To: Distribution

From: Monte Davidoff

Date: 6 February 1980

Subject: Automatic formatting of PL/I programs

INTRODUCTION

This MTB describes a command called format_pl1 which automatically formats PL/I programs. Each user may provide formatting controls to tailor the formatting according to his personal style. Control comments may be inserted in a program that indicate what formatting style should be used. This allows the style of a program's author to be preserved even if developers with different styles occasionally modify the program.

PURPOSE

Using a program to format a PL/I program automatically is more productive than doing it by hand. A program can do the job much faster and more accurately. Automatic formatting visually displays the nesting of control structures in the program as the compiler will interpret them. Control structure nesting errors, such as dangling else problems, become very easy to spot. Some common programming errors, such as unbalanced quotes, unbalanced parentheses and mismatched end statements, are detected using fewer resources than a compilation would take.

The indent command is not flexible. No one style is adequate for all the different programmer preferences and ways of writing PL/I. If one prefers some aspects of indent but not others, there is no way to turn off the undesirable features. The indent command does not understand very much about PL/I. This causes many bugs and deficiencies in indent. There are no plans to fix any of these. The heavy use of PL/I on Multics justifies a more intelligent formatting program.

Therefore, a new formatting program was written. The differences between format_pl1, in style4, and the indent command are minor, for the most part, and all of the known differences are felt to be bugs in indent. After format_pl1 has been installed for a release or two, the indent command should be made obsolete.

If a programmer inherits a subsystem which he considers unreadable, he will have the option of quickly and mechanically reformatting the entire subsystem into a style he finds more productive. If someone in the field such as a SiteSA changes a program, he will be able to figure out what the style is and mechanically preserve the style.

MCR 3503, which was approved on 25 October 1978, addressed the need for formatting standards for installed PL/I programs. Many of the reasons for a formatting standard are the same as those stated above for using an automatic formatting program. Much time has been spent in the past trying to arrive at a single formatting style acceptable to most programmers. All attempts to arrive at a single standard style have failed. The complexity and wide variety of formatting controls available in format_pl1 indicates just how diverse programmers' styles are. The standard proposed by MCR 3503 requires an installed formatting command that allows for a variety of styles. By default, the command would not make any irreversible changes. The default style could be changed by command line arguments or formatting control comments in the source. The proposed standard is to format all installed PL/I source by the installed standard formatting command. Formatting control comments in the source would specify how the program's style differed from the the default action of the command. The auditor would insure that the style chosen by the author was readable.

The proposed format_pl1 command is intended to become the installed standard formatting command. An MCR to install format_pl1 will not automatically make it the installed standard formatting command. A separate MCR must be approved to make it the standard. No program can implement the style of any programmer perfectly. PL/I is too complicated a language. Sometimes the same code needs to be formatted differently for different purposes. The format_pl1 command does not even implement exactly the style of its author. In order to get the benefit of an automatic formatting program and the benefit of a formatting standard, we must agree on a reasonable, small range of styles and exercise good judgement to avoid styles that most would find difficult to read.

One purpose of this MTB is to find out how close format_pl1 is to satisfying the needs of Multics developers. Disagreements will be settled by consensus at a format_pl1 design review. Speak up if format_pl1 is incompatible with an important part of your style. Actually, any standard imposed by format_pl1 is not very tight. It is possible to specify a style that does not change the format of a program at all. One programmer has said this safety valve should be closed. I prefer to leave in this option and let auditors enforce standards, rather than programs.

Once format_pl1 is approved as the standard formatting command, it should be phased in gradually. A conservative approach should be used to decrease the chance of accidentally making irreversible changes. It would not be wise to immediately format all installed sources with any style. Programs should be converted from their present hand-formatted style or indent's style as they are opened and changed, at the discretion of the programmers and the auditors. Each programmer can then go at his own pace, converting to format_pl1 on a program by program basis or a whole subsystem at time.

HISTORY

The indent command was written by Don Widrig and Stan Dunten in MAD on CTSS in 1966. Its formatting style dates back to its authors' understanding of PL/I at that time. Its emphasis on speed reflects the constraints of its development environment. It was converted to PL/I for Multics in June 1969 by Tom Van Vleck.

