A User's Guide to the

Lout

Document Formatting System

Jeffrey H. Kingston

Version 3.20
April, 2000

 ISBN 0 86758 951 5.

Contents

Chapter 1. Computer Programs

This chapter describes how to typeset computer program text using Lout in conjunction with the prg2lout[1] filter program, which is always installed wherever Lout is.

It is possible to simply print out one or more program files independently of any document. Alternatively, the program text may be printed as part of a larger Lout document. Either way, Lout does not lay out the programs in the sense of choosing line breaks and indenting; it uses whatever line breaks and indenting you give to the program. What Lout does do is cope with characters in the program text that it would ordinarily either reject or interpret in some way (braces and so on), ensuring that you can include program texts with absolutely no modifications; plus, if you wish, Lout will print keywords in bold, identifiers in italics, add line numbers, etc.

At the time of writing, the available programming languages are:

Language name	*Setup file name*	*Lout symbol*	*Default style*	*‘ ’ escapes*
C, C++	cprint	@CP	fixed	No
Eiffel	eiffel	@Eiffel	varying	Yes
Blue	blue	@Blue	varying	Yes
Perl	perl	@Perl	fixed	No
Pod	pod	@Pod	varying	No

C and C++ are handled together since, for formatting purposes, they differ only in that C++ has some additional keywords plus an extra way to make comments. Whenever we mention C from now on, we mean both C and C++. See Section 1.11 for more on Perl and its handmaiden Pod. The second to fifth columns of this table will be explained at various points later in this chapter.

The list of languages is likely to expand, because the prg2lout program has been designed to make it relatively easy to add new languages (you don’t have to write executable code, just declare a lot of records describing your language). Consult the instructions at the top of the source file of that program (*prg2lout.c*) if you want to try it yourself.

1.1. Typesetting computer programs independently of any document

Printing of program files independently of any document is accomplished by the following Unix pipeline:

```
prg2lout -l language options files | lout -s > out.ps
```

where language stands for any one of the programming language names in the first column of the

[1]Prior to Version 3.18 of Lout, this chapter described how to typeset programs written in the C programming language using the c2lout filter, and Eiffel programs using the eif2lout filter. These have now been withdrawn and replaced by prg2lout, which handles multiple languages. Ordinary Lout documents require no modifications as a result of this change.

table above. As usual with Lout, the output will be a PostScript file. Each input file will begin on a new page of the output, starting with its name in bold type. The options provide control over the final appearance, as follows:

-p*style* Select a printing style. Your choices are -pfixed, -pvarying, and -psymbol, with the default value varying with the language as given in the fourth column of the table above. Consult Section 1.2 for examples of these styles.

-n Do not print file names.

-f*font* Select a Lout font family. The default is -fCourier for -pfixed, and -fTimes for -pvarying and -psymbol.

-v*vsize* Select an inter-line spacing size in Lout units. The default is -v1.1fx meaning 1.1 times the font size measured from baseline to baseline.

-L*number* Add line numbers to the program print, starting with *number*, or 1 if *number* is omitted.

-S*file* Use *file* as the setup file for printing your language. This allows you to change all the options mentioned in subsequent sections, rather than just the few given here.

There are also -t and -T options for dealing with tab characters (Section 1.5).

1.2. Typesetting computer programs as part of a larger document

When the program texts are to be part of a larger Lout document, the procedure is somewhat different. You need to include the setup file appropriate to your language, like this:

```
@SysInclude { cprint }
@SysInclude { doc }
@Doc @Text @Begin
...
@End @Text
```

The cprint setup file includes everything needed to set up for C program formatting; for the other languages, consult the second column of the table at the start of this chapter.

The program texts within the Lout document are enclosed in braces preceded by the Lout symbol from the third column of the table, like this for the C language:

```
@IndentedDisplay @CP {
#include <stdio.h>

treeprint(p)        /* print tree p recursively */
struct tnode *p;
{
        if (p != NULL) {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
        }
}
}
```

Although computer programs violate the rules of legal Lout input in many ways, these rules are suspended by the @CP, @Eiffel etc. symbols,A allowing the program text to be incorporated with absolutely no modifications. The result is

```
#include <stdio.h>

treeprint(p)        /* print tree p recursively */
struct tnode *p;
{
        if (p != NULL) {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
        }
}
```

We have chosen to use the @IndentedDisplay symbol from Section ?? to obtain an indented display, but in fact @CP, @Eiffel and the rest may appear anywhere at all: the result is an object in the usual way, which may go anywhere. When including a program text within a paragraph, use @OneCol @CP { ... } (or @OneCol @Eiffel { ... } etc. for other languages) to prevent it being broken across two lines, if desired.

