An Expert's Guide to the

Lout

Document Formatting System

Jeffrey H. Kingston

Version 3.17
September, 1999

Preface

This manual is addressed to those who wish to become expert users of the Lout document formatting system. An expert user is someone who understands the principles of document formatting that Lout embodies, and is able to apply them, for example to design a document format or a special-purpose package. In contrast, a non-expert user is someone who simply uses Lout to format documents.

Chapter 1 explains these principles, and it should be read carefully and in sequence. Chapters 2 and 3 are for reference; respectively, they contain descriptions of the detailed operation of Lout's major components, and a complete description of each predefined symbol. The final chapter presents a collection of advanced examples.

This manual presents Version 3 of Basser Lout, publicly released in September 1994 [3]. This manual was rendered into PostScript by Version 3.17 of the Basser Lout interpreter, using the symbols described in the User's Guide [4].

Acknowledgment. Version 3 has benefited from hundreds of comments received since the release of Version 1 in October 1991. Not every suggestion could be followed, but many have been, and the encouragement was greatly appreciated.

Contents

Chapter 1. Principles

The Lout document formatting language is based on just four key ideas: objects, definitions, cross references, and galleys. This chapter concentrates on them, postponing the inevitable details.

1.1. Objects

Since our aim is to produce neatly formatted documents, we should begin by looking at a typical example of such a document:

PURCELL[1]

In the world of music England is supposed to be a mere province. If she produces an indifferent composer

[1]Blom, Eric. *Some Great Composers.* Oxford, 1944.

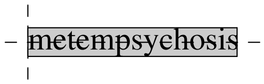
or performer, that is regarded elsewhere as perfectly normal and natural; but if foreign students of musical history have to acknowledge a British musical genius, he is considered a freak.

Such a freak is Henry Purcell. Yet if we

make a choice of fifteen of the world's musical classics, as here, we find that we cannot omit this English master.

It is a large rectangle made from three smaller rectangles – its pages. Each page is made of lines; each line is made of words, although it makes sense for any rectangle (even a complete document) to be part of a line, provided it is not too large.

Lout deals with something a little more complicated than rectangles: *objects*. An object is a rectangle with at least one *column mark* protruding above and below it, and at least one *row* *mark* protruding to the left and right. The simplest objects contain words like metempsychosis, and have one mark of each type:

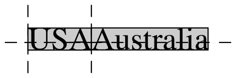


The rectangle exactly encloses the word; its column mark is at the left edge, and its row mark passes through the middle of the lower-case letters. The rectangle and marks do not appear on the printed page, but to understand what Lout is doing you have to imagine them.

To place two objects side by side, we separate them by the symbol |, which denotes the act of *horizontal concatenation.* So, if we write

```
USA | Australia
```

the result will be the object

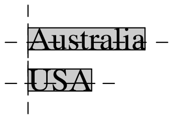


Notice that this object has two column marks, but still only one row mark, because | merges the two row marks together. This merging of row marks fixes the vertical position of each object with respect to the other, but it does not determine how far apart they are. This distance, or *gap*, may be given just after the symbol, as in |0.5i for example, which specifies horizontal concatenation with a gap of half an inch. If no gap is given, it is assumed to be 0i.

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

```
Australia /0.1i USA
```

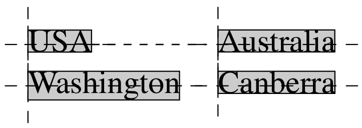
has result



The usual merging of marks occurs, and now the gap determines the vertical separation. Horizontal and vertical can be combined:

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

has result



There are several things to note carefully here. White space (including tabs and newlines) adjacent to a concatenation symbol is ignored, so it may be used to lay out the expression clearly. The symbol | takes precedence over /, which means that the rows are formed first, then vertically concatenated. The symbol / will merge two or more column marks, creating multiple columns (and | will merge two or more row marks). This implies that the gap 0.2i used above is between columns, not individual items in columns; a gap in the second row would therefore be redundant, and so is omitted.

A variant of / called // left-justifies two objects instead of merging their marks.

By enclosing an object in braces, it is possible to override the set precedences. Here is another expression for the table above, in which the columns are formed first:

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

Braces have no effect other than to alter the grouping.

Paragraph breaking occurs when an object is too wide to fit into the space available to it; by breaking its paragraphs into lines, its width is reduced to an acceptable amount. The available space is determined by the @Wide symbol, whose form is

length @Wide *object*

and whose result is the given object modified to have exactly the given length. For example,

```
5i @Wide {
Macbeth was very ambitious. This led him to wish to become king of
Scotland. The witches told him that this wish of his would come true. The
king of Scotland at this time was Duncan. Encouraged by his wife, Macbeth
murdered Duncan. He was thus enabled to succeed Duncan as king. (51 words)
|0.5i
Encouraged by his wife, Macbeth achieved his ambition and realized the
prediction of the witches by murdering Duncan and becoming king of Scotland
in his place. (26 words)
}
```

has for its result the following five inch wide object [7]:

Macbeth was very ambitious. This led him to wish to become king of Scotland. The witches told him that this wish of his would come true. The king of Scotland at this time was Duncan. Encouraged by his wife, Macbeth murdered Duncan. He was thus enabled to succeed Duncan as king. (51 words)	Encouraged by his wife, Macbeth achieved his ambition and realized the prediction of the witches by murdering Duncan and becoming king of Scotland in his place. (26 words)

A paragraph of text can be included anywhere, and it will be broken automatically if necessary to fit the available space. The spaces between words are converted into concatenation symbols.

These are the most significant of Lout's object-building symbols. There are others, for changing fonts, controlling paragraph breaking, printing graphical objects like boxes and circles, and so on, but they do not add anything new in principle.

1.2. Definitions

The features of Lout are very general. They do not assume that documents are composed of pages, nor that there are such things as margins and footnotes, for example. *Definitions* bridge the gap between Lout's general features and the special features – footnotes, equations, pages – that particular documents require. They hold the instructions for producing these special features, conveniently packaged ready for use.

For example, consider the challenge posed by 'TEX', which is the name of one of Lout's most illustrious rivals [5]. Lout solves it easily enough, like this:

```
T{ /0.2fo E }X
```

but to type this every time TEX is mentioned would be tedious and error-prone. So we place a definition at the beginning of the document:

```
def @TeX { T{ /0.2fo E }X }
```

Now @TeX stands for the object following it between braces, and we may write

```
consider the challenge posed by '@TeX', ...
```

as the author did earlier in this paragraph.

A *symbol* is a name, like @TeX, which stands for something other than itself. The initial @ is not compulsory, but it does make the name stand out clearly. A *definition* of a symbol declares a name to be a symbol, and says what the symbol stands for. The *body* of a definition is the part following the name, between the braces. To *invoke* a symbol is to make use of it.

Another expression ripe for packaging in a definition is

```
@OneRow { | -2p @Font n ^/0.5fk 2 }
```

which produces 2^n (see Chapter 2). But this time we would like to be able to write

object @Super *object*

so that a @Super 2 would come out as a^2, and so on, for in this way the usefulness of the definition is greatly increased. Here is how it is done:

```
def @Super
  left x
  right y
{ @OneRow { | -2p @Font y ^/0.5fk x }
}
```

This definition says that @Super has two *parameters*, x and y. When @Super is invoked, all occurrences of x in the body will be replaced by the object just to the left of @Super, and all occurrences of y will be replaced by the object just to the right. So, for example, the expression

```
2 @Super { Slope @Font n }
```

is equal to

```
@OneRow { | -2p @Font { Slope @Font n } ^/0.5fk 2 }
```

and so comes out as 2^n.

Lout permits definitions to invoke themselves, a peculiarly circular thing to do which goes by the name of *recursion.* Here is an example of a recursive definition:

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

The usual rule is that the value of an invocation of a symbol is a copy of the body of the symbol's definition, so the value of @Leaders must be

```
.. @Leaders
```

But now this rule applies to this new invocation of @Leaders; substituting its body gives

```
.. .. @Leaders
```

and so on forever. In order to make this useful, an invocation of a recursive symbol is replaced by its body only if sufficient space is available. So, for example,

```
4i @Wide { Chapter 7 @Leaders 62 }
```

has for its result the object

Chapter 7 62

with Lout checking before each replacement of @Leaders by .. @Leaders that the total length afterwards, including the other words, would not exceed four inches.

The remaining issue is what happens when Lout decides that it is time to stop. The obvious thing to do is to replace the last invocation by an empty object:

```
.. .. .. .. .. .. .. .. {}
```

As the example shows, this would leave a small trailing space, which is a major headache. Lout fixes this by replacing the last invocation with a different kind of empty object, called @Null, whose effect is to make an adjacent concatenation symbol disappear, preferably one preceding the @Null. Thus, when Lout replaces @Leaders by @Null in the expression

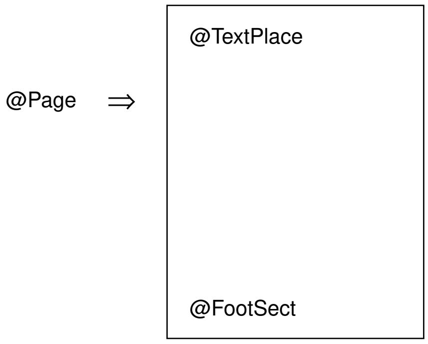
```
.. .. .. .. .. .. .. .. @Leaders
```

the trailing space, which is really a horizontal concatenation symbol, disappears as well. This is taken into account when deciding whether there is room to replace @Leaders by its body.

The remainder of this section is devoted to showing how definitions may be used to specify the *page layout* of a document. To begin with, we can define a page like this:

```
def @Page
{
   //1i ||1i
   6i @Wide 9.5i @High
   { @TextPlace //1rt @FootSect }
   ||1i //1i
}
```

Now @Page is an eight by eleven and a half inch object, with one inch margins, a place at the top for text, and a section at the bottom for footnotes (since //1rt bottom-justifies the following object). It will be convenient for us to show the effect of invoking @Page like this:

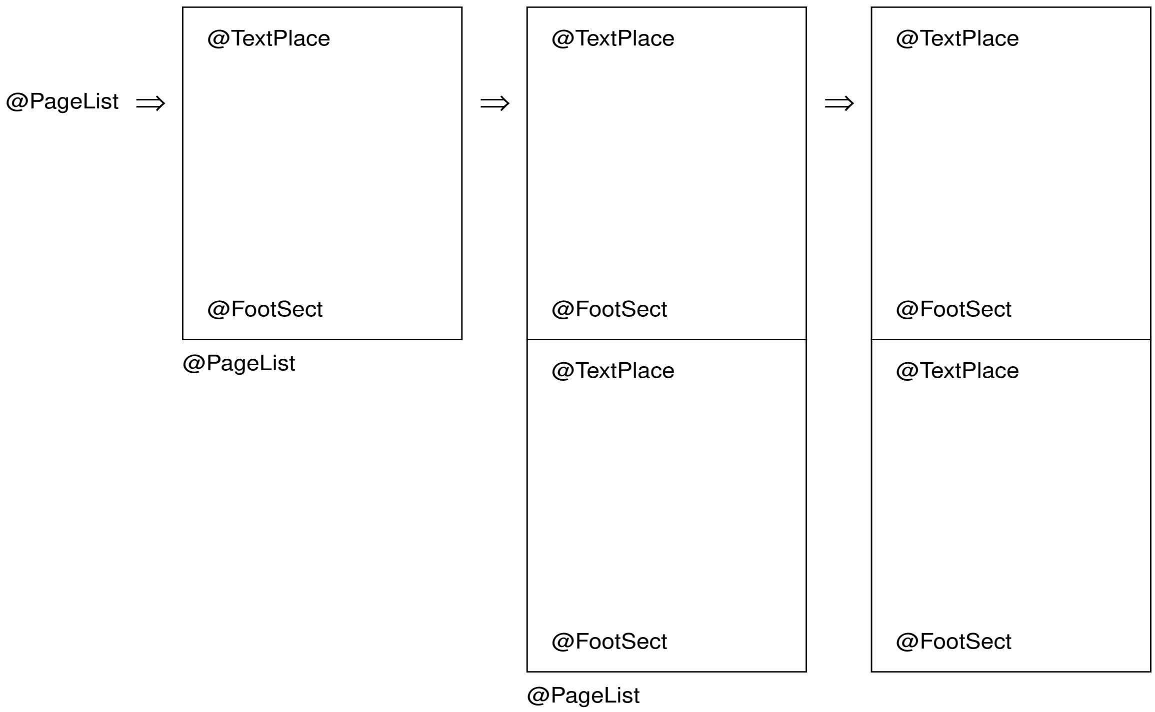


with the invoked symbol appearing to the left of the arrow, and its body to the right.

The definition of a vertical list of pages should come as no surprise:

```
def @PageList
{
   @Page // @PageList
}
```

This allows invocations like the following:



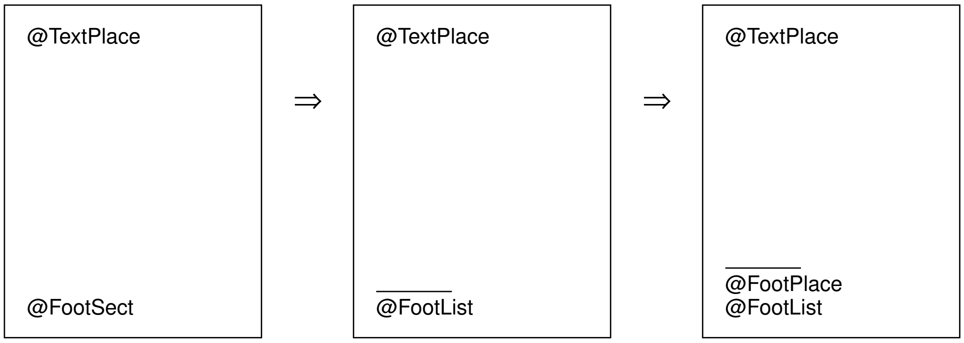
setting @PageList to @Null on the last step. Any number of pages can be generated.

A definition for @TextPlace is beyond us at present, since @TextPlace must be replaced by different parts of the text of the document on different pages. But we can define @FootSect to be a small space followed by a horizontal line followed by a list of places where footnotes go:

```
def @FootList
{
  @FootPlace  //0.3v  @FootList
}

def @FootSect
{
  //0.3v 1i @Wide @HLine
  //0.3v @FootList
}
```

assuming that @HLine will produce a horizontal line of the indicated width. With this definition we can generate pages like this:



and so on for arbitrarily many footnotes.

We will see in the next section how invocations of @PageList, @FootSect and @FootList are replaced by their bodies only when the need to insert text and footnotes obliges Lout to do so; otherwise the invocations are replaced by @Null. In this way, the right number of pages is made, the small line appears only on pages that have at least one footnote, and unnecessary concatenation symbols disappear.

This approach to page layout is the most original contribution Lout has made to document formatting. It is extraordinarily flexible. Two-column pages? Use

```
{2.8i @Wide @TextPlace}  ||0.4i  {2.8i @Wide @TextPlace}
```

instead of @TextPlace. Footnotes in smaller type? Use -2p @Font @FootPlace instead of @FootPlace. And on and on.

1.3. Cross references

A cross reference in common terminology is something like 'see Table 6' or 'see page 57' – a reference within a document to some other part of it. Readers find them very useful, but they are a major problem for authors. As the document is revised, Table 6 becomes Table 7, the thing on page 57 moves to page 63, and all the cross references must be changed.

The Scribe document formatter, developed by Brian K. Reid [6], introduced a scheme for keeping track of cross references. It allows you to give names to tables, figures, etc., and to refer to them by name. The formatter inserts the appropriate numbers in place of the names, so that as the document is revised, the cross references are kept up to date automatically. Lout has adopted and extended this scheme.

In Lout, automatic cross referencing works in the following way. First define a symbol with a parameter with the special name @Tag:

```
def @Table
  left @Tag
  right @Value
{
  ||1i  @Value
}
```

When this symbol is invoked, the value given to @Tag should be a simple word like cities, or several simple words juxtaposed like cities compare; it serves to name the invocation:

```
{ cities compare } @Table
{
   Washington  |0.5i  Canberra
}
```

We may now refer to this invocation elsewhere in the document, using the *cross reference* @Table&&{ cities compare }. Here && is the *cross reference symbol*; its left parameter is a symbol and its right parameter is the value of the @Tag parameter of some invocation of that symbol. Of course it's simplest if you use just a one-word tag; then no braces are needed.

A cross reference is not an object; the reader should think of it as an arrow in the final printed document, beginning at the cross reference and ending at the top of the target invocation. Three special values may be given to the right parameter of &&: preceding, following, and foll_or_prec. The cross reference @Table&&preceding points to some table appearing earlier in the final printed document than itself; that is, the arrow is guaranteed to point backwards through the document. Usually it points to the nearest preceding invocation. Similarly, @Table&&following points forwards, usually to the nearest following invocation. @Table&&foll_or_prec is the same as @Table&&following if it exists, otherwise it is the same as @Table&&preceding.

