

ABSTRACT

A pretty printer is presented which makes it easy for a user to control the format of the output produced. The printer can be used as a general mechanism for printing data structures as well as programs. It is divided into two parts: a set of formatting functions, and an output routine. Each formatting function creates a sequence of directions which specify how an object is to be formatted if it can fit on one line and how it is to be formatted if it must be broken up across multiple lines. Based on the line length available, the output routine decides what structures have to be broken up across multiple lines and produces the actual output following the directions created by the formatting functions. The directions passed from the formatting functions to the output routine form a well defined interface: a language for specifying formatting options.

Three levels of user format-control are provided. A simple template mechanism makes it easy for a user to control certain aspects of the format produced. A user can exercise much more complete control over how a particular type of object is formatted by writing a special formatting function for it. He can make global changes in format by modifying the formatting process as a whole.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-80-C-0505.

The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, or the United States Government.

Introduction

This paper shows how a pretty printer can be structured to provide mechanisms which allow (but do not require) the user to exercise extensive control over the format of the output produced. There are three basic reasons why user format-control mechanisms are useful: They make it possible for the user to adapt the printer so that it can deal aesthetically with new structures created by the user which were not foreseen by the implementors of the printer. They make it possible for the user to modify the way in which standard structures are formatted so that he can get the kind of output he really wants to see. They can be used to specify how the standard structures should be formatted. This leads to greater internal clarity in the pretty printer itself by separating this formatting information from the rest of the algorithm.

Most pretty printers do not give the user any significant control over the format produced [1, 2, 4-9]. This lack has been ameliorated by the fact that most of these printers are not intended to be able to deal with novel structures; they are used solely as system facilities for printing program text written in a language which does not allow the programmer to create new programming forms.

A pretty printer can be used much more advantageously if it is set up so that it can be used to print data structures as well as programs and so that it can be called by user programs. If used this way the issue of dealing with novel structures cannot be ignored, and user format-control mechanisms become essential.

A pretty printer can be viewed as a general mechanism for formatting and then printing hierarchical structures of variable shape and size. Program parse trees are a prototypical example of such a structure. However, they are by no means the only example. A pretty printer can be extended to deal with arbitrary data structures by adding default formats for each basic data structure (e.g. for records, arrays, etc.). However, due to the idiosyncratic nature of user defined data structures in general, user format-control is needed in order for the output produced to be aesthetic.

Every language has facilities for specifying how output is to be formatted on the page. In general, these facilities are oriented toward printing data structures whose shape is known in advance, on a page whose width is known in advance. If either of these have to be parameterized, then the programmer has to write special code to deal with it. Pretty printers are specifically designed to deal with these two kinds of variability. User format-control mechanisms make it possible for the user to specify his aesthetic criteria in a way which is independent of the exact shape of the data, and the size of the page.

The sections below describe how a particular pretty printer (GPRINT) provides for user format-control and discusses some of the general issues involved. GPRINT was originally implemented in 1975 as an attempt to improve on an earlier pretty printer implemented by Goldstein [3]. Goldstein's pretty printer is one of the few pretty printers which provides significant user format control mechanisms. However, these mechanisms are at the same time too complex for easy use in simple situations and too weak in situations where the user needs extensive control. GPRINT has been rewritten four times (most recently in 1981) in a continuing attempt to create a user controllable pretty printer with very good human engineering.

This paper is not a complete description of GPRINT. In the interest of brevity and clarity, the discussion is simplified wherever possible by omitting details and capabilities of GPRINT which do not directly relate to user control of formatting. An appendix summarizes the full facilities provided by GPRINT.

GPRINT is written in Lisp, and was developed in the context of the Lisp programming environment. In this paper, the Lisp language is used to display parts of the algorithm and Lisp data is used in examples of how objects are printed. This is done because Lisp has several features which make the implementation and explication of a pretty printer particularly easy. However, the ideas embodied in GPRINT are not limited to the Lisp domain. In particular, these ideas grow principally out of the requirements for a highly interactive *programming environment*, rather than out of the Lisp *language*.

The Basic Algorithm

The central feature of the algorithm used by GPRINT is that the pretty printing process is divided into two parts: a set of formatting functions and an output routine. The formatting functions specify how each part of the input is to be printed if it will fit on one line, and how it should be printed if it must be broken up across multiple lines. This information is passed to the output routine as a sequence of entries in a queue. The output routine operates as a coroutine processing the queue entries as they are created. It decides how to fit things into the actual space available and then prints them. An important feature of the algorithm used by the output routine is that it is linear in the number of entries put in the queue. The key to this is that it uses a strictly limited form of look ahead so that the length of the queue is bounded by the length of the line available for printing and is independent of the actual number of entries created.

This basic algorithm has been independently developed by several people [4, 7] in addition to the author. In [7], Oppen gives a lucid description of the algorithm, and how the output routine operates. The only difference between his output routine and GPRINT's output routine is that GPRINT's queue entries are more general. This paper focuses on the unique aspect of GPRINT: the way the formatting process allows for user format-control.

The following simplified example illustrates how formatting information is encoded in queue entries. In order to format the list (ITEM1 (ITEM2 ITEM3 ITEM4) ITEM5) the following queue entries could be used.

```
{, '(ITEM1 ', {, '(ITEM2 ', 'ITEM3 ', N, 'ITEM4)', }, ' ', N, 'ITEM5)', }
```

There are three basic kinds of queue entries:

- { ... } - These two entries mark the beginning and end of a group of queue entries which are to be treated as a single substructure.
- '...' - The indicated string is to be printed in the output.
- N - A conditional line break. A line break is to be inserted in the output if the structure immediately containing this entry cannot be printed on a single line. When this is done, indentation is inserted so that the next item to be printed will line up under the second item in the substructure.

The queue entries in the example above specify that the list should be printed in one of three ways depending on how much horizontal space is available. If the available space is at least 33 characters than the entire list will be printed on a single line as follows:

```
(ITEM1 (ITEM2 ITEM3 ITEM4) ITEM5)
```

If the width is less than 33, then the top level structure must be broken up across lines. The last N queue entry indicates that in this case a line break should be inserted before ITEM5. As long as the width is at least 27 then the sublist does not have to be broken up and the output is as follows:

```
(ITEM1 (ITEM2 ITEM3 ITEM4)
       ITEM5)
```

If the width is less than 27, then the sublist must also be broken up across lines. The first N queue entry indicates that in this case a line break should be inserted before ITEM4 yielding the following:

```
(ITEM1 (ITEM2 ITEM3
              ITEM4)
       ITEM5)
```

If the available width is less than 21, then there is no way to print out the structure which is consistent with the queue entries. This is an example of the *finite line length problem.* Pretty printers in general suffer from

this problem and there is no simple solution to it. However, the problem is usually not severe as long as the line length available is several times larger than the largest indivisible item which must be printed on a single line. GPRINT has a number of built in features which try to ameliorate the problem by keeping the indentation small in order to maximize the line length available.