In cases where the program text has unbalanced braces, it is necessary to use the alternative form @CP @Begin ... @End @CP (or the equivalent for other languages), so that Lout does not confuse program braces with Lout braces. In that case the program text must not contain @End; and in either case the program text must not include @Include or @SysInclude unless you are really including a file at that point (Section 1.9).

If your Lout document contains program texts in several languages, simply add one @SysInclude line for each of them and proceed as before. If your programming language is not currently supported, a viable alternative is

```
@F @Verbatim { ... }
```

These symbols cause the text between braces to be set verbatim in a fixed-width font, as explained elsewhere in this guide. This fallback method will not handle tab and formfeed characters very well. Again, use @Verbatim @Begin ... @End @Verbatim if your program text contains unbalanced braces.

1.3. Changing the appearance of a program

The @CP, @Eiffel etc. symbols have a number of options for changing the appearance of the printed program. These options are the same for all symbols, although their default values may vary. The style option changes the printing style; its value may be fixed (fixed-width font), varying (varying-width font), or symbol (varying-width font with mathematical symbols used for some operators). Its default value depends on the language, and may be found in the fourth column of the table at the start of this chapter. The example in the previous section was in fixed style; we can switch styles like this:

```
@CP
   style { varying }
{
#include <stdio.h>

treeprint(p)        /* print tree p recursively */
struct tnode *p;
{
        if (p != NULL) {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
        }
}
}
```

The result in this case will be

```
#include <stdio.h>

treeprint(p)        /* print tree p recursively */
struct tnode *p;
{
        if (p != NULL) {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
        }
}
```

If we use style { symbol } we get this:

```
#include <stdio.h>

treeprint(p)      /* print tree p recursively */
struct tnode *p;
{
    if (p ≠ NULL) {
        treeprint(p→left);
        printf("%4d %s\n", p→count, p→word);
        treeprint(p→right);
    }
}
```

with mathematical symbols replacing some of the operators.

The @CP, @Eiffel etc. symbols have additional options which allow a finer control over the style. Here they all are, with their default values:

```
@CP      [ or @Eiffel, @Blue, etc. ]
  style { fixed }
  numbered { No }
  font { Courier }
  size { -1.0p }
  line { 1.0vx }
  tabin { 8 }
  tabout { 8s }
  identifiers { Base }
  keywords { Base }
  operators { Base }
  numbers { Base }
  strings { Base }
  comments { Base }
{
  ...
}
```

We are already familiar with style. After that comes numbered, whose value may be No (the default), Yes, or a number, and which determines whether or not line numbers are to be added an if so the value of the first one. Next we have font, which determines the font family to use, size, the font size to use, and line, the inter-line spacing. The default value for size asks for one point smaller than in the surrounding document; this was done to compensate for Courier's relatively large appearance compared to other fonts of the same nominal size.

The tabin and tabout options are the subject of Section 1.5. After them come six options giving the particular font faces in which to print identifiers, keywords, operators, numbers, strings, and comments. Base means the basic face; other commonly available choices are Slope and Bold. These options may all be set to different faces if desired. The default values shown are correct for style { fixed } only; the other styles have other defaults (Section 1.4).

1.4. Changing the appearance of all programs simultaneously

We have just seen that the @CP, @Eiffel etc. symbols have many options for changing the appearance of the program text. However, most people would not want to have a different style for every program text in their document; they want to define the style once at the start, and have all their program texts come out in that style without laboriously setting options on every symbol. You do this by copying the setup file and changing it.