Complete disgust with the style of the indent command motivated format_pl1. Paul Green started writing the first version in November 1977. From the start, one goal of format_pl1 was to understand the syntax of PL/I better than indent. This means format_pl1 will not get confused by constructs that confuse the indent command. It also means format_pl1 can do more processing since it has the knowledge.

The author started working on the command in July 1978. The style in which the command formatted programs gradually improved. A staff meeting talk was given by the author at CISL in September 1978 on the new formatting command. The need for the command to support a variety of styles was the main result of the somewhat heated discussion at the meeting. Soon afterwards, Tom Casey submitted MCR 3503 to ensure that a new command, implementing an incompatible formatting style, would not become a new de facto standard.

Since that time, work on format_pl1 has centered around producing a specification of what the command should do and implementing facilities to allow multiple styles. Why was format_pl1 implemented first, before the MTB and Design Review, at variance with the usual procedures of the Multics group? The problem of formatting PL/I programs is too complicated and emotional an issue to trust to a written specification. PL/I is a very complicated language. Nearly everyone has exceptions to their own formatting guidelines. The evolution of the current format_pl1 command can be described as recognizing more and more special cases that have to be treated differently. To have a specification and ask if this is what you want would not have worked. There are just too many special cases. The only way to

approach the problem is to ask people what they don't like. I want people to know what they are getting. It's better to talk about specifics, rather than "have a style like indent."

THE PHILOSOPHY BEHIND FORMAT PL/I

By default, format_pl1 will not make any drastic changes in a program. For example, if a program is formatted with indent, then formatted by format_pl1 using its default style or the style similar to indent, and then reformatted with indent, the result will be very close to the original. The format_pl1 command can make drastic changes in the source if it is asked to. It can delete intrastatement vertical white space to ensure as much of a statement will fit on a line as possible, and it can insert newlines into the source if a statement is too long for a line or if there is more than one statement on a line. Its algorithm for inserting newlines produces good results most of the time. These formatting modes can save a lot of time if you like what they do. If only a few places in the program look unacceptable when format_pl1 deletes and inserts newlines, you can insert control comments in those places to prevent format_pl1 from inserting and deleting newlines in those places.

As mentioned earlier, control comments in the program can specify the style to format the program. Styles specified in the program always override those specified on the command line. This very conservative approach was taken to ensure that a program's style will not be accidentally changed. This prevents format_pl1 from doing what the author of a program may consider to be irreparable damage. If you really want to change its style, you have to edit the program.

MCR 3503 influenced the development of format_pl1. The control comment mechanism, a default style which makes no irreversible changes and the interaction of styles specified on the command line and specified in the program are consistent with the philosophy of MCR 3503. The issues raised by MCR 3503 contributed to the conservative approach taken by format_pl1.

The predefined styles in format_pl1 are designated by numbers. There is no way to pick a mnemonic identifier to describe a style other than something like the name of the person whose style it is. There is no "style_indent" because the format_pl1 style closest to indent, style4, is not exactly like the indent command. Most programmers consider the minor differences to be bug fixes.

USING FORMAT PL/I

I urge everyone to try format_pl1 using their favorite style. The current version of the command is in >udd>m>mnd>lib on all three development sites. Info segments describing the current version are in >udd>m>mnd>info. Also try the command using the delnl,insnl modes. You may find that the time it will save editing will be worth it.

Since there is so much disagreement about how a program should be formatted, the default style will please very few people. Since control comments in the program override the default style and the style specified on the command line, the choice of a default is not important. As a result, format_pl1 has been designed to be used with abbrev. Everyone should have an abbrev similar to:

```
.ab FP format_pl1 -record_style -modes YOUR_STYLE
```

or:

```
.ab FP format_pl1 -version -record_style -modes YOUR_STYLE
```

If you have been using the indent command and like its style, just substitute "style4" for YOUR_STYLE.

The -record_style control argument tells format_pl1 to insert a style control comment in the program indicating how to format the program if the program does not already contain such a control comment. Presumably, the program does not contain a style control comment because it was just typed in or because it predates format_pl1. This control comment is inserted after the initial comments in the program and before the first token in the program so it will not interfere with copyright comments. The modes string in the control comment specifies every mode so that changes to the default style or different styles specified on the command line won't have any effect. The modes string uses the closest predefined style so a minimum number of modes are specified.

If the -version control argument is specified, format_pl1 will identify itself after it has checked its arguments. This is done in a manner similar to other translators such as PL/I.