The Dispatching Function

The structure of the set of formatting functions is based on the idea that any object to be printed by GPRINT can be viewed as a directed graph where each leaf node is a primitive data object (such as a number or a symbol) and each non-terminal node is a composite data structure (such as a record structure or a non-terminal node in a program parse tree). The formatting functions are organized around a central dispatching function (GDISPATCH). At each node, GDISPATCH selects a particular formatting function based on various features of the node (such as its data type). The selected formatting function creates queue entries which specify what to print corresponding to the node and how it should be formatted. It calls the dispatching function recursively in order to format the composite components of the node.

The discussion in this paper focuses on the familiar use of a pretty printer to format programs and therefore on the Lisp data type list. This is a convenient microcosm in which the user format-control mechanisms can be discussed. Formatting programs is particularly straightforward in the Lisp domain. Due to the fact that program lists are data objects like any other, it is not necessary to implement either a program text parser or a special data structure representing program parse trees as part of the pretty printer.

Consider the following simplified version of GDISPATCH. It assumes that the item to be formatted must be either an atom or a list. GDISPATCH first tests the data type of the item. If it is an atom then it is entered into the queue as an item to be printed out. If it is a list then GDISPATCH looks at the CAR of the list in order to pick a specific formatting function. A formatting function for a particular type of list can be specified by defining it as the :GFORMAT property of the CAR of the list. If there is no such property then GDISPATCH uses either a default format for function applications or for data structures. Unfortunately, in a Lisp system there is no completely reliable way to distinguish the representation of a program from other kinds of data. As a heuristic, GDISPATCH looks to see whether the CAR of the list is the name of a function. (The formatting functions referred to here are discussed below.)

```
(defun Gdispatch (item)
  (funcall (cond ((atom item) ':Gatom-format)
                 ((not (symbolp (car item))) ':Gdata-format)
                 ((get (car item) ':Gformat))
                 ((fboundp (car item)) ':Gfn-format)
                 (T ':Gdata-format))
           item))
```

The actual version of GDISPATCH used by GPRINT dispatches on many other data types as well as lists (for example, arrays, MacLisp hunks, and Lisp Machine named structures). Standard formatting functions are provided for each of these data types (see the appendix).

It is important to note that the dispatching function depends on the availability of run time type information. In Lisp, this information is readily available. In many other languages, most of the type information is used only by the compiler and is not available at run time. In order for a pretty printer to be usable on the data in user programs, the printer would have to have access to type information tables created by the compiler, just as a dynamic debugger has to have access to the compiler's symbol table in order to use the programmer's variable names.

A Language For Specifying Formatting Options

In the GPRINT system, queue entries are created through the use of formatting templates. A template for a list serves two purposes. It matches against the list selecting out the items in the list to be printed. It also specifies auxiliary queue entries such as those specifying substructures in the queue and conditional line breaks.

The clarity of the formatting process depends on the perspicuity of the language used to build templates. This language is a concise language for specifying formatting options. Each template is a string built up out of the following formatting codes:

P - Print the corresponding part of the object in the output.

* - Call the dispatching function to determine how to format this part of the object.

_*n* - Print *n* (default 1) blanks in the output.

'...' - Print the indicated literal in the output.

A - Always put a line break here.

N - (*normal* line break) - Put a line break here if the structure immediately containing this code cannot be printed on one line.

(*n*...) - This delineates a unit of substructure, both in the object being formatted and in the queue entries which specify how to print the object. The part of the object being formatted which corresponds to this part of the template must be a list. It is decomposed into its elements. The template between the parentheses specifies how these are to be formatted. The queue entries for the individual elements (along with entries specifying that parentheses should be printed before and after the list) are grouped together into a single structure in the queue. This structure is treated as a unit when decisions about where to insert line breaks are made.

The number following the open parenthesis specifies how much the indentation should be increased while printing items inside the list when they will not fit on a single line. The indentation increment can be omitted in which case it will default to the sum of the lengths of: the open parenthesis, the first item in the list and any blank space after it.

<...> - This is used inside of (...) to specify a template for a list of unknown length. The part of the template between the brackets is taken as repeating indefinitely, creating a subpattern of infinite length. For example, "(<P_>)" is the same as "(P_P_P_P_P_P_ ...)".

It is important to note that when a subtemplate inside parentheses is used to format a list, the part of the subtemplate extending beyond the P, *, or (...) corresponding to the last element of the list is ignored. For example, if the template "(<P_>)" is used to format the list (X Y) then a blank will not be printed between the Y and the close parenthesis of the list.

Consider the list (ITEM1 (ITEM2 ITEM3 ITEM4) ITEM5), the template "(P_(P_P_NP)_NP)" could be used in order to create queue entries like those given in the example above. Note the way the template matches against the list as a whole, and the way the subtemplate matches against the second element in the list.

Defining a Template Corresponding to a Type of List

The macro (DEFGF *atom template*) can be used to specify a template which will be used to format lists which begin with the indicated atom. For example, lists beginning with SETQ are typically formatted so that each successive variable value pair appears on a separate line. This can be specified by using the A format code in a template as shown.

```
          (SETQ X 1
                Y 2)

(defGF setq "(P_<*_*A>)")
```

Similarly, a COND expression is formatted with each clause on a separate line. In addition, the individual expressions in each clause are lined up one under the other if they will not fit on a single line. This is specified by using a subtemplate with an explicit indentation increment of 1.

```
          (COND ((MINUSP X)
                 (- X))
                (T X))

(defGF cond "(P_<(1<*_N>)A>)")
```

The template for LET specifies that the bound variable initial value pairs are printed one to a line, and that the statements inside the LET are printed one to a line if the LET cannot be printed on one line. The indentation is specified to be only 2 in order to keep the total indentation small.

```
          (LET ((X 1)
                (Y 2))
            (CONS X Y))

(defGF let "(2P_(1<*_A>)<_N*>)")
```

Whenever the format for a particular type of list can be specified by using a single template, then the macro DEFGF can be used. In more complex situations the user can define a complete formatting function.

Formatting Functions

As discussed above, a formatting function is a function which, when called by GDISPATCH, will create a sequence of queue entries describing how to format an object. The formatting function for lists which begin with a particular atom as their CAR can be specified by making the function be the :GFORMAT property of the atom. The formatting function itself can contain arbitrary computation. In order to actually create queue entries, the formatting function calls the macro (GF *template &rest args*). GF matches the template against zero or more arguments and produces a series of queue entries.

