MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 279 February 1973

PRETTY-PRINTING
CONVERTING LIST TO LINEAR STRUCTURE

Ira Goldstein

ABSTRACT

Pretty-printing is the conversion of list structure to a readable format. This paper outlines the computational problems encountered in such a task and documents the current algorithm in use.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

PRETTY-PRINTING

CONVERTING LIST TO LINEAR STRUCTURE

ABSTRACT

Pretty-printing is the conversion of list structure to a readable format. This paper outlines the computational problems encountered in such a task and documents the current algorithm in use.

IRA GOLDSTEIN
JANUARY 31, 1973

CONTENTS

I. Introduction

II. Computational Analysis

- A. The Basic Task
- B. Finite Width
- C. Linear Format
- D. Finite Length
- E. The RECURSIVE RE-PREDICTOR, A Top-Down Approach
- F. The Table Scheme, A Bottom-Up Approach
- G. Semantics
- H. Comments
- I. History

III. Documentation

- A. Top level functions
  1. GRIND and GRINDO
  2. GRINDEF
  3. Formatting
  4. REMGRIND
  5. Functions, atoms and properties reserved by grind.

- B. Predefined formats
  1. Standard formats
  2. Special GRINDFNs
  3. Inverting read macros
  4. System packages

- C. Comments
  1. Single semi comments
  2. Double semi comments
  3. Triple semi comments

- D. Grind control

- E. Defining new formats
  1. GRINDFNs
  2. Vocabulary
  3. Examples
  4. GRINDMACROs

IV. Suggested future improvements

- A. Conceptual
- B. Implementation

I. INTRODUCTION

Pretty-printing is a fundamental debugging aid for LISP. List structure presented as an unformatted linear string is very difficult for a person to understand. The purpose of pretty-printing is to clarify the structure of a LISP expression. The simplest class of pretty-printers accomplishes this by the judicious insertion of spaces and carriage returns. Section II analyzes the computational complexity of such algorithms. [See section IV for suggestions for more sophisticated schemes which break the code into separate expressions.] The existence of algorithms which are only linearly more expensive than the standard LISP printing routines is demonstrated. Various extensions for adding semantic knowledge to the pretty-printer are then considered. Section III documents the pretty-print package currently available for MACLISP. Section IV suggests additional improvements to be considered for the future.

II. COMPUTATIONAL ANALYSIS

A. THE BASIC TASK

The LISP PRINT'ing primitives print expressions as strings. Their only concession to clarity is the insertion of a carriage return each time the right margin is reached. This results in code which is not very readable when longer than a single line. Indeed, the carriage returns can even be inserted directly into the middle of a word. [The LISP reader ignores carriage returns on input]. The following example is the definition of FACTORIAL PRINT'ed by LISP. The dots represent the left and right margins.

```
.                         .
 (DEFUN FACTORIAL (X) (CO
 ND ((= X 0) 1) ((* (FACT
 ORIAL (1- X)) X))))
.                         .
```

Let L be a list of the following form:

```
(<FUNCTION> <ARG(1)> <ARG(2)> ... <ARG(N)>)
```

The objective of the pretty-printer is to present L in a fashion which emphasizes its procedural role. The "standard format" for accomplishing this is aligning the arguments one under the next.

```
(<FUNCTION> <PRETTY-PRINT ARG(1)>
            <PRETTY-PRINT ARG(2)>
                        .
                        .
                        .
            <PRETTY-PRINT ARG(N)>)
```

Using this format, the FACTORIAL function takes on the following, more understandable, appearance:

```
(DEFUN FACTORIAL
       (X)
       (COND ((= X 0) 1)
             ((* (FACTORIAL (1- X))
                 X))))
```

Note that any format used by pretty-print must leave L re-readable by LISP. Hence, the following structure would be illegal:

```
(<FUNCTION> <PRETTY-PRINT ARG(1)> <PRETTY-PRINT ARG(N/2 + 1)>
            <PRETTY-PRINT ARG(2)> <PRETTY-PRINT ARG(N/2 + 2)>
                     .                         .
                     .                         .
                     .                         .
            <PRETTY-PRINT ARG(N/2)> <PRETTY-PRINT ARG(N)>)
```

If the only problem which the pretty-printer faced was the insertion of extra spaces and carriage returns, the computational cost in excess of the standard LISP PRINT would be negligible. The difficulty arises from the finite width of the page. For sufficiently large s-expressions, every sublist cannot be printed in standard format. Instead, the less desirable miser format must be used.

```
(<FUNCTION>
 <PRETTY-PRINT ARG (1)>
            .
            .
            .
 <PRETTY-PRINT ARG (N)>)
```

This format is minimal with respect to the indentation of the arguments. All arguments begin only one space over from the opening parenthesis.

There are rare instances of lists that cannot be pretty-printed even in miser format. If the depth of the list exceeds the width of the page, indenting one for each level is impossible. See suggestion A-3 in section IV for a technique for handling expressions of great depth.