This section has been concerned with what a cross reference is – an arrow from one point in a document to another – but not with how it is used. One simple way to use a cross reference is to put it where an object is expected, like this:

```
a  |  @Table&&cities  |  c
```

In this case the cross reference will be replaced by a copy of the invocation it points to: in the example just given, a table will appear between a and c. Other applications of cross references may be found in Chapter 4, including finding the number of the page where something appears, producing running page headers and footers, and accessing databases of Roman numerals, references, etc. Cross references are also used by galleys, as will be explained in the next section.

The implementation of cross referencing copies every symbol invocation with a @Tag parameter into the *cross-reference database*, a collection of files whose names end in .ld indexed by one file whose name is lout.li. It is generally the case that the bulk content of a symbol such as the table above is contained in its right or body parameter, and that this bulk content is not needed by cross references to the symbol. Hence, to save space in the database, Lout replaces the right parameter of each symbol it writes into it by the word ??? whenever the right parameter appears to be large. The table above would appear as ??? because of this optimization, and in general, the user must ensure that any content required by cross references is contained in parameters other than the right or body parameter. This optimization does not apply when the symbol being written into the cross-reference database is a galley.

1.4. Galleys

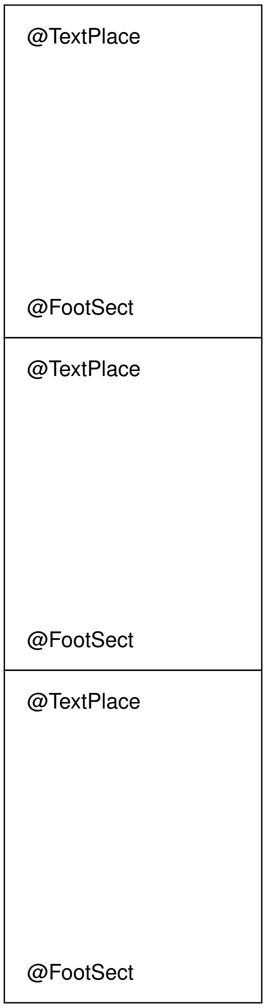
It is time to pause and ask ourselves how close we are to achieving our aim of producing neatly formatted documents. We can certainly produce the pieces of a document:

PURCELL[1]

In the world of music England is supposed to be a mere province. If she produces an indifferent composer or performer, that is regarded elsewhere as perfectly normal and natural; but if foreign students of musical history have to acknowledge a British musical genius, he is considered a freak.

Such a freak is Henry Purcell. Yet if we make a choice of fifteen of the world's musical classics, as here, we find that we cannot omit this English master.

[1]Blom, Eric. *Some Great Composers.* Oxford, 1944.



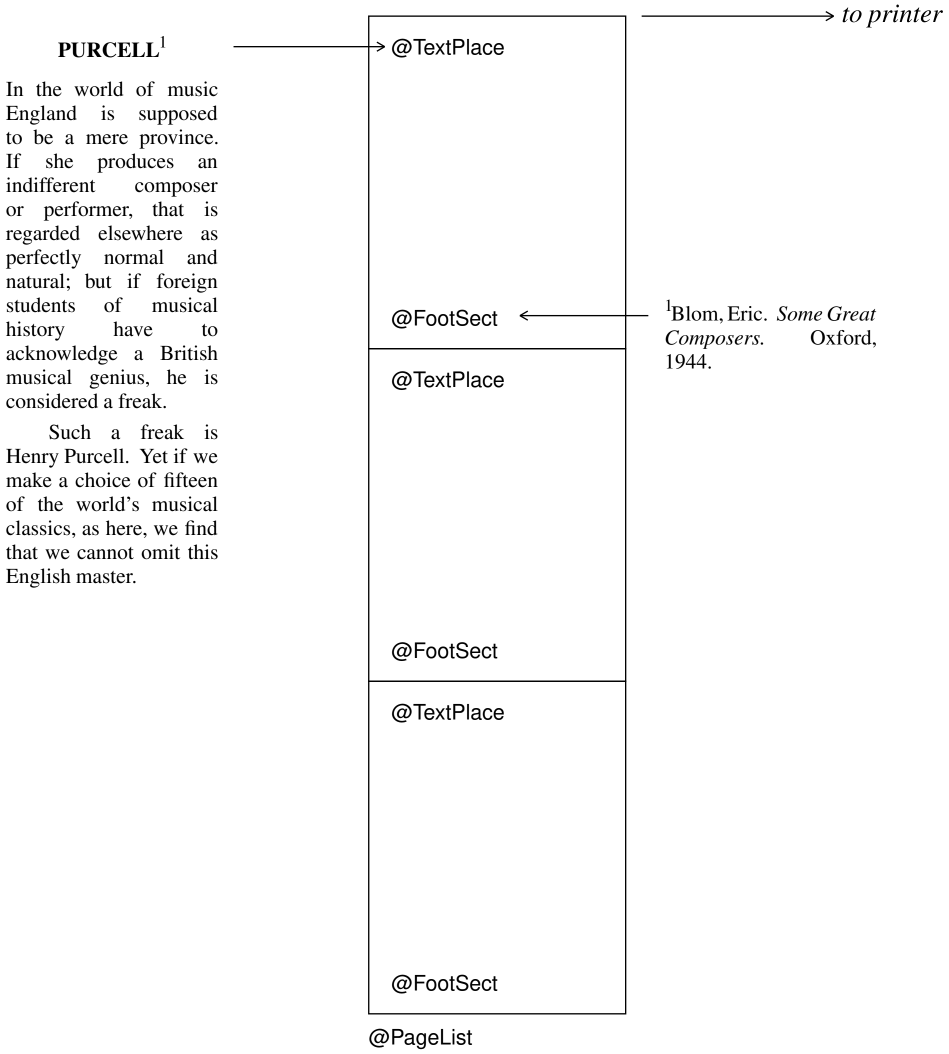
@PageList

but when we try to merge them together, we encounter two obstacles.

First, when an object is entered at a certain place in the document, it appears at that place. But a footnote is naturally entered immediately after the point it refers to ('PURCELL' in this case), yet it appears somewhere else: at the bottom of a page.

Second, all our features build up larger objects out of smaller ones, but the PURCELL object, for example, must be broken down into page-sized pieces. This occurs when the available space at the 'somewhere else' is insufficient to hold the entire object, so this second obstacle arises out of the first.

Lout's last major feature, which we introduce to overcome these obstacles, is the *galley* (the name is borrowed from the galleys used in manual typesetting). A galley is an object plus a cross reference which points to where the object is to appear. The example above has three galleys:



A galley replaces the invocation pointed to by its cross reference. If space is not sufficient there to hold it all, the remainder of the galley is split off (the vertical concatenation symbol preceding it being discarded) and it replaces later invocations of the same symbol. This is exactly what is required to get text and footnotes onto pages.

To create a galley, first define a symbol with a special into clause, like this:

```
def @FootNote into { @FootPlace&&following }
  right x
{
  8p @Font x
}
```

An invocation of such a symbol will then be a galley whose object is the result of the invocation, and whose cross reference is given by the into clause. The right parameter of the cross reference must be one of preceding, following, and foll_or_prec.

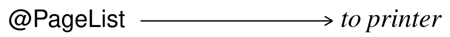
A symbol, like @FootPlace, which is the *target* of a galley, must contain the special symbol @Galley exactly once in its body; often this is all that the body contains:

```
def @FootPlace { @Galley }
```

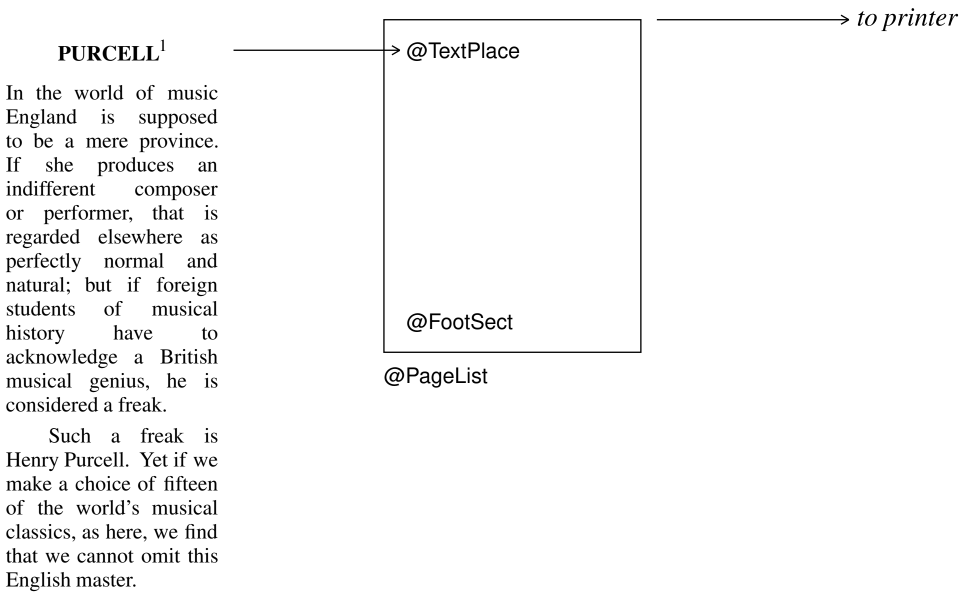
It is this special symbol that is replaced by the incoming galley, in fact, not the @FootPlace symbol as a whole.

A symbol which contains @Galley, either directly within its body or indirectly within the body of a symbol it invokes, is called a *receptive* symbol, meaning receptive to galleys. @FootPlace is receptive, which makes @FootList, @FootSect and @PageList receptive since they invoke @FootPlace. If no galley replaces any @Galley within some invocation of a receptive symbol, that invocation is replaced by @Null. The advantages of this rule for page layout were explained at the end of Section 1.2.

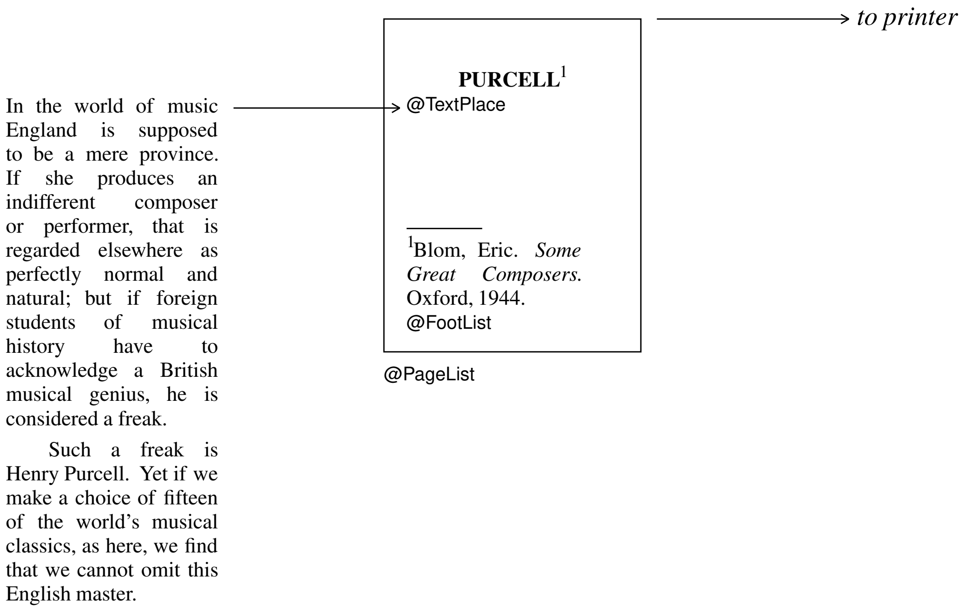
Let us now follow through the construction of our example document. Initially there is just the one *root* galley, containing an unexpanded invocation of @PageList:



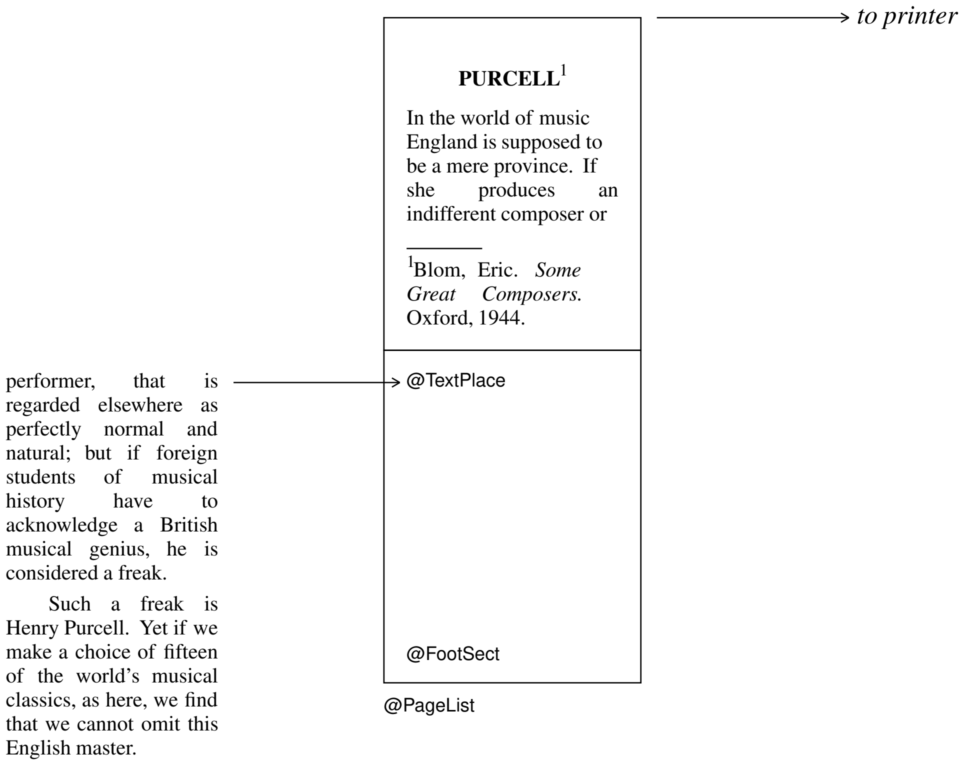
Then the PURCELL galley appears, targeted to a @TextPlace. Lout knows that there is a @TextPlace hidden inside @PageList, so it expands @PageList:



After promoting the first line into @TextPlace, the footnote galley attached to it appears and demands an invocation of @FootPlace following its attachment point ('PURCELL'). Such a @FootPlace is found at the bottom of the first page, inside @FootSect, which is accordingly expanded, and the footnote is promoted onto the page:



Now the promotion of the PURCELL galley resumes. When the first page is filled, Lout searches forwards for another @TextPlace to receive the remainder, once again expanding a @PageList:



and so on. All these expansions and replacements are done with total integrity. For example, if Lout finds after expanding @FootSect that the page is too full to accept even the first line of the footnote, @FootSect is reset to unexpanded and the search for a target for the footnote moves on. And the cross reference direction, preceding or following, is always obeyed (although lack of space sometimes prevents Lout from choosing the nearest target). Only the root galley contains receptive symbols in our running example, but any galley may contain them.

Chapter 2. Details

2.1. Lexical structure (words, spaces, symbols) and macros

The input to Lout consists of a sequence of *textual units*, which may be either *white spaces*, *identifiers, delimiters,* or *literal words*. Each is a sequence of *characters* chosen from:

<table>
<tr><td>letter</td><td>@ab-zAB-Z_</td></tr>
<tr><td>white space</td><td>space formfeed tab newline</td></tr>
<tr><td>quote</td><td>"</td></tr>
<tr><td>escape</td><td>\</td></tr>
<tr><td>comment</td><td>#</td></tr>
<tr><td>other</td><td>!$%&'()*+,-./0123456789:;<=>?[]^`{|}~</td></tr>
</table>

Notice that @ and _ are classed as letters. Basser Lout accepts the accented letters of the ISO-LATIN-1 character set (depending on how it is installed), and these are also classed as letters. The ten digits are classed as 'other' characters, and in fact the 'other' class contains all 8-bit characters (except octal 0) not assigned to previous classes.

A *white space* is a sequence of one or more white space characters. Lout treats the formfeed character exactly like the space character; it is useful for getting page breaks when printing Lout source code.

A *delimiter* is a sequence of one or more 'other' characters which is the name of a symbol. For example, { and // are delimiters. When defining a delimiter, the name must be enclosed in quotes:

```
def "^" { {} ^& {} }
```

but quotes are not used when the delimiter is invoked. A delimiter may have delimiters and any other characters adjacent, whereas identifiers may not be adjacent to letters or other identifiers. The complete list of predefined delimiters is

<table>
<tr><td>/</td><td>|</td><td>&</td><td>&&</td></tr>
<tr><td>//</td><td>||</td><td>^&</td><td>{</td></tr>
<tr><td>^/</td><td>^|</td><td></td><td>}</td></tr>
<tr><td>^//</td><td>^||</td><td></td><td></td></tr>
</table>

A longer delimiter like <= will be recognised in preference to a shorter one like <.