If you do not want to inadvertently change the style of a program without a style control comment, then your abbrev should include the -require_style_comment control argument. This will cause an error message to be printed if the program does not already contain a style control comment. If only one pathname was specified on the command line, the source will not be overwritten. You can then decide if you really want to format the program with format_pl1, and if so, specify the -no_require_style_comment control argument.

The format_pl1 command has a few features designed to make common programming mistakes easier to find and correct. Parentheses are checked to be sure that they balance. If you find you have omitted a quote, the -check_strings control argument can help find the character string containing half of your program. If a labeled end statement closes more than one block or group, format_pl1 will warn you. It will tell you each block or group that is closed by the labeled end statement, except of course, the one it should close. The presence of this warning means that labeled end statements should be used whenever possible. If you leave out an end statement, you will get a message saying which block or group is missing an end statement, rather than the message from PL/I saying you left out one end statement in your 17,000 line program.

The MPM documentation for format_pl1 indicates that the severity active function can be used with the "format_pl1" keyword. The severity active function must be changed to recognize this keyword before it will work as documented below.

MPM DOCUMENTATION

Name: format_pl1, fp

Syntax: fp in_path {out_path} {-control_args}

Function: formats a PL/I, create_data_segment or reduction_compiler source segment to make it more readable. Alternate methods of formatting particular language constructs are selected by means of modes; several popular styles (consisting of groups of modes) are defined. Modes and styles are specified on the command line and in comments in the source segment.

Arguments:

in_path
: pathname of source segment. Suffixes for PL/I, create_data_segment and the reduction_compiler are recognized. If in_path does not have a recognized suffix, format_pl1 attempts to use in_path.pl1 or in_path.cds, in that order.

out_path
: pathname of the formatted source segment. The suffix of in_path is assumed if not given. If omitted and there were no errors, in_path is overwritten. If omitted and there were errors, a formatted copy is left in the process directory.

Control arguments:
-version, -ver
   prints the version of format_pl1.
-no_version, -nver
   doesn't print the version of format_pl1. (Default)
-modes STR
   prefixes modes found in the program by modes string STR. See the "Prevailing style" section below.
-record_style, -recsyl
   puts a control comment in the source specifying the prevailing style if the source does not already have a prevailing style control comment. The comment is placed immediately before the first token of the program so it doesn't interfere with copyright notices.
-no_record_style, -nrecsyl
   don't put a control comment in the source specifying the prevailing style. (Default)

-check_strings, -ckstr
   print a warning if a character-string constant contains vertical white space. This control argument is useful after receiving an error message indicating a quote has been omitted from a character-string constant.
-no_check_strings, -nckstr
   don't print a warning if a character-string constant contains vertical white space. (Default)
-require_style_comment, -reqsylcom
   print an error message if the source does not already contain a prevailing style control comment. This is useful if one is concerned with accidentally destroying the style of a hand-formatted program.
-no_require_style_comment, -nreqsylcom
   format the source even if it does not already contain a prevailing style control comment. (Default)

Modes string:
A modes string changes the style format_pl1 uses to format a program. It consists of mode names separated by commas. Many modes can be preceded by "^" to turn the specified mode off. The modes string is processed from left to right. Thus, if two or more contradictory modes appear within the same modes string, the rightmost mode prevails. Modes not specified by the modes string are left unchanged.

Control comments:
A control comment has the form "/* format: STR */" where STR is a modes string. Control comments may occur only before the first token of the program, between a semicolon and the first token of the next statement or after the last semicolon in the program.

Control comments may not occur in the middle of a statement. Optional horizontal white space may precede "format:" or surround STR. Some modes changed by a control comment may not take effect immediately. For example, end statements are formatted according the modes in effect when the matching do, begin or procedure statement was formatted.

There are two special control comments that are used in if statements. If a comment containing "/* case */" or "/* tree */" immediately follows the word "if" in an if statement, then the current style is changed for the duration of that if statement. Exactly one space must precede and follow "case" and "tree". See the description of the case and tree modes in the "List of if statement modes" section below.