The role of miser format is illustrated by our FACTORIAL examples. When first shown PRINTed, the pagewidth was 24 spaces. However, the pagewidth was increased to 35 in order to demonstrate standard format. Without the extra width, it is impossible to use standard format on the list and all of its sub-expressions without exceeding the right-hand margin. Hence, the pretty-printer is faced with the necessity to use miser format on some sub-expressions if the entire list is to fit on the page. This prediction represents the basic extra-cost above the standard LISP PRINT which the pretty-printer requires. The following format for FACTORIAL ilustrates the cautious use of miser format until sufficient width becomes available to switch to standard form.

```
.                               .
 (DEFUN
  FACTORIAL
  (X)
  (COND
   ((= X 0) 1)
   ((* (FACTORIAL (1- X)
       X))))
.                               .
```

B. FINITE WIDTH

What, then, are the basic computational costs for pretty-printing on a page of finite width? If lists are described as trees, then the cost of printing is simply that of visiting each tip of the tree in left-to-right order. The cost of pretty-printing will be analyzed with respect to this basic "tree traversal" overhead. Upon first arriving at any non-terminal node of the tree, the pretty-printer has no knowledge of the size of the subtree beginning there. Hence, it cannot know whether there is sufficient space to use standard format. The pretty-printer must apply a prediction function to the subtree to estimate the width required to print it in standard format. If that width is more than is currently available, miser format must be used. The additional cost of pretty-printing, then, is simply the cost of prediction.

One criterion for judging different pretty-print algorithms is the number of times each node of the tree must be revisited. In these terms, a minimal algorithm would perform only two tree traversals - one to obtain prediction information and one to actually print the subtree.

The following analysis will proceed at a qualitative level. The assumption will be that list operations represent the major cost, with numerical operations being cheap. The intention is to give the reader the flavor of this computational problem. However, to turn these assertions into theorems would require a more formal attack. For example, a precise comparison of the cost of numerical versus list operations would be necessary. Otherwise, one pass through the the tree could be used to Godelize it. Subsequent computation could then be entirely numerical.

Is a "minimal" two-traversal algorithm possible? The answer is yes. [This yes assumes that the number of lines needed to print the expression is ignored. The section on "finite length" considers this additional complexity.] One pass can be made to associate with each sublist the minimal width needed to print it in standard format. This information completely determines how the s-expression is printed. The pretty-printer uses the more economical miser format from the top down, until the available width exceeds the minimum needed to use standard format. At that point, the printer is assured of room to print all remaining sublists in standard format. This is the structure which was used to print FACTORIAL in the last example.

A single prediction pass is sensible providing the cost of storing and accessing the minimum width computed for each piece of substructure is less than the cost of recomputing the number. Fortunately, this is the case. For example, a hash table accessed by a numerical computation on the pointer to the sublist takes fixed time, regardless of the size of the list structure. Of course, for sufficiently small list structures, the fixed cost of accessing and clearing a hash table will not be worthwhile. But this is uninteresting mathematically. Indeed, even from a practical standpoint, the hash scheme is so fast that its overhead is not noticed on small lists.

C. LINEAR FORMAT

Analysis of the pretty-printing task was begun in reaction to the uninformative use of "linear format" by the LISP primitives. However, when a sub-expression can fit in the space remaining on the line, linear format is sensible. As we shall see, even with this additional complexity, two tree traversals are sufficient.

The prediction pass must now save two pieces of data - the linear width of the sub-expression as well as its minimum width. These two numbers can be computed on the same pass through the tree. The printing pass is extended in the obvious way. First preference is given to linear format if sufficient width is available. Otherwise, the algorithm is as before.

D. FINITE LENGTH

There is an additional element of complexity in pretty-printing that has not yet been considered. When LISP code is spread out over many lines or, worse, many pages, it again becomes indecipherable. Hence, a pretty-print algorithm should also attempt to format s-expressions in the least number of lines. To achieve the minimum number of lines, we shall have to allow an increase in computational cost. Nevertheless, we will propose a scheme which still requires only two tree traversals and is therefore linear in the size of the tree.

The predictor described above can compute the number of lines needed to print an expression in minimal width. The difficulty, however, is that there may be extra width available. This can allow the use of linear format to decrease the number of lines needed to print the expression. For example, for FACTORIAL, the pretty-printer always prints the second argument of "*" under the first. However, with sufficient width, a line is saved by printing (* (FACTORIAL (1- X)) X) in linear format.

```
(DEFUN FACTORIAL
       (X)
       (COND ((= X 0) 1)
             ((* (FACTORIAL (1- X)) X))))
```

For functions with many arguments such as (PLUS 1 2 3 4 5 6 7), the use of linear format over standard format can make a significant difference in the number of lines and, consequently, the readability. Thus, remembering a single datum corresponding to the number of lines needed to print a given sub-expression in minimal width is not sufficient information. At first blush, it would appear necessary to reexamine each sub-expression every time the available width changes.