An *identifier* is a sequence of one or more letters which is the name of a symbol. It is conventional but not essential to begin identifiers with @; Basser Lout will print a warning message if it finds an unquoted literal word (see below) beginning with @, since such words are usually misspelt identifiers. The ten digits are not letters and may not appear in identifiers; and although the underscore character is a letter and may be used in identifiers, it is not conventional to do so. The complete list of predefined identifiers is

@BackEnd	@Include	@Rump
@Background	@IncludeGraphic	@Scale
@Begin	@Insert	@SetColor
@Break	@KernShrink	@SetColour
@Case	@Key	@Space
@Common	@Language	@StartHSpan
@Char	@LClos	@StartHVSpan
@CurrFace	@LEnv	@StartVSpan
@CurrFamily	@LInput	@SysDatabase
@CurrLang	@LVis	@SysInclude
@Database	@LUse	@SysIncludeGraphic
@End	@Meld	@SysPrependGraphic
@Enclose	@Merge	@Tag
@Filter	@Minus	@Tagged
@FilterErr	@Moment	@Target
@FilterIn	@Next	@Underline
@FilterOut	@NotRevealed	@Use
@Font	@Null	@VAdjust
@ForceGalley	@OneCol	@VContract
@Galley	@OneOf	@VCover
@Graphic	@OneRow	@Verbatim
@HAdjust	@Open	@VExpand
@HContract	@Optimize	@VLimited
@HCover	@PAdjust	@VScale
@HExpand	@PageLabel	@VShift
@High	@PlainGraphic	@VSpan
@HLimited	@Plus	@Wide
@HScale	@PrependGraphic	@Yield
@HShift	@RawVerbatim	@YUnit
@HSpan	@Rotate	@ZUnit

plus the names of the parameters of @Moment. The symbols @LClos, @LEnv, @LInput, @LVis and @LUse appear in cross reference databases generated by Lout and are not for use elsewhere.

A sequence of characters which is neither a white space, an identifier, nor a delimiter, is by default a *literal word*, which means that it will pass through Lout unchanged. An arbitrary sequence of characters enclosed in double quotes, for example "{ }", is also a literal word. Space characters may be included, but not tabs or newlines. There are special character sequences, used only between quotes, for obtaining otherwise inaccessible characters:

\"	produces "
\\	\
\ddd	the character whose ASCII code is the up to three digit octal number ddd

So, for example, "\"@PP\"" produces "@PP".

When the comment character # is encountered, everything from that point to the end of the line is ignored. This is useful for including reminders to oneself, like this:

```
# Lout user manual
# J. Kingston, June 1989
```

for temporarily deleting parts of the document, and so on.

Macros provide a means of defining symbols which stand for a sequence of textual units rather than an object. For example, the macro definition

```
macro @PP { //1.3vx 2.0f @Wide &0i }
```

makes Lout replace the symbol @PP by the given textual units before assembling its input into objects. A similar macro to this one is used to separate the paragraphs of the present document. The enclosing braces and any spaces adjacent to them are dropped, which can be a problem: @PP2i has result //1.3vx 2.0f @Wide &0i2i which is erroneous.

The meaning of symbols used within the body of a macro is determined by where the macro is defined, not by where it is used. Due to implementation problems, @Open symbols will not work within macros. Named and body parameters will work if the symbol that they are parameters of is also present. There is no way to get a left or right brace into the body of a macro without the matching brace.

Macros may be nested within other definitions and exported, but they may not be parameters. They may not have parameters or nested definitions of their own, and consequently a preceding export clause (Section 2.3) would be pointless; however, an import clause is permitted.

2.2. Named parameters

In addition to left and right (or body) parameters, a symbol may have any number of *named parameters*:

```
def @Chapter
   named @Tag {}
   named @Title {}
   right x
{
   ...
}
```

Their definitions appear in between those of any left and right parameters, and each is followed by a *default value* between braces. When @Chapter is invoked, its named parameters are given values in the following way:

```
@Chapter
   @Tag { intro }
   @Title { Introduction }
{
   ...
}
```

That is, a list of named parameters appears immediately following the symbol, each with its value enclosed in braces. Any right parameter follows after them. They do not have to appear in the order they were defined, and they can even be omitted altogether, in which case the default value from the definition is used instead.

If the keyword compulsory appears after named and before the parameter's name, Lout will print a warning message whenever this parameter is missing. However it will still use the default value as just described.

A named @Tag parameter does not take its default value from the definition; instead, if a default value is needed, Lout invents a simple word which differs from every other tag. This is important, for example, in the production of numbered chapters and sections (Section 4.4). The same thing occurs if there is a @Tag parameter but its value is the empty object: the value will be replaced by an invented one.

Named parameters may have parameters, as in the following definition:

```
def @Strange
   named @Format right @Val { [@Val] }
   right x
{
   @Format x
}
```

The named parameter @Format has right parameter @Val, and the default value of @Format is this parameter enclosed in brackets. When @Format is invoked it must be supplied with a right parameter, which will replace @Val. Thus,

```
@Strange 27
```

equals @Format 27 and so has result

[27]

The @Format symbol is like a definition with parameters whose body can be changed:

```
@Strange
   @Format { Slope @Font @Val. }
27
```

still equals @Format 27, but this time the result is

27.

In practice, examples of named parameters with parameters all have this flavour of format being separated from content; running headers (Section 4.3) and printing styles for bibliographies (Section 4.5) are two major ones.

2.3. Nested definitions, body parameters, extend, import, and export

A definition may contain other definitions at the beginning of its body:

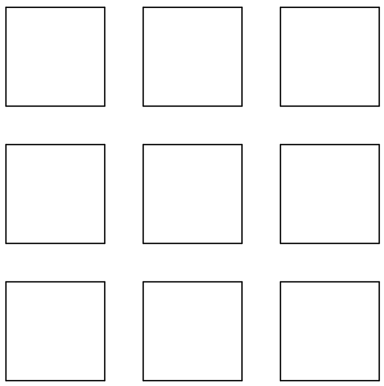
```
def @NineSquare
  right x
{
  def @Three { x |0.2i x |0.2i x }

  @Three /0.2i @Three /0.2i @Three
}
```

A parameter like x may be invoked anywhere within the body of the symbol it is a parameter of, including within nested definitions. A nested symbol like @Three may be invoked anywhere from the beginning of its own body to the end of the body of the symbol it is defined within. So, assuming an appropriate definition of @Box,

```
@NineSquare @Box
```

has result



Nested definitions may themselves contain nested definitions, to arbitrary depth.

There are three special features which permit a nested symbol or parameter to be invoked outside its normal range; that is, outside the body of the enclosing symbol. The first and simplest of these features is the *body parameter*, an alternative form of right parameter. The Eq equation formatting package [4, Chapter 7] is a classic example of the use of a body parameter. In outline, it looks like this:

```
export "+" sup over

def @Eq
   body x
{
   def "+" ...
   def sup ...
   def over ...
   ...

   Slope @Font x
}
```

First we list those nested symbols and parameters that we intend to refer to outside the body of @Eq in an export clause, preceding the definition as shown. Only exported symbols may be invoked outside the body of @Eq. The body parameter is like a right parameter except that the exported symbols are visible within it:

```
@Eq { {x sup 2 + y sup 2} over 2 }
```

calls on the nested definitions of @Eq to produce the result

$$\frac{x^2 + y^2}{2}$$

The body parameter's value must be enclosed in braces. The term 'body parameter' is a reminder that the value is interpreted as if it was within the body of the symbol.

A body parameter may not be exported, and in fact a body parameter may be invoked only within the body of the enclosing symbol, not within any nested definitions. For example, x above may not be invoked within sup. This restriction is needed to avoid the possibility of recursion, when the actual body parameter invokes an exported nested definition which invokes the body parameter, etc.

The second place where exported symbols may be used is in the right parameter of the @Open symbol, and following its alternative form, @Use (Section 3.34).

Exported nested symbols and parameters may be made visible within a subsequent definition or macro by preceding it with an import clause, like this:

```
import @Eq
def pythag { sqrt { x sup 2 + y sup 2 } }
```

Note however that pythag can only be used with some invocation of @Eq: within the body parameter of an invocation of @Eq, within the right parameter of an @Eq&&tag @Open, or following a @Use { @Eq ... }. There may be several symbols in the import clause.

In a similar way to import, a definition may be preceded by extend followed by a symbol name:

```
extend @Eq
def pythag { sqrt { x sup 2 + y sup 2 } }
```

The effect of this is just as though the definition of pythag had occurred directly after the existing definitions within @Eq, with pythag added to @Eq's export list. This is useful for extending the capabilities of a package of definitions like @Eq without modifying its source file. The essential differences to import are that all the symbols of @Eq become visible within pythag, not just the exported ones, and only one symbol may follow the extend keyword.

Actually, more than one symbol may follow extend, but this usage indicates a 'path name' of the symbol. For example,

```
extend @DocumentLayout @ReportLayout
def @Keywords ...
```

causes the definition of @Keywords to occur directly after the existing definitions of @ReportLayout, which itself lies within @DocumentLayout.

A named parameter may also be preceded by an import clause. As usual, the meaning is that the visible local definitions of the import symbol(s) are visible within the body (the default value) of the named parameter. But furthermore, those symbols will be visible within all invocations of the parameter. For example, suppose we define

```
def @Diag
   import @Algebra named linewidth { 1p }
   import @Algebra named dashlength { 2p }
   ...
```

Then, if @Algebra exports symbols +, -, and so on, we may write

```
@Diag
   linewidth { 1f - 2p }
   dashlength { 1f + 2p }
```

using the symbols from @Algebra. There may be several symbols after the import keyword. All these symbols share an important restriction: they may not have parameters. This is necessary because Lout would be unable to determine suitable values for any such parameters, if they did exist.

As an exception to the rule just given, a named parameter may import the symbol it is a parameter of:

```
export @Cell
def @Tbl
   import @Tbl named @Format { ... }
```

In this example the exported definitions of @Tbl (i.e. @Cell) will be visible within @Format. However, they may only be used in actual parameters, not in the default value of the named parameter. This is owing to implementation problems: at the time the default value of @Format is read, the exported symbols have not been read and are consequently not known.

Since @Cell is nested within @Tbl, the value of an invocation of @Cell may depend on the value of parameters of @Tbl. If @Cell is used within an actual @Format parameter, its value depends on the value of parameters of the invocation of @Tbl of which the @Format parameter is a part.

A definition, macro, or named parameter may have several alternative names, like this:

```
macro @CD @CentredDisplay @CenteredDisplay { ... }
```

This is useful for abbreviated and alternative spellings, as shown. The names appear together, and they may subsequently be used interchangeably.

If one name of a symbol appears in an export or import list, its other names are automaticaly included as well, and should not also appear in the list.

2.4. Filtered right and body parameters

A right or body parameter may be filtered by some other computer program before being included by Lout. As an example of such a program we will use the Unix sort command:

```
sort  -o outfile  infile
```

This causes file outfile to contain a sorted copy of file infile. We incorporate this into a Lout definition as follows:

```
def @Sort
   named @Options {}
   right x
{
   def @Filter { sort @Options  -o @FilterOut  @FilterIn }

   lines @Break x
}
```

The presence within @Sort of a definition of a symbol called @Filter tells Lout that the right parameter of @Sort is to be filtered before inclusion. When @Sort is invoked, @Filter is evaluated and its value executed as a system command. In addition to the symbols ordinarily available within the body of @Filter, there are three others:

@FilterIn — the name of a file which will, at the time the system command is executed, contain the actual right or body parameter of the symbol, exactly as it appears in the input file;

@FilterOut — the name of a file of Lout text whose contents Lout will read after the system command has finished, as a replacement for what was put into file @FilterIn;

@FilterErr — the name of a file that Lout will attempt to read after the system command has finished, containing error messages produced by the command that Lout will pass on to the user as non-fatal errors. Use of this file is optional.

It is a fatal error for the system command to return a non-zero status.

Now the sort command has options -u for deleting duplicate lines, and -r for reversing the sorting order. So the result of

```
@Sort
   @Options { -r -u }
{
Austen, Jane
Dickens, Charles
Eliot, George
Hardy, Thomas
Bront{@Char edieresis}, Charlotte
}
```

is

Hardy, Thomas
Eliot, George
Dickens, Charles
Brontë, Charlotte
Austen, Jane

Unlike all the other examples in this manual, this output is simulated. This was done so that the ability to format this manual is not dependent on the existence of the Unix sort command, and it highlights the fact that filtered actual parameters are by their nature of uncertain portability.

There is no need for an actual filtered parameter to obey the lexical rules of Lout, since it is passed directly to the other program. However, Lout must be able to work out where the parameter ends, which gives rise to the following rules. As with a body parameter, a symbol @Sym with a filtered parameter must be invoked in either the form @Sym { ... } or the form @Sym @Begin ... @End @Sym, plus options as usual. In the former case, braces within the actual parameter must match; in the latter case, the actual parameter may not contain @End.

If an actual filtered parameter contains @Include, this is taken to begin a Lout @Include directive in the usual form (Section 3.40):

```
@Sort {
Austen, Jane
@Include { authors }
Hardy, Thomas
}
```

The included file becomes part of @FilterIn, but any braces, @Include, or @End within it are not noticed by Lout.

The first character of file @FilterIn will be the first non-white space character following the opening { or @Begin, or the first character of an included file if @Include comes first. The second-last character of file @FilterIn will be the last non-white space character preceding the closing } or @End @Sym, or the last character of an included file if @Include comes last. One newline character is always appended and is the last character of file @FilterIn. This effects a compromise between the Lout convention, that spaces following { or preceding } are not significant, with the Unix convention that all text files end with a newline character.

2.5. Precedence and associativity of symbols

Every symbol in Lout has a *precedence*, which is a positive whole number. When two symbols compete for an object, the one with the higher precedence wins it. For example,

```
a | b / c
```

is equivalent to { a | b } / c rather than a | { b / c }, because | has higher precedence than / and thus wins the b.

When the two competing symbols have equal precedence, Lout applies a second rule. Each symbol is either *left-associative* or *right-associative*. The value of a op1 b op2 c is taken to be { a op1 b } op2 c if the symbols are both left-associative, and a op1 { b op2 c } if they are right-associative. In cases not covered by these two rules, use braces.

It sometimes happens that the result is the same regardless of how the expression is grouped. For example, { a | b } | c and a | { b | c } are always the same, for any combination of objects, gaps, and variants of |. In such cases the symbols are said to be *associative*, and we can confidently omit the braces.

User-defined symbols may be given a precedence and associativity:

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

They come just after any into clause and before any parameter definitions. The precedence may be any whole number between 10 and 100, and if omitted is assigned the value 100. The higher the number, the higher the precedence. The associativity may be left or right, and if omitted defaults to right. Lout's symbols have the following precedences and associativities:

Precedence	Associativity	Symbols
5	associative	/ ^/ // ^//
6	associative	| ^| || ^||
7	associative	& ^&
7	associative	& in the form of one or more white space characters
10-100	left or right	user-defined symbols
100	right	@Wide, @High, @Graphic, etc.
101	-	&&
102	associative	& in the form of 0 spaces
103	-	Body parameters and right parameters of @Open

Actually the precedence of juxtaposition (two objects separated by zero spaces) is a little more complicated. If either of the two objects is enclosed in braces, the precedence is 7 as for one or more spaces. If neither object is enclosed in braces, the precedence is 102 as shown above. This complicated rule seems to accord better with what people expect and need in practice than a pure precedence rule can do.

2.6. The style and size of objects

This section explains how Lout determines the style and size of each object. Together, these attributes determine the object's final appearance in the output.

The style of an object comprises the following:

- Which font family, face and size to use (also defining the f unit);
- Whether small capitals are in effect or not;
- What gap to replace a single space between two objects by (also defining the s unit);
- The interpretation to place on white space separating two objects (lout, compress, separate, troff, or tex as in Section 3.5);
- The current value of the y and z units of measurement (Section 3.6);
- The kind of paragraph breaking to employ (adjust, ragged, etc.)
- What gap to insert between the lines of paragraphs (also defining the v unit);
- Whether to permit hyphenation or not;
- What colour the object is to appear in;
- The language of the object;
- Whether @VAdjust, @HAdjust and @PAdjust are in effect.

The style of an object depends on where it appears in the final document. For example, the style of a parameter depends on where it is used; the style of a galley is the style of the first target that it attempts to attach itself to. Of course, the style of any object can be changed by using the @Font, @Break, @Space, @SetColour or @SetColor, and @Language symbols.

There are no standard default values for style, except that small capitals are initially off, the interpretation of white space is initially lout, and the values of the y and z units are zero. Therefore one must ensure that the root galley or each of its components is enclosed in @Font, @Break, @SetColour or @SetColor, and @Language symbols. From there the style is passed to incoming galleys and the objects within them. Enclosure in @Space is not required because the s unit is also set by @Font (Section 3.5).

The remainder of this section explains how the size of each object (its width and height on the printed page) is determined. We will treat width only, since height is determined in exactly the same way, except that the complications introduced by paragraph breaking are absent.