As an example of a simple formatting function, consider :GFN-FORMAT, the default formatting function used by GDISPATCH for formatting function applications. It specifies that if the application cannot be printed on a single line, then the arguments should be printed one to a line.

```
          (LIST X
                Y
                Z)

(defun :Gfn-format (item) (GF "(P_<*_N>)" item))
```

Each call on the macro DEFGF expands into the definition of a simple stereotyped formatting function as illustrated below.

(defGF *atom template*) <==> (defun (*atom* :Gformat) (item) (GF *template* item))

An important thing to keep in mind about formatting functions in general, and about GF in particular is that they do not print anything. Rather, they merely create queue entries. In order to print something you call the function GPRINT. It calls GDISPATCH which calls formatting functions which create queue entries which are interpreted by the output routine in order to determine what to print. It is the output routine which actually does the printing. When you define a formatting function, you are not defining a function which will print something, but rather defining a set of directions to be followed when GPRINT prints it.

Blocking and Tabbing

In order to save space, long lists of data are often formatted in *block* form where as many items as possible are put on each line. The language which is used to create formatting templates has two format codes which are useful for specifying this kind of format.

B - (*block* line break) - A line break is inserted here if and only if the structure immediately following this code will not fit on the end of the current line.

T*n* - Blank space is inserted until the position relative to the current indentation is congruent to zero modulo the argument *n*. The argument can be omitted in which case a default tab size will be calculated based on the lengths of the other items in the structure containing this code.

As shown below, the function :GDATA-FORMAT can be defined using the B format code so that block form will be used as the default format when printing data.

```
(ORANGE PEAR (RED APPLE) GRAPEFRUIT
 (HAWAIIAN PINEAPPLE) BANANA
 CANTALOUPE POMEGRANATE TANGERINE)

(defun :Gdata-format (item) (GF "(1<B*_>)" item))
```

There is a problem with printing lists of data in block format. If the elements of a list are themselves lists with a depth of greater than one, then the output is not very aesthetic because it is not easy to identify the elements of the top level list. For example, consider the following list:

```
((ORANGE (SELL 3)) (PEAR (BUY 10)) ((RED APPLE) (BUY 5))
 (GRAPEFRUIT (BUY 10)) ((HAWAIIAN PINEAPPLE) (SELL 8))
 (BANANA (SELL 5)) (CANTALOUPE (BUY 4)))
```

The actual definition of :GDATA-FORMAT uses the T format code so that lists of data are printed out in a tabular form. This makes the output much easier to read without taking up very much more space.

```
((ORANGE (SELL 3))              (PEAR (BUY 10))
 ((RED APPLE) (BUY 5))          (GRAPEFRUIT (BUY 10))
 ((HAWAIIAN PINEAPPLE) (SELL 8))
 (BANANA (SELL 5))              (CANTALOUPE (BUY 4)))

(defun :Gdata-format (item) (GF "(1<TB*_>)" item))
```

The template specifies that the printer should select a default tab size. Unfortunately, due to the fact that the printer uses only limited look ahead, the tab size must usually be chosen before all of the elements in the list have been entered in the queue. As a result, it is not guaranteed to be large enough. In this example, the fourth element in the list was not completely entered in the queue at the time when it was determined that the list had to be put on more than one line. As a result, only the first three elements were used to determine the tab size which turned out to be too small to accommodate the fifth element.

Nonstandard Substructures and Delimiters

A basic limitation of the format codes introduced so far is that the (...) format code conflates three ideas: decomposing the input, defining a structure in the output, and printing parentheses as delimiters in the output. Format codes are provided which separate these ideas:

[...] - This is used to decompose a list argument to GF into its components. For example, (GF "[P_[P_P]]" '(1 (2 3))) is the same as (GF "P_P_P" 1 2 3). This format code has meaning only to the macro GF and does not create any queue entries by itself. In conjunction with <...>, [...] can be used to decompose a list of arbitrary length.

{*n*...} - This delimits a substructure in the output. The number specifies the indentation increment to use if the structure will not fit on one line. If it is omitted, a default value is computed as the sum of the length of any literal open delimiter, the length of the first item in the substructure, and the length of any space after this item.

(*n*...) - This can now be defined as an abbreviation for {*n*'('[...]')'}.

As a simple example of a format requiring this separation of ideas, consider the function QUOTE. A list which begins with the atom QUOTE is not printed with parentheses around it. Rather, the argument to QUOTE is printed out following a "'". The formatting function for QUOTE specifies this format by setting up a substructure (so that the output will be treated as a unit), printing a literal "'", and then calling GDISPATCH (via the * format code) to format the argument to QUOTE. (Note that inside of a literal in a template, "''" stands for "'".)

```
        (GPRINT '(QUOTE (A (QUOTE B))))
prints: '(A 'B)

(defun (quote :Gformat) (item) (GF "{''''*}" (cadr item)))
```

Consider the formatting function below which implements a format for a user defined data structure. The user data structure is implemented as an ordinary list. Instances of the structure can be recognized because they begin with the atom NAMED-FORM. The second element in the data structure specifies a *name*. This name is either a single symbol, or a list of a symbol and a suffix. The final element of the structure is an arbitrary Lisp expression.

The formatting function sets up a substructure with an indentation of 2 but without any delimiters around it. It then formats the *name* part of the structure either as a single symbol, or as compound combination of the symbol and the suffix. Finally, it specifies that a "-" should be printed and calls GDISPATCH (via the * format code) to format the last element in the structure.

```
        (GPRINT '(NAMED-FORM EXPRESSION (+ X 2)))
prints: EXPRESSION - (+ X 2)

        (GPRINT '(NAMED-FORM (CALLING-FORM 3) (CONS X (+ X 2))))
prints: CALLING-FORM-3 -
          (CONS X (+ X 2))

(defun (named-form :Gformat) (item)
  (GF "{2")
  (cond ((atom (cadr item)) (GF "P" (cadr item)))
        (T (GF "[P'-'P]" (cadr item))))
  (GF "_'-'_N*}" (caddr item)))
```

Note that this formatting function is not a single call on GF but rather looks at part of the structure being formatted in order to decide how to format it. Also note that the queue entries indicating the beginning and end of the substructure of queue entries being created are specified in separate calls on GF. This is a common

occurrence and is in marked contrast to [...] (and therefore (...)) which must be properly nested in a single call on GF.

Computed Indentations and Spacing

The following format code makes it possible to specify spacing and indentation as the result of a computation rather than just as a literal number.

- This can be used in place of an argument to _, T, (...), or {...}. It specifies that the value is to be taken from the next input to GF. For example, (GF "P_#P" 'X 6 'Y) specifies that 6 spaces should be printed out between the X and the Y.