Prevailing style:
The style in which format_pl1 formats a PL/I program is formed from a combination of three sources: format_pl1's default style, modes specified on the command line and control comments in the program. The first control comment of the program preceding the first token of the program is called the prevailing style control comment. A program might not have a prevailing style control comment. The style specified by the concatenation of the default style, the command line modes and the prevailing style control comment is called the prevailing style. This is the style in which most of the program is formatted.

Note that since a styleN mode specifies the setting of every possible mode, if the prevailing style control comment contains a styleN mode, the default format_pl1 style and the command line modes are ignored. If the program does not already have a prevailing style control comment, the command line:

```
format_pl1 in_path -modes MY_STYLE -record_style
```

formats a program in MY_STYLE, and records the style in a prevailing style control comment. If the program had a prevailing style control comment, the program is formatted in the style specified by its prevailing style control comment, and the -record_style control argument has no effect. The prevailing style control comment created as a result of the -record_style control argument always begins with a styleN mode.

Notes on examples:
The examples show how various program fragments are formatted. If a control comment is not given, then style1, the default style, is assumed. If a control comment is given, the default is used for all unspecified modes. Unless delnl,insnl mode is being

used, each line of the input source segment contains the same tokens as the corresponding line of the example. If delnl,insnl mode is being used, then newline characters are inserted and deleted as required by the style.

List of modes: (Modes for various language constructs are listed separately.)

styleN
specifies formatting style N. See "Styles" section below.

revert
changes the formatting style to the prevailing style. This mode may not be specified in the -modes control argument's modes string or in the prevailing style control comment. Note that the on mode is changed to the phase specified in the prevailing style.

off, ^on
leave the source exactly as it is until a control comment changes the style to on. When format_pl1 is in the off mode, block and group entries and exits are noticed so the program following the on mode control comment is formatted correctly.

on, ^off
start formatting the source again. (Default)

indN
N is the number of columns to indent for each block or group indentation level. An independent statement in a then or else clause that does not have a condition or label prefix is indented a minimum of five columns even if indN is less than five. This avoids placing the then clause or else clause on the line after the "then" or "else". The five columns are measured from the column the "else" would start in if the else clause was an independent statement. (Default 5)

Example:
```
/* format: ind3 */
if v = 2
then
   do;
      x = 12;
      y = 128;
   end;
else z = 12;
```

llN
N is the output line length. (Default 126)

initlmN
N is the initial column that statements occuring before the first procedure statement should start at. This is most useful for include files. (Default 6)

Declare statements:
Depending upon delnl and insnl modes, each level one identifier

that is declared is placed on a line by itself. In indattr mode, all attributes are indented to the same column. If a declaration list has all the attributes factored, i.e. each declaration component in the declaration list consists only of an identifier, the declaration list doesn't contain any comments and none of the identifiers contain a dollar sign, then the declaration list is placed on as few lines as possible instead of placing each identifier on a separate line. To put it another way, if the parenthesized list contains only identifiers and doesn't contain any attributes or comments, the parenthesized list is placed on as few lines as possible.
Example: declare (hbound, index, null) builtin;

List of declare statement modes:
indattr
always indent the attributes so they start in the same column. (Default)
^indattr
don't indent the attributes from the identifier being declared.
inddcls
indent declare statements so they start in the same column any other statement would start in. (Default)
^inddcls
always start declare statements in column 1.
declareindN
indent N columns after "declare". (Default 8)
dclindN
indent N columns after "dcl". (Default 8)
idindN
indent N columns after the start of an identifier before starting the attributes. Ignored if in ^indattr mode. (Default 23)
struclvlindN
indent N columns for each level in a structure. (Default 2)

List of if statement modes:
ifthenstmt
if the if statement meets certain criteria, put the then clause on the same line as the "if" if it fits. The criteria are: The then clause must be an independent statement and cannot be another if statement. The then clause must not have a condition or label prefix. If in tree mode, the if statement must not have an else clause. If in case mode, the if statement must fall into one of the following categories: there is no else clause, the else clause consists of an if statement, or the if statement under consideration is an else clause of another if statement.
Example: /* format: ifthenstmt */
if x > 3 then return;