For general information about how to make your own setup file, consult Section ??. The options that determine the default values are in the @Use clause which occupies most of the setup file. Here is part of the @Use clause from cprint:

```
@Use {  @CPSetup
    # pipe                      {           }
    # numbered                  { No        }
    # style                     { fixed     }

    # fixedfont                 { Courier   }
    # fixedsize                 { -1.0p     }
    # fixedline                 { 1.0vx     }
    # fixedtabin                { 8         }
    # fixedtabout               { 8s        }

    # fixedidentifiers          { Base      }
    # fixedkeywords             { Base      }
    # fixedoperators            { Base      }
    # fixednumbers              { Base      }
    # fixedstrings              { Base      }
    # fixedcomments             { Base      }
    # fixedlinenumbers          { Base      }

    # fixedidentifiersformat    { @Body     }
    # fixedkeywordsformat       { @Body     }
    # fixedoperatorsformat      { @Body     }
    # fixednumbersformat        { @Body     }
    # fixedstringsformat        { @Body     }
    # fixedcommentsformat       { @Body     }
    # fixedlinenumbersformat    { @Body     }

...

}
```

The pipe option will be explained in Section 1.9. The options whose name begins with fixed apply only when style is fixed; there are corresponding options, not shown, which apply when style is varying and symbol.

We can see in this extract that the default value of style is fixed, and of numbers is No. We can also see the default font family, font face, font size, line spacing, and tab settings when the style is fixed. The font family name for fixed style is Courier, but for the other styles (not shown) it is empty. This causes the fixed style to always switch to Courier, and the other styles to use the same font family as in the surrounding document.

The options from fixedidentifiers to fixedlinenumbers allow you to set the font face to use for each of these parts of your program. People who want fixed-width fonts do not usually want very exciting font faces either, so the default values above are all Base, but for the varying and symbol styles, the default identifier face is Slope, the default keyword face is Bold, and so on. You can actually give a family name before the face name in these options, allowing you to switch font families for different parts of the program if you wish.

The fixedidentifiersformat option allows you to make a more radical change to the format of identifiers than just the font. Within this option, @Body stands for the identifier being formatted, and by applying Lout symbols to it, you apply them to every identifier. For example,

```
fixedidentifiersformat { red @Colour @Body }
```

will cause identifiers to be printed red.[1] If you do use exotic formats, remember that in some programming languages, comments and even strings may occupy more than one line: @Box, for example, will give a logical but probably unwanted result when formatting a multi-line string.

As always with setup files, to change a default value, delete the preceding # and change the part between braces. For example, suppose you are happy with fixed except that you want bold keywords. Then one line needs to be changed, to

```
fixedkeywords { Bold }
```

Or suppose you like varying as it stands, but would like it to be the default style rather than fixed. Again, only one line needs to be changed, to style { varying }.

It is probably not a good idea to change the default value of numbered to Yes, because small fragments of code within paragraphs will then get line numbers as well as large displayed programs. If you do have many large numbered programs as well as small fragments, a better approach would be to place

```
import @CPSetup
macro @NCP { @CP numbered { Yes } }
```

(or the equivalent for your language) in your mydefs file, so that you can type @NCP instead of @CP numbered { Yes }.

The setup files for the other languages are identical to this one, except that the symbol after @Use is different, and some of the default values may be different. Changing an option affects only the language of that setup file; if you have multiple languages you can have multiple setup

[1]@Colour is not a Lout primitive like, say, @Font; it is defined when you write @SysInclude { doc } or the equivalent for the other document types. This is true of quite a few generally useful symbols, including @Box and @I. If you want to use these symbols here, you must include your setup file *after* @SysInclude { doc } or whatever, the reverse of the usual arrangement, so that they are defined before Lout reads your setup file. This reversal is carried out automatically when formatting programs independently of any document, so you can use these symbols in a setup file given by a -S command line flag.

files and change their options quite independently of each other.

1.5. Dealing with tab characters in programs

Tab characters provide a convenient way to indent and align parts of computer programs. With care, this alignment can be preserved in the final print even with varying-width fonts.

The distance between two tab stops in the program text is by default taken to be 8 characters, which is standard for Unix. This can be changed with the tabin option. For example,

```
@CP tabin { 4 }
```

informs Lout that tab stops occur every 4 characters in the program text. All the symbols (@CP, @Eiffel, etc.) and their setup files have this option and the next; but to save repetition we will stick with C for the rest of this section.