With three exceptions (see below), the width of an object is as large as it possibly could be without violating a @Wide symbol or intruding into the space occupied by neighbouring gaps or objects. As an aid to investigating this rule, we will use the definition

```
def @TightBox right x
{
  "0 0 moveto xsize 0 lineto xsize ysize lineto 0 ysize lineto closepath stroke"
  @Graphic x
}
```

which draws a box around the boundary of its right parameter (Section 3.36) with no margin. The result of

```
5c @Wide @TightBox metempsychosis
```

is

metempsychosis

The widest that @TightBox metempsychosis could possibly be is five centimetres, and accordingly that is its width. The same applies to metempsychosis, which is five centimetres wide as well. Note carefully that there is no object in this example whose width is equal to the sum of the widths of the letters of metempsychosis.

The first of the three exceptions to the 'as wide as possible' rule is the @HContract symbol, which causes the width of its right parameter to be reduced to a reasonable minimum (a formal definition will not be attempted):

```
5c @Wide @HContract @TightBox metempsychosis
```

produces

metempsychosis

The object @HContract @TightBox metempsychosis is still five centimetres wide, but the object @TightBox metempsychosis has been reduced.

The second of the three exceptions is the horizontal concatenation symbol | (and also &). Consider this example:

```
5c @Wide @TightBox { A |1c B |1c C }
```

As usual, the right parameter of @Wide is five centimetres wide, and the result looks like this:

A B C

Lout has to apportion the size minus inter-column gaps among the three columns.

If the columns are wide enough to require paragraph breaking, Lout will assign sizes to the columns in such a way as to leave narrow columns unbroken and break wider columns to equal width, occupying the full size. Otherwise, paragraph breaking is not required, and each column will be assigned a reasonable minimum size in the manner of @HContract, except that the last column receives all the leftover width. For example,

```
5c @Wide { @TightBox A |1c @TightBox B |1c @TightBox C }
```

has result

A B C

If it is desired that the leftover width remain unused, rather than going into the last column, an empty column can be appended, or the last column can be enclosed in @HContract. Two other ways to apportion the leftover width are provided by the @HExpand and @HAdjust symbols (Sections 3.12 and 3.15).

The third and final exception to the 'as wide as possible' rule concerns the components of the root galley. Each is considered to be enclosed in @HContract and @VContract symbols.

Up to this point we have treated width as a single quantity, but of course it has two parts: width to left and right of the mark. The 'as wide as possible' rule applies to both directions:

```
@HContract { @TightBox 953^.05 /0.5c @TightBox 2^.8286 }
```

has result

953.05

2.8286

Leftover width usually goes to the right, as we have seen, but here some width was available only to the left of 2.8286 owing to the column mark alignment.

2.7. Galleys and targets

The behaviour of galleys and their targets, as described in Section 1.4, can be summarized in three laws:

First Law: The first target is the closest invocation of the target symbol, either preceding or following the invocation point of the galley as required, which has sufficient space to receive the first component;

Second Law: Each subsequent target is the closest invocation of the target symbol, following the previous target and lying within the same galley, which has sufficient space to receive the first remaining component;

Third Law: A receptive symbol that does not receive at least one component of any galley is replaced by @Null.

The terms 'closest,' 'preceding,' and 'following' refer to position in the final printed document. This section explains the operation of these laws in Basser Lout.

When a galley cannot be fitted into just one target, Lout must find points in the galley where it can be split in two. The object lying between two neighbouring potential split points is called a *component* of the galley. By definition, a component cannot be split.

To determine the components of a galley, expand all symbols other than recursive and receptive ones, discard all @Font, @Break, @Space, @SetColor, @SetColour, and @Language symbols, perform paragraph breaking as required, and discard all redundant braces. Then view the galley as a sequence of one or more objects separated by vertical concatenation symbols; these are the components and split points, except that concatenation symbols whose gaps are unbreakable (Section 3.2) are not eligible to be split points. For example, given the definition

```
def @Section into { @SectionPlace&&preceding }
   named @Title {}
   right @Body
{
   15p @Font { @Title //0.7f }
   //
   @Body
}
```

the galley

```
@Section
   @Title { Introduction }
{ This is a subject that really
needs no introduction. }
```

becomes

```
Introduction
//0.7f
{}
//
This is a subject that really needs
//1vx
no introduction.
```

with four components. If @Body had been preceded by |1.0c in the definition, the result would have been

```
Introduction
//0.7f
{}
//
|1.0c { This is a subject that really needs //1vx no introduction. }
```

with //1vx buried within one component and hence not a potential split point. If 0.7f had been 0.7fu, the gap would have been unbreakable and //0.7fu would not have been a potential split point.

Version 3.03 has liberalized this somewhat in the following way. When a component consists of a horizontal sequence of two or more objects $A_1, \dots, A_n$ separated by | (not ||, not &), Lout will investigate the component to see whether it can be broken up. It looks at each A_i to see whether it is a vertical concatenation of objects $A_{i1}, \dots, A_{im}$; if two or more of the A_i satisfy this condition, the component will not be broken up. So now suppose we have just one A_i which is a vertical concatenation. Lout will break the component into one component for each of the $A_{i1}, \dots, A_{im}$, provided that they are separated by // symbols (not /), and provided this can be done without introducing any apparent change into the appearance of the component (this second rule will be satisfied if the other A_j are not very large). The example above satisfies all these rules and will be broken up into two components, so the //1vx becomes a potential split point after all.

The lines of a paragraph become separate components if the paragraph occupies an entire component before breaking; otherwise they are enclosed in a @OneRow symbol within one component. The same is true of incoming components of other galleys. If a @Galley symbol occupies an entire component by the rules above, then the incoming components that replace it become components of their new home:

An example		An example
//0.5c	⇒	//0.5c
@Galley		Incoming components
//0.5c		//0.2c
@SomethingList		from some other galley
		//0.5c
		@SomethingList

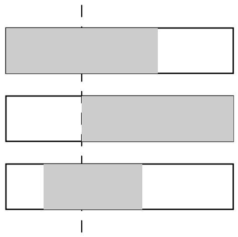
Otherwise the incoming components are grouped within a @OneRow symbol and lie within one component.

This distinction has a marked effect on the vertical concatenation symbol //1.1b, which calls for more space than is available (Section 3.2). There is no room for this symbol within any component, so it will force a split and be discarded in that case. But it can be promoted to between two components.

Components may be separated by / as well as by //, giving rise to column mark alignment between adjacent components:

When aligned components are promoted into different targets, the meaning of alignment becomes very doubtful. For example, what if the targets are in different columns of one page, or what if one lies within 90d @Rotate?

The truth is that / causes all the objects that share a mark to have equal width:



This is a consequence of the 'as wide as possible' rule (Section 2.6). Mark alignment occurs *incidentally*, whenever the fragments are placed into similar contexts.

In this connection we must also consider the special case of a @Galley symbol which shares its column mark with some other object:

```
@Galley
/0.2c
@SomethingList
```

(The @Galley may or may not occupy an entire component; that doesn't matter here.) If incoming components are separated by // rather than by /, the meaning is so doubtful that this is forbidden. In fact, a galley whose components replace such a @Galley must have a single column mark running its full length; that is, its components must all share a single column mark. This mark will be merged with the column mark passing through each @Galley that these components replace; all the objects on the resulting merged mark will have equal width.

The root galley, where everything collects immediately prior to output, is created automatically, not by a definition. Its target is the output file, and its object is the entire input, which typically looks like this:

```
@PageList
//
@Text {
  Body text of the document ...
}
```

where @PageList expands to a sequence of pages containing @TextPlace symbols (see Section 1.2), and @Text is a galley:

```
def @TextPlace { @Galley }

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

The spot vacated by a galley – its invocation point – becomes a @Null object, so this root galley is effectively @PageList alone, as required. The @Text galley will find its first target preceding its invocation point, within @PageList.

Printing the root galley on the output file is somewhat problematical, because Lout has no way of knowing how large the paper is. Basser Lout simply prints one root galley component per page (except it skips components of height zero), and the user is responsible for ensuring that each component is page-sized. Gaps between root galley components, even unbreakable ones, have no effect on the result.

Basser Lout will promote a component only after any receptive symbols within it have been replaced, either by galleys or by @Null, since until then the component is not complete. A component which shares a mark with following components is held up until they are all complete, since until then their width is uncertain.

Consider a page with @TextPlace and @FootSect receptive symbols. The rule just given will prevent the page from being printed until @TextPlace is replaced by body text, quite rightly; but @FootSect will also prevent its printing, even when there are no footnotes.

Basser Lout is keen to write out pages as soon as possible, to save memory, and it cannot afford to wait forever for non-existent footnotes. A variant of the galley concept, called a *forcing galley*, is introduced to solve this problem. A forcing galley is defined like this:

```
def @Text force into { @TextPlace&&preceding }
   ...
```

and so on. When such a galley replaces a @Galley symbol, Lout replaces every receptive symbol preceding the @Galley by @Null, thus ensuring that as soon as text enters a page, for example, everything up to and including the preceding page can be printed. This does not take care of the very last page, but Basser Lout replaces all receptive symbols by @Null when it realizes that its input has all been read, thus allowing the last page to print.

A forcing galley causes the Third Law to be applied earlier than expected, and this creates two problems. First, the replacement by @Null may be premature: a galley may turn up later wanting one of the defunct targets. Such galleys (entries in tables of contents are typical examples) are copied into the cross reference database and read in during the next run just before their targets are closed, and so they find their targets in the end. Care must be taken to ensure that large galleys such as chapters and sections do not have defunct targets, since the cost of copying them to and from the database is unacceptably high.

It is actually an over-simplification to say that these replacements occur when the forcing galley replaces its @Galley. What really happens is that from this moment on Lout understands that it has the right to make these replacements, and it will do each one at the first moment when not doing it would hold things up. So there is a short period of grace when galleys, such as the entries in tables of contents just alluded to, can sneak into these receptive symbols.

The into and force into forms are actually just abbreviations for the true way that galleys are defined, which is by giving the symbol that is to be a galley a parameter or nested definition with the special name @Target:

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

   x
}
```

A forcing galley is obtained by using &&& instead of &&. @Target may be an arbitrary object, provided that it yields such a cross reference when evaluated. In this way, different invocations may have different targets.

The forcing galley effect can be obtained in another way, by replacing the @Galley symbol to which the galley is attached by @ForceGalley. The advantage of this form is that the galley can then be forcing at some places and not at others, using the formula

```
def @SomePlace right x
{
   x @Case {
      noforce  @Yield @Galley
      force    @Yield @ForceGalley
   }
}
```

Now a galley may have @SomePlace for its target, and if it happens to attach to

```
@SomePlace force
```

it will have the effect of a forcing galley, while if it happens to attach to

```
@SomePlace noforce
```

it will not.

Although it doesn't matter whether a galley is declared as a forcing galley or merely arrives at a @ForceGalley symbol from the point of view of the effect on nearby targets, there is one way in which Lout treats the two cases differently. If a forcing galley's first component does not fit into the available space, that component will be scaled vertically until it does. The rationale for this is that forcing galleys are meant to carry the bulk of the document and cannot afford to be held up because the user has inadvertently included an over-high component, which for all Lout knows to the contrary may not fit on any page. If this scaling is not wanted but forcing is, the galley may be declared not forcing but all its targets may be set to contain @ForceGalley.

Within a galley, a symbol whose name is @Enclose has a special meaning: when components of the galley replace a @Galley or @ForceGalley symbol, that symbol is first replaced by @Enclose @Galley or @Enclose @ForceGalley. For example,

```
def @Enclose
   right x
{
   @Box x
}
```

within some galley definition causes each @Galley or @ForceGalley symbol that receives components of the galley to be replaced by @Box @Galley or @Box @ForceGalley, assuming an appropriate definition of @Box. This is useful, for example, when producing multi-page boxed displays, figures, and tables.

An @Enclose symbol may have only one parameter, which must be a right parameter. It would not make sense to allow more parameters, since there is no suitable value to assign to them. However, the @Enclose symbol may contain inner definitions, and it may make use of any symbol that is available at that point, in the usual way.

A following galley may fail to find a first target lying in a following component of the same galley as its invocation point. This is a deficiency of Basser Lout, which occurs if the target has not been read from input at the time the galley tries to find it. A workaround is to use a preceding galley instead, defined like this:

```
def @AGalley into { @AGalleyPlace&&preceding }
   right @Body
{
   //1.1b
   @Body
}
```

and invoked like this:

```
@AGalleyPlace | @AGalley { content of galley }
//
...
@AGalleyPlace
```

The first @AGalleyPlace receives only the initial empty object, since the //1.1b forces a split; and the Second Law puts Basser Lout on the right track thereafter.

2.8. Sorted galleys

When footnotes are placed at the bottom of a page, they appear there in first come, first served order. To make galleys appear in sorted order, as is needed in bibliographies and indexes, a parameter or nested definition with the special name @Key is added to the galley definition, like this:

```
def @IndexEntry into { @IndexPlace&&following }
  left @Key
  right x
{ x }
```

@Key must be set to a simple word, or several words with nothing more complex than font changes within them, when the galley is invoked:

```
{ cities compare } @IndexEntry { cities, comparison of, 27 }
```

and this key is used to sort the galleys.

If several sorted galleys with the same key are sent to the same place, the default behaviour is to print only the first of them; the assumption is that the others are probably unwanted duplicates. This holds good for sorted reference lists, for example: we don't want two copies of a reference just because we happen to cite it twice. However, the other common example of sorted galleys, index entries, requires something different: *merged* galleys.

Suppose that at some point of the document we insert the index entry

```
aardvarks @IndexEntry { Aardvarks, 23 }
```

while at another point we insert

```
aardvarks @IndexEntry { Aardvarks, 359 }
```

How the page numbers are worked out is not relevant here. Clearly we would like to merge these two entries into one entry that comes out as

Aardvarks, 23, 359

The following definition will merge two objects in this way:

```
def @Merge left x right y
{
   {x @Rump y} @Case
   {
      ""     @Yield x
      else   @Yield { x, x @Rump y }
   }
}
```

The @Rump symbol is the subject of Section 3.23; this says 'if the two things to be merged are equal, the result is one of them; otherwise it is the first followed by a comma and space and then the rump of the second.' Our only problem is that this symbol has to be applied to two galleys from widely separated parts of the document.

Lout makes this possible by the following special rule: if a sorted galley contains a nested definition of a symbol whose name is @Merge (@Merge must have just two parameters, left and right), and if that sorted galley is preceded in the list of sorted galleys destined for some target by another sorted galley with the same key, then rather than being discarded, the second galley is merged into the first using the @Merge symbol.

The natural thing to do when more than two galleys have the same key is to merge the first two, then merge the third with the result of that, then the fourth with the result of that, and so on. For efficiency reasons beyond our scope here, Lout does the merging in a different order: it merges n galleys by merging the first $\lfloor n/2 \rfloor$ together, then the last $\lceil n/2 \rceil$ together, then merging the result. Of course, if the @Merge symbol is associative this has the same effect. The @Merge symbol above is not strictly associative, but it is close enough in practice. The total time it takes to merge n galleys with equal keys is $O(n^2)$ or somewhat higher (but always polynomial in n) depending on how many times the parameters occur within the body of @Merge; to do it in the natural linear order would take Lout exponential time.

2.9. Horizontal galleys

All the galleys so far have been *vertical galleys*: galleys whose components are separated by vertical concatenation symbols. There are also horizontal galleys, whose components are separated by the horizontal concatenation operator & (or equivalently, by spaces). These work in the same way as vertical galleys, except for the change of direction. For example, the following defines the equivalent of an ordinary outdented paragraph, except that an option is provided for varying the size of the outdent:

```
def @OutdentPar
   named outdent { 2f }
   right x
{
   def @ParPlace { @Galley }

   def @LineList
   {
      outdent @Wide {} | @PAdjust @ParPlace
      //1vx @LineList
   }

   def @ParGalley force horizontally into { @ParPlace&&preceding }
      right x
   {
      x
   }

      @PAdjust @ParPlace
   //  @ParGalley { x &1rt }
   //1vx  @LineList
}
```

Notice the use of &1rt to cancel the effect of @PAdjust on the last line of the paragraph. This definition has a problem in that there will be a concluding unexpanded @LineList symbol which will hold up promotion of the enclosing galley; this problem may be fixed by the same method used to end a list.

In an ideal world, there would be nothing further to say about horizontal galleys. However there are a few differences which arise from various practical considerations and limitations. Perhaps some day a more perfect symmetry will be implemented.

Each vertical galley has a fixed finite width, and every component is broken to that width. This is needed basically to trigger paragraph breaking. However, there is no equivalent of paragraph breaking in the vertical direction, so horizontal galleys do not have any particular fixed height. Instead, each component has its own individual height.