E. THE RECURSIVE RE-PREDICTOR, A TOP-DOWN ALGORITHM

Let us begin by examining approaches that do reexamine sublists many times. One obvious algorithm is to consider all possible format choices at each node, computing the resulting number of lines required. By brute search, this approach is guaranteed to find the sequence of formats that yields the minimum number of lines. However, the exponential cost is certainly prohibitive. A less powerful but less costly alternative is the RECURSIVE RE-PREDICTOR.

The RECURSIVE RE-PREDICTOR works in the following way. Upon arriving at a given node, the algorithm knows N, the remaining available width. Linear format is used if N is sufficiently large. Otherwise, it estimates how many lines it would take to print the arguments

      in width (N - 1) corresponding to the use of miser format
  and in width (N - <linear width of the function>) corresponding
      to standard format.

The estimate is made by guessing that all sublists are printed in the following way:

    linear format if sufficient width;
    else standard format.

This scheme is not guaranteed to find the sequence of format choices that results in the minimum number of lines. It does not consider all possible sequences. When insufficent space occurs, it prints the toplevel expression in miser format. It ignores the possibility of printing the toplevel expression in standard format while printing the sublists in miser form.

Computationally, the RECURSIVE RE-PREDICTOR can reexamine a given subtree many times. Thus, the cost is still exponential in the worst case. Nevertheless, for various reasons, this approach is possible:

1. Lists beginning with non-atomic elements such as LAMBDA expressions can always be printed in miser format. This avoids prediction costs for these sublists.

2. There is no longer any point to remembering the minimum width needed for standard format. Since the predictor must reexamine each sublist for the number of lines, it can at the same time check that the list fits in the given width.

3. A hash table can still used to remember the linear width.

4. Empirically, much LISP code is broad but not deep. PROG's are typical examples. After predicting and printing the first level or two, it is often the case that the remaining elements almost all fit in linear format. Thus, little recursive re-prediction is needed.

This RECURSIVE RE-PREDICTOR is the current pretty-print algorithm in use. Empirical observations indicate that it is only some four to five times slower than PRINT. Thus, it is of practical use. The next section describes an algorithm that is theoretically linear in the size of the list. It has not yet been implemented, and, in practice, may not be worth implementing. The use of tables and numerical operations is required. The overhead of these computations might be prohibitive for handling the average LISP expression. Also, such numerical operations are more efficient hand-coded in LAP than written directly in LISP. In any case, the final verdict must await implementation.

F. THE TABLE ALGORITHM, A BOTTOM-UP APPROACH

A bottom-up attack can yield a predictor which is linear in the size of the tree. One prediction pass is used. The trick will be to remember more than just the minimal width and corresponding length. Instead, a step function must be built for each node which provides the minimal number of lines resulting for different widths. For example,

(PLUS 2 3 4)

WIDTH	# OF LINES	FORMAT
0-4	impossible	
5-7	4	miser
8-11	3	standard
12-LINEWIDTH	1	linear

Such tables are finite. The number of intervals is limited by the finiteness of LINEWIDTH. The tables for all of the daughters of a given node determine the table for the parent. Before giving more details of this table scheme, notice that the cost is only a linear increase in the basic "tree traversal" computation. [This assumes that the cost of numerical COMPARE's needed to merge tables is roughly comparable to moving up and down levels in the tree.]

The table for the parent is built by merging the tables for the daughters, creating their "refinement". For example, the table for (PLUS 2 3 4) given above is built from the tables for the atoms PLUS, "2", "3" and "4". For each possible width, the table entry is the minimum number of lines to pretty-print the given subtree. This is determined by checking the number of lines resulting from each format. The number of lines to print a given tree in a given format is completely determined by the choice of format and the tables for the daughters. A given initial width and a given format imply a specific width for each daughter. The predictor, then, looks up the number of lines that the daughter requires for that width. The total number of lines is obtained by summing over all the daughters.

The format used to obtain the minimum number of lines is recorded as well. Ultimately, this bottom-up approach yields a table for the toplevel list. The entry for the total LINEWIDTH gives the number of lines to print the expression as well as the program for doing it.

Some savings in cost is possible. This can be done by determining limits for the widths that a given table must consider. The maximum width is:

LINEWIDTH - DEPTH.

This is true since each level of the tree costs at least one unit of width in order to print the opening parenthesis. Alternatively, it can be viewed as the width corresponding to using only miser format. A lower bound on the table is obtained by considering the use of only standard format. This results in maximal indentation. For each use of standard format, the available width decreases by

```
  1      ;for the opening parenthesis
+ FLAT   ;where FLAT equals the linear width of the first element
+ 1      ;for the space between the first and second elements.
```

These upper and lower bounds are computed as the predictor travels down the tree. The tables are computed on the return trip back up. Thus no extra tree traversing is necessary. An additional bound on the minimum width that need be considered for a given table is obtained by the left-to-right analysis of the daughters of each node. Suppose the table for daughter(1) asserts that it is impossible to pretty-print this subtree in less width than MIN. Then, it is unnecessary to consider widths less than MIN for the remaining daughters.

