# A New Approach to Document Formatting

*Jeffrey H. Kingston*

Basser Department of Computer Science
University of Sydney 2006
Australia

*ABSTRACT*

This paper describes a new approach to document formatting, in which features are written in a small, coherent, high-level language called Lout. The resulting increase in productivity has permitted many advanced features to be developed quickly and accurately, including page layout of unprecedented flexibility, equation formatting, automatically generated tables of contents, running page headers and footers, cross references, sorted indexes, and access to bibliographic databases. A fully operational production implementation of the Lout system including all these features and many others is freely available.

22 December, 1992

# A New Approach to Document Formatting

*Jeffrey H. Kingston*

Basser Department of Computer Science
University of Sydney 2006
Australia

## 1. Introduction

The personal computer and the laser printer have sparked a revolution in the production of documents. Many authors now routinely take their work from conception to camera-ready copy, many publishers are using desktop publishing systems, and it is probable that manual assembly of documents will become uncommon in the near future.

As control moves into the hands of an ever-increasing number of non-technical people, the stresses on document formatting software increase. On the one hand, this software must be so simple that anyone can use it; on the other, it must supply a bewildering array of features. A book, for example, demands fonts, paragraph and page breaking, floating figures and tables, footnotes, chapters and sections, running page headers and footers, an automatically generated table of contents, and a sorted index. Add to this an open-ended list of specialized features, beginning with mathematical typesetting, diagrams, and access to bibliographic databases, and the result is a nightmare for the software developer.

One solution to this feature explosion problem is to implement as a system primitive every feature that will ever be required. Although all of the successful interactive document editors known to the author take this approach (admittedly with some attempt to generalize and unify their features), it has clearly reached its limit. Few such systems provide equation formatting, fewer still will format a Pascal program, and other specialized features will simply never be implemented.

A second solution is to provide a relatively small system equipped with a means of defining new features, as in programming languages. This approach has been taken by the batch formatters (those which do not display a continuously updated image of the printed document while editing) found in academia, notably troff [11], TEX [10], and Scribe [12]. Features such as footnotes and automatic tables of contents have been added to these systems using macro definitions. Unfortunately, such extensions are very difficult and error-prone in practice: TEX's footnote macro alone contains half a page of dense, obscure code, while those who have extended troff have abandoned the language itself and taken refuge in preprocessors. A more productive basis for developing new features is needed.

This article presents a high-level language for document formatting, called Lout, which is intended to form such a basis. Lout is quite accessible to non-expert users, but its unique property is the ease with which expert users can add new features. We begin with a presentation of Lout as it appears to the non-expert user who employs the standard packages without understanding Lout's principles. Later sections switch to the expert's view, showing by examples the principles of Lout and how advanced features are defined.

A Unix-compatible[1] batch formatter for

[1] Unix is a trademark of AT&T Bell Laboratories.

Lout (called Basser Lout) has been written which produces PostScript[1] output suitable for printing on most laser printers and many other devices. A library of standard packages written in Lout provides all of the features listed above and many others. This system is not an experimental prototype, it is a fully operational production implementation. The software and its supporting documentation [3, 9, 4, 5, 6, 7, 8] are available free of charge from the author.

## 2. The non-expert's view

The non-expert user perceives Lout as text interspersed with special symbols, in a style reminiscent of many other batch formatters:

```
@Doc @Text @Begin
@Heading { Standard Integrals }
@PP
The following list of standard
integrals should be memorized:
@NumberList
@Item @Eq {int e sup x dx = e sup x}
@Item @Eq {int dx over
  sqrt { 1 - x sup 2 } = arc sin x}
@EndList
@End @Text
```

Braces are used for grouping parameters to the features. The symbols are all taken from two of the standard packages: DocumentLayout, which provides headings, paragraphs, lists, footnotes, sections, and so on, and Eq, which provides mathematical typesetting in a style copied from the eqn language of Kernighan and Cherry [2].