When two objects are separated by /, they are assigned the same width (Section 2.7), and this holds true even if the two objects are subsequently separated by being promoted into different targets. For example, two aligned equations will have the same width, and hence their alignment will be preserved, even if they appear in different columns or pages. However, even though & aligns the marks of its two parameters, it does not assign them a common height. This means that the height of any component of a horizontal galley promoted into one target does not affect the height consumed by the components promoted into any other target. The other horizontal concatenation operator, |, does assign a common height to its two parameters; but sequences of objects separated by this operator cannot be the components of a horizontal galley.

Lout is able to read vertical galleys one paragraph at a time; in this way it processes the document in small chunks, never holding more than a few pages in memory at any time. However, horizontal galleys are always read in completely, so they should not be extremely long.

In principle Lout should be able to hyphenate the components of horizontal galleys when they are simple words, but this is not implemented at present.

In an ideal world, every paragraph would be treated as a horizontal galley. However, to do so in practice would be too slow and would lead to excessive clumsiness in notation, so at present Lout has two competing mechanisms in this area: the built-in paragraph breaker with its limited set of options as given under the @Break operator, and horizontal galleys. As the example above shows, horizontal galleys are in principle capable of implementing many more paragraph styles than the built-in paragraph breaker could ever hope to do. The recommended practical strategy is to use the built-in paragraph breaker most of the time, and switch to horizontal galleys only for occasional tricks, such as paragraphs with drop capitals, circular outlines, etc.

2.10. Optimal galley breaking

As explained in Section 2.7, the components of a galley are promoted one by one into a target. When space runs out there, the galley searches for a new target and promotion resumes.

This process is exactly analogous to placing words onto a line until space runs out, then moving to another line. But, as we know, that simple method is inferior to the optimal paragraph breaking used by Lout (copied from the TEX system), which examines the entire paragraph and determines the most even assignment of words to lines.

Lout offers *optimal galley breaking*, the equivalent for galleys of optimal paragraph breaking. Optimal galley breaking can reduce the size of ugly blank spaces at the bottom of pages preceding large unbreakable displays, sometimes quite dramatically.

Optimal galley breaking is applied to each galley, horizontal or vertical, that possesses a parameter or nested symbol called @Optimize whose value is Yes. Like cross referencing, it takes two runs to have effect. On the first run, Lout records the sizes of the galley's components and gaps, and also the space available at each of its targets. At end of run this information is used to find an optimal break, which is written to the cross-reference database. On the second run, the optimal break is retrieved and used.

Considering that this process must cope with floating figures, new page and conditional new page symbols, breaks for new chapters, and evolving documents, it is surprisingly robust. If it does go badly wrong, removing file lout.li then running Lout twice without changing the document may solve the problem. However, cases are known where the optimization never converges. These are usually related to figures and footnotes whose anchor points fall near page boundaries.

Chapter 3. Predefined symbols

3.1. @Begin and @End

The body of a symbol @Sym may be enclosed in @Begin and @End @Sym instead of the more usual braces:

```
def @Section
   named @Title {}
   right @Body
@Begin
   @Title //2v @Body
@End @Section
```

They may also enclose the right or body parameter of a symbol invocation:

```
@Chapter
   @Title { Introduction }
@Begin
This subject needs no introduction.
@End @Chapter
```

Apart from their utility as documentation aids, these forms allow Basser Lout to pinpoint mismatched braces, which can otherwise create total havoc. For this reason, they should enclose the major parts of documents, such as chapters and sections. Note that braces cannot be replaced by @Begin and @End in general.

3.2. Concatenation symbols and paragraphs

There are ten concatenation symbols, in three families:

/	^/	//	^//	Vertical concatenation
|	^|	||	^||	Horizontal concatenation
&	^&			In-paragraph concatenation

Each symbol produces an object which combines together the two parameters. The right parameter must be separated from the symbol by at least one white space character.

The vertical concatenation symbol / places its left parameter above its right parameter with their column marks aligned. If one parameter has more column marks than the other, empty columns are inserted at the right to equalize the numbers. The variant // ignores column marks and left-justifies the objects.

The horizontal concatenation symbols | and || are horizontal analogues of / and //: they place their two parameters side by side, with row mark alignment or top-justification respectively. The in-paragraph concatenation symbol & produces horizontal concatenation within a paragraph; its special properties are treated in detail at the end of this section.

The concatenation symbols in any one family are *mutually associative*, which means that

{ *x* |*p* *y* } |*q* *z*

is always the same as

x |*p* { *y* |*q* *z* }

for any objects *x*, *y*, and *z*, any gaps *p* and *q* (defined below), and any choice of |, ^|, ||, and ^||. In practice we always omit such braces, since they are redundant and can be misleading. The result of the complete sequence of concatenations will be called the *whole concatenation object*, and the objects which make it up will be called the *components*.

One mark is designated as the *principal mark*, usually the mark of the first component. A later mark can be chosen for this honour by attaching ^ to the preceding concatenation symbol. See Section 3.9 for examples.

A *gap*, specifying the distance between the two parameters, may follow any concatenation symbol. There may be no spaces between a concatenation symbol and its gap. A missing gap is taken to be 0ie. The gap is effectively a third parameter of the concatenation symbol, and it may be an arbitrary object provided that it evaluates to a juxtaposition of simple words. In general, the gap must be enclosed in braces, like this:

```
//{ @Style&&mystyle @Open { @TopMargin } }
```

but the braces may be omitted when the object is a juxtaposition of simple words or an invocation of a symbol without parameters, as in //0.3vx and ||@Indent.

A gap consists of a length plus a gap mode plus an optional indication of unbreakability. A *length* is represented by an decimal number (which may not be negative) followed by a unit of measurement. For example, 2.5c represents the length 2.5 centimetres. Figure 3.1 gives the full selection of units of measurement.

After the length comes an optional *gap mode*, which is a single letter following the length, indicating how the length is to be measured. As shown in Figure 3.2, with edge-to-edge gap mode the length *l* is measured from the trailing edge of the first object to the leading edge of the second. Edge-to-edge is the default mode: the e may be omitted. Hyphenation gap mode is similar, except as explained at the end of this section.

Mark-to-mark, overstrike, and kerning measure the length from the last mark of the first object to the first mark of the second. In the case of mark-to-mark, if the length is too small to prevent the objects almost overlapping, it is widened until they no longer do. (The extra *l/10* is not applied when plain text output is in effect.) Kerning also widens, with the aim of preventing the mark of either object from overlapping the other object; this mode is used for subscripts and superscripts.

Tabulation ignores the first object and places the leading edge of the second object at a distance *l* from the left edge of the whole concatenation object. It is the main user of the b and r units of measurement; for example, |1rt will right-justify the following component, and |0.5rt

c	Centimetres.
i	Inches.
p	Points (72p = 1i).
m	Ems (12m = 1i).
f	One f equals the size of the current font, as specified by the @Font symbol (Section 3.3). This unit is appropriate for lengths that should change with the font size.
s	One s equals the preferred gap between two words in the current font, as specified in the definition of the font, or by the @Space symbol (Section 3.4).
v	One v equals the current gap between lines introduced during paragraph breaking, as specified by the @Break symbol (Section 3.4). This unit is appropriate for lengths, such as the spaces between paragraphs, which should change with the inter-line gap.
w	One w equals the width of the following component, or its height if the symbol is vertical concatenation.
b	One b equals the width of the whole concatenation object, or its height if the symbol is vertical concatenation.
r	One r equals one b minus one w. This unit is used for centring, and for left and right justification.
d	Degrees. This unit may only be used with the @Rotate symbol.
y	One y equals the current value set by the @YUnit symbol (Section 3.6). This unit is not used internally by Lout; it is included for the convenience of application packages.
z	One z equals the current value set by the @ZUnit symbol (Section 3.6). This unit is not used internally by Lout; it is included for the convenience of application packages.

Figure 3.1. The thirteen units of measurement provided by Lout.

will centre it.

The value |0rt separating the first and second items in a sequence of horizontally concatenated objects is somewhat special in that it denotes left justification of the object to its left in the available space. This is identical with |0ie when the object to the left also has the principal mark; but when it does not, |0rt will cause the object to the left to appear further to the left than it would otherwise have done, if space to do so is available.

A gap is optionally concluded with an indication of unbreakability, which is a letter u appended to the gap. A paragraph will never be broken at an unbreakable gap, nor will a galley be broken across two targets at such a gap. Basser Lout's implementation is slightly defective in that it ignores any unbreakable indication in the gap separating the first component promoted into any target from the second.

When two objects are separated only by zero or more white space characters (spaces, tabs, newlines, and formfeeds), Lout inserts &*k*s between the two objects, where *k* is the number of spaces. Precisely, *k* is determined by discarding all space characters and tabs that precede newlines (these are invisible so are better ignored), then counting 1 for each newline, formfeed or space, and 8 for each tab character. The gap will be unbreakable if *k* is zero.

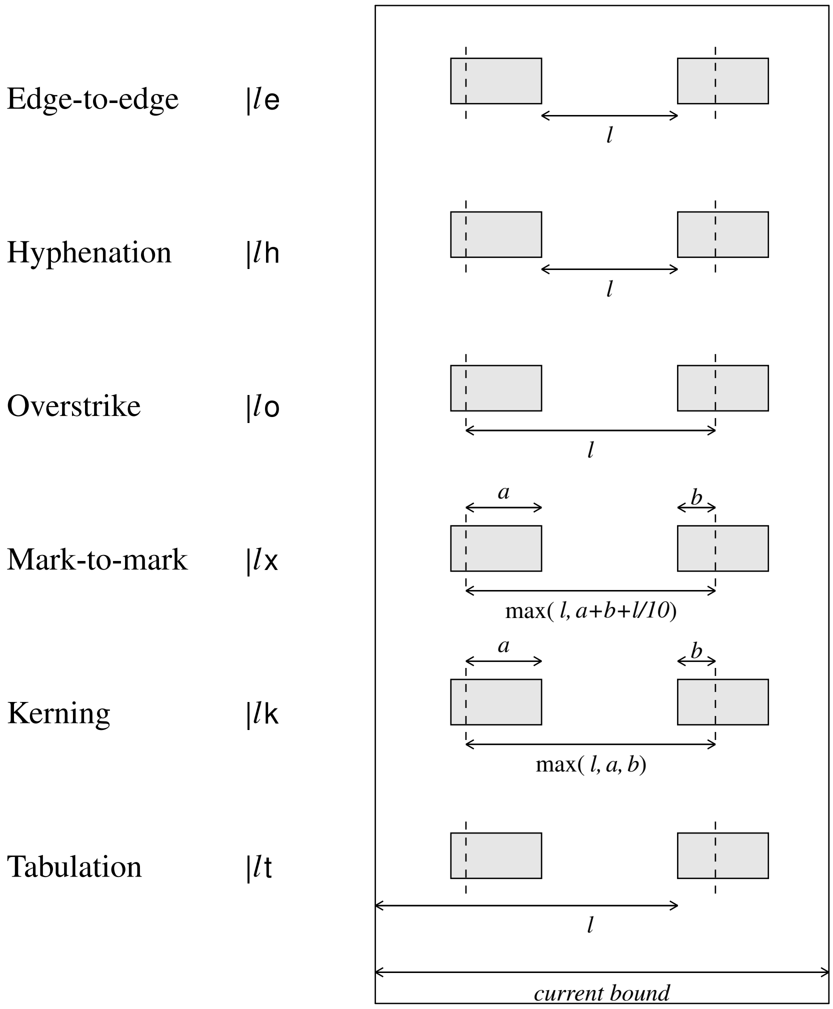


Figure 3.2. The six gap modes provided by Lout.

A sequence of two or more objects separated by & symbols is a *paragraph*. Lout breaks paragraphs into lines automatically as required, by converting some of the & symbols into //1vx. Unbreakable gaps are not eligible for this conversion. 'Optimal' line breaks are chosen, using a method adapted from TEX [5].

If an & symbol whose gap has hyphenation mode is chosen for replacement by //1vx, a hyphen will be appended to the preceding object, unless that object is a word which already ends with a hyphen. For example,

```
Long words may be hyph &0ih enat &0ih ed.
```

could have the following result, depending where the line breaks fall:

Long words may be hyphenat-
ed.

Basser Lout inserts hyphenation gaps automatically as required, again following the method of TEX, which approximates the hyphenations in Webster's dictionary. To prevent the hyphenation of a single word, enclose it in quotes. Further control over paragraph breaking and hyphenation is provided by the @Break and @Space symbols (Sections 3.4 and 3.5).

3.3. @Font and @Char

A *font* is a collection of characters which may be printed. Many fonts come in *families*, which are groups of fonts that have been designed to go together. For example, the Times family includes the following fonts:

Times Base
Times Slope
Times Bold
Times BoldSlope

Thus, each font has two names: its *family name* (Times, Helvetica, etc.) and its *face name* (Base, Slope, etc.). Times Base is more commonly called Times Roman, and Times Slope is more commonly called Times Italic. Lout avoids these names in favour of generic names which can be applied to many font families.

Ligatures, such as ﬂ for fl and ﬁ for fi, are considered by Basser Lout to be an integral part of the font: if the font definition (see below) mentions them, they will be used. Similarly, kerning (fine adjustment of the space between adjacent characters to improve the appearance) is done whenever indicated in the font definition. Enclosing one of the letters in @OneCol is one sure way to disable a ligature or kern.

The @Font symbol returns its right parameter in a font and size specified by its left:

{ Times Base 12p } @Font *object*

The family and face names must have appeared together in a fontdef; the size is arbitrary and may be given in any one of the c, i, p, m, f, s, and v units of measurement (Section 3.2), although 10p and 12p are the most common sizes for text. There may be empty objects and @Null objects in the left parameter of @Font; these are ignored.

When a @Font symbol is nested inside the right parameter of another @Font symbol, the inner one determines the font of its own right parameter. However, it may be abbreviated so as to inherit part of the outer symbol:

```
{ Times Base 12p } @Font
{ hello, Slope @Font hello, 15p @Font hello }
```

has result

hello, *hello,* hello

The first inner @Font inherits the outer family and size, changing only the face; the second inherits the outer family and face. When a family name is given, it must be followed immediately by a face name. A size change may appear first or last.

Sizes of the form +*length* and –*length* may also be used, meaning that the font size is to be *length* larger or smaller than the inherited value. For example, –2p is often used for superscripts and subscripts. These forms are highly recommended, since they don't need to be changed if a decision is made to alter the font size of the document as a whole.

The @Font symbol also switches to and from small capitals:

```
smallcaps @Font ...
nosmallcaps @Font ...
```

These may be nested, and they cooperate with other font changes. The precise effect depends on the font (see below). There is a default value (nosmallcaps), so it is not necessary to mention this attribute when giving an initial font.

There are two predefined symbols, @CurrFamily and @CurrFace, which respectively return the family and face names of the current font. For example, right now @CurrFamily is Times and @CurrFace is Base.

When Lout runs, the first thing it reads is a list of font definitions, like these:

fontdef Times Base { *implementation-dependent* }
fontdef Times Slope { *implementation-dependent* }

Each line tells Lout of the existence of a font, and assigns it a family name and a face name. There are a few fonts which are the only members of their families; even though these fonts do not need a face name, they must be given one by their fontdef. The part between braces may vary with different implementations of Lout; it is supposed to contain the information Lout needs to work with the font.

In Basser Lout Version 3, this implementation-dependent part consists of a PostScript font name, an Adobe font metrics (formerly AFM) file whose FontName entry must agree with the PostScript font name just mentioned, the name of a Lout Character Mapping (LCM) file, and the word Recode or NoRecode:

```
fontdef Times Base { Times-Roman Ti-Ro LtLatin1.LCM Recode }
```

It seems to have become an invariable rule at Adobe Systems that the font name and the file name should be the same, but some operating systems do not permit such long file names, so Lout does not use them. The files are searched for in standard places. Consult the PostScript Reference Manual [1] for general information about fonts and encoding vectors; briefly, an 8-bit character code *c* in Lout's input is mapped to the character in the Adobe font metrics file whose name appears on the line labelled *c* in the LCM file. The LCM file also defines various character-to-character mappings, such as upper-case to lower-case, which are used for various purposes, such as the production of small capitals. If the word NoRecode appears, Lout assumes that the given encoding vector is already associated with this font in the PostScript interpreter, and optimizes its output accordingly.

If there are two font definitions with the same family and face name, it is an error unless the second one has the same font name and file name as the first, in which case it is silently ignored. A PostScript font name and file may appear in two or more font definitions, allowing one PostScript font to have two or more equally valid Lout names, and different LCM files.

The @Char symbol allows a character to be specified by its name (its PostScript name in Basser Lout) rather than by its code:

@Char nine

is equivalent to 9 in most fonts. This is useful as a documentation aid and to be sure of getting the right character even if the encoding vector of the font is changed. However @Char will fail if the character named is not in the encoding vector of the current font.

