# The Design and Implementation of the Lout Document Formatting Language

*Jeffrey H. Kingston*

Basser Department of Computer Science,
The University of Sydney 2006,
Australia

27 January, 1993

## SUMMARY

Lout is a high-level language for document formatting, whose ease of use has permitted an unprecedented number of advanced features to be added quickly and reliably. This paper charts the evolution of the design and implementation of Lout from conception in mid-1984 to public release in October 1991. It includes extensive discussions of remaining problems and possible solutions.

**Keywords** document formatting typesetting

**The Design and Implementation**

**of the**

**Lout Document Formatting Language**

*Jeffrey H. Kingston*

Basser Department of Computer Science,
The University of Sydney 2006,
Australia

27 January, 1993

## 1. Introduction

Lout [1, 2] is a high-level language for document formatting, designed and implemented by the author. The implementation, known as Basser Lout, is a fully operational production version written in C for the Unix operating system,[1] which translates Lout source code into PostScript,[2] a device-independent graphics rendering language accepted by many high-resolution output devices, including most laser printers. Basser Lout is available free of charge [3]. It includes installation instructions, C source, seven standard packages, and complete documentation in the form of six technical reports and a manual page.

The Lout project arose out of the author's desire to bring to document formatting languages the elegance of expression found in programming languages like Algol-60 and Pascal. This emphasis on expressiveness has produced an order of magnitude reduction in the cost of developing document formatting applications. For example, an equation formatting application, which may be difficult or impossible to add to other systems, can be written in Lout in a few days.

When expert users can implement such applications quickly, non-experts benefit. Although Lout itself provides only a small kernel of carefully chosen primitives, packages written in Lout and distributed with Basser Lout provide an unprecedented array of advanced features in a form accessible to non-expert users. The features include rotation and scaling, fonts, paragraph and page breaking, displays and lists, floating figures and tables, footnotes, chapters and sections (automatically numbered), running page headers and footers, odd-even page layouts, automatically generated tables of contents, sorted indexes and reference lists, bibliographic and other databases (including databases of formats for printing references), equations, tables, diagrams, formatting of Pascal programs, and automatically maintained cross references.

This paper charts the evolution of Lout from conception in mid-1984 to the public release of Basser Lout in October 1991. Lout is organized around four key concepts – objects, definitions,

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

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

galleys, and cross references – and they were developed in the order listed, so this paper will treat each in turn, discussing its design, implementation, problems, and prospects for further improvement.

## 2. Objects

The essence of any move to a higher level is the introduction of some abstraction which serves to organize the low-level operations, resulting in a more succinct expression of their common combinations at the cost of some loss of detailed control. The early part of the Lout project was spent in the development of such an abstraction for the building blocks of documents, one which could explain, not just the simple phenomena of words, lines, and paragraphs, but also the alignment of columns in tables, and the complex nested structures of equations.

### 2.1. The genesis of the object abstraction

When one examines previous document formatting systems [4] looking for ideas for abstractions, as the author did in 1984, the Eqn formatting language [5] stands out like a beacon. In Eqn, a mathematical formula such as

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

is produced by typing

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

in the input file; `sup` and `over` are binary operators, and braces are used for grouping. This is document formatting at a very high level, close to the language of mathematics itself, with all reference to font changes and spacing suppressed.

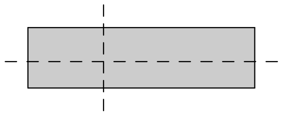
Eqn provides a single data type (let us call it the *expression*), built up recursively in context-free style: where one expression may appear, any expression may appear. This approach is common in algebra and programming languages, where its simplicity and expressiveness have long been appreciated; but Eqn was the first language to demonstrate its utility in document formatting.

Each expression is treated by Eqn as a rectangle with a *horizontal axis*, used for alignment with adjacent expressions:

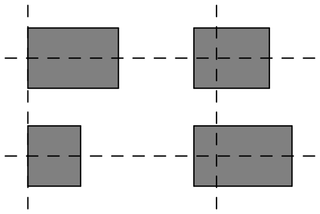
$$-\ \boxed{\frac{x^2 + 1}{4}}\ -$$

The size and rendering of the expression on the printed page are known only to the implementation, never explicitly calculated or accessed by the user. This prohibition is crucial to the maintenance of the context-free property in practice. In Lout, for example, equations, figures, tables, and arbitrary objects may be mixed together freely. This would be impossible if size information was hidden from the implementation in user calculations.

The object abstraction of Lout is a direct descendant of the Eqn expression. It employs the same context-free recursive style of construction, and each object is treated by Lout as a rectangle:



The horizontal axis, called a *row mark* in Lout, has a vertical analogue called a *column mark*, creating a valuable symmetry between horizontal and vertical. Multiple column and row marks are permitted:



so that objects are able to represent tables.

This abstraction has some limitations, the most obvious being the restriction of size calculations to rectangular bounding boxes. Non-rectangular and disconnected shapes arise naturally in figures and in the characters of fonts; the extension to them is conceptually straightforward and might help to explain some fine points of layout such as kerning. However, there are implementation and language design problems, particularly when filling non-rectangular shapes with text, and so the author chose to keep to Eqn's rectangles.

A more fundamental limitation of the object abstraction arises from the inability of recursive data types to describe cross-linked structures, which seem to require some means of naming the multiply referenced parts. Lout is obliged to introduce additional abstractions to cope with cross linking: galleys for inserting text into pages (Section 5.1), cross references (Section 6.1), and labelled points in figure drawing [6]. An abstraction closer to hypertext might form a more unified basis for these features.

## 2.2. Grammatical and lexical structure

If objects are to be constructed like mathematical expressions, the natural notation is a functional language based on operators, as in Eqn. The grammar of Lout objects is accordingly

$$
\begin{array}{lll}
\textit{object} & \rightarrow & \textit{object infixop object} \\
 & \rightarrow & \textit{prefixop object} \\
 & \rightarrow & \textit{object postfixop} \\
 & \rightarrow & \textit{noparsop} \\
 & \rightarrow & \textit{literalword} \\
 & \rightarrow & \{\ \textit{object}\ \} \\
 & \rightarrow & \textit{object object} \\
 & \rightarrow & 
\end{array}
$$

where *infixop*, *prefixop*, *postfixop*, and *noparsop* are identifiers naming operators which take 0, 1 or 2 parameters, as shown, and *literalword* is a sequence of non-space characters, or an arbitrary sequence of characters enclosed in double quotes. Ambiguities are resolved by precedence and associativity.

The last production allows a meaning for expressions such as {}, in which an object is missing. The value of this *empty object* is a rectangle of size 0 by 0, with one column mark and one row mark, that prints as nothing.

The second-last production generates sequences of arbitrary objects separated by white space, called *paragraphs*. Ignoring paragraph breaking for now, the natural meaning is that the two objects should appear side by side, and Lout's parser accordingly interpolates an infix horizontal concatenation operator (see below) between them. This operator is associative, so the grammatical ambiguity does no harm. However, the Algol-60 rule that white space should be significant only as a separator is necessarily broken by Lout in just this one place.

Algol-like languages distinguish literal strings from identifiers by enclosing them in quotes, but literals are far too frequent in document formatting for this to be viable. The conventional solution is to begin identifiers with a special character, and Lout follows Scribe [7] in using '@' rather than the '\' of troff [8] and TeX [9].

However, Lout takes the unusual step of making an initial '@' optional. The designers of Eqn apparently considered such characters disfiguring in fine-grained input like equations, and this author agrees. The implementation is straightforward: '@' is classed as just another letter, and every word is searched for in the symbol table. If it is found, it is an identifier, otherwise it is a literal. A warning message is printed when a literal beginning with '@' is found, since it is probably a mis-spelt identifier. No such safety net is possible for identifiers without '@'.

Equation formatting also demands symbols made from punctuation characters, such as + and <=. It is traditional to allow such symbols to be juxtaposed, which means that the input

```
<=++
```

for example must be interpreted within the lexical analyser by searching the symbol table for its prefixes in the order <=++, <=+, <=. Although this takes quadratic time, in practice such sequences are too short to make a more sophisticated linear method like tries worthwhile.

### 2.3. Basic structural operators

A programming language may be considered complete when it attains the power of a Turing machine, but no such criterion seems relevant to document formatting. Instead, as the language develops and new applications are attempted, deficiencies are exposed and the operator set is revised to overcome them.

Lout has a repertoire of 23 primitive operators (Figure 2.1), which has proven adequate for a wide variety of features, including equations, tables, and page layout, and so seems to be reasonably complete in this pragmatic sense. In this section we introduce the eight concatenation and mark-hiding operators. To them falls the basic task of assembling complex objects from simple ones, and they were the first to be designed and implemented.

Many of the operators of Eqn can be viewed as building small tables. A built-up fraction,

| | |
|---|---|
| *object* /*gap* *object* | Vertical concatenation with mark alignment |
| *object* //*gap* *object* | Vertical concatenation with left justification |
| *object* \|*gap* *object* | Horizontal concatenation with mark alignment |
| *object* \|\|*gap* *object* | Horizontal concatenation with top-justification |
| *object* &*gap* *object* | Horizontal concatenation within paragraphs |
| @OneCol *object* | Hide all but one column mark of *object* |
| @OneRow *object* | Hide all but one row mark of *object* |
| *font* @Font *object* | Render *object* in nominated font |
| *breakstyle* @Break *object* | Break paragraphs of *object* in nominated style |
| *spacestyle* @Space *object* | Render spaces between words in nominated style |
| *length* @Wide *object* | Render *object* to width *length* |
| *length* @High *object* | Render *object* to height *length* |
| @HExpand *object* | Expand horizontal gaps to fill available space |
| @VExpand *object* | Expand vertical gaps to fill available space |
| @HScale *object* | Horizontal geometrical scaling to fill available space |
| @VScale *object* | Vertical geometrical scaling to fill available space |
| *angle* @Rotate *object* | Rotate *object* by *angle* |
| *PostScript* @Graphic *object* | Escape to graphics language |
| @Next *object* | Add 1 to an object denoting a number |
| *object* @Case *alternatives* | Select from a set of alternative objects |
| *identifier* && *object* | Cross reference |
| *cross-reference* @Open *object* | Retrieve value from cross reference |
| *cross-reference* @Tagged *object* | Attach cross referencing tag to object |

**Figure 2.1.** The 23 primitive operators of Lout, in order of increasing precedence.

for example, has one column and three rows (numerator, line, and denominator). Numerous investigations of this kind convinced the author that operators capable of assembling the rows and columns of tables would suffice for building all kinds of objects.

The simplest objects are empty objects and literal words like metempsychosis, which have one column mark and one row mark:

metempsychosis

To place two arbitrary objects side by side, we use the infix operator |, denoting horizontal concatenation. For example,

USA |0.2i Australia

produces the object

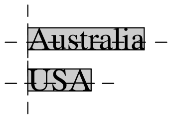
USA Australia

The row marks are merged into one, fixing the vertical position of the objects relative to each other; their horizontal separation is determined by the *gap* attached to the operator, in this case 0.2 inches. We think of the gap as part of the operator, although strictly it is a third parameter. It may be omitted, defaulting to 0i.