The distance between two tab stops on the printed page is quite a different thing, and it is determined by the value of the tabout option, which must be a Lout length. For example,

```
@CP tabout { 0.5i }
```

requests that tab stops be placed at half-inch intervals. In other words, a distance of one tab stop in the program text will be equivalent to a distance of half an inch on the printed page. For example,

```
@CP style { varying } tabout { 3f } numbered { Yes }
```

might produce the following, where tab characters in the program text have been used for indenting and also to align the comments:

```
struct tnode {                  /* the basic node */
        char *word;             /* points to the text */
        int count;              /* number of occurrences */
        struct tnode *left;     /* left child */
        struct tnode *right;    /* right child */
};
```

We've used numbered { Yes } to demonstrate that the features for dealing with tabs work even with line numbers. The value 3f means three times the current font size, and it is the default value of tabout for the varying and symbol styles (Section 1.4). In a 12 point font this is 36 points, or half an inch.

If tabout is too small, there is a danger that the alignment might fail. For example,

```
@CP style { varying } tabout { 0.2i }
```

produces

```
struct tnode {/* the basic node */
    char *word;/* points to the text */
    int count;/* number of occurrences */
    struct tnode *left;/* left child */
    struct tnode *right;/* right child */
};
```

given the same C text as the previous example. The problem here is that we are asking for `/*` to appear four tab stops or 0.8 inches from the left edge, and yet the material to its left on the line is wider than this. This causes `/*` to be shifted further to the right than expected, and the alignment is lost. The only solution is to increase tabout.

When typesetting computer program texts independently of any document, there are -t and -T options to the prg2lout program equivalent to tabin and tabout respectively. For example, -T0.5i produces a half-inch tab width.

1.6. Dealing with formfeed characters in programs

The formfeed (Control-L) character is traditionally taken to be a request to start a new page. This is explicitly recognized by the formal definition of the C language and many others, which treat this character as white space from a language point of view, with the understanding that it will cause a page break when printed.

There are no prg2lout options for dealing with formfeed characters. They will be converted into @NP (new page) symbols, causing a new page or column to be begun in the printing.

Whether formfeed characters end their line or not is a problem. Consider this example, where ^L stands for one formfeed character:

```
abc
def^Lghi
jhk
```

How many lines does this example contain? Your text editor would probably say ‘three’, but when you print it you will see four. It is not desirable to have printed programs (especially those with line numbers attached) disagreeing with text editors about line numbers. The solution adopted by prg2lout to this problem is to treat the formfeed character as including a newline, but to assign the same line number to both parts of the original line (the parts before and after the formfeed). If the part after the formfeed is empty (that is, if the formfeed character is immediately followed by a newline or another formfeed), and if the formfeed is not inside any lexical unit, then the empty line after the formfeed will not be printed at all.

The most common case is that of a formfeed character, outside any lexical unit, on a line by itself. Let’s see what this rule produces in the following example of this case:

```
abc
def
^L
ghi
```

There will be one blank line numbered 3 at the end of the first page, and a line numbered 4 and containing ghi at the start of the next page. The blank line is a necessity, at least when lines are being numbered, because we want the last line in the example to be numbered 4 to agree with text editors, but we don't want the line numbers on our print to skip from 2 on the first page to 4 on the second, because that would make readers anxious about the apparently missing line 3. If you don't want that empty line, move the formfeed character to the end of the preceding line.

1.7. Embedding Lout commands within program comments

It is possible to embed Lout text inside program comments. How this is done could in principle vary from language to language, but in every language supported so far it is done by starting off the comment with an @ character. If the language has several ways to get a comment, this will work every way. The entire comment after the @ character should then be Lout text. For example, to force Lout to start a new page at some point within a C program, place

```
/*@ @NP */
```

at that point. (In this case you can also simply include a formfeed character, control-L, without any comment; see Section 1.6 for more on this.) Or, to make a heading in an Eiffel program, do this:

```
--@ @Display @Heading { treeprint }
```

(Eiffel comments begin with -- and end at the end of the line.) Other possible uses for this feature include index entries and margin notes. Incredible as it may seem, you can even write

```
/*@ @CD @Heading { Function @CP { treeprint() } } */
```

with a @CP symbol and some C code inside the Lout code inside the C code. You probably can't go further, however, at least not in C, since that would require a C comment inside a C comment.