At the time of writing, packages exist for formatting general documents, technical reports, and books, the latter providing an automatic table of contents, running page headers and footers, access to bibliographic databases, and a sorted index, among many other features. Specialized packages exist for mathematical typesetting, drawing figures, and formatting Pascal programs.

The advanced features maintain the simple style established above. To produce a footnote, for example, one simply types

```
@FootNote { ... }
```

at the appropriate point, and it will be numbered and placed at the bottom of the page; to add an item to the index,

```
expert @Index { Expert user }
```

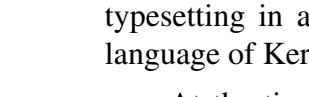
is typed, and the right parameter will appear in the index, with a page number, at a place determined by the alphabetical ranking of the left parameter. No technical knowledge is required to use these features.

## 3. Objects

To the expert user, Lout is a high-level functional language with a relatively small repertoire of primitive features organized around four key concepts: *objects*, *definitions*, *cross references*, and *galleys*. An object is a rectangle with at least one horizontal and one vertical mark protruding from it. For example,

```
Australia
```

is an object which is viewed by Lout like this:

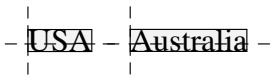



Horizontal and vertical concatenation operators, denoted by the symbols | and /, are used to assemble larger objects:

```
USA  |0.2i  Australia
```

is the object

[1] PostScript is a trademark of Adobe Systems, Incorporated.

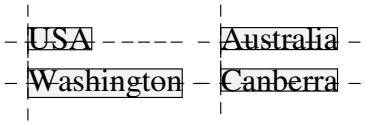



The parameters are separated by the length given after the concatenation symbol (0.2 inches in this example), and their horizontal marks are aligned.

Tables are made by combining the two operators, with | having the higher precedence:

```
    USA          |0.2i  Australia
/0.1i  Washington |     Canberra
```

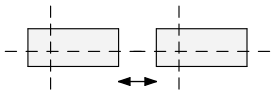
is the object



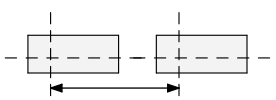

The second horizontal concatenation symbol needs no length, since the first one determines the separation between the two columns created by the alignment of the vertical marks. Objects of arbitrary complexity may be assembled using these and other operators, and braces used for grouping, in a manner analogous to the assembly of expressions in programming languages.

The lengths attached to concatenation symbols have features which permit objects to be positioned very precisely. In addition to the usual units of measurement (inches, centimetres, points, and ems), lengths may be measured in units of the current font size, space width, inter-line space, and available width (for centering and right justification).

There are also six *gap modes*, which determine where the lengths are measured from. Previous examples have used edge-to-edge mode:



Lout also provides a mark-to-mark mode, obtained by appending x to the length:



The length will be widened if necessary to prevent the parameters from overlapping, thus implementing the baseline-to-baseline spacing used between lines of text. Other modes provide tabulation from the left margin, overstriking, and hyphenation.

The final appearance of an object is affected by a limited amount of information inherited from the context, principally the font and the width available for the object to occupy. There are operators for setting these attributes:

```
Slope @Font {
Hello, world
}
```

produces

*Hello, world*

and in a similar way

```
1.5i @Wide {
(1) |0.1i A small
indented paragraph
of text.
}
```

produces

(1) A small indented paragraph of text.

with the paragraph inheriting and being broken to an available width of 1.4 inches minus the width of (1). This size inheritance remains secure through all the complexities of gap modes, mark alignment, the @Wide and other operators, and so on, providing a high-level service comparable in value with strong typing in programming languages.

## 4. Definitions

Lout permits the user to define operators which take objects as parameters and return objects as results. This feature,

unremarkable in itself, has some surprising applications, most notably to a problem which is the litmus test of flexibility in document formatting: the specification of page layout.