*Vertical concatenation*, denoted by the infix operator /, is the same apart from the change of direction:

```
Australia /0.1i USA
```

produces the object

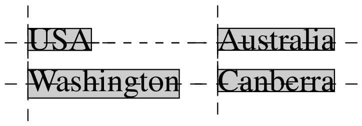



with column marks merged and a 0.1 inch gap.

Consider now what happens when horizontal and vertical are combined:

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

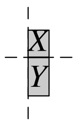
The two parameters of / now have two column marks each, and they will be merged with the corresponding marks in the other parameter, yielding the object




The 0.2i gap separates columns, not individual items in columns, so a gap attached to the second | would serve no purpose; any such gap is ignored. If the number of marks to be merged differs, empty columns are added at the right to equalize the number. The four marks protruding from the result are all available for merging with neighbouring marks by other concatenation operators. The precedence of | is higher than the precedence of /, so the braces could be omitted.

When lines of text are concatenated, it is conventional to measure their separation from baseline to baseline (mark to mark in Lout), rather than from edge to edge as above. This idea of different reference points for measurement evolved over the years into a system of six *gap modes* (Figure 2.2), expressed by appending a letter to the length. For example, |0.2i is an abbreviation for |0.2ie, meaning 0.2 inches measured from edge to edge; |0.3ix produces a 0.3 inch gap measured from mark to mark and widened if necessary to prevent overstriking; and |2.5it places its right parameter 2.5 inches from the current left margin, irrespective of the position of the left parameter. There is also a choice of eleven units of measurement (inches, centimetres, multiples of the current font size, etc.), the most interesting being the r unit: one r is the column width minus the width of the following object, so that |1rt produces sufficient space to right justify the following object, and |0.5rt to center it. These features implement spacings needed in practice rather than suggested by theory. They work with all five concatenation operators, horizontal and vertical.

When we construct a built-up fraction, the result has three row marks, but only the second should be visible outside the object:




This is a common problem, and accordingly a @OneRow operator was introduced for hiding all but one of the row marks of its parameter. Normally, the first mark is the survivor, but a later mark can be chosen by prefixing ^ to the preceding concatenation operator:

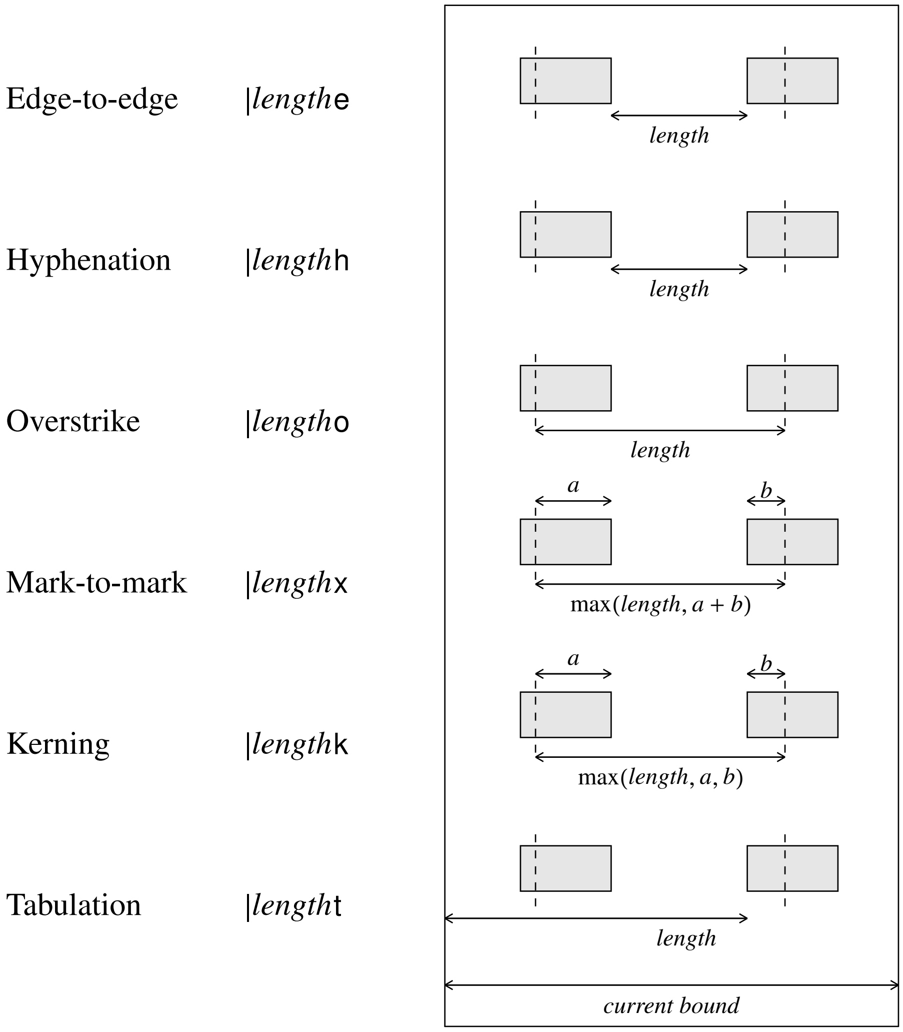



**Figure 2.2.** The six gap modes (*length* is any length). Hyphenation mode has an extra property not shown here.

```
@OneRow { X  ^/2p  @HLine  /2p  Y }
```

has the desired result, where 2p is two points and @HLine is an easy combination of Lout's graphics operators. A similar operator, @OneCol, hides column marks.

A variant of / called // is provided which performs vertical concatenation but ignores all column marks and simply left-justifies its two parameters:

```
Heading  //0.1i
A  |0.2i  B  /0.1i
C  |  D
```

has result

Heading
A    B
C    D

showing that spanning columns in tables motivate the inclusion of this operator. There is an analogous || operator. The author would have preferred to leave out these operators, since they complicate the implementation, and it is interesting to examine the prospects of doing so.

The // operator is formally redundant, because in general the expression x // y can be replaced by

```
@OneCol { | x }  /
@OneCol { | y }
```

for any objects x and y. By concatenating an empty object at the left of x and hiding all but that empty object's column mark, we effectively shift x's column mark to its left edge. The same goes for y, so the / operator has just one column mark to merge, at the extreme left, and its effect is indistinguishable from //.

Unfortunately, if y consists of two rows separated by /, as in the example above, both rows must be placed inside the @OneCol, and the table cannot be entered in the simple row-by-row manner that non-expert users naturally expect. Another advantage of // is that its left parameter can be printed before its right parameter is known; this is important when the left parameter is an entire page.

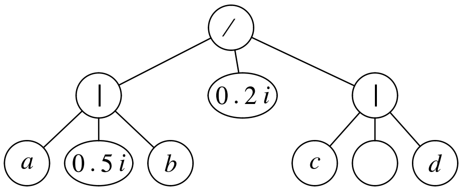
The fifth and final concatenation operator, &, is an explicit version of the horizontal concatenation operator interpolated when objects are separated by white space. It is formally identical to | except for taking higher precedence and being subject to replacement by //1vx during paragraph breaking (Section 2.5).

### 2.4. Implementation of objects and concatenation

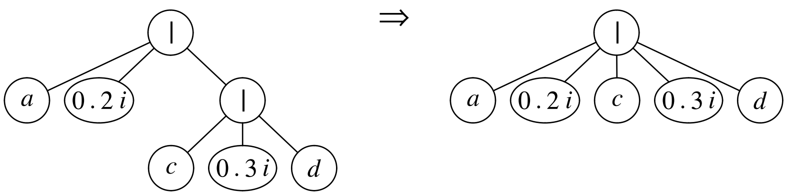
In this section we discuss the implementation of objects and concatenation, and especially mark alignment. The first step is to use an operator precedence parser to convert input such as

```
a  |0.5i  b  /0.2i  c  |  d
```

into parse trees such as




Missing objects are replaced by empty objects, and sequences of concatenation operators are consolidated:



to make manifest their associativity and reduce the depth of the tree for efficiency later.

The required semantic information is the size of each subobject, consisting of four integers: width to left and right of the distinguished column mark, and height above and below the distinguished row mark. These numbers are always non-negative in Basser Lout, but this restriction is unnecessary and should be dropped.

For the leaves, which are simple words, the numbers are obtained from font tables. For the higher levels we apply recursive rules. Suppose that $hgap(x, g, y)$ returns the desired distance between the column marks of objects $x$ and $y$ when they are separated by gap $g$: $right(x) + length(g) + left(y)$ when the gap mode is edge-to-edge, the larger of $length(g)$ and $right(x) + left(y)$ when the mode is mark-to-mark, and so on. Given an object

$$X = x_1 \ |g_1 \ \ldots \ \hat{}|g_{i-1} \ x_i \ \ldots \ |g_{n-1} \ x_n$$

we may calculate its size as follows:

$$\begin{aligned}
left(X) &= left(x_1) + hgap(x_1, g_1, x_2) + \ldots + hgap(x_{i-1}, g_{i-1}, x_i) \\
right(X) &= hgap(x_i, g_i, x_{i+1}) + \ldots + hgap(x_{n-1}, g_{n-1}, x_n) + right(x_n) \\
above(X) &= above(x_1) \uparrow \ldots \uparrow above(x_n) \\
below(X) &= below(x_1) \uparrow \ldots \uparrow below(x_n)
\end{aligned}$$

where $\uparrow$ returns the larger of its two parameters. Similar formulas are easily derived for the other operators.

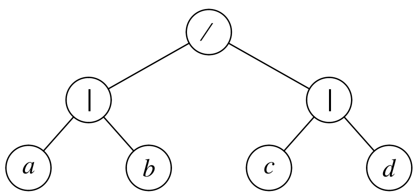
For purposes of exposition we will now make the simplifying assumptions that all gaps are 0i, all column marks lie at the left edge, and all row marks lie at the top edge. Then the size of each object can be expressed by just two numbers, width and height, and the four formulas reduce to

$$\begin{aligned}
width(x_1 \mid \ldots \mid x_n) &= width(x_1) + \ldots + width(x_n) \\
height(x_1 \mid \ldots \mid x_n) &= height(x_1) \uparrow \ldots \uparrow height(x_n)
\end{aligned}$$

The corresponding formulas for vertical concatenation are

$$\begin{aligned}
width(x_1 \,/\, \ldots \,/\, x_n) &= width(x_1) \uparrow \ldots \uparrow width(x_n) \\
height(x_1 \,/\, \ldots \,/\, x_n) &= height(x_1) + \ldots + height(x_n)
\end{aligned}$$

According to these formulas, the height of



is

$$[height(a) \uparrow height(b)] + [height(c) \uparrow height(d)]$$

which is correct, but for width they yield

$$[width(a) + width(b)] \uparrow [width(c) + width(d)]$$

which is not, since it does not take the merging of column marks into account. The asymmetry between horizontal and vertical has come about because the row entries, such as $a$ and $b$, are adjacent in the tree, but the column entries, such as $a$ and $c$, are not. It would be possible to solve this cross-linking problem by augmenting the size information stored in each node to record the number of marks and the size of each, but the author has preferred the following method which makes structural changes to the tree instead.