The use of this format code is illustrated by the following formatting function for LET. This formatting function extends the one given above so that it tests the current indentation level and prints out the LET at a reduced indentation if the indentation is already greater than half of the total line length. The function GESTIMATE-INDENT looks at the queue and determines what indentation will be used when printing out the LET. Note that this must be computed from the queue because there may be many entries in the queue which have not yet been printed.

If the indentation is not greater than one third of the line length then the LET is simply formatted as in the example format definition above. If the indentation is greater than one third of the line length, then the formatting function sets up a substructure with a negative indentation increment which reduces the total indentation to zero and forces a line break. It then prints out a comment line which indicates that left shifting is occurring, and uses a "|" to show the indentation which otherwise would have been used. On the next line, the format spaces over five spaces and prints the LET itself. Finally, it prints another comment line and terminates the substructure.

```
(DEFUN ROOTS (A B C)  ;computes the real roots of AXX + BX + C
  (COND ((NOT (ZEROP A))
         (LET ((DISCRIMINANT (-$ (*$ B B) (*$ 4.0 A C))))
           (COND ((PLUSP DISCRIMINANT)
;----------        |
       (LET ((TERM1 (-$ B))
             (TERM2 (SQRT DISCRIMINANT))
             (TERM3 (*$ 2.0 A)))
         (LIST (//$ (+$ TERM1 TERM2) TERM3)
               (//$ (-$ TERM1 TERM2) TERM3)))
;----------        |
                  ))))))
```

```
(defun (let :Gformat) (item)
  (let ((ind (Gestimate-indent)))
    (cond ((< ind (// Glinelen 3)) (GF "(2P_(1<*_A>)<_N*>)" item))
          (T (GF "A{#A';----------'_#'|'" (- ind) (- ind 11.))
             (GF "A'     '(2P_(1<*_A>)<_N*>)" item)
             (GF "A';----------'_#'|'}A" (- ind 11.))))))
```

This formatting function for LET is an attempt to deal with the finite line length problem. It uses nonstandard formatting in order to significantly reduce the indentation when it begins to get too large. This indentation shift reduces readability. However, this is ameliorated by the fact that an entire logical unit is being left shifted, not some arbitrary part of the program. GPRINT uses similar formatting functions for each of the program constructs which correspond to internal code blocks (such as PROG, DO, and LAMBDA).

Modifying the Dispatching Function

Writing formatting functions makes it possible for the user to implement a wide variety of formats, however, they are restricted in several ways. Most important, formatting functions are essentially local. A user can implement global formatting decisions by modifying the standard dispatch function and formatting functions. This section gives an example of this by showing how GDISPATCH can be modified in order to implement an abbreviation mechanism based on the depth of a structure.

The global variable PRINLEVEL controls the abbreviation. If a list is at a depth equal to PRINLEVEL in the object being printed, then "#" is printed in place of it as shown below:

```
PRINLEVEL > 4      (Z (A (B (C (D E)))) F G)
PRINLEVEL = 3      (Z (A (B #)) F G)
```

In order to implement the abbreviation, a global variable LEVEL is used to keep track of the nesting level of each substructure. LEVEL is initialized to 0. GDISPATCH is modified so that LEVEL is incremented each time the function recurses into a sublist. In addition it is changed so that it prints a "#" whenever LEVEL reaches PRINLEVEL.

```
(SETQ LEVEL 0)

(defun Gdispatch (item)
  (LET ((LEVEL (+ LEVEL 1)))
    (funcall (cond ((atom item) ':Gatom-format)
                   ((> LEVEL PRINLEVEL) ':GLEVEL-ABBREV-FORMAT)
                   ((not (symbolp (car item))) ':Gdata-format)
                   ((get (car item) ':Gformat))
                   ((fboundp (car item)) ':Gfn-format)
                   (T ':Gdata-format))
             item)))

(DEFUN :GLEVEL-ABBREV-FORMAT (ITEM)
  (GF "'#'"))
```

Potentially, any pretty printer can be modified by reimplementing parts of it. The point here is that GPRINT is specifically designed so that this can be done easily. Both the formatting functions and GDISPATCH are designed so that they are individually simple and so that their interaction is clearly defined. As a result, it is easy to modify one or more of them. The really complex parts of the printing algorithm are relegated to the output routine which is not intended to be modified.

Conclusion

GPRINT includes a large number of standard formats and features (such as the ones used as examples above). As a result, a user does not have to write any of his own formats in order to get reasonable output in ordinary situations. However, no amount of anticipation can ever satisfy every user. This is particularly true when a pretty printer is being used in an interactive programming environment to print data as well as programs, and when it is called by user programs as well as by the system itself.

The principal goal of the design of GPRINT has been to produce a system with good human engineering which gives the user powerful facilities for controlling the format of output and which at the same time makes the specification of simple formats simple. Two key ideas comprise GPRINT's approach to this problem: the basic algorithm chosen, and the existence of multiple levels at which a user can specify formatting information.

At the same time, the key features of the algorithm underlie the basic simplicity of GPRINT's approach and fundamentally limit its scope. The division of the algorithm into two pieces communicating through a queue

makes it possible to separate the simple parts of the algorithm from the complex ones. The decision to use a linear time algorithm in the output routine makes it possible for GPRINT to run with acceptable speed. However, it fundamentally limits the kind of formatting decisions which can be made by the output routine. In particular, when making its decisions, it can only look ahead a very limited distance. An example of this was discussed in the section on block form output.

In line with the limited abilities of the output routine the queue entries are designed so that they encode only two formatting options for a given structure: how to print it on one line, and how to print it on multiple lines. This design is an important basis for the understandability of the printer because it presents the user with a simple model of how formatting decisions are made. However, one could easily imagine wanting to be able to specify more complex formatting information. For example, one might specify two different multi-line formats: one to use if there was a lot of room available and the other to use when there is only a little space.

The printer provides three basic levels at which a user can specify formatting information. He can use a simple template. This makes it very easy to describe certain aspects of how a structure is to be formatted. He can write a formatting function. This allows him to exercise much more control over the format to be used, at the cost of greater complexity. Finally, he can modify the dispatching function and the standard formatting functions in order to make global changes in the format produced.

The use of multiple levels of interaction is a generally useful technique for increasing the understandability and availability of a system to a wide range of users. It makes it possible for users who have simple needs to satisfy them without having to learn very much about the system. Users who take the time to learn more can then do more.

It is important to note that, though the discussion above was cast in the domain of the Lisp language, the ideas are substantially programming language independent. It should be possible to use these ideas to construct a flexible pretty printer allowing significant user format-control in any programming language environment.

References

[1] Conrow, K., and Smith, R.G., "NEATER2: A PL/I Source Program Reformatter", CACM V13 #11, November 1970, 669-675.

[2] Donzeau-Gouge, V. et al, "A Structure-Oriented Program Editor; A First Step Towards Computer Assisted Programming", Proc. Inter. Computing Symp., Antibes, 1975.