However, it is clear that such savings, though useful from a practical standpoint, still leave the algorithm linear in the size of the tree. Indeed, the table algorithm is essentially minimal in its cost. This can be illustrated by a worst case analysis. Suppose that an intermediate width W in a table for the sublist L is not computed. Obtaining the minimum number of lines can be made to hinge on just this piece of information. A sketch of the argument is:

> Construct a supertree for L for which a sequence of miser-standard choices could be made resulting in width W being possible.
>
> Construct the sisters of L such that they pretty-print optimally in this width.
>
> Then, if L behaves well for width W, it should be chosen. But if the number of lines to print L in width W is large, then it is not worth choosing.
>
> Hence, the choice of format depends on how L behaves in this width.

G. SEMANTICS

So far, we have introduced only three formats for lists:

```
standard format
miser format
linear format
```

Knowledge of the semantics of various types of s-expressions leads to additional forms. For example, argument lists for PROG's and LAMBDA's are preferably presented as blocks.

```
(PROG (***** ***** ***** *****
       ***** ***** ***** *****)
      . . . . . . .
TAG   . . . . . . .           ;tags are unindented.
      . . . . . . . )
```

Similarly, the preferred format for SETQ should be:

```
(SETQ NAME(1) <PRETTY-PRINT OF VALUE(1)>
      NAME(2) <PRETTY-PRINT OF VALUE(2)>
                 .
                 .
                 . )
```

This additional versatility can be achieved by extending the pretty-print algorithm. In the current PRETTY-PRINT package, special formats have been designed for many LISP primitives. [This includes informing the predictor of the special way such functions as PROG and SETQ are handled.] If sufficient space is available, these formats are preferred over standard or miser format. See section III for details.

H. COMMENTS

The importance of documenting code cannot be underestimated. Hence, the pretty-printer, when applied to files, formats semi-colon comments. These comments can be inserted in the code or printed on the right-hand half of the page. Again see section III for details.

I. HISTORY

Bill Gosper developed one of the earliest pretty-print algorithms for LISP. It used the recursive re-prediction scheme to minimize the number of lines. Eugene Charniak modified the program to process semi-colon comments. Ira Goldstein extended the comment formats, made the pretty-printer programmable with respect to adding

new formats for special functions, added a hash scheme for linear width and developed the table algorithm discussed above. Carl Hewitt, Guy Steele, John White, Gerry Sussman, Terry Winograd, Bruce Roberts, and Stavros Macrakis provided many helpful suggestions.

III. Documentation

The new grind package differs from earlier ones in providing a larger number of formats in which s-expressions and comments can be ground. A variety of predefined formats exist which can be associated with any LISP function. For unusual formats, the user can design his own procedures to control grinding.

The grind package is automatically loaded into LISP upon executing GRIND or GRINDEF. Alternatively, the user can obtain the file via:

```
(FASLOAD F GRIND COM)
```

The REM feature can subsequently be used to eliminate unwanted code (see section A-4). Send suggestions and bugs to IRA.

A. Top level functions

1. GRIND and GRINDO - fexprs

GRIND and GRINDO convert files to pretty-printed form. Their input format is that of the LISP file manipulating functions like UREAD and UWRITE.

```
(GRIND <filename1> <filename2> <device> <uname>)
```

UFILE's a pretty-printed form of the file under the same name. The usual LISP conventions for default device, user and file names are used. To avoid possible disasters, use ">" as your second file name. GRINDO does not UFILE. Hence, it is useful for filing the pretty-printed file under a different name. For example,

```
(GRINDO GEO > DSK IRA) (UFILE GEO PRINT)
```

results in the pretty-printed version being filed as GEO PRINT.

2. GRINDEF - fexpr

GRINDEF takes atoms as arguments. It then pretty-prints their EXPR, FEXPR, MACRO and VALUE properties. For example,

```
(GRINDEF PROGRAM1 PROGRAM2)
```

pretty-prints these two LISP functions.

The default properties pretty-printed by GRINDEF can be modified in two ways.

```
(GRINDEF <LIST OF ADDITIONAL PROPERTIES> <ATOM1> <ATOM2> ...)
```

appends the additional properties to the list of default properties for the duration of the current call to GRINDEF. A permanent change to the default properties pretty-printed by GRINDEF is made by setting the atom "GRINDPROPERTIES" to a new list of properties.

"(GRINDEF)" will repeat the last call to GRINDEF. This saves typing when repeatedly GRINDEF'ing the same functions.