The use of operators in document formatting seems to have been pioneered by the eqn equation formatting language of Kernighan and Cherry [2]. In eqn, the mathematical formula

$$\frac{x^2 + 1}{4}$$

for example is expressed as

```
{ x sup 2 + 1 } over 4
```

This identical expression is also used in Lout, but in Lout the operators (sup, over and so on) have visible definitions which are easy to modify and extend.

For example, here is the definition of the over operator as it appears in the Eq equation formatting package [6]:

```
def over
    precedence 54
    associativity left
    left x
    right y
{
    @OneRow @OneCol
    {
        |0.5rt  x
        ^//0.2f  @HLine
        //0.2f  |0.5rt  y
    }
}
```

Invocations of over return the object between the braces, with the formal parameters x and y replaced by actual parameters which are objects found to the left and right of the over symbol.

The body of over makes a good demonstration of the way in which Lout's operators combine together. All are Lout primitives except @HLine, which calls upon Lout's graphics primitives to draw a horizontal line. The // and ^// operators are variants of vertical concatenation which omit mark alignment; the separation is 0.2 times the current font size. The two |0.5rt operators center each parameter in the column. Finally, the @OneRow and ^// operators work together to ensure that only one horizontal mark protrudes, rather than three; the result has the structure

$$- \boxed{\frac{x^2 + 1}{4}} -$$

and so will be aligned correctly with adjacent parts of the equation.

As is usual in functional languages, sequences are obtained by recursive definitions. For example, the 'leaders' often seen in tables of contents can be generated by the definition

```
def @Leaders
{
    ..  @Leaders
}
```

White space after { and before } is not significant. The recursion stops when space runs out, so

```
1.5i @Wide {
Chapter 1  @Leaders  5
}
```

has result

Chapter 1 .. .. .. 5

The final invocation of @Leaders is deleted along with any preceding concatenation operator (or white space in this case).

The specification of page layout is a major problem for many document formatters, because the model of a document as a sequence of pages is built-in, and an

armada of tedious commands is required to communicate with this model: commands for setting page width and height, margins, columns, page header and footer lines, and so on. Even with all these commands, the formatter will be unable to do such a simple thing as draw a box around each page, if there is no command for it.

Lout has no built-in model and no such commands. Instead, a page is an object like any other:

```
def @Page
   right x
{
   8i @Wide 11i @High
   {
      //1i ||1i x ||1i
      //1i
   }
}
```

The result of @Page is an eight by eleven inch object containing the right parameter within one inch margins. A document is a vertical concatenation of numbered pages:

```
def @PageList
   right @PageNum
{
   @Page
   {
      |0.5rt  - @PageNum -
      //0.2i  @TextPlace
      //1rt   @FootSect
   }
   //0i
   @PageList
      @Next @PageNum
}
```

The @Next operator is a Lout primitive that returns its right parameter plus one; all automatic numbering is effected by combining this operator with recursion. The result of @PageList 1 is the object


@PageList 3

which has the potential to expand to infinitely many pages.

We conclude this example by defining @FootSect to be a small horizontal line above a list of @FootPlace symbols:

```
def @FootList
{
   @FootPlace
   //0.1i @FootList
}

def @FootSect
{
   1i @Wide @HLine
   //0.1i @FootList
}
```

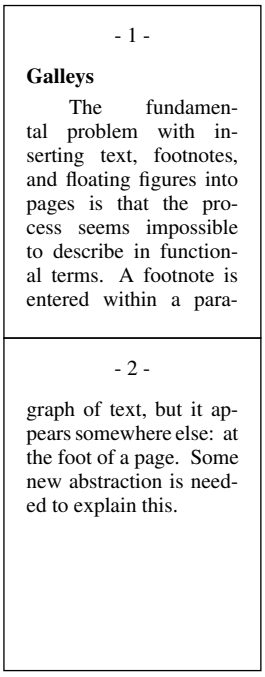
This method of specifying page layout is infinitely more flexible than the other, since the full resources of the language may be brought to bear. Of course, the @PageList, @FootSect, and @FootList symbols must be expanded only on demand, and we have yet to see how the @TextPlace and @FootPlace symbols can be replaced by actual text and footnotes.