[3] Goldstein, I., "Pretty Printing, Converting List to Linear Structure", MIT/AI/MEMO-279, February 1973.

[4] Hearn, A.C. and Norman, A.C., "A One-Pass Pretty Printer", Report UUCS-79-112, Univ. of Utah, Salt Lake City, Utah, 1979.

[5] Heuras, J., and Ledgard, H., "An Automatic Formatting Program for Pascal", SIGPLAN Notices V12 #7, July 1977, 82-84.

[6] McKeeman, W. "Algorithm 268, Algol-60 Reference Language Editor [R2]", CACM V8 #11, November 1965, 667-669.

[7] Oppen, D., "Prettyprinting", ACM Transactions on Programming Languages and Systems, V2 #4, October 1980, 465-483.

[8] Scowen, R. et al, "SOAP - A Program Which Documents and Edits Algol60 Programs", Comput. J. V14 #2, 1971, 133-135.

[9] Teitlebaum, T., "The Cornell Program Synthesizer", Tech. Rep. 79-370, Dept. of Computer Science, Cornell Univ., 1979.

Appendix

This appendix describes all of the facilities available in GPRINT, and how to use them. In conjunction with the main body of the paper, it is intended to be a manual for the system.

Using GPRINT

GPRINT is implemented in Lisp and is fully compatible with both MacLisp and the Lisp Machine. It is integrated fully with both systems, following all of the standard input/output conventions. In order to use GPRINT all you have to do is load a file. For MacLisp the file is "LIBLSP;GPRINT FASL". For the Lisp Machine the file is "LMLIB;GPRINT QFASL". The program text is available for inspection on LIBDOC; and LMLIB;. Bug reports should be sent to BUG-GPRINT@AI.

When the file loads in it initializes all of the variables it needs and sets up its environment. In order to install GPRINT as the standard top level Lisp printer you need only do:

```
(setq prin1 'Gprin1)
```

In addition to the Lisp top level, many of the standard system facilities which do output check this variable to see if there is a user specified output function. Note that you *cannot* redefine the function PRIN1 because GPRINT calls PRIN1.

In order to install GPRINT totally, you can call the function of no arguments GSET-UP-PRINTER. This sets PRIN1 and sets up some control keys (see the section on stopping and resuming printing).

The version of GPRINT described here obsoletes an older version which has been in use for several years. (This old version still exists as "LIBLSP;OGPRIN FASL", however, it does not run on the Lisp Machine and is no longer supported in any way.) If you just use the top level functions in the file then the new version is closely compatible. However, if you used user created templates or formatting functions then the new system is not compatible at all even though it is identical in conception.

Basic Top Level Functions

GPRINT/GPRINC/GPRINT1 *object &optional file format level length endline startline*

These are exactly analogous to PRINT, PRINC, and PRIN1 except that they do pretty printing. (Unfortunately, the standard MacLisp grind package has already used up the name GPRIN1.) The first argument is the object to be printed. The second argument specifies the file (stream) to use for output. If it is missing then the standard system defaults are used (e.g. TYO etc. on MacLisp and STANDARD-OUTPUT on the Lisp Machine).

The third argument is a formatting function which defaults to NIL. If non-NIL it will be used by GDISPATCH to format the object. For example (GPRINT FOO STANDARD-OUTPUT ':GFN-FORMAT) will use functional format for the top level of FOO no matter what the CAR of foo is. The last four arguments can be used to control abbreviation. Their meaning is discussed in the section on abbreviation below.

PL *object &optional file format*

Print in full - this is an abbreviation for (GPRINT OBJECT FILE FORMAT NIL NIL NIL NIL). It specifies that the object should be printed without abbreviation. It is quite handy at top level.

GEXPLODE/GEXPLODEC *object &optional format level length*

These are analogous to the MacLisp functions EXPLODE and EXPLODEC.

PLP *&quote &rest args*

Plist print - this is very similar to GRINDEF. Since the definitions of GRINDEF are totally different in MacLisp and on the Lisp Machine, there was no way to make a new function compatible with both, and so I just made my own definition that is basically similar but follows my sense of aesthetics.

The Dispatcher Revisited

The description in the main body of this paper of the behavior of GDISPATCH on lists ignored two issues. A more complete version of the code is shown below.

```
(defun Gdispatch (Gsuggested-format item)
  (cond ((depth-abbreviation-necessaryp) (GF "C" Gprinlevel-abbrev))
        ((Gselect Gspecial-formatters item))
        ((or (symbolp item) (numberp item)) (GF "P" item))
        ((listp item)
         (cond ((Gselect Goverriding-list-formatters item))
               (Gsuggested-format (funcall Gsuggested-format item))
               ((Gselect Glist-formatters item))
               (T (funcall
                    (cond ((and (listp (car item))
                                (memq (car item) '(lambda named-lambda)))
                           Gapply-format)
                          ((not (symbolp (car item))) Gnon-symbol-car-format)
                          ((get (car item) ':Gformat))
                          ((fboundp (car item)) Gfn-format)
                          (t Gsymbol-car-format))
                    item))))
        (dispatchers-for-other-data-types)))

(defun Gselect (formatting-function-list item)
  (do ((fns formatting-function-list (cdr fns)))
      ((null fns) nil)
    (cond ((funcall (car fns) item) (return T)))))
```

There are a number of things to notice about this function. The checking for depth abbreviation is done first so that it is orthogonal to everything else. Note that the formatting of depth abbreviation and atoms is specified directly in GDISPATCH and not through a function call. The variable GPRINLEVEL-ABBREV holds the thing to print when depth abbreviation occurs The main body of the paper discussed the way in which GDISPATCH looks at the :GFORMAT property of the CAR of a list in order to determine what format to use for it. This dispatching is extended in that it also looks at the CAAR of the list in order to see if it is a literal LAMBDA application. In this case the value of the special variable GAPPLY-FORMAT is used as the format. Instead of using the constant :GFN-FORMAT to format function applications, GDISPATCH indirects through the variable GFN-FORMAT which is bound to :GFN-FORMAT by default. Rather than using a single format (called :GDATA-FORMAT above) to format all random lists, the dispatcher indirects through two special variables: GSYMBOL-CAR-FORMAT and GNON-SYMBOL-CAR-FORMAT. By default these are set to the value :G1TBLOCK which does tabular blocking. You can set them to anything you want.

There is also a completely different kind of dispatching going on which was not discussed above. There are several special variables which can be used to specify formatting functions which will be used in place of the normal ones. Each variable holds a list of special formatting functions. Each of these functions has two purposes. First, they are called with the list to format as their argument, and they must test it in order to determine if they are applicable to it. If they are then they return T. Second, when they are applicable, they do the actual formatting of the list, creating the appropriate queue entries. GDISPATCH processes each of these special lists by calling the functions one at at time in order to determine if any of them is applicable (see

the function GSELECT). As soon as one of them returns T then the dispatcher knows that formatting has been completed. If none do, then the dispatcher continues on.