3.4. @Break

The @Break symbol influences the appearance of paragraphs (Section 3.2), offering a fixed set of nine styles:

adjust @Break *object*	Break the paragraphs of *object* into lines, and apply @PAdjust (Section 3.15) to every line except the last in each paragraph;
outdent @Break *object*	Like adjust, except that 2.0f @Wide {} &0i is inserted at the beginning of every line except the first, creating an outdented paragraph;
ragged @Break *object*	Break the paragraphs of *object* into lines, but do not adjust the lines ('ragged right');
cragged @Break *object*	Like ragged, except that each line will be centred with respect to the others;
rragged @Break *object*	Like ragged, except that each line will be right-justified with respect to the others ('ragged left');
oragged @Break *object*	The obvious combination of ragged and outdent;
lines @Break *object*	Break the paragraphs of *object* into lines at the same points that they are broken into lines in the input, and also at concatenation symbols of the form &*k*b for any *k* greater than 1. Do not adjust the lines. Any spaces at the start of a line other than the first line will appear in the output;
clines @Break *object*	Break the paragraphs of *object* into lines as for lines @Break, then centre each line with respect to the others;
rlines @Break *object*	Break the paragraphs of *object* into lines as for lines @Break, then right-justify each line with respect to the others.

If the paragraph was an entire component of a galley, so will each of its lines be; otherwise the lines are enclosed in a @OneRow symbol after breaking.

The length of the gap used to separate the lines produced by paragraph breaking is always 1v. However, the v unit itself and the gap mode may be changed:

gap @Break *object* — Within *object*, take the value of the v unit to be the length of *gap*;

+*gap* @Break *object* — Within *object*, take the value of the v unit to be larger by the length of *gap* than it would otherwise have been;

–*gap* @Break *object* — Within *object*, take the value of the v unit to be smaller by the length of *gap* than it would otherwise have been.

In each case, the mode of *gap* is adopted within *object*.

The @Break symbol also controls hyphenation:

hyphen @Break *object* — Permit hyphenation within the paragraphs of *object*;

nohyphen @Break *object* — Prohibit hyphenation within the paragraphs of *object*; all hyphenation gaps without exception revert to edge-to-edge mode.

The @Break also has options which control widow and orphan lines:

unbreakablefirst @Break *object* — Prevent column and page breaks (i.e. prevent a galley from splitting) between the first and second lines of the paragraphs of *object*;

unbreakablelast @Break *object* — Prevent column and page breaks between the last and second last lines of the paragraphs of *object*.

These options work by adding the u (unbreakable) suffix to the appropriate gaps during paragraph breaking, so their precise effect is as described for this suffix. These options may be countermanded by breakablefirst @Break and breakablelast @Break.

Several options may be given to the @Break symbol simultaneously, in any order. For example,

```
{ adjust 1.2fx hyphen }  @Break  ...
```

is a typical initial value. There may be empty objects and @Null objects in the left parameter of @Break; these are ignored.

3.5. @Space

The @Space symbol changes the value of the s unit of measurement (Section 3.2) within its right parameter to the value given by the left parameter:

```
1c @Space { a b c d }
```

has result

a   b   c   d

As for the @Break symbol, the left parameter of @Space may be given relative to the enclosing s unit, and it may include a gap mode. Note that the @Font symbol also sets the s unit.

The left parameter of the @Space symbol may also hold any one of the five special values lout, compress, separate, troff, and tex, which control the way in which Lout treats white space separating two objects. The names troff and tex indicate that the behaviour of these options is inspired by these other document formatting systems.

The default setting, lout, produces as many spaces in the output as there are in the input. The compress setting causes all sequences of two or more white space characters to be treated the same as one white space character. The separate setting is like compress but also causes zero white spaces between two objects (but not within one word) to be treated the same as one white space character.

The troff setting is the same as lout except that wherever a sentence ends at the end of a line, one extra space is added. Formally, when two objects are separated by white space characters which include at least one newline character, and the first object is a word ending in any one of a certain set of sequences of characters, the extra space is added. The set of sequences of characters depends on the current language and is defined in the langdef for that language (see Section 3.8).

The tex option is the most complicated. First, the compress option is applied. Then, at every sentence ending, whether or not at the end of a line, one extra space is added. A sentence ending is defined as for troff except that, in addition to the preceding word having to end in one of a certain set of sequences of characters, the character preceding that sequence must exist and must be a lower-case letter. A character is a lower-case letter if, in the Lout Character Mapping file (Section 3.3) associated with the current font, an upper-case equivalent of the character is defined.

3.6. @YUnit and @ZUnit

The @YUnit symbol changes the value of the y unit of measurement (Section 3.2) within its right parameter to the value given by the left parameter:

```
1c @YUnit { ... }
```

ensures that the value of 1y within the right parameter will be 1c. The @ZUnit symbol is similar, setting the value of the z unit in its right parameter. Both units have default value zero. The left parameter may not include a gap mode, nor may it use the w, b, r, or of course d units, but it may begin with + or - to indicate that value is to be added to or subtracted from the current value. Any negative result of using - will be silently replaced by zero.

These units are not used internally by Lout. They are supplied as part of the style information for the convenience of application packages. For example, the Eq equation formatting package uses them to fine-tune the appearance of equations.

3.7. @SetColour and @SetColor

The @SetColour and @SetColor symbols, which have identical effect, return their right parameter in the colour specified by their left parameter. The form of the left parameter is implementation-dependent; in Basser Lout it must be an object whose value is a sequence of words comprising a PostScript command for setting colour. For example,

```
{ 1.0 0.0 0.0 setrgbcolor } @SetColour { hello, world }
```

produces the red result

hello, world

Of course, a colour output device is needed to see the effect; on a monochrome device the result will be some shade of grey. The @SetColour command will not accept an empty left parameter (it refuses to believe that this could cause a change of colour), but otherwise it trusts the user to supply something sensible.

The @SetColour command accepts the special value nochange for the left parameter. This value causes the right parameter to have the colour it would have had without the @SetColour command.

There is no default colour, so the user must ensure that the root galley or each of its components is enclosed in a @SetColour symbol whose left parameter is not nochange.

Lout makes no attempt to understand colour, it simply prints the PostScript commands when appropriate. This has the advantage of permitting access to any of PostScript's colour models (some require initialization which can be supplied using @PrependGraphic), but the disadvantage of offering no way to make relative changes ('as before only redder,' and so on).

For those who wish to obtain colour without working very hard, the setrgbcolor command used above is available in every version of PostScript, requires no initialization, and is simple to use. The three numbers, which range from 0.0 to 1.0, determine the intensity of red, green, and blue respectively. Some useful values for the left parameter are

1.0 0.0 0.0 setrgbcolor	*red*
0.0 1.0 0.0 setrgbcolor	*green*
0.0 0.0 1.0 setrgbcolor	*blue*
1.0 1.0 1.0 setrgbcolor	*white*
0.5 0.5 0.5 setrgbcolor	*grey*
0.0 0.0 0.0 setrgbcolor	*black*

Colouring an object white is useful for producing an empty space whose size is that of some object.

3.8. @Language and @CurrLang

The @Language symbol informs Lout that its right parameter is written in the language of its left parameter:

```
Danish @Language { ... }
```

Basser Lout Version 3 uses this information in two ways: to hyphenate words appropriately to that language, and to change the value of the @CurrLang symbol (see below). Other uses, such as right-to-left formatting of certain languages, may be added in the future.

The left parameter must either be empty (which means to leave the current language unchanged) or else it must have been given in a langdef language definition at the beginning of the input:

langdef Danish Dansk { *implementation-dependent* }

After langdef comes a sequence of one or more simple words, which are alternative names for the language being defined. Following them comes an implementation-dependent part between braces. In Basser Lout Version 3 this part contains the name of the Lout hyphenation information file (minus its .lh suffix) to be used when hyphenating words in this language, followed by a sequence of words which define the ends of sentences. For example:

```
langdef English { english . : ? ! .) ?) !) }
```

defines a language called English with hyphenation patterns file english.lh and seven ways to end a sentence. The use of these sentence endings is described in Section 3.5. If there is no hyphenation file available, this is indicated by writing - for the file name; if there are no sentence ends, they are simply omitted.

The @CurrLang symbol, which has no parameters, evaluates to the first name given in the langdef of the language in force at the point where it is invoked:

```
Dansk @Language { This is @CurrLang. }
```

has result

This is Danish.

It is typically used with the @Case symbol like this:

```
@CurrLang @Case {
   Danish @Yield tirsdag
   English @Yield Tuesday
   French @Yield Mardi
}
```

This example evaluates to the name of the third day of the week in the current language.

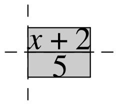
The current language is part of the style of an object, like its font. As explained in Section 2.6, style is inherited through the point of appearance, which for language can be unexpected. For example, an index entry which originates in a French chapter but appears in an English index will have English for its language, so must be explicitly set to French using @Language.

3.9. @OneCol and @OneRow

The @OneRow symbol returns its right parameter modified so that only the principal row mark protrudes. This is normally the first row mark, but another one may be chosen by preceding it with ^/ or ^//. For example,

```
@OneRow {  |0.5rt  Slope @Font x + 2  ^//1p  @HLine  //1p  |0.5rt  5 }
```

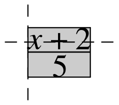
has result



with one row mark protruding from the bar as shown. Compare this with

```
@OneRow {  |0.5rt  Slope @Font x + 2  //1p  @HLine  //1p  |0.5rt  5 }
```

where the mark protrudes from the numerator:



@OneCol has the same effect on columns as @OneRow does on rows, with the symbols ^| and ^|| (or ^&) determining which mark is chosen.

3.10. @Wide and @High

The @Wide symbol returns its right parameter modified to have the width given by its left parameter, which must be a length (Section 3.2) whose unit of measurement is c, i, p, m, f, s, or v. If the right parameter is not as wide as required, white space is added at the right; if it is too wide, its paragraphs are broken (Section 3.4) so that it fits. A @OneCol operation is included in the effect of @Wide, since it does not make sense for an object of fixed width to have two column marks.

The @High symbol similarly ensures that its result is of a given height, by adding white space at the bottom. In this case it is an error for the right parameter to be too large. A @OneRow operation is included.

3.11. @HShift and @VShift

The @HShift symbol returns its right parameter with principal mark shifted as prescribed by its left parameter:

+*length* @HShift *object*	Principal mark shifted to the right by *length*;
-*length* @HShift *object*	Principal mark shifted to the left by *length*;
length @HShift *object*	Principal mark shifted so as to lie *length* to the right of the left edge of *object*;

In each chase @HShift includes a @OneCol effect.

The units of measurement of *length* may be c, i, p, m, f, s, v, or w. In the latter case, 1w is taken to be the width of the right parameter, so that, for example, 0.5w @HShift will centre the principal column mark within the right parameter.

The @VShift symbol is similar except that it applies vertically to the principal row mark: +*length* shifts it down, -*length* shifts it up, and *length* shifts it to *length* below the top edge of the object. With @VShift, 1w is taken to be the height of the right parameter.

3.12. @HExpand and @VExpand

The @HExpand symbol causes its right parameter to be as wide as it possibly could be without violating a @Wide symbol or intruding into the space occupied by neighbouring gaps or objects. The @VExpand symbol is similar, but it affects height. For example, in the object

```
8i @Wide 11i @High {
   //1i  ||1i @HExpand @VExpand x ||1i
   //1i
}
```

object x could have any size up to six inches wide by nine inches high, so the @HExpand and @VExpand symbols cause it to have exactly this size. This is important, for example, if x contains |1rt or /1rt; without the expansion these might not move as far across or down as expected.

As Section 2.6 explains in detail, most objects are already as large as they possibly could be. Consequently these symbols are needed only rarely. @HExpand includes a @OneCol effect, and @VExpand includes a @OneRow effect.

3.13. @HContract and @VContract

The @HContract symbol reduces the size of its right parameter to a reasonable minimum (after paragraph breaking). For example,

```
5i @Wide @HContract { A |1rt B }
```

has result

AB

in which the B is much closer to the A than it would otherwise have been. @VContract is similar, but in a vertical direction. See Section 2.6 for a more extensive discussion.

3.14. @HLimited and @VLimited

The @HLimited symbol limits the width available to recursive and receptive symbols within its right parameter to whatever is available without increasing the existing size of the @HLimited object. So this symbol acts like @Wide with respect to limiting the space occupied by recursive and receptive symbols, except that instead of enforcing a fixed constant limit, it enforces whatever size is already in place.

The @VLimited symbol is exactly the same, except that it applies vertically rather than horizontally.

3.15. @HAdjust, @VAdjust, and @PAdjust

These symbols spread their right parameter apart until it occupies all the space available to it; @HAdjust adjusts | sequences, @VAdjust adjusts / sequences, and @PAdjust adjusts & sequences. For example,

```
4i @Wide @PAdjust { 1 2 3 4 5 6 7 8 }
```

has result

1 2 3 4 5 6 7 8

More precisely, the widening is effected by enlarging the size of each component except the last by an equal fraction of the space that would otherwise be left over – just the opposite of the usual procedure, which assigns all the leftover space to the last component (Section 2.6).

@PAdjust is used by the adjust and outdent options of the @Break symbol (Section 3.4). It has a slight peculiarity: it will not enlarge components when the immediately following gap has width 0. This is to prevent space from appearing (for example) between a word and an immediately following comma. The other two symbols will enlarge such components.

3.16. @HScale and @VScale

@HScale causes its right parameter to expand to fill the space available, by geometricallly scaling it:

```
4i @Wide @HScale { 1 2 3 4 5 6 7 8 }
```

has result

1 2 3 4 5 6 7 8

and

```
0.5i @Wide @HScale { 1 2 3 4 5 6 7 8 }
```

has result

12345678

@HScale first applies @HContract to its parameter, then horizontally scales it to the actual size. The principal mark of the right parameter has no effect on the result; the parameter is scaled to the actual size and positioned to fill the space available. (Taking account of alignment of the principal mark only causes trouble in practice.)

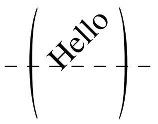
@VScale is similar, but in a vertical direction. @HScale and @VScale each have both a @OneCol and a @OneRow effect.

3.17. @HCover and @VCover

The @VCover symbol vertically scales its right parameter so that it covers every object that shares its row mark. For example,

```
@VCover ( 45d @Rotate Hello @VCover )
```

produces



The row mark has been added to show clearly where it lies. This should be compared with

```
@VScale ( 45d @Rotate Hello @VScale )
```

which produces

Scaling abandons mark alignment and so is able to exactly cover the rest of the row, whereas covering preserves mark alignment and so is obliged in general to more than cover the rest of the row.

If the parameter of @VCover has zero vertical size, this is taken to mean that covering is not required after all and the @VCover is silently ignored. If however the parameter has non-zero size above the mark but zero size below, or vice versa, this is taken to be an error since scaling cannot make the parameter cover the rest of the row.

@HCover is similar, horizontally covering all objects that share its column mark. Neither symbol works well near galley targets, because the scale factor to apply is determined before any galley flushing takes place.

3.18. @StartHSpan,@StartVSpan, @StartHVSpan, @HSpan, and @VSpan

These symbols work together to produce spanning columns and rows in a more flexible way than is possible in practice with // and ||. An object

```
@StartHSpan object
```

causes object to be printed, but occupying all the horizontal space to the right on the row mark on which it lies up to and including the rightmost @HSpan symbol on that mark not preceded by @StartHVSpan, @StartHSpan, @StartVSpan, or @VSpan. The column mark of this spanning object is not constrained to align with any of the column marks of the columns it spans.

If there is no @HSpan symbol anywhere to the right of @StartHSpan, then the object spans only its own column. This means that it occupies that column as usual but its mark is not constrained to align with those of the other objects in the column.

Similarly, the @StartVSpan symbol causes its object to occupy all the vertical space below it on the column mark on which it lies, down to and including the bottommost @VSpan symbol on that mark not preceded by a @StartHVSpan, @StartHSpan, @StartVSpan, or @HSpan; and if there is no @VSpan symbol anywhere below it on that mark, then the object spans only its own row, occupying its row but with its mark not constrained to align with the row mark.

The @StartHVSpan symbol combines the effects of @StartHSpan and @StartVSpan, allowing an object to span both columns and rows simultaneously. For example, in

```
@StartHVSpan x  |  | @HSpan
/
@VSpan  |  |
```

the object x will occupy a rectangular area spanning three columns, two rows, and the gaps between them.

The objects lying in the region spanned should all be empty, or the @HSpan and @VSpan symbols can be used to document the spanning that is occurring. At present there may be no galley targets or recursive symbols within the right parameter of @StartHSpan, @StartVSpan, or @StartHVSpan. However, the right parameter may otherwise be an arbitrary object, including paragraphs of text that require breaking.

If the right parameter of @StartHSpan, @StartVSpan, or @StartHVSpan occupies more horizontal or vertical space than all of the spanned columns or rows combined require, the extra space goes into the last spanned column or row. Overlapping spanning rows and columns are permitted. Gaps spanned by span objects are unbreakable (their u indicator is set automatically and cannot be revoked).