## 5. Galleys

The fundamental problem with inserting text, footnotes, and floating figures into pages is that the process seems impossible to describe in functional terms. A footnote is entered within a paragraph of text, but it appears somewhere else: at the foot of a page. Some new abstraction is needed to explain this.

The landscape of features that previous document formatting systems have introduced at this point can best be described metaphorically, as an antediluvian swamp populated by dinosaurs and demons, whose air is filled with the piteous cries of document format designers in torment.

Lout's solution to this problem is a feature called the *galley,* after the metal trays used in manual typesetting. A galley consists of an object plus an indication that the object is to appear somewhere other than its invocation point. For example,




is the result of the expression

```
@PageList 1
//
@Text {
@Heading { Galleys }
@PP
The fundamental ...
... to explain this.
}
```

The only new definitions required are these:

```
def @TextPlace { @Galley }

def @Text
   into { @TextPlace&&preceding }
   right x
{ x }
```

They say that @TextPlace (which the reader will recall as lying within the pages of @PageList) is a placeholder for an incoming galley, and that @Text is a galley whose result is to appear, not where @Text is invoked, but rather at some @TextPlace preceding that point of invocation in the final printed document.

Although the abstraction can be understood in a static way, it is helpful to trace what happens when the Basser Lout batch formatter reads the expression above.

Since @PageList 1 indirectly contains the special @Galley symbol, it will be expanded only upon demand. The discovery of the @Text galley initiates a search for a @TextPlace, which is found within @PageList 1 and so forces one such expansion. The available width at this @TextPlace is six inches, so the @Text galley is broken into six-inch components which are promoted one by one until the available height (nine inches) is exhausted. Since the @Text galley is not entirely consumed, a forward search of the document is made, another @TextPlace is found within the as yet unexpanded @PageList 2, and the process is repeated.

The treatment of footnotes is the same, except that

```
def @FootNote
   into { @FootPlace&&following }
```

is used to make the footnote appear later in the finished document than its invocation point. Basser Lout suspends the promotion of text into pages just after the component containing the footnote's invocation point is promoted, switches to the placement of the footnote galley, then resumes the body text.

A collection of galleys all targeted to the same place may optionally appear sorted on a designated key, thus implementing sorted reference lists and indexes. The author was obliged to make the sorting option a primitive feature, since it otherwise seems to require boolean operators which he preferred to exclude.

The @PageList object which receives the @Text galley can itself be viewed as a galley whose components are pages, and this leads to a dynamic view of Lout document assembly as a tree of galleys, each promoting into its parent, with the root galley promoting into the output file. For example, the BookLayout package [4] has @Section galleys promoting into @Chapter galleys promoting into a single @PageList galley, which promotes into the output; no limit is imposed on the height of the tree.

## 6. Cross references

The terms @TextPlace&&preceding and @FootPlace&&following used above can be thought of as arrows in the final printed document, pointing from themselves to the place they name. Expressed in this way, free of any reference to the internal action of the document formatter, they are easy to comprehend and work with. These arrows are called cross references in Lout.

A galley is transported forwards along its arrow, but it turns out that a reverse flow of information can also be very useful. For example, large documents often have cross references such as 'see Table 6 on page 57.' If the numbers are replaced by arrows pointing to the table in question, it should be possible to have their values filled in automatically (an idea introduced by Scribe [12]). An arrow pointing outside the document could retrieve an entry from a database of references, Roman numerals, etc. And a running page header like 'Chapter 8: Sorting' might obtain its value from an arrow pointing from the page header line down into the body text of the page, where the current chapter is known.