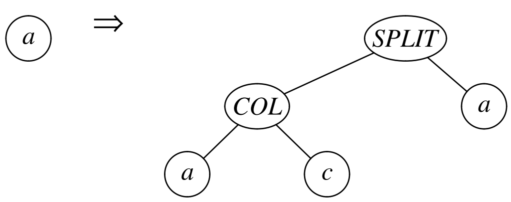
If $a$ and $c$ share a column mark, they each might as well have width $width(a) \uparrow width(c)$, since all width calculations apply to entire columns. Accordingly, we introduce a new operator, *COL*, defined by

$$width(x_1 \; COL \; \ldots \; COL \; x_n) = width(x_1) \uparrow \ldots \uparrow width(x_n)$$

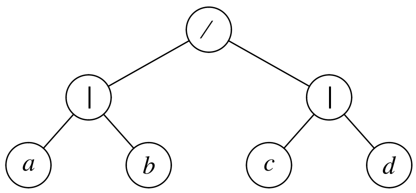
and replace both $a$ and $c$ by $a \;\; COL \;\; c$. To prevent *COL* operators from disturbing height calculations, we define a binary operator called *SPLIT* by

$$width(x \; SPLIT \; y) = width(x)$$
$$height(x \; SPLIT \; y) = height(y)$$

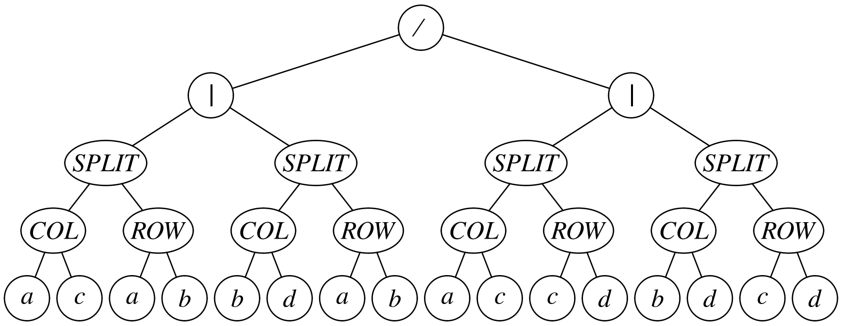
which switches height and width calculations onto different subtrees. Then the transformation




widens $a$ to $width(a) \uparrow width(c)$ without affecting its height; it is applied to every object that shares its column mark with at least one other object. A similar transformation involving a *ROW* operator deals with shared row marks. The effect on our little table is to replace




by




In fact, common subexpressions are identified (trivially) and the result is a directed acyclic graph; each affected leaf has two parents, one for width and one for height; and each *COL* or *ROW* node has one parent and one child for each object lying on the corresponding mark. The data structure roughly doubles in size, and this occurs only rarely in practice.

This method can cope with any legal input, including

```
{ a // c | d } | { b / e }
/ { f / i } | { g | h // j }
```

which produces overlapping spanning columns:

<table>
<tr><td colspan="2">a</td><td>b</td></tr>
<tr><td>c</td><td>d</td><td>e</td></tr>
<tr><td>f</td><td>g</td><td>h</td></tr>
<tr><td>i</td><td colspan="2">j</td></tr>
</table>

The boxes have been added to clarify the structure. The width of this object is formally

$$((width(a) \uparrow (x + y)) + z) \uparrow (x + ((y + z) \uparrow width(j)))$$

where

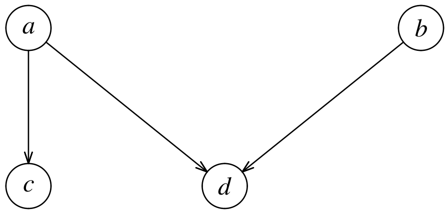
$$x = width(c) \uparrow width(f) \uparrow width(i)$$

$$y = width(d) \uparrow width(g)$$

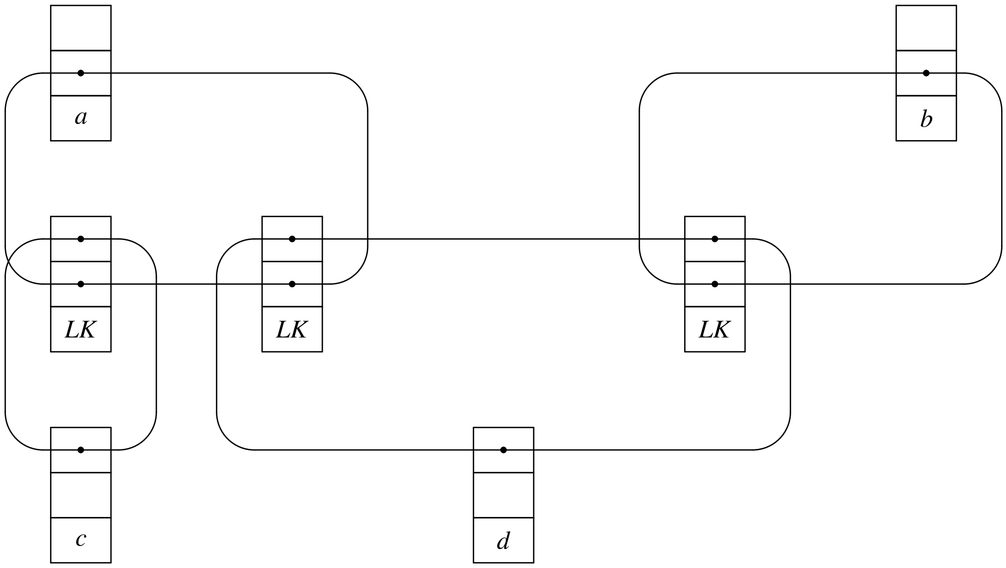
$$z = width(b) \uparrow width(e) \uparrow width(h)$$

It seems clear that $y$ at least must appear twice in any expression for the width of this object made out of simple addition and maxing operations, showing that an ordinary tree structure is insufficient for overlapping spanning columns. The Basser Lout interpreter actually rejects such structures, owing to the author's doubts about the implementability of *Constrained* and *AdjustSize* (Section 5.3) on them; but with hindsight this caution was unnecessary.

The directed acyclic graph is ordered in the sense that the order of the edges entering and leaving each node matters. The structure is highly dynamic, and traversals both with and against the arrows are required. After a few ad-hoc attempts to extend the usual tree representation had failed, the author developed a representation based on doubly linked lists of records denoting links, whose flexibility more than compensated for the somewhat excessive memory consumption. For example,




is represented by




where *LK* tags a record representing a link. The first list in any node contains all the incoming links, the second contains the outgoing ones. The node serves as the header for both lists. The required operations reduce to simple appends, deletes, and traversals of doubly linked lists, all having small constant cost. There is a highly tuned memory allocator, and care is taken to dispose of each node when the last incoming link is deleted, so that there is no need for garbage collection.

In normal use the number of nodes at higher levels of the dag is small in comparison with the leaves and their incoming links, so we may estimate the space complexity at about 60 bytes per input word (20 bytes per link, 40 per leaf node). Careful optimization could easily halve this, but since memory is reclaimed after printing each page there is little need.

## 2.5. Context-sensitive attributes of objects

Although we are free to place any object in any context, the context must influence the appearance of the object, since otherwise

A short paragraph of text.

could not appear in a variety of fonts, column widths, etc. This influence cannot take the purely static form that block-structured languages use to associate values with identifiers, for then an operator could not influence the appearance of its parameters; and a state variable solution is not compatible with the overall functional design.

The information needed from the context seems quite limited, comprising the font family, face, and size to use, the style of paragraph breaking required, how much space to substitute between the words of paragraphs, and how much horizontal and vertical space is available to receive the object. These four items constitute the so-called ‘style information’ of Lout. As graphics rendering hardware improves, the style information will probably grow to include colour and texture information.

The way to deal with fonts at least is very clear:

```
{ Times Slope 12p } @Font { Hello, world }
```

should have result

*Hello, world*

Lout also provides @Break and @Space symbols for controlling the paragraph breaking and space styles mentioned above. These work in the same way, returning their right parameters in the style of their left. The implementation is very simple: one merely broadcasts the style information down into the parse tree of the right parameter. A font, for example, is converted to an 8-bit internal name and stored in each leaf, while a breaking style is stored in the root node of each paragraph.

The same language design can be used for available width and height, only here the implementation is much more demanding:

```
2i @Wide {
(1)  |0.1i  An example
containing a small
paragraph of filled text.
}
```

is guaranteed to be two inches wide:

(1) An example containing a small paragraph of filled text.

One must calculate that 1.9 inches minus the width of (1) is available to the paragraph, and break it accordingly; Basser Lout does this in two stages. In the first, upward-moving stage, widths are calculated using the formulae of Section 2.3, which assume that available space is infinite. If the upward movement reaches a *WIDE* node, corresponding to a @Wide operator, and the calculated width exceeds that allowed, a second, downward-moving stage is initiated which attempts to reduce the width by finding and breaking paragraphs. This second stage is quite routine except at | nodes, whose children are the columns of a table. It is necessary to apportion the available width (minus inter-column gaps) among the columns. Basser Lout leaves narrow columns unbroken and breaks the remaining columns to equal width, using up all of the available space.

The size of an object is not clearly determined when the upward-moving size is less than the downward-moving available space, and the object contains constructs that depend on available space (e.g. right justification). For example, in

```
2i @Wide { Heading // a |1rt b }
```

it seems natural to assign a width of two inches to the subobject a |1rt b because of the right justification, but it would be equally plausible if the width of Heading was assigned to the subobject instead. The author is conscious of having failed to resolve this matter properly; an extra operator for controlling available space is probably necessary.

The actual paragraph breaking is just a simple transformation on the parse tree; the real issue is how to describe the various styles: ragged right, adjusted, outdented, and so on. Their diversity suggests that they should somehow be defined using more basic features; but then there are algorithms for high-quality paragraph breaking, which presumably must be built-in. This dilemma was not clearly grasped by the author in 1985, and he included a built-in paragraph breaker, with the @Break operator selecting from a fixed set of styles. A much better solution based on galleys will be given in Section 5.5, but, regrettably, it is not implemented.

## 3. Definitions

The need to provide a means of packaging useful pieces of code for easy repeated use was recognised in the very earliest programming languages. This need is even more acute in document formatting, if that is possible, because the majority of users are not programmers and do not understand the code they invoke.