^ifthenstmt
don't put the then clause on the same line as the "if". (Default)
Example:
```
if x > 3
then return;
```

ifthendo
if the then clause of an if statement is a noniterative do group without a condition or label prefix, then put the "then do" on the same line as the "if" if it fits. If the else clause of an if statement is a noniterative do group without a condition or label prefix, then put the "else do" on the same line if it fits even if indN is less than five. In ^delnl mode, the "then" or the "else" must already be on the same line as the "do".
Example:
```
/* format: ifthendo,^indnoniterdo */
if v = 2 then do;
     x = 8;
     y = 9;
end;
else do;
     x = 9;
     y = 92;
end;

/* format: ind3,ifthendo,^indnoniterdo */
if v = 2 then do;
   x = 8;
   y = 9;
end;
else do;
   x = 9;
   y = 92;
end;
```

^ifthendo
don't put the "then do" on the same line as the "if". (Default)
Example:
```
/* format: ^indnoniterdo */
if v = 2
then do;
     x = 8;
     y = 9;
end;
else do;
     x = 9;
     y = 92;
end;
```

ifthen
put the "then" on the same line as the "if".

```
   Example:  /* format: ind3,ifthen */
             if v = 2 then
                do;
                   x = 12;
                   y = 128;
                end;
             else
                do;
                   x = 128;
                   y = 12;
                end;
```

^ifthen
line the "then" up with the "if". (Default)

```
   Example:  if v = 2
             then x = 8;
             else x = 9;
```

indnoniterdo
if a then or else clause contains a noniterative do group, then start the statements of the do group two indentation levels from the column in which the "if" starts. Indent three indentation levels instead of two if in indthenelse mode. (Default)

```
   Example:  if v = 2
             then do;
                       x = 3;
                       y = 4;
                  end;
             else do;
                       x = 35;
                       y = 27;
                  end;
```

^indnoniterdo
if a then or else clause contains a noniterative do group without a condition or label prefix, then start the statements of the do group one indentation level from the column in which the "if" starts. Indent two indentation levels instead of one if in indthenelse mode.

```
   Example:  /* format: ^indnoniterdo */
             if v = 2
             then do;
                  x = 3;
                  y = 4;
             end;
             else do;
                  x = 35;
                  y = 27;
             end;
```

indend
if a then or else clause contains a noniterative do group without a condition or label prefix, then start the end statement of the noniterative do group in the same column as the statements of the noniterative do group.
Example:
```
/* format: ^indnoniterdo,indend */
if v = 2
then do;
     x = 8;
     y = 9;
     end;
else do;
     x = 9;
     y = 92;
     end;
```

^indend
if a then or else clause contains a noniterative do group, then start the end statement of the noniterative do group in the column that is one indentation level before the column the statements of the noniterative do group start in. (Default)
Example:
```
/* format: ^indnoniterdo */
if v = 2
then do;
     x = 8;
     y = 9;
end;
else do;
     x = 9;
     y = 92;
end;
```

indthenelse
start the then and else clauses two indentation levels from the column in which the "if" is placed. Place the "else" one indentation level from the column in which the "if" is started. If in ^ifthen mode and the ifthenstmt and ifthendo modes do not apply to the if statement, place the "then" in the same column as the "else". If in case mode and the if statement under consideration is the else clause of another if statement, then indent from the column in which the preceding "else" is placed instead of the column in which the "if" is placed. In case mode, this mode is ignored for the else clause if the else clause consists of an if statement or the if statement under consideration is an else clause of another if statement.
Example:
```
/* format: indthenelse */
if v = 2
     then x = 8;
     else do;
               x = 9;
```

```
                         call default;
                    end;

          /* format: indthenelse */
          if v = 2
               then x = 8;
          else if v = 3
               then x = 25;
          else call error;
```

^indthenelse
start the then and else clauses one indentation level from the column in which the "if" is placed. Place the "else" in the same column as the "if" is placed. If in ^ifthen mode and the ifthenstmt and ifthendo modes do not apply to the if statement, place the "then" in the same column as the "else". If in case mode and the if statement under consideration is the else clause of another if statement, then indent from the column in which the preceding "else" is placed instead of the column in which the "if" is placed. (Default)
Example:

```
          if v = 2
          then x = 8;
          else do;
                    x = 9;
                    call default;
               end;

          if v = 2
          then x = 8;
          else if v = 3
          then x = 25;
          else call error;
```