3. Formatting

The pretty-printer can be programmed in the following ways:

a. (<grind-control-fn> <arguments>) executes the grind-control-fn on the given arguments. A typical grind control function is PROGRAMSPACE. (PROGRAMSPACE 80) sets the width available for pretty-printing code to 80. Complete documentation follows in III-C.

b. (<GRINDFN or GRINDMACRO> <function> <grind-format>) assigns the grind-format to the function as either a GRINDFN or GRINDMACRO. Whenever the pretty-printer encounters the function as the first element of a list, the list is printed using the special format. The grind-format can either be the name of a function of no inputs or the body of a lambda definition. A variety of predefined formats such as PROG-FORM are described in the next section. The mechanism for building new formats is presented in section III-E.

c. (UNFORMAT <function>) removes any special GRINDFN or GRINDMACRO properties of the function.

For all of the above specifications, <function> can be replaced by <list of functions>. The grind specification is then applied to each function in the list.

Typically, format statements are either placed in a "GRIND (INIT)" file read by the grind package when loaded; or inserted directly into the user's file as

```
;;*(GRINDFN THPROG PROG-FORM) (PROGRAMSPACE 80) <cr>.
```

Comments beginning with ";;*" cause the pretty-printer to evaluate the remainder of the line. If the line consists of only a single s-expression, the toplevel parentheses are optional.

```
;;*GRINDFN THPROG PROG-FORM
```

The normal LISP READ-EVAL-PRINT loop ignores semi-colon comments. Hence, ;;* comments only have effect when the file is ground.

4. REMGRIND - fexpr

(REMGRIND) removes all of the grind package's functions from a user's LISP. Alternatively, the user can be more selective in pruning the space occupied by the grind package by erasing only those features he does not need. This is done as follows:

> (REMGRIND FILE)- erases GRIND and GRINDO. Useful when only GRINDEF is needed.
>
> (REMGRIND UCONTROL) - erases the formatting functions. It does not erase those special formats already defined by the user. But it prevents him from defining any more. Useful after the user has created his special formats.
>
> (REMGRIND FORMAT) - erases both the formatting functions as well as any all special formats.
>
> (REMGRIND SEMI) - erases special functions for handling semi-colon comments.

5. Functions, atoms and properties reserved by grind.

The functions and atoms reserved by grind can be found in the DECLARE statement in the grind file. The grind package also uses the indicators "GRINDFN" and "GRINDMACRO" for specifying special grind formats.

B. Predefined formats

1. Standard formats

The following formats are used by the pretty-printer in the absence of any special formatting instructions. Choice depends on the avaliable width and the cost in number of lines. The algorithm is described in section II.

a. LINEAR-FORM - The expression is printed with no extra insertion of carriage-returns and spaces. this is the format used by the LISP printing primitives. It is used by GRIND only when there is sufficient width remaining on the line.

b. STANDARD-FORM - This is the preferred format for lists beginning with atomic functions. It is also used on other lists if fewer lines are needed to print the code this way.

```
        (<function> <pretty-print of arg(1)>
                    <pretty-print of arg(2)>
```

```
                        .
                        .
                  <pretty-print of arg(2)>)
```

c. MISER-FORM - This format conserves the space remaining on the line. When in width trouble, function lists are printed this way.

```
        (<pretty-print of element(1)>
         <pretty-print of element(2)>
                                .
                                .
         <pretty-print of element(n)>)
```

d. FUNNY-FORM - Occasionaly, this format decreases the number of lines needed to print an expression. It is used whenever this is the case. If PREDICT is NIL, computation is saved by ignoring it.

```
  (<ELEMENT(1)> <ELEMENT(2)> ... <PRETTY-PRINT OF ELEMENT(N)>)
```

2. Special GRINDFNs

Each of the following grind-formats can be assigned to any function by:

```
        (GRINDFN <function> <grind-format>)
```

a. BLOCK-FORM - the entire expression is ground as text where the left margin follows the opening parenthesis of the expression. For example,

```
                (A B C D E F G
                 H I J K L M N
                 O P Q R S T U
                 V W X Y Z)
```

Typically, argument lists and planner patterns are ground as blocks.

b. DEF-FORM - Def-form is the standard format for grinding definitions. The "defun", function-name, indicators and argument list are always ground on the first line. The argument list is ground as a block. The remaining elements of the definition are ground as a "body", i. e. depending on their size, they are ground one under the other in :

i. either the space remaining on the line, e. g.

```
        (DEFUN FNNAME <ARGLIST GROUND AS BLOCK> ******
                                                ******
```

```
                                                    ******)
```

ii. in standard format, i. e. aligned under the function name:

```
(DEFUN FNNAME INDICATOR <ARGLIST GROUND AS BLOCK>
       ******
       ******
       ******)
```

iii. or in miser format, i. e. aligned under the defun:

```
(DEFUN FNNAME INDICATOR <ARGLIST GROUND AS BLOCK>
 ******
 ******
 ******)
```

c. LAMBDA-FORM - the LAMBDA and its arglist are ground on the first line. The arglist is ground as a block. The remaining elements of the LAMBDA are ground as a "body" i. e. depending on their size, and in order of preference,:

i. in either the space remaining on the line, e. g.

```
(LAMBDA <ARGLIST GROUND AS BLOCK> ******
                                  ******
                                  ******)
```

ii. in standard format:

```
(LAMBDA <ARGLIST GROUND AS BLOCK>
        ***********
        ***********)
```

iii. or in miser format:

```
(LAMBDA <ARGLIST GROUND AS BLOCK>
 *************
 *************)
```

d. PROG-FORM - This format used for PROG's is similar to LAMBDA-FORM, except that tags are unindented.