All these ideas are realized in Lout, but here we will just sketch a simplified version of the running page header definitions found in the BookLayout package [4]. A symbol called @Runner is first defined:

```
def @Runner
   right @Val
{}
```

@Runner produces nothing at all, which means that we may place the invocation

```
@Runner { Chapter
8:  Sorting }
```

at the end of a chapter without harm. This invisible invocation will be carried along with the chapter and will end up on some page of the final printed document.

By modifying the definition of @PageList, we can add to each page a header line containing the expression

```
@Runner&&following
@Open { @Val }
```

This means 'find the nearest following invocation of @Runner in the final printed document and retrieve its @Val parameter.' Every page of Chapter 8 will find the correct

running header, since @Runner was placed at the end of the chapter. The invocation @Runner {} placed at the beginning of the chapter will suppress the header on the first page of the chapter, as it is conventional to do.

These invocations of @Runner are hidden from the non-expert user within the definition of the @Chapter operator. The result is a reliable implementation of a notoriously difficult feature.

## 7. Conclusion

The Lout document formatting system permits features as diverse as page layout and equation formatting to be implemented by definitions written in a high-level language. The consequent improvement in productivity has allowed an unprecedented repertoire of advanced features to be presented to the non-expert user.

To future research in document formatting, Lout offers evidence of the utility of the functional paradigm, as well as two new abstractions: galleys and cross references. These provide a secure foundation for features which have proven very difficult to implement in the past.

A number of improvements to the current system can be envisaged. Better paragraph and page breaking algorithms could be added to the formatter without any change to the language; non-rectangular objects would be useful in some places. Perhaps the most useful improvement would be the representation of paragraphs as horizontal galleys, since this would allow the full power of the language to be brought to bear on paragraph layout, in contrast to the present built-in system which offers only the traditional styles (ragged right, justified, and so on).

The author of a recent interactive document editor [1] has recommended that the interface be supported by a functional base language, accessible to the expert user, for such purposes as page layout definition and fine control over formatting. Lout appears to be an excellent candidate for such a language, because of its small size, precision, and functional semantics.

## References

1. Brooks, Kenneth P., Lilac: a two-view document editor. *IEEE Computer*, 7–19 (1991).

2. Kernighan, Brian W. and Cherry, Lorinda L., A system for typesetting mathematics. *Communications of the ACM* **18**, 182–193 (1975).

3. Kingston, Jeffrey H., *Document Formatting with Lout (Second Edition)*. Tech. Rep. 449 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia.

4. Kingston, Jeffrey H., *A beginners' guide to Lout*. Tech. Rep. 450 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia.

5. Kingston, Jeffrey H., *The design and implementation of the Lout document formatting language*. Tech. Rep. 442 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia. To appear in *Software—Practice and Experience.*

6. Kingston, Jeffrey H., *Eq – a Lout package for typesetting mathematics*. Tech. Rep. 452 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia. Contains an appendix describing the Pas Pascal formatter.

7. Kingston, Jeffrey H., *Fig – a Lout package for drawing figures*. Tech. Rep. 453 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia.

8. Kingston, Jeffrey H., *Tab – a Lout package for formatting tables*. Tech. Rep. 451 (1992), Basser Department of Computer Science F09, University of Sydney 2006, Australia.

9. Kingston, Jeffrey H., *The Basser Lout Document Formatter, Version 2.05*, 1993. Computer program, publicly available in the *jeff* subdirectory of the home directory of *ftp* to host *ftp.cs.su.oz.au* with login name *anonymous* or *ftp* and any non-empty password (e.g. *none*). Lout distributions are also available from the comp.sources.misc newsgroup. All enquiries to jeff@cs.su.oz.au.

10. Knuth, Donald E., *The TEXBook*. Addison-Wesley, 1984.

11. Joseph F. Ossanna, *Nroff/Troff User's Manual*. Tech. Rep. 54 (1976), Bell Laboratories, Murray Hill, NJ 07974.

12. Reid, Brian K., A High-Level Approach to Computer Document Production. *Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL), Las Vegas NV*, 1980, 24–31.