case, ^tree
indents "else if" clauses like a case statement. (Default)
Example:

```
          if char = "a"
          then call char_a;
          else if char = "b"
          then call char_b;
          else if char = "c"
          then call char_c;
          else call error;

          /* format: ifthenstmt */
          if char = "a" then call char_a;
          else if char = "b" then call char_b;
          else if char = "c" then call char_c;
          else call error;

          /* Decision tree formatted like a case statement. */
          if condition_1
          then if condition_2
```

```
                    then call condition (0);
                    else call condition (1);
               else if condition_2
               then call condition (2);
               else call condition (3);
tree, ^case
   indents "else if" clauses like a decision tree.
   Example:  if /* tree */ condition_1
             then if condition_2
                  then call condition (0);
                  else call condition (1);
             else if condition_2
                  then call condition (2);
                  else call condition (3);

             /* Case statement formatted like a decision tree. */
             /* format: tree */
             if char = "a"
             then call char_a;
             else if char = "b"
                  then call char_b;
                  else if char = "c"
                       then call char_c;
                       else call error;
```

Horizontal white space:
All horizontal white space, except within character-string constants and comments, is removed from the program. Spaces are inserted before left parentheses, after commas, around operators and in other places to improve readability. Where possible, horizontal tabs are used to conserve space in the output segment.

Statements continued onto another line are indented *indN* from the current left margin. The left margin at which a statement is indented is increased by indN for every nested begin block, group and then or else clause. Procedure and entry statements are always placed in column indN+1. The left margin before a procedure statement is saved; it is restored after the procedure's end statement. The left margin after a procedure statement is reset to 2*indN. End statements are started in the same column as the statement which began the block or group, except as required by indend mode. Condition and label prefixes are placed on lines by themselves, except possibly in ^insnl mode.

Vertical white space:
Vertical white space within character-string constants and comments is never changed. Other vertical white space is divided into two categories: intrastatement and interstatement vertical

white space. In on mode, vertical tabs and newlines before newpages are removed, newlines before vertical tabs are removed, and multiple newpages are reduced to one. A newline is inserted before and after a sequence of vertical tabs and newpages if there is not already one there. Interstatement vertical white space is never changed except for the above canonicalizations. Intrastatement vertical white space is also canonicalized as above and processed depending upon delnl and insnl modes.

List of vertical white space modes:
delnl
deletes all existing intrastatement vertical white space.
^delnl
leaves existing intrastatement vertical white space in the program. (Default)
insnl
insert newlines in the program if necessary. Newlines are inserted when statements are too long to fit on a line. Various heuristics are used to determine where newlines are inserted. The heuristics use the statement type and the precedence of the tokens in the statement to determine where to insert newlines. The driving force of format_pl1 is what column statements or other language constructs should start in. Newlines are inserted to start a statement, subset of a statement or a comment in a particular column.
^insnl
don't insert newlines into the program. (Default)

Comments:
Comments are classified by where they occur within a PL/I program and where they are placed in the output segment. They are divided into three categories: intrastatement comments, indented comments and column one comments. Intrastatement comments occur between the first token of a statement and the semicolon ending the statement. They are placed according to linecom mode. Comments that follow a semicolon and are separated by at most one newline character are considered indented comments. They are placed in column comcolN. All other comments are column one comments. They are started in column one. All comments before the first token of the program, all comments following a blank line and all comments following a column one comment are column one comments. Placing a comment in column N means that the "/*" starts in column N.

In certain special cases, intrastatement comments are treated as indented comments and placed in column comcolN. These cases are: comments following a comma, comments preceding the right parenthesis of a declaration component, comments following the colon in a condition or label prefix and in if statements that

the ifthenstmt or ifthendo modes do not apply to, comments following the "then" in ifthen mode and comments preceding the "then" in ^ifthen mode.

List of comment modes:

comcolN
N is the column comments are placed at if they are not placed in column 1 and the comment does not occur within a statement. (Default 61)

linecom
intrastatement comments at the end of a line in the original source segment apply to an entire line. These comments are treated as indented comments and are placed in column comcolN.
Example:
```
/* format: linecom */
if line_status < 3                    /* Is line active? */
     | char_count > 0
then return;
```