Before doing any of its normal dispatching, GDISPATCH looks at the list GSPECIAL-FORMATTERS. The purpose of this list is to make it possible for a user to take complete control of formatting for any *type* of object he chooses. The variables GOVERRIDING-LIST-FORMATTERS and GLIST-FORMATTERS are used to specify special formatting functions for lists. They are needed when you want to dispatch off of something other than the CAR of a list.

There is one more aspect to the dispatching for lists. GDISPATCH takes an additional argument GSUGGESTED-FORMAT. This argument holds a formatter which will be used for a list no matter what its CAR is. The purpose of this is to allow the format for a form like PROG to specify how its subparts should be formatted. Without this, a PROG bound variable list would be formatted in functional application format if it happened to start with an atom that was a function name. The difference between GOVERRIDING-LIST-FORMATTERS and GLIST-FORMATTERS is that any format put in the former will override any format in GSUGGESTED-FORMAT. GLIST-FORMATTERS is the variable which should be used for supplying special formatters in all normal situations.

Another way to deal with the fact that the formatting functions for complex forms have to have a way to specify how their sublists should be formatted would be to have these formatters format the sublists themselves directly without calling GDISPATCH. This is the approach which was implicitly taken in the examples above. Unfortunately, this approach is error prone. If a form has an atom where a list is normally expected, then the formatter will blow up. In order to avoid this, the formatting for all sublists is channeled through GDISPATCH (see the discussion of the $ format code in the next section).

The Function GF Revisited

The description in the main body of this paper of the options available for making templates left out several features and simplified the descriptions of several others. In particular the format codes ".", C, S, I, &, %, and $ are not described in the main paper. The discussion below gathers together in one place a complete description of all of the options available including the ones which were not described above.

GF *template &rest args*

This is a macro that creates code which formats the args as specified by the template. The template is a string of single character commands, some of which can be followed by a parameter. There are three kinds of parameters:

n - Some commands take a number as a parameter. This number should be an integer optionally beginning with a "-" and/or ending with a ".". Alternately, it can be omitted in which case a default value is used.

f - Some commands take a function name as a parameter. This name is an arbitrary symbol possibly containing ":". Case does not matter. The symbol must be terminated by a blank. Function name parameters cannot be omitted. They have no default values.

- This can be used in place of any numeric parameter or any function name parameter. It indicates that the next input to GF should be used as the parameter, instead of a literal value.

The commands which can be used in a template are divided into several categories. The first set is used to parse the structure of the arguments to GF so that their parts can be accessed.

[...] - This is used to access the internal elements of an item which is a list. The template inside the brackets refers to the elements of the list. If the item is not a list, then no formatting of it, or anything inside it, is done. Processing begins by considering each element of this list in turn. As soon as the list is exhausted, control skips out of the subtemplate and continues after its end. This is done even if there is more stuff left in the subtemplate. Special code is included to deal with the possibility of unexpectedly encountering a non-NIL atomic CDR. If this happens it is automatically formatted to appear after a ".". [...] also produces special code to deal with length abbreviation. They only way to get it automatically is to use [...].

. - This is valid only inside a [...]. It specifies that the next item is the whole sublist left to process by [...] rather than its CAR. For example, (GF "[P_P_.P]" '(1 2 . 3)) is the same as (GF "P_P_P" 1 2 3). Note that when a "." is used, normal checking for the end of the list in the [...] is suppressed. For example, (GF "[P_.P_]" '(1)) is equivalent to (GF "P_P_" 1 NIL). The NIL at the end of the list is explicitly picked up by the ".", and a blank will be printed at the end. This happens even though the [...] template would normally have terminated right after the first P.

<...> - This can only be used directly inside [...] (or (...)). It specifies an indefinite repeat block. This is used to specify a template for a list of unknown length.

The next set of commands are used to specify how individual items are printed out.

P - Print the corresponding item using the default printer (PRINC if GPRINC was called or PRIN1 if GPRINT or GPRINT1 was called).

C - Print the corresponding item using PRINC.

S - Print the corresponding item using PRINC but do not count it as one of the items printed from the point of view of length abbreviation.

I - Ignore the corresponding item.

'...' - Print the indicated literal using PRINC and do not count it as one of the items printed from the point of view of length abbreviation. Note that in the literal "''" stands for "'".

* - This specifies that GDISPATCH should be called in order to format the corresponding item.

&*f* - Call the function *f* with no arguments at this point.

%*f* - This specifies that the function *f* should be called in order to format the corresponding item. (Note if *f* is # then the argument which is used as the function follows the argument which is formatted.)

$*f* - The dollar sign command specifies that GDISPATCH should be called in order to format the corresponding item, but that the function *f* should be passed to GDISPATCH as a suggestion of how to format the item. (Note if *f* is # then the argument which is used as the function follows the argument which is formatted.) The difference between $*f* and %*f* is that with $*f* GDISPATCH gets control. As a result, if the item is not a list, or if some function on GOVERRIDING-LIST-FORMATTERS formats it, then the function *f* will not get used.

$/"...//" - In addition to the name of a function, the parameter to $ can be a literal template which is converted into a function to use. (Note that the quotes have to be slashified in order to read in inside a quoted string.) The formatting function produced is compiled out of line. As a result, if there is a # format code in it, the argument to GF that this refers to will be compiled out of line. In order for this to work any variables this refers to must be declared special.

The next commands are used to specify the nested structure of the output (which need not be the same as that of the input).

{*n*...} - This indicates a substructural unit in the output. The parameter specifies what indentation to use when printing out the items inside the substructure if the substructure cannot be printed on a single line. (If the indentation is specified to be zero then the substructure is not counted as increasing the depth from the point of view of depth abbreviation.) The default parameter value is calculated as the sum of the lengths of the first thing printed in the substructure, and any literals before it and any spaces after it.

(*n*...) - This is a useful abbreviation in the situation where the nested structure of the output is the same as the nested structure of the input, and when you want to print parentheses around the structure. It is an abbreviation for {*n*'('[...]')'}. Additionally, if the (*n*...) is nested more directly inside [...] than inside $ then it is treated as an abbreviation for $/"{*n*'('[...]')'}/". In other words, if the item whose format is being specified by the (*n*...) was passed through GDISPATCH for dispatching then the $ format code is used to force the list to dispatch through GDISPATCH. This prevents the format from blowing up when the item is not a list. (Note the comment about # inside $/"../" above.)

The final set of commands specifies spacing and where and when carriage returns should be printed.

A - Do a line break here always.