### 3.1. Operators

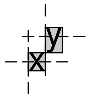
It is evident from the example of Eqn that user-defined operators are needed that mimic the primitive ones in taking objects as parameters and returning objects as results. For example, to define a superscript operator so that

```
2 sup n
```

appears as $2^n$, the following operator definition may be used:

```
def sup
   precedence 50
   associativity right
   left x
   right y
{
   @OneRow { | {-2p @Font y} ^/0.5fk x }
}
```

The sup operator has precedence 50, is right associative, takes two objects as parameters passed on the left and right, and returns the object between braces as result. This object has the structure



but with the first row mark hidden by the @OneRow operator, and y two points smaller than it would otherwise have been. The length 0.5f specifies half the current font size; Figure 2.2 describes the k gap mode. In the Eq equation formatting package [10] the equation as a whole is set in italic font, and 2 is an identifier whose body contains a font change back to Roman. The digits 0 to 9 are classed as punctuation characters, permitting 234 for example to be interpreted as a sequence of three identifiers.

These definitions are easily implemented by a standard symbol table and an operator precedence parser. Algol block structure with the usual scope rules was adopted as a matter of course.

Operators are limited to at most two parameters, left and right, and the parameters cannot be given default values. *Named* parameters solve both problems:

```
def @Preface
   named @Tag {}
   named @Title { Preface }
   right @Body
{
   Bold @Font @Title
   //0.3v  @Body
}
```

The default value appears just after the parameter's declaration, between braces. Invocations have a natural syntax:

```
@Preface
   @Title { About this book }
{
   Few observers would have supposed in 1984, that ...
}
```

with the actual named parameters following directly after the operator, before any right parameter. In this example, @Tag will receive its default value, and a less expert user could safely omit the @Title parameter as well.

Lout permits named parameters to have parameters, a feature with applications to bibliographic databases, running headers, and other places where a format has to be supplied before content is known. One could go further and provide a complete lambda calculus, with functions as first-class objects, provided care was taken not to intimidate the non-expert user.

### 3.2. Recursion and page layout

Design and implementation should proceed together in exploratory projects, since otherwise the design too easily becomes unrealistic. Sometimes the implementation does more than its designer intended. The author wrote the following purely as a testing scaffold:

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

Only afterwards did he realize its significance: the concept of a page had been defined outside the implementation, removing the need for commands for setting page width and height, margins, and so on.

Defining a sequence of pages is harder, since their number is not known in advance. A simple version of this same problem is afforded by the leaders found in tables of contents:

Chapter 7 .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. 53

This seemed to require recursion, specifically the definition

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

Note that both .. and @Leaders are objects, so the two spaces separating them are significant. No base case is given, and indeed we have no boolean or conditional operators with which to express it; but we can adopt the implicit base ‘if space is not sufficient, delete @Leaders and any preceding space’. Then the expression

```
4i @Wide { Chapter 7  @Leaders  53 }
```

will produce the object shown above. It is hard to see how this base could be made explicit, without violating the general principle of keeping all size information internal. In the implementation, @Leaders remains unexpanded while sizes are being calculated; then it is treated similarly to a receptive symbol, with its body as an incoming galley (Section 5.2).

With this settled, it is now clear how to define a document which is a numbered sequence of pages. Let @Next be a prefix operator which returns its parameter plus one. Then

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

when invoked in the expression @PageList 1, has for its result the potentially infinite object


@PageList 3

Similarly, we may define @FootSect like this:

```
def @FootSect
{
   def @FootList
      right @Num
   {
      @FootPlace
      //1v
      @FootList @Next @Num
   }

   1i @Wide @HLine
   //1v
   @FootList 1
}
```

so that an invocation of @FootSect produces

```
@FootPlace
@FootPlace
@FootPlace
...
```

The expansion process is very similar to a BNF derivation, and would be attempted only on demand.

Clearly, deciding which expansions to take and replacing @TextPlace and @FootPlace by the appropriate actual text will not be easy; this is the subject of Section 5.1. The important point for now is that we have here a very simple and flexible method of specifying the layout of pages, which requires no specialized language features.

### 3.3. Modules

It is well accepted that the visibility of symbols is not adequately controlled by Algol block structure. The author is aware of several major problems of this kind in document formatting.

One problem is that some symbols should be visible only within restricted parts of a document. For example, we naturally expect equation formatting to be accomplished like this:

```
surrounding text
@Eq { {x sup 2  +  1} over 4 }
surrounding text
```

with the symbols sup, over, etc., visible only within the equation, not in the surrounding text.

It seems natural to define these symbols within @Eq, since they are local to equations. It only remains then to decree that symbols local to @Eq are to be visible within its actual right parameter, and this is done by replacing the right formal parameter with a *body* parameter:

```
export sup over
def @Eq
   body @Body
{
   def sup ...
   def over ...

   Slope @Font @Body
}
```

The export clause lists the identifiers which are permitted to be visible outside their usual range, the body of @Eq; and the body declaration imports them into (makes them visible within) the actual right parameter of each invocation of @Eq. This arrangement has proven very convenient for defining a variety of special-purpose packages.

Another problem arises when global symbols, such as the ones used for headings and paragraph separators, call on values that the non-expert user will need to modify, such as the initial font or paragraph indent. These values are like parameters of the document as a whole, so it is natural to try this:

```
export @Heading @PP ...
def @BookLayout
   named @InitialFont { Times Base 12p }
   named @InitialBreak { adjust 14p }
   named @ColumnWidth { 6i }
   ...
{
   def @Heading ...
   def @PP ...
}
```

Now @Heading and @PP may invoke @InitialFont and the other parameters. To make @Heading and @PP visible throughout the document, we need only add a body parameter to @BookLayout and present the entire document as

```
@BookLayout
   @InitialFont { Helvetica Base 10p }
   @InitialBreak { adjust 12p }
{
   The document.
}
```

but for practical reasons given below we prefer not to enclose the entire document in braces. Instead, we write

```
@Use { @BookLayout
   @InitialFont { Helvetica Base 10p }
   @InitialBreak { adjust 12p }
}
The document.
```

which has the same effect: @Use makes the exported symbols of @BookLayout visible for the remainder of the document, and is permitted only at the beginning.

The third feature that affects visibility, and which will prove useful for cross referencing (Section 6.1), is the @Open symbol. It makes the exported symbols of its left parameter visible within its right parameter, and is therefore similar to the Pascal with statement.

It could be argued that Lout is over-supplied with these visibility modifying features: the body parameter, @Use and @Open do not seem sufficiently different from each another. The @Open symbol is the most general, being capable of replacing the other two. For example,

```
@Use { x }
@Use { y }
Body of document
```

can be replaced by

```
x @Open {
y @Open {
Body of document
}}
```

and, taking the @Eq symbol above as example, we could eliminate its body parameter, add

```
def @Body right x { Slope @Font x }
```

to the exported definitions of @Eq, and replace

```
@Eq { object }
```

by

```
@Eq @Open { @Body { object } }
```

If @Eq is a galley (Section 5.1), @Body must take over that function. But one would not want to write these clumsy expressions in practice, and the enclosure of large quantities of input in extra braces could cause Basser Lout to run out of memory (Section 5.4).

A quite separate kind of visibility problem arises when expert users wish to define an object or operator for repeated use within, say, equations:

```
def isum { sum from i=1 to n }
```

As it stands this can only be placed within the @Eq package itself, where sum and the other symbols are visible, but it is not desirable to modify the source code of a standard package. Lout provides an import clause to solve this problem:

```
import @Eq
def isum { sum from i=1 to n }
```

may appear after @Eq is defined, and it will make the exported symbols of @Eq visible within the body of isum. This feature complicates the treatment of environments (Section 3.4), and even introduces an insecurity, when isum is invoked outside an equation. A simpler approach would be to allow only one symbol in an import clause, and treat the following definition exactly like a local definition of that symbol; but then it would not be possible to define symbols using the resources of more than one of the standard packages.

### 3.4. Implementation of definitions

Input is processed by a hybrid parser which employs operator precedence for objects and simple recursive descent for the headers of definitions. A symbol table stores the body of each definition as a parse tree, except for macros which are lists of tokens, and manages the usual stack of static scopes, accepting *PushScope* and *PopScope* operations as the parser enters and leaves scope regions, including actual body parameters and the right parameter of the @Open operator.

As the parse proceeds, a complete call graph is constructed, recording, for each symbol, which symbols are invoked within its body. Immediately after the last definition is read, the transitive closure of the call graph is computed, and used to determine whether each non-parameter symbol is recursive or receptive (Section 5.1), and whether each parameter is invoked exactly once or not.

Purely functional systems may evaluate symbol invocations in applicative order (where parameters are evaluated before substitution into bodies), or in normal order (substitution before evaluation), and they may also share the value of a parameter among all uses of it. But in Basser Lout, the presence of context-sensitive style information (Section 2.5) forces normal order evaluation and prevents sharing of parameter values.

To evaluate an unsized object (pure parse tree), its *environment*, the equivalent of the stack frames in Algol-like languages, must be available, containing the actual values of all formal parameters that are visible within the unsized object. Environment handling is a well-known implementation technique, so it will be discussed only briefly here.

Environments are extra subtrees hung from the objects they refer to. This organization makes excellent use of the ordered dag to permit environments to be shared, and deleted when the last reference to them is removed. Several optimizations have been implemented. Actual parameters known to be invoked only once are moved in from the environment, not copied; copying could lead to quadratic time complexity. Actual parameters of the form @Next *object* receive an applicative pre-evaluation which prevents long chains of @Next symbols from forming during the generation of large page numbers. Some environments which provably contribute nothing are deleted, most notably when a symbol invocation has no symbols within its actual parameters and no import list, so that only the environment of its body need be kept; this saves a great deal of space when objects with environments are written to auxiliary files (Section 6.1).

## 4. Implementation of the functional subset

The objects and definitions of Lout are very similar to those found in other functional languages, and they form a natural subset of the language. So we pause here and present an overview of the Basser Lout object evaluation algorithm.

The problem is to take an unsized object (pure parse tree), its environment (Section 3.4), and its style (Section 2.5), and to produce a PostScript file for rendering the object on an output device. This file is essentially a sequence of instructions to print a given string of characters in a given font at a given point.

Before the algorithm begins, the parse tree must be obtained, either by parsing input or by copying from the symbol table. Afterwards the data structure must be disposed. The algorithm proper consists of five passes, each a recursive traversal of the structure from the root down to the leaves and back.

*1. Evaluation of unsized objects.* On the way down, calculate environments and replace non-recursive, non-receptive symbols by their bodies (Section 3.4); broadcast fonts to the leaves, and paragraph breaking and spacing styles to the paragraph nodes. On the way back up, delete *FONT*, *BREAK*, and *SPACE* nodes, and insert *SPLIT*, *COL*, and *ROW* nodes (Section 2.3).

*2. Width calculations and breaking.* Calculate the width of every subobject from the bottom up. As described in Section 2.3, *WIDE* nodes may trigger object breaking sub-traversals during this pass.