^linecom
intrastatement comments apply to the preceding token. (Default)
Example:
```
/* format: ^delnl */
if char = "040"b3 /* space */
     | char_count > 0
then return;
```

indcomtxt
if there is no horizontal or vertical white space between the "/*" and the comment text or between the end of the comment text and the "*/", then insert a space. Indent the text of continuation lines of a multiline comment so they line up. Indenting the text of continuation lines does not apply to intrastatement comments. The horizontal white space between the "/*" and the comment text on the first line of a comment is not reduced, however, leading horizontal white space on subsequent lines is replaced by sufficient horizontal white space to indent the line. If the comment is placed in column N, the text of each line of the comment begins in column N+3.
Example:
Input:
```
/* format: indcomtxt */
a = 3; /* Here we have a very
complicated assignment
statement. */
```

Output:
```
/* format: indcomtxt */
a = 3;                    /* Here we have a very
                             complicated assignment
                             statement. */
```

^indcomtxt
leave the white space at the beginning of each line of a comment alone. The character-string between the "/*" and the "*/" of a comment is never changed in this mode. (Default)

Styles:
style1: on,ind5,ll126,initlm6,indattr,inddcls,declareind8, dclind8,idind23,struclvlind2,^ifthenstmt,^ifthendo, ^ifthen,indnoniterdo,^indend,^indthenelse,case,^delnl, ^insnl,comcol61,^linecom,^indcomtxt (Default)

style2: style1,delnl,insnl
style3: style2,^inddcls,declareind10,dclind10,idind20
style4: style1,^indattr,^inddcls,declareind9,dclind5,ifthendo, ^indnoniterdo,linecom,indcomtxt

Irreversible changes:
Several modes can cause irreversible changes to be made to the source program. Suppose that program p.pl1 was formatted with style S and does not contain a prevailing style control comment. Then style T causes an irreversible change if the following command line produces a program q.pl1 that differs substantially from p.pl1.

```
format_pl1 p q -modes T; format_pl1 q -modes S
```

The following modes can cause irreversible changes: delnl, insnl, ^linecom,delnl and indcomtxt. If a program was not formatted with format_pl1, the on mode may also cause irreversible changes.

Style summaries:
Style1, the default style, indents declare statements, indents the attributes of declare statements, lines up the end statement of a noniterative do group in a then or else clause under the "do" and indents the statements of the do group. No irreversible changes, such as with the delnl, insnl or indcomtxt modes, are made. Example:

```
/* format: style1 */
          declare entryname                  char (32);
          if x = 2
          then do;
                    a = 43;
                    b = 21;
               end;
```

Style2 is the same as style1 except using the delnl,insnl modes. This style can cause irreversible changes.

Style3 is the same as style2 except that declare statements are started in column one and the columns the identifiers and

attributes start in are aligned on tab stops. This style can cause irreversible changes. Example:

```
/* format: style3 */
declare   entryname              char (32);
          if x = 2
          then do;
                    a = 43;
                    b = 21;
               end;
```

Style4 starts declare statements in column one, doesn't indent the attributes in declare statements, formats noniterative do groups in then or else clauses by indenting the statements of the noniterative do group one indentation level from the "if" or the "else" and starts the end statement in the same column as the "if" or the "else". This style uses the ^delnl,^insnl modes, but still can cause irreversible changes from indcomtxt mode. This style resembles that of the indent command. Example:

```
/* format: style4 */
declare  entryname char (32);
          if x = 2 then do;
               a = 43;
               b = 21;
          end;
```

Error checking:
Parenthesis balance checking is done for statements that are not partially contained in include files. A warning is printed if an end statement with a closure label terminates more than one block or group. If out_path is omitted and there were errors, the source segment is not overwritten, and a formatted copy is left in the process directory. An error message is printed if a control comment is incorrect.

Error severities:
The following severity values are returned by the severity active function when the "format_pl1" keyword is used:

Value	Meaning
0	No error or format_pl1 has not been used yet.
1	Warning.
2	Correctable error.
3	Fatal error.
4	Unrecoverable error.
5	Bad control arguments, could not find source or other severe errors.

Notes:
If a control argument and its opposite are both present on the command line, the rightmost one is chosen. This command does not work properly with include files that contain partial statements or unbalanced blocks or groups. Also the %page macro or the %skip macro must not occur within a statement. Throughout this document, the term "token" excludes comments. See the Multics PL/I Language Specification, Order No. AG94, for definitions of words describing syntactic constructs in a PL/I program, e.g. independent statement, declaration list, declaration component, condition prefix list, label prefix list, block, group and noniterative do group.