e. MEM-FORM - The first argument is ground as code. The remainder are also ground as code unless quoted, in which case, they are ground as a block. For example,

```
        (MEMBER X
                '(A B C D E F G H I J K L
                  M N O P Q R S T U V W X
                  Y Z))
```

By default, MEMQ, MEMBER, the MAP functions, and the ASSOC functions are ground in this format.

f. COMMENT-FORM - The CDR of the expression is ground as a block. For example,

```
        (COMMENT THIS IS A VERY LONG
                 COMMENT THAT TAKES
                 SEVERAL LINES)
```

COMMENT, REMOB and *FEXPR, *EXPR, *LEXPR, **ARRAY, SPECIAL and UNSPECIAL clauses of DECLARE's are ground in this format.

g. SETQ-FORM - Space permitting, variables and values are ground as pairs. For example,

```
                (SETQ A (PLUS 1 1)
                      B 0)
```

If there is insufficient space, standard or miser format is used.

3. Inverting read macros

QUOTE-type read macros can be inverted when pretty-printed.

```
                        reader                          grind
        <char> <expr> - - -> (function <expr>) - - -> <char> <expr>
```

This is accomplished via the READMACRO instruction:

```
        (READMACRO <function> <macro character or characters>)
```

The macro character is PRINC'ed and then the <expr> is pretty-printed. Two examples are:

```
            (READMACRO QUOTE /') & (READMACRO THV /$/?)
```

4. System packages

A package of special formats currently exists for MICRO-PLNR. To utilize them, place either (PLNR) in your GRIND (INIT) file or ;;*PLNR directly in your micro-plnr files.

C. Comments

Semi-colon comments are defined as a semi-colon followed by text and concluded by a carriage return. These comments can be inserted anywhere in an s-expression or appear alone at the top level. They are completely ignored by the LISP reader. The grind package pretty-prints these comments in several formats depending on whether the comment begins with 1, 2 or 3 semi-colons.

1. Single semi's

Comments beginning with a single semi-colon are printed to the right of the code. Sequences of single-semi's are merged. The code is normally ground in the first 70 spaces of the line (PROGRAMSPACE) while the single semi's are ground in the final 49 spaces (COMSPACE). GAP = 1 is the space between code and comments.

```
    ------programspace------   --gap--   -------comspace-----
              70                  1              49

    --------------------pagewidth = 120--------------------
```

These values can be altered, for example, by inserting the following comment into a file:

;;*(PAGEWIDTH 120 89 1 30)

This results in PROGRAMSPACE becoming 89, GAP 1 and COMSPACE 30.

For code that contains no single semi's, a PROGRAMSPACE of 80. is preferable.

2. Double semi's

These comments are printed as part of the code with the proper indentation. Sequences of double semi's are merged. at the top level, TOPWIDTH = PAGEWIDTH is used. Inside code, double semi's are limited to PROGRAMSPACE. To alter TOPWIDTH, execute:

(TOPWIDTH <newvalue>)

3. Triple semi's

";;;..." are similar to ";;..." with respect to indentation. However, they are otherwise not modified by grind. Spaces are not filled and sequences of comments are never merged. They are thus useful when the user desires his comment to be printed exactly as originally typed.

D. Grind control

These functions set various switches and variables for the pretty-printer.

1. FILL causes multiple spaces appearing in single and double semi's to be merged. Periods ending sentences are followed by two spaces. This is the default case.

2. NOFILL causes multiple spaces to be treated as such. Triple semi's are always NOFILL'ed.

3. MERGE causes double semi's to be merged, if sufficient COMSPACE remains on the line.

4. NOMERGE causes double semi's not to be merged. This is the manner in which triple semi's are handled. The full pagewidth is used.

5. PAGE causes the output of a formfeed.

6. FF causes grind to insert formfeeds approximately every 60 lines. Formfeeds are only inserted at the toplevel, never appearing within s-expressions. This is the default case.

7. NOFF limits the insertion of formfeeds to explicit calls of PAGE.

8. PPAGE causes grind to preserve original paging of user's file.

9. NOPREDICT - This switch makes the grind dumber but faster. The algorithm no longer consider as many alternatives for grinding each expression. For PROG-FORM and DEF-FORM, format 1 is no longer considered. Similarly, FUNNY-FORMAT is never considered. Dumb mode is the default state.