*3. Height calculations.* Calculate the height of every subobject, from the bottom up.

*4. Horizontal coordinates.* Calculate the horizontal coordinate of each subobject from the top down, and store each leaf's coordinate in the leaf.

*5. Vertical coordinates and PostScript generation.* Calculate the vertical coordinate of every subobject from the top down, and at each leaf, retrieve the character string, font, and horizontal coordinate, and print the PostScript instruction for rendering that leaf.

Figure 5.1 gives the amount of code required for each pass. Symmetry between horizontal and vertical is exploited throughout Basser Lout, and passes 2 and 3, as well as 4 and 5, are executed on shared code.

The author can see no simple way to reduce the number of passes. The introduction of horizontal galleys (Section 5.5) would remove the need for the object breaking transformations within this algorithm that are the principal obstacles in the way of the merging of passes 2 and 3.

## 5. Galleys

With objects and definitions under control, the author faced the problem of getting body text, footnotes, floating figures and tables, references, index entries, and entries in the table of contents into their places. The resulting investigation occupied three months of full-time design work, and proceeded approximately as described in Section 5.1; the implementation occupied the years 1987-89.

### 5.1. The galley abstraction

Let us take the footnote as a representative example. At some point in the document, we wish to write

```
preceding text
@FootNote { footnote text }
following text
```

and we expect the formatter to remove the footnote from this context and place it at the bottom of the current page, possibly splitting some or all of it onto a following page if space is insufficient.

An object appears in the final document at the point it is invoked, but this basic property does not hold for footnotes: the point of invocation and the point of appearance are different. In some way, the footnote is attached to the document at both points, introducing a cross linking (Section 2.1) that cannot be described in purely functional terms.

Since the interpretation of any object depends on an environment and style inherited from the context, the first question must be whether the footnote inherits them through the invocation point or through the point(s) of appearance.

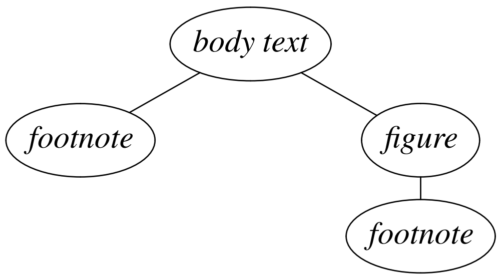
If symbols are to be interpreted statically as heretofore, then environments must be inherited through the invocation point alone. Dynamic inheritance through the point of appearance is enticing in some ways: it might replace the body parameter, and it might help with automatic

| | | |
|---|---|---|
| 1. | Initialization | 200 |
| 2. | Memory allocation, ordered dag operations | 400 |
| 3. | Lexical analysis, macros, file handling | 1,350 |
| 4. | Parsing of objects and definitions | 1,150 |
| 5. | Symbol table and call graph | 600 |
| 6. | Evaluation of pure parse trees | 1,650 |
| 7. | Reading, storing, and scaling of fonts | 600 |
| 8. | Cross references and databases | 1,000 |
| 9. | Width and height calculations, and breaking | 700 |
| 10. | *Constrained* and *AdjustSize* | 700 |
| 11. | Transfer of sized objects into galley tree | 450 |
| 12. | Galley flushing algorithm | 1,500 |
| 13. | Coordinate calculations and PostScript output | 700 |
| 14. | Debugging and error handling | 1,200 |
| | | 12,200 |

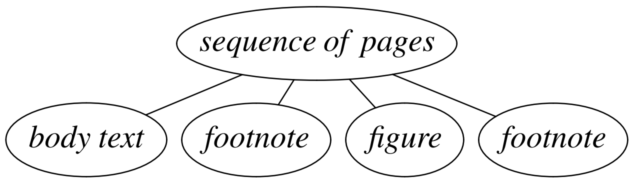
**Figure 5.1.** Major components of the Basser Lout interpreter, showing the approximate number of lines of C code.

numbering, since the number of a footnote is known only at the point of appearance; but the implementation problems are severe, and static inheritance seems much simpler and more comprehensible to the user. Style, at least its available width and height part, must of necessity be inherited through the point of appearance. For consistency, the entire style should be inherited in this way. There is a suggestive analogy here with actual parameters, which have a point of invocation from which they inherit an environment, and a point of appearance within the body of the enclosing definition, from which they inherit a style. It may be possible to treat a footnote as the actual parameter of some symbol, therefore, although the details seem very obscure.

But the most profound consequence of having two types of attachment point is that it leads to two distinctive tree structures. Considering invocation points only leads to static trees like this one:




which shows that the body text contains a footnote and a figure, the latter itself containing a footnote. Considering points of appearance only gives a completely different, dynamic tree:



The tree can be deeper, for example with sections appearing within chapters which appear within the body text, which appears within the final sequence of pages. Document formatting languages generally shirk the issues raised by this dual tree structure, by making the dynamic tree built-in, by limiting one or both trees to two levels, and so on, providing a classic example of the impoverishing effect of failing to permit language features to attain their natural level of generality.

We are thus led to propose a second abstraction for document formatting, which we name the *galley* in recognition of its similarity to the galleys used in manual typesetting. A galley consists of an object (such as a footnote) together with a sequence of places where that object may appear (such as the bottoms of the current and following pages). Splitting occurs quite naturally when space at any place is insufficient to hold the entire object.