3.19. @Scale

This symbol geometrically scales its right parameter by the scale factor given in its left parameter:

```
1.0 @Scale Hello  2.0 @Scale Hello  0.5 @Scale Hello
```

has result

Hello Hello Hello

The left parameter can be two scale factors, in which case the first applies horizontally, and the second vertically:

```
{0.5 2.0} @Scale Hello
```

has result

Hello

The left parameter may be empty, in which case Lout will scale the object by a common factor horizontally and vertically so as to occupy all available horizontal space:

```
{} @Scale { Hello world }
```

has result

Hello world

The right parameter may be any object. @Scale has both a @OneCol and a @OneRow effect, and the marks of the result coincide with the principal marks of the right parameter.

3.20. @Rotate

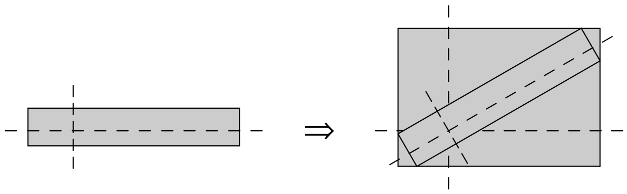
The @Rotate symbol will rotate its right parameter counterclockwise an amount given in degrees (positive or negative) by its left parameter. For example,

```
30d @Rotate { hello, world }
```

has result

hello, world

Before rotating the object, @OneCol and @OneRow are applied to it. The result is a rectangle whose marks pass through the point where the original marks crossed:



As this example shows, rotation by an angle other than a multiple of ninety degrees introduces quite a lot of white space. So, for example, the result of

-30d @Rotate 30d @Rotate *object*

is a much larger object than *object*, despite the fact that one rotation cancels the other.

Rotation of objects containing receptive and recursive symbols is permitted, but for angles other than multiples of ninety degrees it is best to make the size of the rotated object clear with @Wide and @High symbols:

```
30d @Rotate 5i @Wide 4i @High
```

```
{   //1i @TextPlace
    //1i
}
```

This is because for angles other than multiples of ninety degrees the space available for @TextPlace to occupy is indeterminate, and the result is poor.

3.21. @Background

The @Background symbol will print its left parameter in the background of its right parameter. That is, the result has the size of the right parameter, but the left parameter will be printed first in the same space, with its marks aligned with the marks of the right parameter.

3.22. @KernShrink

This symbol returns its right parameter unchanged in appearance but occupying a slightly smaller bounding box. The reduction is by the amount of kerning that would be applied if the right parameter was immediately *followed* by the left parameter. For example,

```
. @KernShrink P
```

has result

P

where a box of size 0 has been drawn around the result to make its extent clear. Compare this with 'P' alone:

P

in which the bounding box exactly encloses the object, or at least is supposed to. The bounding box is smaller on the right by the amount of kerning that would be applied between 'P' and '.'.

The only known use for this symbol is to produce tucked-in subscripts in the Eq equation formatting package.

3.23. @Common, @Rump, and @Meld

The @Common and @Rump symbols compare two paragraph objects:

```
{ Aardvark, 29 } @Common { Aardvark, 359 }
```

If either parameter is not a paragraph object, it is converted into a single-object paragraph first. The result of @Common is the common prefix of the two paragraphs; that is, those initial objects which are equal in the two paragraphs. In the example above, the result is Aardvark,. The result of @Rump is that part of the second object which is not included in @Common; the result of

```
{ Aardvark, 29 } @Rump { Aardvark, 359 }
```

is 359.

If the two objects have nothing in common, the result of @Common will be an empty object and the result of @Rump will be the second object. If the two objects are identical, the result of @Common will be the first object, and the result of @Rump will be an empty object.

The only known use for @Rump and @Common is to implement merged index entries (Section 2.8).

The @Meld symbol returns the minimum meld of two paragraphs, that is the shortest paragraph that contains the two original paragraphs as subsequences. For example,

```
{ Aardvark , 1 , 2 } @Meld { Aardvark , 2 , 3 }
```

produces

Aardvark , 1 , 2 , 3

The result is related to the well-known longest common substring, in that the meld contains everything not in the lcs plus one copy of everything in the lcs. This function is well-suited to melding complex index entries. Note that there must be a non-zero amount of space before each comma, otherwise each will become part of the preceding word, and, since @Meld compares the two paragraphs word by word, the result will be different and less satisfactory.

3.24. @Insert

The @Insert symbol inserts its left parameter at the beginning of the first paragraph of its right parameter:

```
X @Insert { A B // C // D }
```

is equivalent to

```
{ XA B // C // D }
```

Notice that a zero-width space separates X from the first paragraph, so if some wider space is required it must be placed at the end of X. The @Insert operation is applied to the value of the right parameter after evaluation.

The only known use for this symbol is to attach something like **Figure 6** to the front of a multi-paragraph caption.

3.25. @OneOf

The @OneOf symbol returns one of the sequence of objects which is its right parameter as its result:

```
@OneOf {
   @ResultA
```

```
        @ResultB
        @ResultC
    }
```

The choice is made to ensure that whatever galley target is required at the moment is found. For example, if we are evaluating @OneOf as part of an attempt to attach a galley whose target is @SomeTarget, then the result above will be @ResultA if it contains @SomeTarget, or else @ResultB if it contains @SomeTarget, or else @ResultC (whether or not it contains the target, or if there is no target).

Use of @OneOf in conjunction with recursive symbols can lead to problems. Consider this example:

```
def @Recursive {

    def @ChoiceA { @APlace // @Recursive }

    def @ChoiceB { @BPlace // @Recursive }

    @OneOf {
        @ChoiceA
        @ChoiceB
    }
}
```

Lout believes that expanding @Recursive is the right thing to do when searching for either of the galley targets @APlace and @BPlace. When searching for @BPlace this leads Lout to expand @Recursive, then @ChoiceA, then the @Recursive symbol within @ChoiceA, and so on infinitely. This problem can be avoided by attaching a @NotRevealed symbol to each of the inner @Recursive symbols: these are then not available for expansion until a decision has been made to expand the symbol they lie within. In this particular example it would be simpler to write

```
def @Recursive {

    @OneOf {
        @APlace
        @BPlace
    }
    // @Recursive
}
```

but this factoring is not possible when the recursive calls have parameters that are required to differ in the two cases.

3.26. @Next

The @Next symbol returns its parameter plus one. It is rather clever at working this out: it hunts through the parameter from right to left, looking for a number to increment:

```
@Next (3.99)
```

has result (3.100). If @Next cannot find a digit inside its parameter, it is an error. Roman numerals are handled by storing them in a database, as explained in Section 4.2; @Next will not increment a Roman numeral.

3.27. @Case

The @Case symbol selects its result from a list of alternatives, depending on a tag:

```
@Day @Case {
   { 1 21 31 }  @Yield  st
   { 2 22 }  @Yield  nd
   { 3 23 }  @Yield  rd
   else  @Yield th
}
```

In this example the result will be st if @Day is 1, 21, or 31, and nd if @Day is 2 or 22, etc. The effect is similar to accessing a database, though in a more compact form. The right parameter is a sequence of @Yield symbols, each with a left parameter whose value is a sequence of one or more juxtapositions of simple words, and a right parameter which may be any object.

We first describe the behaviour when the value of the left parameter of @Case is a juxtaposition of one or more simple words. Then the result of the @Case is the right parameter of the first @Yield whose left parameter contains either the value of the left parameter of the @Case, or the special value else. If there is no such @Yield it is an error.

When the left parameter of @Case is not a juxtaposition of simple words, the result is the right parameter of the first @Yield whose left parameter is else, or an error otherwise. This permits examples like

```
@RunningTitle @Case {
   dft @Yield @Title
   else @Yield @RunningTitle
}
```

where a running title is returned unless it has the value dft (which presumably means that no running title was supplied), in which case an ordinary title is returned instead.

When a receptive symbol is placed within a @Case, it should be included in each alternative, since otherwise Basser Lout may become confused when trying to predict whether the symbol will be a part of the result or not. Alternatively, if it can be guaranteed that the receptive symbol will never be searched for when the cases that it does not lie within are selected, that is all right too.

3.28. @Moment

The predefined symbol @Moment has the following definition:

```
def @Moment
  named @Tag {}
  named @Second {}
  named @Minute {}
  named @Hour {}
  named @Day {}
  named @Month {}
  named @Year {}
  named @Century {}
  named @WeekDay {}
  named @YearDay {}
  named @DaylightSaving {}
{}
```

It may be used like any other symbol. Lout provides an invocation of @Moment with tag now, whose other parameters are numbers encoding the current date and time:

@Second	the current second, usually between 00 and 59
@Minute	the current minute, between 00 and 59
@Hour	the current hour, between 00 and 23
@Day	the current day of the month, between 1 and 31
@Month	the current month, between 1 (January) and 12 (December)
@Year	the current year of the century, between 00 and 99
@Century	the current century, e.g. 19 or 20
@WeekDay	the current day of the week, between 1 (Sunday) and 7 (Saturday)
@YearDay	the current day of the year, between 0 and 365
@DaylightSaving	an implementation-dependent number that may encode the daylight saving currently in effect

Unix manual entries state that @Second can be as high as 61, to allow for leap seconds. Judicious use of databases can convert these numbers into useful dates. For example,

```
@Moment&&now @Open { @Day {@Months&&@Month}, @Century{@Year} }
```

produces something like 17 September, 1999 given a suitable database of months.

3.29. @Null

This symbol provides a convenient way to remove unwanted concatenation symbols. If there is a concatenation symbol preceding @Null, the @Null and the concatenation symbol are both deleted. Otherwise, if there is a following concatenation symbol, it and the @Null are both deleted. Otherwise, @Null becomes an empty object.

These rules apply to a fully parenthesized version of the expression. For example, in

```
... //1vx @Null |0.5i ...
```

it is the horizontal concatenation symbol following @Null that disappears, because in the fully parenthesized version

```
... //1vx { @Null |0.5i ... }
```

there is no concatenation symbol preceding the @Null.

3.30. @Galley and @ForceGalley

These symbols both act as a placeholder for a galley. That is, they may be replaced by components of a galley. In the case of @ForceGalley the galley will then have a forcing galley effect at this point although it need not be declared using force into. See Section 2.7 for a detailed discussion of galleys, forcing galleys, and targets.

3.31. @NotRevealed

The @NotRevealed symbol exerts fine control over the process of expanding receptive symbols. It may appear only within the body of a definition, immediately following the name of a receptive symbol. For example:

```
def A { @Galley }

def B { @Galley }

def ABList
{
   A
   //  B @NotRevealed
   //  ABList
}
```

The meaning is that the symbol immediately preceding @NotRevealed, B in this example, is not revealed to galleys which encounter ABList while searching for targets; to such galleys it appears that ABList contains A only, not B, hence only galleys targeted to A will expand ABList. However, after ABList is expanded by such a galley, B will be available as a target in the usual way.

Apart from this meaning, @NotRevealed has no effect at all, and the body of the definition may be understood by deleting @NotRevealed and any preceding space. Thus, the symbol preceding @NotRevealed may have named and right parameters in the usual way; these would follow after the @NotRevealed symbol.

This symbol was introduced to overcome a problem with floating figures treated as displays. It turned out to be essential to specify the layout of a column (in part) as

```
@BodyTextPlace
// @FigurePlace
// @BodyTextPlace
// @FigurePlace
// @BodyTextPlace
...
```

so that figures could alternate with body text down the column. However, some means was needed to ensure that in the absence of any figures there could only be one @BodyTextPlace in the column, since otherwise various problems arose, for example the @NP symbol merely causing a skip from one @BodyTextPlace to the next in the same column, rather than to the first in the next column. Also, without this feature the optimal page breaker's attempts to end a column early would be frustrated by Lout then discovering that plenty of space existed at a following @BodyTextPlace in the same column. The solution is based on ABList above; each occurrence of @BodyTextPlace after a @FigurePlace is not revealed in the enclosing definition, and so cannot be found by body text galleys unless a figure has previously attached to the preceding @FigurePlace.

3.32. The cross reference symbols && and &&&

The cross reference symbol && takes the name of a symbol (not an object) for its left parameter, and an object whose value must be a simple word, or several simple words, for its right parameter. The result is a cross reference, which may be thought of as an arrow pointing from the cross reference symbol to the beginning of an invocation of the named symbol.

The invocation pointed to, known as the *target* of the cross reference, is generally one whose @Tag parameter has value equal to the right parameter of the cross reference symbol. Three special tags, preceding, following, and foll_or_prec, point respectively to the first invocation preceding the cross reference in the final printed document, to the first invocation following it, and to the first following it if such exists else to the first preceding it.

A cross reference may be used in four ways: where an object is expected, in which case its value is a copy of the target; with the @Open and @Use symbols; with the @Tagged symbol; and in the into clause or @Target symbol of a galley definition, in which case the value of the tag must be preceding, following, or foll_or_prec.

Within an into clause or @Target symbol, the alternative form &&& is acceptable and indicates a forcing galley (Section 2.7).

Except within an into clause or @Target symbol, the symbol referred to must have a @Tag parameter. This is so even if the right parameter of the cross reference is preceding, following, or foll_or_prec.

3.33. @Tagged

The @Tagged symbol takes a cross reference for its left parameter and an object, whose value must be a juxtaposition of simple words, or several words, or an empty object, for its right parameter. It has the effect of attaching its right parameter as an additional tag to the invocation denoted by its left parameter, unless the right parameter is empty, in which case @Tagged does nothing. The result of @Tagged is always @Null, which makes it effectively invisible.

3.34. @Open and @Use

The @Open symbol takes a cross reference or symbol invocation for its left parameter, and an arbitrary object, which must be enclosed in braces, for its right parameter. The right parameter may refer to the exported parameters and nested definitions of the invocation denoted by the left parameter, and its value is the @Open symbol's result. The target of the cross reference may lie in an external database (Section 3.35). Any symbol available outside the @Open which happens to have the same name as one of the symbols made available by the @Open will be unavailable within the @Open.

The @Use symbol is an @Open symbol in a different form. It may only appear among or after the definitions in Lout's input, and it is equivalent to enclosing the remainder of the input in an @Open symbol. For example,

definitions
@Use { *x* }
@Use { *y* }
rest of input

is equivalent to

```
definitions
x @Open
{   y @Open
    { rest of input
    }
}
```

The @Use symbol allows a set of standard packages to be opened without the inconvenience of enclosing the entire document in @Open symbols. Such enclosure could cause Basser Lout to run out of memory.

3.35. @Database and @SysDatabase

The @Database symbol is used to declare the existence of a file of symbol invocations that Lout may refer to when evaluating cross references. In Basser Lout, for example,

```
@Database @Months @WeekDays { standard }
```

means that there is a file called standard.ld containing invocations of the previously defined symbols @Months and @WeekDays. A @Database symbol may appear anywhere a definition or a @Use symbol may appear. Different definitions packages may refer to a common database, provided the definitions they give for its symbols are compatible. An entry is interpreted as though it appears at the point where the cross reference that retrieves it does, which allows symbols like @I for Slope @Font to be used in databases. The database file may not contain

@Database or @Include symbols, and each invocation within it must be enclosed in braces.

Basser Lout constructs an *index file*, which in this example is called standard.li, the first time it ever encounters the database, as an aid to searching it. If the database file is changed, its index file must be deleted by the user so that Basser Lout knows to reconstruct it. There is also an installation option which allows this deletion to be done automatically on suitable systems (including Unix).

Basser Lout searches for databases in the current directory first, then in a sequence of standard places. To search the standard places only, use @SysDatabase.

3.36. @Graphic

Lout does not provide the vast repertoire of graphical objects (lines, circles, boxes, etc.) required by diagrams. Instead, it provides an escape route to some other language that does have these features, via its @Graphic symbol:

```
{ 0 0 moveto
  0 ysize lineto
  xsize ysize lineto
  xsize 0 lineto
  closepath
  stroke
}
@Graphic
{ //0.2c
  ||0.2c hello, world ||0.2c
  //0.2c
}
```

The result of the above invocation of the symbol @Graphic is

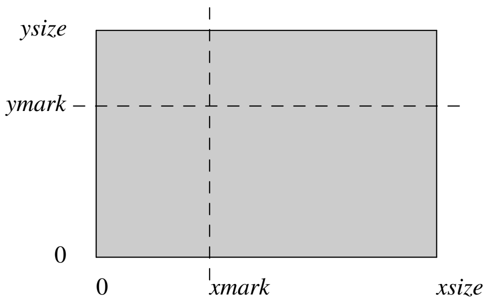
hello, world

The right parameter always appears as part of the result, and indeed the result is always an object whose size is identical to the size of the right parameter with @OneCol and @OneRow applied to it. From now on we refer to this part of the result as the *base*.