10. PREDICT - All of the formats discussed in the previous pages are considered.

11. PAGEWIDTH <pagewidth> <programspace> <gap> <commentspace>

12. PROGRAMSPACE <value> - resets the value of the PROGRAMSPACE. Enlarging PROGRAMSPACE shrinks COMSPACE.

13. COMSPACE <value> resets the width used for single semi comments. The tradeoff is again with the PROGRAMSPACE.

14. TOPWIDTH <value> - resets the width used for toplevel double semi comments.

E. Defining new formats

The user may wish to go beyond the predefined formats discussed in section III-B. To do this, GRINDFN can be used to define special grind functions [SGF's] of his own design. The syntax is as follows:

```
(GRINDFN <atom or list of atoms> <grind-format>)
```

where the definition is either the name of O-input procedure or the body of a LAMBDA expression.

GRINDFNs are processed as follows: assume the atom L1 has a SGF associated with it. Then, whenever expressions of the form (L1 ... LN) are encountered, grind prints "(" and then transfers control to the definition of the SGF. Upon entering the SGF, the following free variables are relevant:

```
L <——  (L1 ... LN)
N <—— CHRCT = remaining line width, following the "(".
```

A SGF generally processes some initial segment of L, CDR'ing L in the process. Note that the SGF must at least process L1. Upon completion, if L has been set to NIL, grind simply prints the closing parenthesis ")". If, on the other hand, L has been rebound to some terminal segment of itself,

L = (Li ... Ln)

then grind prints the remainder of L as the body of a DEF-FORM, i. e. the elements of L are printed one under the other in either

a. the space remaining on the line
b. aligned under L2
or c. aligned under L1.

2. Vocabulary

The following vocabulary is useful for defining SGF's:

1. (REMSEMI) - expr - This function processes any ; comments that occur as initial elements of L, CDR'ing L in the process.

2. (PPRIN S F) - expr - S is printed in the format specified by F where F can be:

'LINE - equivalent to PRIN1
'BLOCK - BLOCK-FORM

```
        'LIST - COMMENT-FORM
        'CODE - applies pretty-printer to S.
```

PPRIN should not be given ; comments as input. (REMSEMI) is generally used to avoid this. PPRIN does not print a space following S.

3. (FORM F) - expr - This function is designed to relieve the user of an explicit concern for comments. It also frees him from printing spaces between elements of L. Its definition is:

```
        (REMSEMI)
        (PPRIN (CAR L) F)
        (AND (SETQ L (CDR L)) (PRINC '/ ))
```

Its action is to first apply REMSEMI, removing any initial comments from L. It then pretty-prints (CAR L) in the specified format F. Finally it CDR's L and prints a space if there is still more to go.

4. (TURPRI) - expr - A carriage return is printed. TERPRI should not be used.

5. (INDENT-TO N) - expr - This function causes CHRCT to be set to N by printing a carriage return if necessary (N > CHRCT) and spaces. Note that CHRCT is the current width. This number is equal to the indentation subtracted from the total line width. A common bug is to treat N as the indentation.

6. (INDENT M) - expr - M spaces are printed. An error results if M exceeds the space remaining on the line.

7. (POPL) - expr - L is set to (CDR L). Then REMSEMI is applied. The net result is to CDR L until its CAR is not a comment.

8 a. (TESTL) - lexpr - returns the first element of L that IS NOT a ";" comment.
  b. (TESTL j) returns the jth element of L that is not a comment.
  c. (TESTL j t) returns the entire remainder of L beginning WITH the jth element.

9. (SEMI? K) - expr - returns T only if K is a semi-colon comment.

3. EXAMPLES

Following are some examples of SGF's. LAMBDA'S are ground by default in DEF-FORM. The user could achieve the same effect by defining the following SGF:

```
        1       (GRINDFN LAMBDA (FORM 'LINE)
        2                       (FORM 'BLOCK))
```

(FORM 'LINE) in line 1 prints LAMBDA and pops L. (FORM 'BLOCK) in line 2 prints the argument list of the LAMBDA in BLOCK-FORM and again pops L. Control is then returned to grind and the remainder of the LAMBDA is printed as a body.

Another example might be where the user wishes to grind all expressions of the form:

```
   (DEFPROP <ATOM> <DEFINITION> <EXPR, FEXPR OR MACRO>)
```

as DEFUN's. This would be done by:

```
 1 (GRINDFN DEFPROP
 2          (COND ((MEMQ (TESTL 4) '(EXPR FEXPR MACRO))
 3                 (SETQ L
 4                       (APPEND (LIST 'DEFUN (TESTL 2))
 5                               (COND ((EQ (TESTL 4) 'EXPR)
 6                                      NIL)
 7                                     ((LIST (TESTL 4))))
 8                                (CDR (TESTL 3))))
 9                 (DEF-FORM))
10                ((FORM LINE))))
```

The MEMQ of line 2 checks for whether the indicator is a function property. If so, L is redefined as the appropriate DEFUN:

```
   (CADR L) = function name
   The cond of line 5 puts fexpr/macro into the DEFUN
   (CDR (CADDR L)) is the argument list of the function
   (CDDR (CADDR L)) is the body of the function
```

and then ground in DEF-FORM. If not, DEFPROP is printed and control is returned to grind.