In Lout, a footnote galley and its place of appearance are defined as follows:

```
def @FootPlace { @Galley }

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

The @FootPlace symbol contains the special symbol @Galley, indicating that it is a point of appearance for a galley. By placing invocations of @FootPlace at the bottoms of pages, as in Section 3.2, we define the desired points of appearance for footnotes. Symbols whose body contains @Galley either directly or indirectly are called receptive symbols, meaning receptive to galleys, and they are expanded only on demand. The effect of the into clause is to make each invocation of @FootNote a galley whose object is the result of the invocation in the usual way, and whose sequence of points of appearance is specified by the into clause; in this example, the sequence of all @FootPlace symbols following the invocation point.

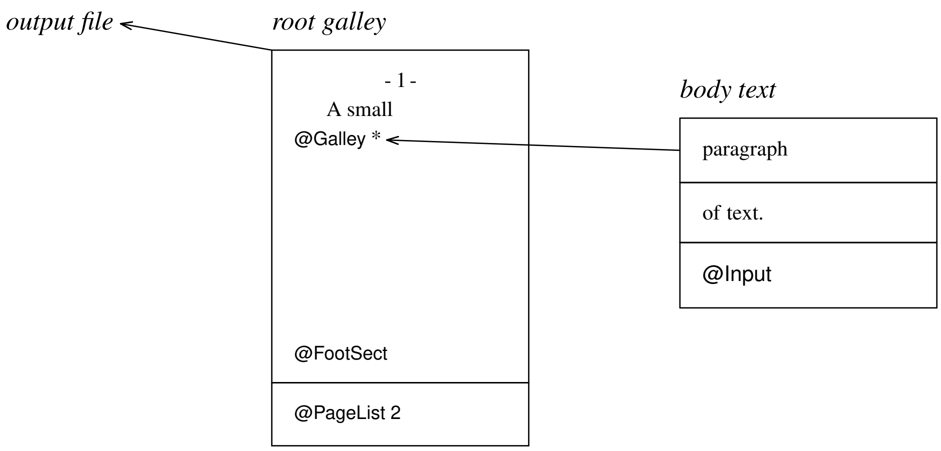
Lout permits galleys to be invoked within other galleys to arbitrary depth, so that one may have footnotes within figures within the body text galley, for example, creating arbitrary static trees. Receptive symbols like @FootPlace may appear within any galley, creating arbitrary dynamic trees as well. The root of the dynamic tree, which would normally consist of the sequence of pages of the complete assembled document, is considered to be a galley whose point of appearance is the output file. Points of appearance may be preceding or following the invocation point; entries in tables of contents are the main users of preceding.

The galley abstraction is adequate for all of the applications listed at the beginning of this section, except that there is no provision for sorting index entries and references. Sorting of galleys has been added to Lout as a built-in feature, invoked by adding a special @Key parameter to the galleys, and using its value as the sort key. The author was at a loss to find any other way, or any useful generalization of this feature. Its implementation will be discussed in Section 6.2.

### 5.2. The galley flushing algorithm

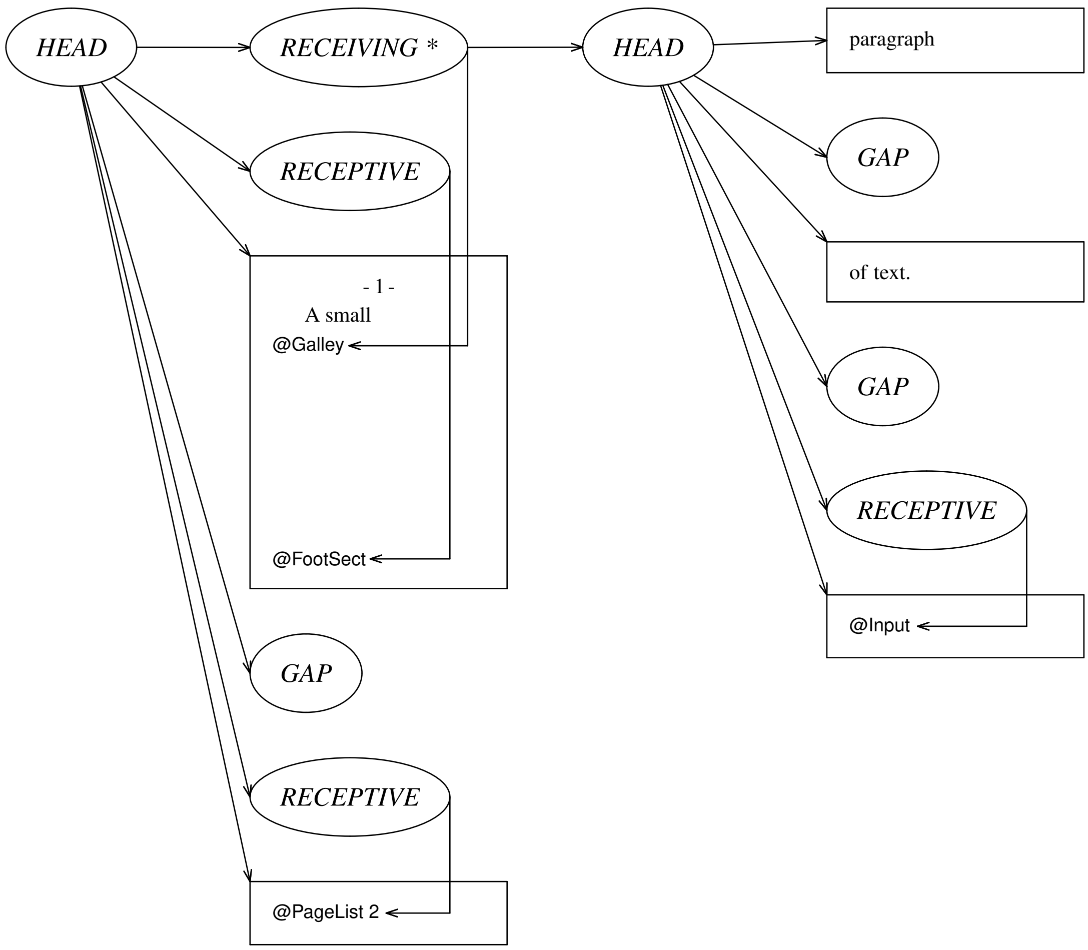
Galley components are promoted one by one into the point of appearance in the dynamic parent galley, then carried along with it, ultimately to the root galley and the output file. This process is called *galley flushing*: the galleys are rivers running together to the sea, and each component is a drop of water.

Here is a snapshot of a small dynamic tree, based on the @PageList definitions of Section 3.2:




The components of the body text galley are lines, except for the special receptive symbol @Input which is a placeholder for as yet unread input (Section 5.4). The components of the root galley are pages, except for the concluding unexpanded invocation of @PageList, which is an inexhaustible source of more pages, expanded on demand.

The concrete data structure used by Basser Lout permits the galley flushing algorithm to navigate the dynamic tree and find significant features quickly:



Each galley has a *HEAD* node whose children are its component objects, separated by *GAP* nodes recording the inter-component gaps.

Each component is preceded by zero or more *galley index nodes* of various types. Every receptive symbol has a *RECEPTIVE* index pointing to it, so that it can be found without searching through its component. If the symbol is currently the target of a galley, it has a *RECEIVING* index instead which is also linked to the incoming galley. Galleys that are currently without a target are linked to the dynamic tree by *UNATTACHED* galley indexes, either just after their most recent target if there has been one, or else at their point of invocation.

Each galley should be thought of as a concurrent process, although the implementation in C uses coroutines implemented by procedures. A galley may promote its first component only if it has a target, sufficient space is available at the target to receive the component, and the component contains no receptive symbols. This last condition seems to be the key to galley synchronization: it forces a bottom-up promotion regime, preventing pages from flushing to output before text flushes into them, for example.

Each galley contains a number of binary semaphores, shown as asterisks in our snapshots when set. At any given moment, a galley process is either running or else is suspended on one of its own semaphores. The *HEAD* node contains a semaphore which is set when the galley has tried to find a target and failed. Each receptive symbol has a semaphore which is set when that symbol is preventing the first component from being promoted.

For example, in the snapshot at the beginning of this section, the root galley is suspended on the @Galley symbol, but the text galley is running. It will suspend on the @Input symbol after the first two components are promoted.

Every galley *G*, be it a list of pages, body text, a footnote, or whatever, executes the following algorithm in parallel with every other galley:

1. Initially *G* is unattached. Search forwards or backwards from its *UNATTACHED* index as required, to find a receptive symbol *S* which can expand to reveal a target for *G*.

2. If no *S* can be found, suspend on the attachment semaphore. Resume later from step 1.

3. Expand *S* to reveal the target of *G*. Preserve *S*'s semaphore by moving it to the first receptive symbol within the expansion of *S*.

4. Calculate the available width and height at the target, and if *G* is still a pure parse tree, use the environment attached to *G* and the style information from the target to evaluate *G* as in Section 4.

5. Examine the components of *G* one by one. For each component there are three possibilities:

*ACCEPT.* If the component fits into the available space, and has no other problems, then promote it into the target. If this is the first component promoted into this target, and *G* is a forcing galley (Section 5.4), delete every receptive symbol preceding the target in the parent galley. If *G* is the root galley, render the component on the output file and dispose it;

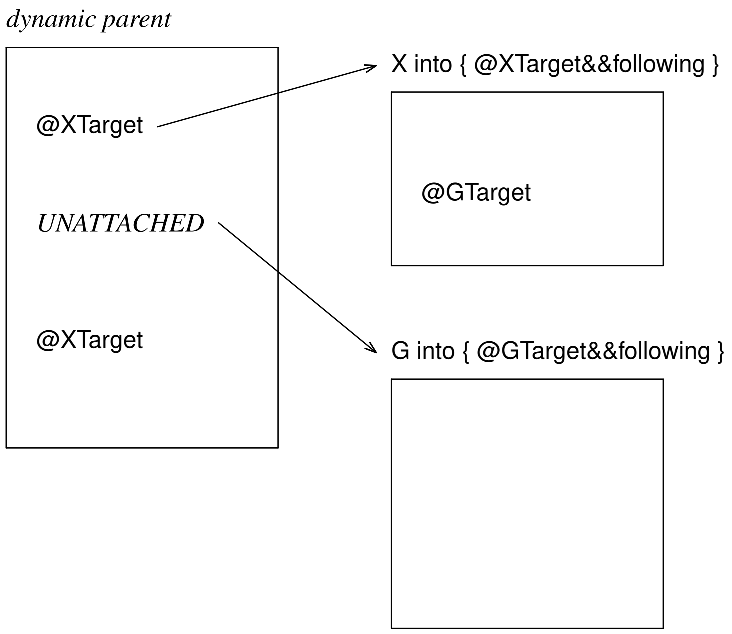
*REJECT.* If the component is too large for the available space, or a *FOLLOWS* index (described below) forbids its promotion into this target, then detach *G* from the target. If this was the first component at this target, *S* has been a complete failure, so undo step 3 (Basser Lout is not able to undo step 4); otherwise delete the target. Return to step 1 and continue immediately;

*SUSPEND.* If the component contains a receptive symbol, it cannot be promoted yet. If this symbol is the target of a galley that was written to an auxiliary file on a previous run, read in that galley and flush it. Otherwise suspend on the receptive symbol's semaphore; resume later from step 4.

6. Terminate when the galley is empty.

At various points in this algorithm, receptive symbols (and their semaphores) are deleted in the dynamic parent galley, possibly permitting it to resume flushing. When this happens, Basser Lout resumes the parent immediately after *G* suspends or terminates. Also, whenever a component is promoted, any child galleys connected to it by *UNATTACHED* indexes must be resumed, since these galleys may be able to find a target now. A good example of this situation occurs when a line of body text with one or more footnotes is promoted onto a page. Basser Lout gives priority to such children, suspending *G* while each is given a chance to flush.

Basser Lout searches for the first target of *G* only in regions of the dynamic tree that will clearly precede or follow *G*'s invocation point in the final printed document, whichever is specified in the into clause; subsequent targets are sought later in the same galley as the first. An exception to this rule, whose necessity will be made clear later, is that a first following target will be sought within a dynamic sibling galley preceding *G*'s invocation point:



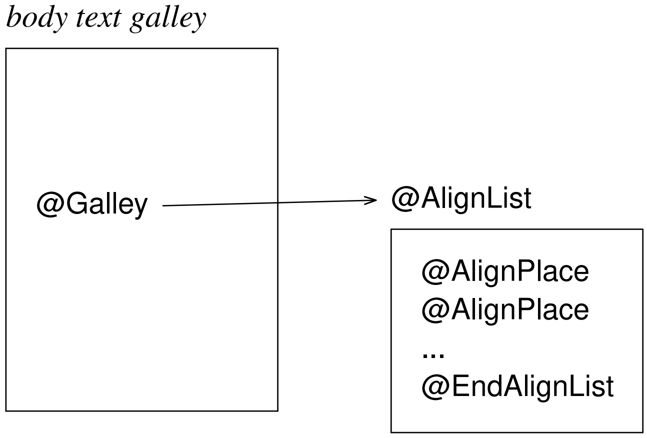

Here *G* will find the @GTarget target within *X*. This is dangerous, since if the first component of *G* is then promoted via *X* into the first @XTarget rather than into the second, *G*'s target will not appear later in the final printed document than its invocation point, as required by the into clause.

Accordingly, when such a target is chosen, two special galley indexes are inserted and linked together: a *PRECEDES* index at *G*'s invocation point, and a *FOLLOWS* index at the first component of *G*. The algorithm checks before promoting any *FOLLOWS* index that its promotion would not place it earlier than the corresponding *PRECEDES* index in the same galley, and rejects the component if it would. Since *PRECEDES* and *FOLLOWS* indexes are rarely used, this check can be implemented by linear search.

When two components are separated by /, as opposed to the more usual //, each influences the horizontal position of the other. Because of this, the *SUSPEND* action is in fact taken if a receptive symbol occurs in any component separated from the first by / operators only. Again, linear search forwards to the first // suffices for this check.

A good illustration of these unusual cases is afforded by the @Align symbols from the standard DocumentLayout package. These are used to produce displayed equations, aligned on their equals signs despite being separated by arbitrary body text.

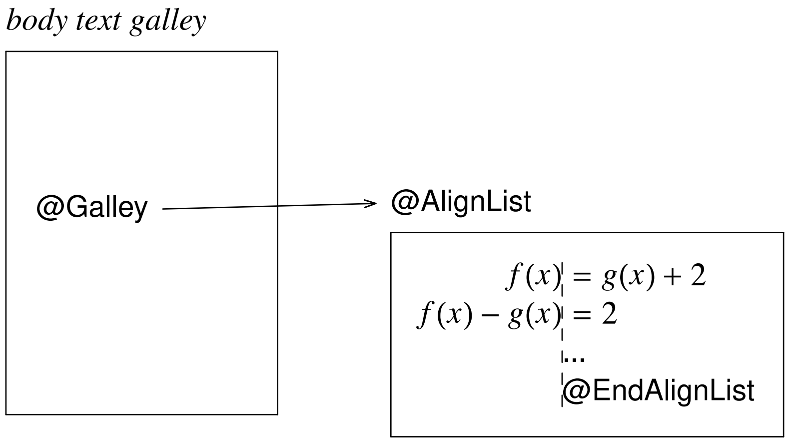
The @Align symbols are packaged neatly for the convenience of the non-expert user, but we will show just the essence of the implementation here. First, an @AlignList galley is created which contains an infinite supply of @AlignPlace receptive symbols separated by / operators:




Then equations like

$$f(x) = g(x) + 2$$

are created and sent to @AlignPlace&&following targets. They collect in the @AlignList galley and are aligned there:



The @AlignList galley does not flush, because its first component is connected to a receptive symbol by / operators.

After the last equation, an empty forcing galley is sent to @EndAlignList, deleting the two remaining receptive symbols from the @AlignList galley and permitting it to flush. *FOLLOWS* indexes ensure that each equation finds a target placed in the body text just after its point of invocation, so the equations return, aligned, to approximately the points where they were invoked. Notice that the flushing of body text is suspended until the list of equations is completed, as it must be, since the horizontal position of the first equation cannot be known until the last equation is added to the list.

Layout quality can occasionally be improved by rejecting a component that could be promoted – for example, a component of body text that carries a footnote too large to fit on the current page. Since Lout does not specify how breaking decisions are made, beyond the basic constraints imposed by available space and into clauses, in principle such high quality breaking could be added to the implementation with no change to the language. However, the generality of the galley flushing algorithm, and its already considerable complexity, make this a daunting problem in practice, although a fascinating one. TEX [9], with its unnested set of 'floating insertions' clearly identifiable as each page is begun, has the advantage in this respect.

### 5.3. Size constraints and size adjustments

The galley flushing algorithm needs to know the available width and height at each receptive symbol. These symbols may lie within arbitrarily complex objects, and they may compete with each other for available space (as body text and footnote targets do), so this information must be extracted from the tree structure when required.

For example, consider the object