N - Do a line break here if required for normal mode printing. I.e. if and only if the structure immediately containing this point cannot be printed on a single line.

B - Do a line break here if required for block mode printing. This is the same as N except that even if the immediately containing structure is being broken up a line break will not be put here as long as the following structure can be printed on the end of the current line and the prior structure at this level was printed on a single line.

_*n* - Print *n* (default 1) spaces. (Note that spaces are elided if they are the first or last thing on a line).

T*n* - Tab over. Moves to a place where the character position relative to the current indentation is congruent to zero modulo *n*. (Doesn't move at all if it doesn't have to.) When necessary, a default tab size is calculated based on the length of the other items in the substructure.

The characters SPACE, TAB, CR, and LF are all ignored. Any other character is an error.

The fact that GF is a macro saves time by parsing the template at compile time, and producing efficient code to do the formatting. This does waste space however. It is to your advantage to make each template as short as possible.

Several of the codes shown above were not discussed in the main body of the paper. The following examples illustrate their use. The codes "." and C are used in the following redefinition of the format for the NAMED-FORM data structure used as an example above. The structure is changed so that the name field is a CONS of a name and a suffix rather than a LIST, and so that the name is a lower case string which needs to be PRINCed rather than PRIN1ed. The "." format is used to parse the name field, and the C format is used to PRINC the name part. (Note that if you call GPRINC on a structure, all of the atoms will be PRINCed.)

```
                (GPRINT '(NAMED-FORM ("calling-form" . 3) (CONS X (+ X 2))))
        prints: calling-form-3 - (CONS X (+ X 2))

(defun (named-form :Gformat) (item)
  (GF "{2")
  (cond ((atom (cadr item)) (GF "C" (cadr item)))
        (T (GF "[C'-'.P]" (cadr item))))
  (GF "_'-'_N*}" (caddr item)))
```

The I format code is illustrated by the following alternative format for QUOTE. It is used to explicitly ignore the first component of the QUOTE list as it is parsed by [...].

```
                (GPRINT '(QUOTE (A (QUOTE B))))
        prints: '(A 'B)

(defGF quote "{''''[I *]}")
```

The use of the $ code is illustrated in the following format which block formats a tree at all levels. It is capable of formatting trees of arbitrary depth because it explicitly calls itself recursively. GDISPATCH is called at each level of the recursion. As a result as soon as an atom is encountered, the recursion is terminated and the atom is printed normally.

```
(defun :Gblock (tree) (GF "(1<$:Gblock _B>)" tree))
```

The following formatting function for PROG uses % so that it can call a sub-format (GPROG-FORMAT2) *without* GDISPATCH being called. This is necessary so that the labels (which are atoms) in the PROG will be processed by the GPROG-FORMAT2. This function prints labels shifted left and keeps track of their lengths so that the form following them can be printed at the right place.

```
        (PROG (A B)
              (form)
         L    (form)
         BIG-LABEL (form)
         A B  (form))

(declare (special Gdefault-spaces Gspaces))

(defun (prog :Gformat) (list)
  (let* ((Gdefault-spaces 5) (Gspaces 5))
    (GF "(0*_$:Gblock <%Gprog-format2 >)" x)))

(defun Gprog-format2 (item)
  (cond ((= Gspaces Gdefault-spaces) (GF "A' '")))
  (cond ((atom item)
         (setq Gspaces (- Gspaces 1 (flatsize item)))
         (GF "P_" item))
        (T (GF "_#*" (max Gspaces 0) item)
           (setq Gspaces Gdefault-spaces))))
```

The formatting function below prints out a special data structure as a list of items separated by a delimiter. There are no parentheses around the list, and the delimiter to print is supplied as part of the structure. The & format code is used in order to call a sub-format which prints the actual delimiters. This function uses the S format code in order to PRINC the delimiter. It uses S, instead of C, so that the delimiters will not be counted as part of the length of the object when length abbreviation is being performed.

```
                (Gprint '(DELIMITED-LIST |,| APPLE BANANA PEAR))
        prints: APPLE, BANANA, PEAR

(declare (special delimiter))

(defun (delimited-list :Gformat) (list)
  (let ((delimiter (cadr list)))
    (GF "{1[<* &delimited-list-format2 _B>]}" (cddr list))))

(defun delimited-list-format2 () (GF "S" delimiter))
```

An important aspect of the last two examples is the way they interact with length abbreviation. Since length abbreviation is implemented by [...], in order to get length abbreviation to apply to the formats you write, you have to use [...]. This is an important reason for writing them in the form given above rather than as a single routine containing a loop which just prints the right things.

Data Types Recognized

In addition to lists, GPRINT has built in formatters for all of the standard Lisp data types. Symbols, numbers, strings, and things of random types not specifically discussed below are treated as indivisible atoms and printed in the standard ways. The way lists are handled is discussed fully above.

Hunks

In MacLisp a hunk is formatted in one of two ways depending on whether or not it is a USRHUNK. If it is a USRHUNK (note that EXTENDs and the like are all USRHUNKs) then GPRINT checks the messages it accepts. If it takes the message :GFORMAT-SELF then GPRINT sends a :GFORMAT-SELF message with the object as argument to the object so that it can format itself. Note that the receiver for this message is in essence a formatting function for the object.

If a USRHUNK doesn't take a :GFORMAT-SELF message, but it does take a :PRINT-SELF or PRINT message then GPRINT treats the hunk as an atomic object and lets the standard printer print it. This makes it possible to use GPRINT on these objects without having to write formatters for them. However it should be noted that since they are treated as atomic objects, no formatting occurs inside them no matter how large their print form may be. For example, a line break will never be inserted inside one.

If a USRHUNK doesn't accept any of these messages, then it is treated as an ordinary hunk. In order to format an ordinary hunk GPRINT first checks to see if there is a formatting function for the hunk. The user sets up a hunk formatter by adding a function to the list in the variable GHUNK-FORMATTERS. The purpose of this function is two fold: to test whether it is applicable to a hunk (in which case it returns T) and in this case to actually format the hunk. GPRINT calls each of these functions in turn passing it the hunk. As soon as one of them returns T it stops. If they all return NIL then the hunk is printed by default in the normal way (e.g. in parentheses with the CXRs separated by periods) in block format.

Named Structures, Entities, and Instances

On the Lisp Machine, named structures, entities, and instances are printed in one of two ways depending on whether or not they know how to format themselves. If the object accepts the message `:GFORMAT-SELF` then `GPRINT` sends a `:GFORMAT-SELF` message with the object as argument to the object so that it can format itself.

If the named structure, entity, or instance doesn't take a `:GFORMAT-SELF` message, then `GPRINT` treats is as an atomic object and lets the standard printer print it. This makes it possible to use `GPRINT` on these objects without having to write formatters for them. However it should be noted that since they are treated as atomic objects, no formatting occurs inside them no matter how large their print form may be. For example, a line break will never be inserted inside one.