Finally, consider a function called CMEANS whose arguments are property lists. It is to be ground as follows:

```
                  (CMEANS
                     (<IND-11> <GRIND PROP-11>
                                    .
                      <IND-1N> <GRIND PROP-1N>)
                                    .
                                    .
                     (<IND-M1> <GRIND PROP-M1>
                                    .
                      <IND-MN> <GRIND PROP-MN>))
```

Suppose the additional subtlety is desired that properties with indicator FOO are ground as blocks while all other properties are ground ordinarily as code. The following SGF achieves this format.

```
 (GRINDFN CMEANS (PROG NIL
1                    (FORM 'LINE)
2                    (SETQ N (*DIF N 4.))
3                    (REMSEMI)
4               A    ((LAMBDA (L)
5                             (PROG NIL
6                                   (INDENT-TO (ADD1 N))
7                                   (PRINC '/()
8                              B    (REMSEMI)
9                                   (INDENT-TO N)
10                                  (COND ((EQ (CAR L) 'FOO)
11                                         (FORM 'LINE)
12                                         (FORM 'BLOCK))
13                                        ((FORM 'LINE)
14                                         (FORM 'CODE)))
15                                  (AND (TESTL) (GO B))
16                                  (PRINC '/))
17                                  (REMSEMI)))
18                    (CAR L))
19                   (COND ((POPL) (GO A)))))
```

Line 1 prints "CMEANS". Line 2 establishes the indentation of the arguments of CMEANS. Line 3 processes any comments preceding the first argument. Line 4 binds the special free variable L to the current argument of CMEANS for use by FORM and REM. Line 6 indents for the current argument. Line 8 processes any initial comments embedded in the argument. The cond of line 10 forks depending on whether or not the indicator is "FOO". In line 15, TESTL returns NIL if L contains no more indicator-property pairs. Line 16 prints the closing parenthesis. 17 processes any remaining comments. By line 19, the current argument of CMEANS has been ground. Hence, L is popped. If there are no more arguments, POPL returns NIL and the SGF is done.

4. GRINDMACROs

A GRINDMACRO differs from the above grindfunctions in that the grind package takes nothing for granted. It does not automatically print the opening parenthesis, the balance of L and the closing parenthesis. If the GRINDMACRO function returns T, then the pretty-printer does nothing more on L. The assumption is that the GRINDMACRO has done all the work. This would be the case for a GRINDMACRO for "QUOTE":

```
(GRINDMACRO QUOTE (PRINC '/')
                  (PPRIN (CADR L) 'CODE)
                  T)
```

Alternatively, if the GRINDMACRO returns NIL, the pretty-printer prints L as though nothing had happened. This mode is useful for a GRINDMACRO used to print "index" information as comments preceding the s-expression.

GRINDMACROs can be defined similarly to GRINDFNs.

```
(GRINDMACRO <ATOM or LIST OF ATOMS> <grind-format>)
```

Again the definition can be either the body of a LAMBDA or a function of 0 inputs.

VI. Possible future improvements

A. CONCEPTUAL

1. The language for specifying formats should be expanded. A pattern-oriented or template approach might be preferable.

2. The table scheme would allow the pretty-printer to consider indentations for the arguements of a function, intermediate between miser and standard format. The algorithm could choose the greatest indentation that does not cause extra lines to be printed.

3. Pretty-printers could do more than just insert spaces and carriage returns. For example, FACTORIAL could be printed as follows:

```
        (DEFUN FACTORIAL (X) (COND ((= X 0) 1) (==>)))

        (* (FACTORIAL (1- X)) X)
```

"==>" is interpreted by the reader to mean that the next expression READ should be inserted here. This suggestion is due to MINSKY.

4. The fact that comments are not read in as part of the list structure presents a serious obstacle to interactive debugging. The user must return to a text editing language to make corrections in his code. Otherwise, he loses any commentary. One possible solution would be for comments to be resident. Paging could be used to store all comments on the same page. This would allow them to be swapped out during runtime. The evaluator would have to be modified to ignore pointers to a comment page. These modifications are probably well worth the effort. The user would be able to move continuously between defining, running and editing programs.

B. IMPLEMENTATION

1. Grind should accept a wider variety of TJ6'like specifications. for example,

```
        ;*DASH            --->   Line of dashes
        ;*CENTER <text>   --->   Centers text in comment
```

2. The current scheme for ; comments leads to enormous list structures since every comment is expanded to a list, one letter per node. Alternatives to this approach are:

   A. TYI rather than readch.
   B. Pack ascii characters into pnames or array.
   C. Use read to pack by turning off syntax of parentheses, Periods, commas.

3. GRIND should use its own obarray and readtable to minimize interference with the user's world.

4. Special grind formats can check for the correct number of inputs.

5. Grind could print page numbers

6. The grind file should be broken into 2 files. the first contains the basic grind. The second contains the fns for the user to define his own formats. This decreases the initial load on free storage when reading in the basic grind.

7. Special formats should be created for DO, CNVR and LAP code.