```
5i @Wide { a / b }
```

and suppose that the width of a is $1i, 2i$ ($1i$ to the left of the mark, $2i$ to the right). What then is the available width at b? If we let the width of b be $l, r$, we must have

$$(1i \uparrow l) + (2i \uparrow r) \leq 5i$$

with the $\uparrow$ (i.e. max) operations arising from mark alignment. Eliminating them gives

$$\begin{aligned} 1i + 2i &\leq 5i \\ l + 2i &\leq 5i \\ 1i + r &\leq 5i \\ l + r &\leq 5i \end{aligned}$$

and since we assume that a fits into the available space, the first inequality may be dropped, leaving

$$\begin{aligned} l &\le 3i \\ l + r &\le 5i \\ r &\le 4i \end{aligned}$$

Object b may have width $l, r$ for any $l$ and $r$ satisfying these inequalities, and no others.

Here is another example:

```
5i @High { a /2ix b }
```

Assuming that a has height $1i, 1i$, the height $l, r$ of b must satisfy

$$1i + ((1i + l) \uparrow 2i) + r \le 5i$$

This time the $\uparrow$ operation arises from the mark-to-mark gap mode, which will widen the $2i$ gap if necessary to prevent a and b from overlapping. This inequality can be rewritten as

$$\begin{aligned} l &\le \infty \\ l + r &\le 3i \\ r &\le 2i \end{aligned}$$

In general, Lout is designed so that the available width or height at any point can be expressed by three inequalities of the form

$$\begin{aligned} l &\le x \\ l + r &\le y \\ r &\le z \end{aligned}$$

where $x$, $y$ and $z$ may be $\infty$. We abbreviate these three inequalities to $l, r \le x, y, z$, and we call $x$ $, y, z$ a *size constraint*.

The two examples above showed how to propagate the size constraint $\infty, 5i, \infty$ for a / b down one level to the child b. Basser Lout contains a complete set of general rules for all node types, too complicated to give here. Instead, we give just one example of how these rules are derived, using the object

$$x_1 \ / \ x_2 \ / \ \ldots \ / \ x_n$$

where $x_j$ has width $l_j, r_j$ for all $j$.

Suppose the whole object has width constraint $X, Y, Z$, and we require the width constraint of $x_i$. Let $L = \max_j l_j$ and $R = \max_j r_j$, so that $L, R$ is the width of the whole object. We assume $L, R \le X, Y, Z$. Then $x_i$ can be enlarged to any size $l_i, r_i$ satisfying

$$(l_i \uparrow L), (r_i \uparrow R) \le X, Y, Z$$

which expands to eight inequalities:

$$
\begin{aligned}
l_i &\le X \\
L &\le X \\
l_i + r_i &\le Y \\
l_i + R &\le Y \\
L + r_i &\le Y \\
L + R &\le Y \\
r_i &\le Z \\
R &\le Z
\end{aligned}
$$

Three are already known, and slightly rearranging the others gives

$$
\begin{aligned}
l_i &\le X \\
l_i &\le Y - R \\
l_i + r_i &\le Y \\
r_i &\le Z \\
r_i &\le Y - L
\end{aligned}
$$

Therefore the width constraint of $x_i$ is

$$\min(X, Y - R), Y, \min(Z, Y - L)$$

The size constraint of any node can be found by climbing the tree to a *WIDE* or *HIGH* node where the constraint is trivial, then propagating it back down to the node, and this is the function of procedure *Constrained* in Basser Lout.

After some components have been promoted into a target, the sizes stored in its parent and higher ancestors must be adjusted to reflect the increased size. This is done by yet another set of recursive rules, upward-moving this time, which cease as soon as some ancestor's size does not change. These rules are embodied in procedure *AdjustSize* of Basser Lout. The adjustment must be done before relinquishing control to any other galley, but not after every component.

## 5.4. The limited lookahead problem

Basser Lout assumes that there will be enough internal memory to hold the symbol table plus a few pages, but not an entire document. This section describes the consequent problems and how they were solved. Other interpreters, notably interactive editors running on virtual memory systems, would not necessarily need this assumption.

Although Basser Lout can read and format any legal input, its memory consumption will be optimized when the bulk of the document resides in galleys whose targets can be identified at the moment they are encountered. Let us take the typical example of a root galley which is a list of pages, a @BodyText galley targeted into the pages, @Chapter galleys targeted into @BodyText, and @Section galleys targeted into the @Chapter galleys:

```
@PageList
//
@BodyText
//
@Chapter {
   @Section { ... }
   @Section { ... }
   ...
   @Section { ... }
}
@Chapter {
   ...
}
```

Basser Lout is able to read and process such galleys one paragraph at a time (strictly, from one // at the outer level of a galley to the next), as we now describe.

When the parser encounters the beginning of a galley, like @Chapter or @Section, it initiates a new galley process. The special receptive symbol @Input is substituted for the as yet unread right parameter of the galley. As each paragraph of the right parameter is read, it is deleted from the parse tree and injected into the galley's @Input. The galley is then resumed. The parser thus acts as an extra concurrent process; it has low priority, so that input is read only when there is nothing else to do. Since galleys may be nested, a stack of @Input symbols is needed, each with its own environment and style. If a galley is encountered for which a target is not immediately identifiable (a footnote, for example), it is read in its entirety and hung in pure parse tree form from an *UNATTACHED* index in the usual way, with an environment but without a style. It will be flushed later when its component is promoted.

In addition to producing a steady flow of components from input, we must also ensure that receptive symbols do not unduly block their promotion. The @FootSect symbol at the foot of each page is a typical example: until it is deleted the page cannot be printed.

Receptive symbols are expanded only on demand, so @FootSect can be deleted as soon as we can prove that it is not wanted. The symbol table can tell us that only @FootNote galleys (with @FootPlace&&following targets) want it, so it might be possible to deduce that @FootSect may be deleted as soon as body text enters the following page.

The author was unable to make this work, so Basser Lout requires the user to identify those galleys which will carry the bulk of the document (@Chapter, @Section, @BodyText) as *forcing galleys*, by writing force into instead of into in their definitions. As described in the previous section, when a forcing galley attaches to a target, all receptive symbols preceding the target in its galley are deleted, removing all impediments to flushing. For example, when a forcing body text galley enters a new page, the @FootSect symbol on the preceding page will be deleted. It seems likely that a system which could afford to wait until all input was read before deleting any receptive symbols would not need forcing galleys.

Galleys whose targets are a long way from their invocation points can be a problem. If the direction is following, such galleys are held in internal memory for a long time, unless they are to be sorted. If the direction is preceding, then either the entire intervening document must be held in memory (prevented by the target from flushing), or else some forcing galley prematurely deletes the target, leaving the galley bereft.

The typical example of the latter case occurs when the galley is an entry in the table of contents, launched backwards from the beginning of a chapter or section. Its target in the table of contents will have been deleted long before, to permit the rest of the document to print, so the galley ultimately emerges as an unattached galley promoted out of the root galley. All such galleys are written to an auxiliary file, indexed by the missing target. On the next run, just before that target is deleted, the auxiliary file is checked and any galleys for it are read in and flushed.

### 5.5. Horizontal galleys

There is a strong analogy between breaking a column of text into page-sized pieces, and breaking a paragraph into line-sized pieces. In fact, the two differ only in direction: vertical for body text, horizontal for paragraphs. In this section we define *horizontal galleys*, and show how they provide an unlimited number of paragraph breaking styles, as well as solve some other problems. Regrettably, lack of time has prevented their incorporation into the Basser Lout interpreter.

Imagine a galley whose components are separated by horizontal concatenation operators instead of vertical ones, perhaps indicated by a horizontally into clause. Then all object breaking, including paragraph breaking, could be replaced by galley component promotion like this:

```
def @Paragraph right x
{
   def @LinePlace { @Galley }

   def @LineList
   {
      @HExpand @LinePlace
      //1vx  @LineList
   }

   def @Par horizontally into { @LinePlace&&preceding }
      right x
   { x }

   @LineList  //  @Par { 0.2i @Wide {}  &0i  x  &1rt }
}
```

The @HExpand operator, which is a primitive of Basser Lout, horizontally expands the gaps in its right parameter until the result fills the available space, thus implementing line adjustment, except when the parameter contains tabulation gaps like &1rt, which cause the parameter to be already expanded. The result of

```
@Paragraph { A short paragraph of text. }
```

would then be something like

A short paragraph
of text.

depending on the available horizontal space. An unlimited range of paragraph breaking styles could be defined, including ragged right, ragged left, break-and-center, and so on.

In Basser Lout, indented paragraphs are produced by preceding them with a horizontal concatenation operator, for example |0.5i. This has the unfortunate effect of making an indented paragraph into a single component of the enclosing galley, so that it will always be kept together on one page. Horizontal galleys solve this problem with a simple change to @LineList:

```
def @LineList
{
   |0.5i  @HExpand @LinePlace
   //1vx  @LineList
}
```

showing the flexibility that comes from bringing the full power of the Lout language to bear on paragraph layout. It is easy to make provision for a tag on the first line.

Although Basser Lout permits receptive symbols within paragraphs, they are of little use, because their available width is calculated after paragraph breaking, and the incoming galley cannot spread over more than one line. With horizontal galleys, such symbols would have infinite available width, and we could easily produce a filled paragraph of footnotes like this:

[1]See Jones and Saunders (1982). [2]Or so Jacobsen (1973) asserts. [3]*ibid*, p. 327.

based on an infinite horizontal sequence of @FootPlace symbols inside a horizontal galley.

When body text is placed on pages, the length of each column varies depending on the available vertical space. Horizontal galleys could analogously produce lines of varying length, and so could fill non-rectangular shapes.

An important theoretical benefit of horizontal galleys is that they would permit horizontal and vertical to be treated in a perfectly symmetrical way, whereas at present paragraph breaking is horizontal only, and galley breaking is vertical only. This must simplify the treatment of non-European languages which fill in unusual directions, although it is not itself sufficient to implement them.

There are a few minor problems with horizontal galleys. First, the syntactic overhead of enclosing each paragraph in @Paragraph { ... } or whatever is unacceptable. Permitting user-defined operators to have lower precedence than the white space between two words might help here. Second, the built-in paragraph breaker includes hyphenation, and it permits line breaks in the input to determine line breaks in the output, if desired. These features must somehow be preserved. Finally, we have explained how the Basser Lout interpreter assigns equal width to the wider columns of tables (Section 2.5). The equivalent situation in vertical galleys occurs when two receptive symbols compete for vertical space (e.g. @TextPlace and @FootSect), and there it is conventional to grant as much as required to the first arrival. It is not clear to the author how these different approaches can be reconciled.

## 6. Cross references

Cross references, such as ‘see page 57’ and ‘see Figure 5,’ are a useful but highly error-prone feature of documents. Scribe [7] introduced a method of keeping them up to date automatically as the document changes: the user gives each referenced entity a tag, and operators are provided that return the page or sequence number of the entity with a given tag.

A cross reference takes an object (such as a page number) from one point in the document and copies it to another, and this generalization suggests other applications. For example, a running header is copied from the title of a nearby chapter, and a reference is copied from a bibliographic database. Making the unity of these applications manifest is an interesting language design problem.

### 6.1. The cross reference abstraction

In developing the cross reference abstraction, it seemed best to begin with the database application, since it is the simplest. Database relations are naturally mapped into Lout definitions:

```
def @Reference
   named @Tag {}
   named @Author {}
   named @Title {}
   named @Journal {}
{}
```

The set of all invocations of @Reference is a relation whose attributes are the parameters, and whose tuples are the invocations. To complete the correspondence, we need only declare that the @Tag parameter is special, serving as the key attribute.

Following the database model, we next need a notation for retrieving the invocation with a given tag:

```
@Reference&&kingston91
```

This *cross reference* is like an arrow pointing to the invocation. To access its attributes, we write

```
@Reference&&kingston91 @Open { @Author, @Title }
```

The @Open operator evaluates its right parameter in an environment which includes the exported parameters of its left.

An invocation is chosen to be a running header because of its proximity to the place where it is used, rather than by its tag. Such proximity is naturally expressed by two special tags, preceding and following; for example, @Sym&&following will point to the closest following invocation of @Sym in the final printed document. This is much simpler conceptually than reference to the internal state of the document formatter at a critical moment, the usual approach to running headers.

It turns out that the above design solves all the cross referencing problems encountered in practice except one, which may be typified by the problem of finding the number of the page on which the chapter whose tag is intro begins. Two cross referencing steps are needed, first to @Chapter&&intro, then from there to @Page&&preceding, where the page number is known.

Given our success so far, this last problem proves to be surprisingly difficult. We first try

```
@Chapter&&intro @Open {
   @Page&&preceding @Open { @PageNum }
}
```

but this fails because @Page&&preceding is evaluated in the present context, not in the context of @Chapter&&intro as required. So our next attempt is

```
def @Chapter
   named @PageNum { @Page&&preceding @Open { @PageNum } }
   ...
```

with the @Page&&preceding cross reference attached to the chapter; we write

```
@Chapter&&intro @Open { @PageNum }
```

This also fails, because parameters are evaluated after substitution, so once again @Page&&preceding is evaluated in the wrong context. We could of course define a new operator specifically for this case:

```
@Page&&{ @Preceding @Chapter&&intro }
```

or some such. This is free of the annoying context-sensitivity, but it seems quite complex, and the expected cross reference @Page&&preceding does not appear.

The author was lost in these obscurities for some time, and ultimately rescued himself by looking ahead to the implementation of the preceding and following tags, to see if a simple extension of it would solve the problem. This led to the @Tagged operator:

```
@Page&&preceding @Tagged intro
```

placed at the beginning of the body of the chapter will attach intro as an extra tag to the closest preceding invocation of @Page, so that

```
@Page&&intro @Open { @PageNum }
```

yields the desired page number. There is something low-level and ad hoc about the @Tagged operator, but the two cross references do appear naturally, and it works.

### 6.2. Implementation of cross references

Before an object can be sized and printed, the values of any cross references within it must be known. If they refer to invocations that have not yet been read, there is a problem. Scribe [7] solves it by capitalizing on the fact that documents are formatted repeatedly during the drafting process. All tagged invocations are copied to an auxiliary file during the first run, and indexed for quick retrieval on the second. A new auxiliary file is written during the second run, for retrieval on the third, and so on. Cross references always lag one run behind the rest of the document; a perfect copy may be produced by formatting the same version twice, except in a few pathological cases that fail to converge.

Cross referencing in Lout is implemented on top of a simple database system. Each database is either writable or readable but not both at once, and holds a set of key-value entries: the keys are ASCII strings, and the values are Lout objects, possibly with environments, written in Lout source. Operations are provided for writing an entry, converting from writable to readable, retrieval by key, and sequential retrieval in key order.

The implementation, which is quite unsophisticated, employs one or more ASCII *database files*, containing the values, and one ASCII *index file* per database, containing the keys. To write an entry, the value is first appended to a database file, then a line like

```
@Chapter&&intro  ch1.ld  57
```

is appended to the index file, giving the file and offset where the value is stored. To convert from writable to readable, the index file is sorted. Then retrieval by key requires a binary search of the index file and one seek into a database file, and sequential retrieval by key is trivial.

This database system is used in several ways. For an external database, say of bibliographic references, the user creates the database file of values (without environments), Lout creates the index file whenever it cannot find one, and retrievals by key proceed as usual. Cross references with tags other than preceding and following are treated as described above, by writing all tagged invocations (with environments) to a single database, which is converted to readable at the end of the run for retrievals on the next run. Sorted galleys, such as index entries, are written out indexed by target and key and retrieved sequentially on the next run. Unsorted galleys with preceding targets which pop off the top of the root galley without finding a target, such as entries in tables of contents, are treated similarly, except that they are indexed by target and a sequence number that preserves their relative order during the sort.

When Lout processes a multi-file document, one cross reference database file is written for each input file, but they share a common index file. At end of run, the new index file is sorted and merged with the old one in such a way as to preserve entries relating to files not read on the current run. This provides some support for piecemeal formatting, but eventually the files must all be formatted together.

When a preceding or following cross reference is found, it is attached to a galley index of type *CROSS_PREC* or *CROSS_FOLL*, together with an automatically generated tag composed of the current file name and a sequence number. When a tagged invocation is found, it is attached to a *CROSS_TARG* index. These galley indexes are carried along through the dynamic tree, and eventually pop off the top of the root galley, at which point it is easy to determine which cross references refer to which invocations, since the indexes are now in final printed document order. Each referenced invocation is then written to the cross reference database, multiply indexed by the generated tags of the associated cross references. On the next run, when the same preceding and following cross references are found, chances are good that the same tags will be generated, and the appropriate values can be retrieved from the database immediately.

This approach was the genesis of the @Tagged operator, whose implementation is now immediate: for each @Tagged operator we produce one *CROSS_PREC* or *CROSS_FOLL* galley index, replacing the generated tag with the right parameter of the @Tagged operator. Nothing more is required.

## 7. Conclusion

Since its public release in October 1991, the Basser Lout interpreter has been ported without incident to a wide variety of Unix systems and hardware. It was tested extensively before release on its own documentation, and the few minor bugs which have emerged since then have all been fixed in the second release, scheduled to appear in mid-1992.

Seven substantial packages of definitions are distributed with Basser Lout. The DocumentLayout package, and its variants ReportLayout and BookLayout, provide the standard features that all documents require: pages, columns, paragraphs, headings, footnotes, floating figures and tables, chapters and sections, displays and lists, access to bibliographic databases, cross references, and so on [11]. The BookLayout package has extra features needed by books, including an automatically generated table of contents, Roman page numbers for the prefatory material, running page headers, odd and even page layouts, and a sorted index. The Eq package formats equations, and Pas formats Pascal programs [10]; Tab formats tables [12]; and Fig draws figures [6].

The non-expert user who uses these packages perceives a system of a standard quite similar to other fully developed batch formatters, although the interface is considerably more coherent than, say, the troff family's [8]. The expert user perceives a system which is radically different from previous ones, in which a great deal can be achieved very quickly. To take an extreme example, Pas was designed, implemented, tested, and documented in one afternoon. Eq took about a week, but most of that time was spent in marshalling the vast repertoire of mathematical symbols, and fine-tuning the spacing. Most of the effort seems to go into designing a good interface; most symbols are implemented in just one or a few lines of Lout.

A group of about 20 satisfied non-expert users has grown up within the author's department, mainly Honours students with no investment in older systems to hold them back. Basser Lout has been advertised on the Internet news as available via anonymous *ftp*, so the extent of its outside user community is hard to gauge. About 50 people have mailed comments or questions to the author; many of these people have ported the program, written small definitions, and modified the standard packages.

Future work could usefully begin with the improvements suggested in this paper: overlapping spanning columns, better semantics for available space, and especially horizontal galleys. Support for non-European languages is also needed. However, the main task is the development of an interactive document editor based on Lout. A structure editor similar to Lilac [13], which already has objects and user-defined symbols, is envisaged; since cross references are easy when the whole document is available, the only major new problem is the treatment of galleys, including the expansion and retraction of receptive symbols.

**Note.** Since the above was written the author has completed a revised version of Basser Lout, in which the problem concerning available space mentioned in Section 2.5 has been resolved.

**Acknowledgment.** The author gratefully acknowledges many valuable discussions with

Douglas W. Jones, especially during the development of the galley abstraction; and also many helpful comments on presentation by the anonymous referee.

**References**

1. Kingston, Jeffrey H.. Document Formatting with Lout. Tech. Rep. 408 (1991), Basser Department of Computer Science, The University of Sydney, Australia.

2. Kingston, Jeffrey H.. A new approach to document formatting. Tech. Rep. 412 (1991), Basser Department of Computer Science, The University of Sydney, Australia.

3. Kingston, Jeffrey H.. The Basser Lout Document Formatter, 1991. Computer program; Version 2 publicly available in the *pub* subdirectory of the home directory of *ftp* to host *ftp.cs.su.oz.au* with login name *anonymous* and no password. Distribution via email is available for non-*ftp* sites. All enquiries to jeff@cs.su.oz.au..

4. Furuta, Richard, Scofield, Jeffrey, and Shaw, Alan. Document formatting systems: survey, concepts, and issues. *Computing Surveys* **14**, 417–472 (1982).

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

6. Kingston, Jeffrey H.. Fig – a Lout package for drawing figures. Tech. Rep. 411 (1991), Basser Department of Computer Science, The University of Sydney, Australia.

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

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

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

10. Kingston, Jeffrey H.. Eq – a Lout package for typesetting mathematics. Tech. Rep. 410 (1991), Basser Department of Computer Science, The University of Sydney, Australia. (Contains an appendix describing the Pas Pascal formatter.).

11. Kingston, Jeffrey H.. A beginners' guide to Lout. Tech. Rep. 409 (1991), Basser Department of Computer Science, The University of Sydney, Australia.

12. Kingston, Jeffrey H.. Tab – a Lout package for formatting tables. Tech. Rep. 413 (1991), Basser Department of Computer Science, The University of Sydney, Australia.

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