1.8. Embedding program text within program comments

The standard reference for the Eiffel language [1] specifies that identifiers within comments may or should be enclosed in ‘ and ’ so that they may be noticed and printed in an italic font:

```
@ID @Eiffel {
deposit(amount: REAL) is
    -- deposit ‘amount’ dollars
}
```

produces

deposit(*amount*: *REAL*) **is**
    -- deposit *amount* dollars

This has been generalized in Lout: arbitrary text within an Eiffel comment between ‘ and ’ will be treated as Eiffel text and printed accordingly. Some other languages may also offer this feature: see the fifth column of the table at the start of this chapter. In principle the precise means of getting it could vary from language to language, but the languages available at the moment either do not have it at all, or else they use ‘ and ’ like Eiffel.

On the subject of Eiffel, the Eiffel reference [1] has some quite detailed style guidelines, and these have been closely followed in the implementation of the @Eiffel symbol. In particular, @Eiffel prints dots larger than usual when they denote feature calls, as the example *account*.*deposit*(20) shows.

1.9. Reading and selecting program text from separate files

We have said that program text within @CP { ... } and the other symbols is passed directly to prg2lout for analysis. However, there is an exception. The program text may contain an @Include or @SysInclude command, which, as for the @Verbatim symbol (Section ??), causes Lout to take the program text from a file:

```
@Eiffel
{
   @Include { "/usr/staff/jeff/Eiffel/hash.e" }
}
```

The included file is not examined for balanced braces or @End or @Include; it is treated entirely verbatim and passed straight on to prg2lout. There may be several @Include commands, and any amount of program text as well, within @CP { ... } and the rest.

When including files in this way it often happens that only part of an actual program file is wanted for display. Rather than placing the wanted part in a separate file, which is error-prone and tedious when the program is changing, Unix users can use the pipe option to pipe the entire file through an arbitrary sequence of Unix commands, which may be used to make the wanted selection before the program text is passed to prg2lout.

For example, suppose that all your Eiffel routines begin with the routine name one tab stop from the left margin and end at the first following **end** indented two tab stops. Then

```
@Eiffel
   pipe { "sed -n /^.insert/,/^..end/p" }
{
   @Include { "/usr/staff/jeff/Eiffel/hash.e" }
}
```

will select just the *insert* routine from the hash.e file. Assuming that your program text has been laid out in a disciplined manner, every line of the selection will begin with a tab character that is not wanted in this display, so an even better pipe is

```
@Eiffel
   pipe { "sed -n /^.insert/,/^..end/p | cut -c2-" }
{
   @Include { "/usr/staff/jeff/Eiffel/hash.e" }
}
```

since it cuts away the unwanted tab characters. Unfortunately, we can't show the result of this on an actual example, since that would prevent this manual from being formatted on a non-Unix system.

1.10. Error messages

In order to understand the error messages produced by program printing, it is necessary to understand that Lout's first step when given a program text is to pass it to the separate prg2lout program for analysis. This separate program is the source of most of the error messages associated with program printing.

The prg2lout program is quite happy to format a fragment of a computer program: there is no need to supply a complete routine, or a complete statement, or any such thing. However, it will complain if you supply only a fragment of one lexical unit, such as a comment or string without its terminating delimiter. It will also complain if there is a character that cannot be classified as part of an identifier, number, etc. according to the rules of the language as they have been given to prg2lout by the implementer. Irrespective of the language rules, prg2lout always interprets spaces, tabs, newlines, and formfeed characters in the usual way.

If an error message is generated by prg2lout, it will contain a line and column number counting from the start of the program text involved. Lout will precede this error message with a file name, line number, and column number pointing to the Lout symbol (@CP, @Eiffel etc.) whose program text caused the error message, like this:

```
lout file "prg_tabs" (from "prg" line 96, from "all" line 46):
  56,23: prg2lout 2,1: program text ended within comment
```

This is an actual message produced when formatting this chapter. The program text in question has only one line, containing an incomplete comment, so when prg2lout tried to start the second line and found nothing, it complained as shown. In general, then, you have to add prg2lout's line number to Lout's line number, and use some initiative, to find the precise point of the problem.