Arrays

If an object is an array (and not a named-structure) it is formatted as follows. `GPRINT` first checks to see if there is a formatting function for the array. The user sets up an array formatter by adding a function to the list in the variable `GARRAY-FORMATTERS`. The purpose of this function is two fold: to test whether it is applicable to the array (in which case it returns `T`) and in this case to actually format the array. `GPRINT` calls each of these functions in turn passing it the object. As soon as one of them returns `T` it stops. If they all return `NIL` then a default formatter is used.

The default array formatter first prints out the array object in the standard way (e.g. as an atom containing the type and the address). Next, if the variable `GPRINT-ARRAY-CONTENTS` is `T` and the array has only one or two dimensions it prints out the contents of the array. The contents are printed as a list (for one dimensional arrays) or a list of lists (for two dimensional ones). Tabular blocking is used to format these lists.

Formatting Functions

The operation and use of formatting functions is described in the main body of this paper, however, several additional points should be made. First, formatting functions are supported by many of the parts of `GPRINT`, not just those that handle lists (see the discussions of `USRHUNK`s, hunks, named structures and arrays above). Second, a number of standard formatting functions are provided including ones for `LAMBDA`, `NAMED-LAMBDA`, `LET`, `LET*`, `SETQ`, `COND`, `PROG`, `DO`, `DEFUN`, `QUOTE`, `FUNCTION`, `BACK-QUOTE`, `MACROEXPANDED`, and `SI:DISPLACED`.

Abbreviation

`GPRINT` provides several different abbreviation mechanisms. First, there is abbreviation based on `PRINLEVEL` and `PRINLENGTH` as in the standard printer. "#" (or "**" on the Lisp Machine) is printed for structures which are too deep, and "..." in place of the ends of lists which are too long. (The *level* and *length* optional arguments to `GPRINT` can be used to set the abbreviation parameters at values other than the ones established by the global variables.)

Second, there is a separate facility based on the variables `PRINSTARTLINE` and `PRINENDLINE`. As `GPRINT` prints, it counts the lines starting with zero for the line the printer is called on. While the line number is less than `PRINSTARTLINE` no actual printing is done. (`PRINSTARTLINE` defaults to `NIL` which is the same as zero.) If the line number ever becomes greater than `PRINENDLINE`, then the printer prints "---" to indicate that truncation has occurred and immediately stops printing and returns normally. (`PRINENDLINE` defaults to `NIL` which is the same as infinity.) I have found setting this to a relatively small number like 4 very useful particularly due to the availability of the continuation facilities described below. (The *endline* and *startline*

optional arguments to GPRINT can be used to set the abbreviation parameters at values other than the ones established by the global variables.)

As a third kind of abbreviation, if the variable GCHECKRECURSION is T then GPRINT checks for circularity in the objects it is printing. (This is rather expensive, but not prohibitively so.) When a circular reference to an object is encountered, it is replaced in the output by ^*n* or %*n*. %*n* is only used in a list. It is used when the CDR of a list is EQ to an earlier CDR in the *same* list. In this case *n* is the number of CDRs separating the two positions. ^*n* is used in other situations. Here, *n* indicates that *n* selector operations (CAR, CXR, AREF; but not CDR) were performed between the first occurrence of the object and the second.

```
the result of (LET ((X '(Y (Z 1 2 3) 4)))
                 (RPLACD (CDR X) (CDR X))
                 (RPLACA (CDADR X) X)
                 (RPLACA (CDDADR X) (CADR X))
                 (RPLACD (CDDADR X) (CDADR X))
                 X)
prints as      (Y (Z ^2 ^1 . %2) . %1)
```

It is possible (but not easy) to reconstruct the exact shape of the object from what was printed. However, the main purpose is just to print something more readable than what you would otherwise see. An important feature of the way this abbreviation is done is that it is completely orthogonal to the rest of the formatting process so that it works no matter what kinds of user formatting functions are written.

Stopping and Resuming Printing

If you execute the function GSET-UP-PRINTER several useful facilities are installed as interrupt characters. GPRINT is set up so that it can be stopped in the middle of printing and caused to return normally without doing any more work. This is triggered by typing CONTROL S in MacLisp and TERMINAL STOP-OUTPUT on the Lisp Machine.

Whenever output is truncated or abbreviated by forcefully aborting as above (or due to abbreviation based on PRINLEVEL, PRINLENGTH, or PRINENDLINE) GPRINT remembers the state of the printing so that it can be resumed. Only a single variable is maintained so that only the most recently abbreviated thing is remembered. If printing was truncated by PRINENDLINE or user intervention, then it can be continued from the point of truncation by typing CONTROL C in MacLisp, and TERMINAL RESUME on the Lisp Machine.

As an additional feature, you can reprint the last abbreviated thing in full with PRINLEVEL, PRINLENGTH, PRINSTARTLINE, and PRINENDLINE set to NIL by typing CONTROL P in MacLisp and TERMINAL 1 RESUME on the Lisp Machine.

Miscellaneous Features

A thing which has always irritated me is that if printing starts within about 7 lines of the bottom of the screen "more" processing is inhibited. By default, this is disabled. If you actually like this feature, than you can continue to get it by setting GFORCE-MORES to NIL.

There is a special formatting function set up for lists that start with the atom :GUNTERPRI. This function forces the printer to go back up a line without printing anything. This is provided so that you can have a function return (:GUNTERPRI) when you don't want to see it print out any results at all (not even the TERPRI that the Lisp top level prints out). On the Lisp Machine you can get this effect better in most situations by having your program return (VALUES).

On the Lisp Machine, GPRINT is set up so that ~N is a FORMAT keyword which invokes GPRINT1 (~G is taken already). ~:N invokes GPRINC. Numeric pre-arguments are taken to be PRINLEVEL, PRINLENGTH, etc.

(Note that even when doing a (FORMAT NIL ...) line breaks may be introduced into the output.)

In order to protect the user from errors, the printer is executed inside an ERRSET which catches any errors that occur. If there is an error than the printer prints out an error message to that effect instead of printing the object. This is useful in general, however, it can make debugging difficult. If you set GSHOW-ERRORS to T, then an ERRSET is not used, and you get a break when the error occurs.

One of the strengths of the GPRINT system is that it is designed to be efficient in both time and space. The basic algorithm is linear in the size of the object being printed, and so it is relatively fast even on large objects. There is however, a rather large constant overhead due to the large amount of dispatching which is done in order to decide how to format each node in the object. An important feature of the algorithm is that it does no CONSing in normal operation and requires memory proportional only to the line length rather than to the size of the object being printed. These savings are particularly important when running in MacLisp.