1.11. Notes on Perl and Pod

The Perl programming language[1] is quite a difficult one for the prg2lout program to deal with, and our boast that programs can be included with 'absolutely no modifications' is not quite true for Perl.

Here is the complete list of problem areas. In most cases their effect is to get the formatting wrong over a short region, which is not perhaps so disastrous; and it should be easy to modify

[1] My thanks to Mark Summerfield for help with Perl and Pod.

your Perl program without changing its meaning, to work around these problems. After all, in Perl there is always more than one way to do it.

1. *Here-documents* such as

   ```
   <<"EOF"
   These lines will be read as though
   enclosed in double quotes
   EOF
   ```

   will be handled correctly only if the string appearing immediately after the << operator (that is, the string used to terminate the here-document) is one of `EOF`, `EOT`, `END`, and the empty string, all optionally enclosed in quotes of any of the three kinds. If this condition is not met, then the here-document will be treated as Perl program text. If the condition is met, there is still another problem: the << symbol and everything after it on the same line will be treated (incorrectly) as a string. The worst consequence of this is that stacked here-documents will not be printed properly.

2. When prg2lout is scanning the program text looking for the beginning of a lexical unit, it may come upon a / character, and this *initial* / (not subsequent ones in the same lexical unit) it finds difficult to interpret, since it may be the beginning of a regular expression, to be formatted like a string, or it may be a complete lexical unit denoting division. The program chooses the regular expression (or equivalently, string) interpretation if the / character is immediately preceded by q, qq, qx, qw, qr, m, s, y, or tr. It also chooses the regular expression interpretation if the / character appears at the start of a line, or if it is immediately preceded by zero, one, or two space or tab characters, which are themselves immediately preceded by a complete lexical unit which is one of (, =, =~, !~, split, if, and, &&, or, ||, not, !, unless, for, foreach, and while. Otherwise it chooses the division interpretation. In the rare cases where this rule fails, you can force prg2lout to choose the regular expression interpretation by placing an m in front of the initial / (this does not change the meaning of the program), and you can force the division interpretation by placing at least three spaces before the / character.

3. Substitution expressions, even such lexically complex ones as `s{{@D}}[{@I}]`, are handled correctly. However, prg2lout does not understand that the letters gimosx in any combination appearing immediately after a substitution expression are part of it; it treats them as the start of a new lexical unit. This new unit will usually be taken to be an identifier, which is harmless enough, but occasionally it is taken to be something else. For example, in

   ```
   s///s;
   ```

   the trailing s will be mistaken for the start of a new substitution expression, with ; delimiting the first pattern. This particular example can be fixed by inserting a space before the semicolon.

Further work may eliminate some of these problems.

The Pod language is used by Perl programmers for creating documentation, and may be found within Perl programs or standing alone. Lout supports both arrangements without any special action by the user. At the beginning of the perl setup line, the following line has been placed:

```
@SysInclude { pod }
```

Thus, asking for Perl always gives you Pod as well. If you are using your own setup files for both languages, it is probably better to break this connection by deleting this line from your copy of the perl setup file and placing

```
@Include { mypod }
@Include { myperl }
```

at the start of your document in the usual way.

Because Pod is a documentation language rather than a programming language, the setup file options listed in Section 1.4 do not really apply. So for Pod only these have been discarded and replaced by a completely different set of options, controlling such things as the size of headings and the gaps between list items, which you can find documented in the pod setup file.

If you ask for line numbers on a Pod program, or on a Perl program that contains Pod, any text blocks in the Pod that would otherwise have appeared as filled paragraphs will come out with the line breaks in the source respected, and lines numbered accordingly. Because prg2lout attaches line numbers before Lout breaks paragraphs, it is not possible to numnber the lines after paragraph breaking.

Owing to problems behind the scenes, if a Pod inclusion in a Perl program has unbalanced braces, prg2lout is forced to insert braces into the Pod text to make them balance. It will insert a left brace directly before any unbalanced right brace, and it will insert right braces at the end of the Pod inclusion to balance any preceding unbalanced left braces. It will tell you if it has to do this. This problem does not afflict Pod when used as a separate language.

References

[1] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

Index