The left parameter is implementation-dependent: that is, its meaning is not defined by Lout, and different implementations could require different values for it. The following description applies to Basser Lout, which uses the PostScript page description language [1]. Similar but more restricted possibilities exist with the PDF back end (see a separate document distributed with Lout); to include both, use the @BackEnd symbol like this:

```
{ @BackEnd @Case {
    PostScript @Yield
    {
      ...
    }
    PDF @Yield
    {
      ...
    }
  }
  @Graphic
  {
    ...
  }
}
```

Returning to PostScript, the left parameter refers to a coordinate system whose origin is the bottom left-hand corner of the base. It may use the symbols xsize and ysize to denote the horizontal and vertical size of the base; similarly, xmark and ymark denote the positions of the base's column and row marks:



In addition to these four symbols and 0, lengths may be denoted in centimetres, inches, points, ems, f's, v's and s's using the notation

l cm	instead of Lout's	*l*c
l in	instead of Lout's	*l*i
l pt	instead of Lout's	*l*p
l em	instead of Lout's	*l*m
l ft	instead of Lout's	*l*f
l vs	instead of Lout's	*l*v
l sp	instead of Lout's	*l*s

Note that there must be a space between the number and its unit, unlike Lout proper.

A point within the base (and, with care, a point outside it) may be denoted by a pair of lengths. For example,

```
xmark  ymark
```

is the point where the marks cross, and

```
0   2 cm
```

is a point on the left edge, two centimetres above the bottom left-hand corner. These two numbers are called the *x coordinate* and the *y coordinate* of the point.

The first step in specifying a graphic object is to define a *path.* A path can be thought of as the track of a pen moving over the page. The pen may be up (not drawing) or down (drawing a line or curve) as it moves. The entire path is a sequence of the following items:

x y moveto	Lift the pen and move it to the indicated point.
x y lineto	Put the pen down and draw a straight line to the indicated point.
x y r angle1 angle2 arc	Put the pen down and draw a circular arc whose centre has coordinates *x* and *y* and whose radius is *r*. The arc begins at the angle *angle1* measuring counterclockwise from the point directly to the right of the centre, and proceeds counterclockwise to *angle2.* If the arc is not the first thing on the path, a straight line will be drawn connecting the current point to the start of the arc.
x y r angle1 angle2 arcn	As for arc, but the arc goes clockwise from *angle1* to *angle2.*
closepath	Draw a straight line back to the point most recently moved to.

The first item should always be a moveto, arc, or arcn. It should be clear from this that the path given earlier:

```
0 0 moveto
0 ysize lineto
xsize ysize lineto
xsize 0 lineto
closepath
```

traces around the boundary of the base with the pen down.

Once a path is set up, we are ready to *paint* it onto the page. There are two choices: we can either *stroke* it, which means to display it as described; or we can *fill* it, which means to paint everything inside it grey or black. For stroking the two main options are

length setlinewidth	The pen will draw lines of the given width.
[*length*] 0 setdash	The pen will draw dashed lines when it is down, with the dashes each of the given length.

These options are followed by the word stroke. So, for example,

```
{ 0 0 moveto xsize 0 lineto
  2 pt setlinewidth [ 5 pt ] 0 setdash stroke
}
@Graphic { 3i @Wide }
```

has result

When filling in the region enclosed by a path, the main option is setgray, which determines the shade of grey to use, on a scale from 0 (black) to 1 (white). So, for example,

```
{ 0 0 moveto xsize 0 lineto 0 ysize lineto closepath
  0.8 setgray fill
}
@Graphic
{ 2c @Wide  2c @High }
```

has result

There are many other options. The value of the left parameter of @Graphic may be any fragment of the PostScript page description language [1]. Here are two other examples:

```
xsize 2 div
```

denoting a length equal to half the horizontal size of the base, and

```
gsave fill grestore stroke
```

which both fills and strokes the path. Since Basser Lout does not check that the left parameter is valid PostScript, it is possible to cause mysterious errors in the printing device, resulting in no output, if an incorrect value is given. It is a good idea to encapsulate graphics objects in carefully tested definitions, like those of the Diag figure drawing package [4, Chapter 9], to be sure of avoiding these errors.

PostScript experts may find the following information helpful when designing advanced graphics features. The left parameter of @Graphic may have two parts, separated by //:

{ *first part* // *second part* } @Graphic *object*

If there is no //, the second part is taken to be empty. The PostScript output has the form

gsave
x y translate
Code which defines xsize, ysize, xmark, ymark, ft, vs, *and* sp
gsave
first part
grestore
Code which renders the right parameter in translated coordinates
second part
grestore

where x,y is the position of the lower left corner of the base. Having two parts permits bracketing operations, like save and restore or begin and end, to enclose an object. See the source file of the Diag package for examples.

3.37. @PlainGraphic

The @PlainGraphic symbol is avery rudimentary analogue for plain text output of the @Graphic symbol for PostScript output. Its result is its right parameter printed on a background created by repeated printings of its left parameter, which must be a simple word. For example,

```
"." @PlainGraphic 5s @Wide
```

would produce five dots. @PlainGraphic is used in the tbl table-drawing package to produce plain-text rules.

3.38. @IncludeGraphic and @SysIncludeGraphic

These symbols instruct Lout to incorporate a separately created illustration:

```
@IncludeGraphic "myportrait.eps"
```

The parameter is implementation-dependent; in Basser Lout it is an object whose value is a simple word denoting the name of a file. This file should ideally be a PostScript EPS Version 3.0 file [1], since then Lout will keep careful track of what resources are required for printing that file. However, any PostScript file containing the %%BoundingBox: comment and not requiring unusual resources is likely to work.

The result of @IncludeGraphic is an ordinary Lout object with marks through its centre. It may be rotated, scaled, and generally treated like any other object. Basser Lout determines its size by consulting the bounding box information in the file. If this cannot be found, a warning message is printed and the result object has zero size.

@IncludeGraphic searches the same directories that @Include does (Section 3.40). @SysIncludeGraphic is the same as @IncludeGraphic, except that it searches only the directories searched by @SysInclude.

If the file name ends in any of .gz, -gz, .z, -z, _z, or .Z, the file will first be uncompressed using the gunzip command into a temporary file called lout.eps in the current directory. This file is removed immediately after it is copied into the output file.

3.39. @PrependGraphic and @SysPrependGraphic

These symbols, which may appear anywhere that a definition or @Use symbol may appear, tell Lout to include the contents of a file in the preamble of its output. For Basser Lout this means that the file must contain PostScript (and ideally it would begin and end with the %%BeginResource and %%EndResource comments of DSC 3.0). For example,

```
@SysPrependGraphic { diagf.lpg }
```

appears at the start of the Diag package; the file diagf.lpg contains a number of PostScript definitions used by Diag for drawing diagrams. It saves a lot of space to include them just once at the start like this, rather than with every diagram. @PrependGraphic and @SysPrependGraphic search for the file in the same places as @Include and @SysInclude respectively.

3.40. @Include and @SysInclude

These symbols instruct Lout to temporarily switch to reading another file, whose name appears in braces following the symbol. For example,

```
@Include { "/usr/lout/fontdefs" }
```

will cause the contents of file /usr/lout/fontdefs to be read at the point it occurs. After that file is read, the current file is resumed. The included file may contain arbitrary Lout text, including other @Include commands. The file is searched for first in the current directory, then in a sequence of standard places which are not necessarily the same places that databases are searched for. @SysInclude searches the standard places only.

3.41. @BackEnd and the PlainText and PDF back ends

The @BackEnd symbol, which takes no parameters, has for its result a string naming the back end currently in use. Three back ends are available, PostScript, PDF and PlainText. The symbol is generally used like this:

```
@BackEnd @Case {
   PlainText @Yield { ... }
   PostScript @Yield { ... }
   PDF @Yield { ... }
}
```

to obtain different objects depending on the back end. No else is required since these are the only possible values.

When a @Case symbol has @BackEnd for its left parameter and the left parameter of each @Yield symbol within it consists of a sequence of one or more literal words (including else), Lout will optimize by evaluating the @Case symbol at the time it is read. This optimization ensures that there is only a small once-only performance penalty for multiple back ends, and it permits these @Case symbols (but no other symbols) to appear within the object following @Include and @PrependGraphic symbols.

The PlainText back end differs from the PostScript one in two main respects. First, there is effectively just one font: although all the font commands work exactly as usual, they don't actually change anything. Each character in this font is taken to be one tenth of one inch wide and 20 points high. Second, the output is an ordinary text file, not a PostScript file.

Clearly, with ordinary text output the possibility of advanced graphics features such as rotation and scaling is curtailed. Nevertheless, all symbols have well-defined (possibly null) effects in the PlainText back end, so there is no additional danger of crashing the system or obtaining grossly unreasonable output by a change to PlainText.

The PlainText back end is obtained by the -p option to Basser Lout. The character size can be changed by adding two lengths to the -p option, like this:

```
lout -p0.1i12p ...
```

which invokes the PlainText back end with each character being 0.1 inches wide and 12 points high. However, experience suggests that the best approach is to define all horizontal lengths as multiples of the s unit (the width of a space, hence the width of all characters) and to define all vertical lengths as multiples of the f unit (the font size, equal to the height of every character), and not to change the character size in the command line.

There is a -P option which is identical with the -p option except that it inserts a form-feed character between each two components of the output, but not before the first or after the last.

The PDF back end is obtained by typing lout -Z. It is similar to PostScript but much more limited in functionality. Consult a separate document distributed with Lout for further information.

3.42. @Verbatim and @RawVerbatim

These symbols instruct Lout to read the following text (enclosed in braces) verbatim, that is, turning off all special character meanings. For example,

```
@Verbatim { "hello" }
```

produces

"hello"

@Verbatim ignores all characters after the opening brace up to but not including the first non-white-space character. @RawVerbatim differs from @Verbatim only in that it ignores all characters after the opening brace up to but not including the first non-white-space character, or up to and including the first newline character, whichever comes first. This variant is useful in cases such as

```
@RawVerbatim {
    var x: Real
begin
}
```

where the first line of the verbatim text begins with white space which would be ignored by @Verbatim. Both symbols ignore all white spaces at the end of the verbatim text, preceding the closing brace.

3.43. @Underline

The @Underline symbol underlines its right parameter, but only if that parameter is a word or a paragraph:

```
We @Underline { really do } mean this.
```

produces

We really do mean this.

It is not possible to underline an arbitrary object using this symbol; the @Underline symbol will be ignored if this is attempted.

It is very easy to *define* a symbol which will underline an arbitrary object, using the @Graphic symbol. This raises the question of why @Underline is needed at all. The answer is that @Underline has two properties that distinguish it from symbols based on @Graphic.

First, when @Underline both contains a paragraph and is used within a paragraph, as in the example above, the inner and outer paragraphs are merged into one, permitting the underlined text to break over several lines. This is how the @Font symbol works too, but symbols based on @Graphic do not permit this merging.

Second, Adobe font files specify the correct position and thickness of underlining for each font, and the @Underline symbol follows these specifications. The font used is the font of the first object underlined, if it is a simple word, or else the font of the enclosing paragraph.

3.44. @PageLabel

The @PageLabel symbol associates a page label in the PostScript output file with the page within which (or just before which) the symbol occurs, so that PostScript viewers are able to index the page by this label. (The label is printed in the %%Page comment preceding the page in the PostScript output file.) For example,

```
@PageLabel iv
```

associates the label iv with the page. The label may be an arbitrary object; if its value is not a simple word, it will be replaced by ?.

@PageLabel is unrelated to Lout's cross referencing mechanism; it is for communicating a label to the PostScript output file, not to other parts of Lout. The result of @PageLabel is a null object.

Chapter 4. Examples

This chapter presents some examples taken from the various packages available with Basser Lout. The reader who masters these examples will be well prepared to read the packages themselves. The examples have not been simplified in any way, since an important part of their purpose is to show Lout in actual practice.

4.1. An equation formatting package

In this section we describe the design and implementation of the Eq equation formatting package. Equation formatting makes a natural first example, partly because its requirements have strongly influenced the design of Lout, and partly because no cross references or galleys are required.

To the author's knowledge, Eq is the first equation formatter to be implemented as a collection of high-level definitions. This approach has significant advantages: the basics of language and layout are trivial, so the implementor can concentrate on fine-tuning; and the definitions, being readily available, can be improved, extended, or even replaced.

As described in the User's Guide [4], an equation is entered in a format based on the one introduced by the eqn language of Kernighan and Cherry [2]:

```
@Eq { { x sup 2  +  y sup 2 }  over  2 }
```

The result is

$$\frac{x^2 + y^2}{2}$$

In outline, the definition of the @Eq symbol is

```
export  sup  over "+"  "2"  "<="
def @Eq
  body @Body
{
  def sup precedence 60 left x right y { ... }
  def over precedence 54 left x right y { ... }
  def "2" { Base @Font "2" }
  def "+" { {Symbol Base} @Font "+" }
  def "<=" { {Symbol Base} @Font "\243" }
  ...

  Slope @Font 1.2f @Break 0c @Space @Body
}
```

A body parameter is used to restrict the visibility of the equation formatting symbols (there are hundreds of them). The equation as a whole is set in Slope (i.e. Italic) font, and symbols such as "2" and "+" are defined when other fonts are needed. Precedences are used to resolve ambiguities such as a sup b over c. Eq takes all spacing decisions on itself, so to prevent white space typed by the user from interfering, the equation is enclosed in 0c @Space. We will discuss the 1.2f @Break later.

Thus have we disposed of the language design part of the equation formatting problem; it remains now to define the twenty or so symbols with parameters, and get the layout right.

Every equation has an *axis*: an imaginary horizontal line through the centre of variables, through the bar of built-up fractions, and so on. We can satisfy this requirement by ensuring that the result of each symbol has a single row mark, on the axis. For example, the superscripting symbol is defined as follows:

```
def sup
   precedence 60
   associativity left
   left x
   named gap { @SupGap }
   right y
{
   @HContract @VContract {
      |  @Smaller y
      ^/gap  x
   }
}
```

The @VContract and ^/ symbols together ensure that the axis of the result is the axis of the left parameter. A gap parameter has been provided for varying the height of the superscript, with default value @SupGap defined elsewhere as 0.40fk. It is important that such gaps be expressed in units that vary with the font size, so that they remain correct when the size changes. Collecting the default values into symbols like @SupGap ensures consistency and assists when tuning the values. Here is another characteristic definition:

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

Both parameters are centred, since we do not know which will be the wider; we use @OneCol to make sure that the entire parameter is centred, not just its first column, and @HContract ensures that the fraction will never expand to fill all the available space, as Lout objects have a natural tendency to do (Section 2.6). @HLine is a horizontal line of the width of the column:

```
def @HLine
   named line { "0.05 ft setlinewidth" }
{
   { "0 0 moveto xsize 0 lineto" line "stroke" } @Graphic {}
}
```

Here we are relying on the expanding tendency just mentioned.

The remaining symbols are quite similar to these ones. We conclude with a few fine points of mathematical typesetting mentioned by a leading authority, D. E. Knuth [5].

Some symbols, such as ≤ and ≠, should have a thick space on each side; others, such as + and –, have a medium space; others have a thin space on the right only. This would be easy to do except that these spaces are not wanted in superscripts and subscripts:

$$r^{n+1} - 1$$

In effect, the definition of such symbols changes depending on the context; but Lout does not permit such a change. Luckily, the so-called 'style' information set by the @Font, @Break, and @Space symbols can change in this way. Accordingly, Eq uses the y unit, which is part of style, for these spaces:

```
def @MedGap { 0.20y }

def "+" { &@MedGap plus &@MedGap }

def @HSqueeze right x { 0.2f @YUnit x }
```

In the equation as a whole, the y unit is initially set to 1f, and so @MedGap ordinarily supplies 20% of this amount. But superscripts and subscripts are enclosed in the @HSqueeze symbol, which, by changing the y unit, ensures that any @MedGap within them is much smaller than usual.

4.2. Paragraphs, displays, and lists

The remaining sections of this chapter are all based on Version 2 of the DocumentLayout package. Version 3, which is similar but more elaborate, is described from the user's perspective in the User's Guide [4]. In 26 pages of Lout, the DocumentLaytout package defines many features required in the formatting of simple documents, technical reports, and books, including displays, lists, page layout, cross references, tables of contents, footnotes, figures, tables, references, chapters, sections, and sorted indexes.

The symbols used for separating paragraphs and producing displays and lists may lack the excitement of more exotic features, but they can teach some important lessons about robust design. The following macro for separating paragraphs produces a 0.3 cm vertical space and a 1 cm indent on the following line, and is clearly on the right track:

```
macro @PP { //0.3c &1c }
```

Nevertheless it has several major problems.

The & symbol is subject to widening during line adjustment, so it should be replaced by 1c @Wide {}. But then white space following the symbol will affect the result, so an extra &0i must be added. If the document is printed double spaced, this paragraph gap will fail to widen: it should be expressed in terms of the v unit, with mark-to-mark spacing mode. Similarly, the paragraph indent should probably be made proportional to the font size.

'Magic numbers' like 0.3c should not be buried in definitions where they cannot be changed easily, or kept consistent with similar definitions during tuning. They are much better placed as symbols, possibly parameters of the enclosing package:

```
def @DocumentLayout
   named @ParaGap { 1.3vx }
   named @ParaIndent { 2f }
   ...
@Begin

   macro @PP { //@ParaGap @ParaIndent @Wide &0i }
   macro @LP { //@ParaGap }
   ...
@End @DocumentLayout
```

and we have arrived at the definition of @PP as it appears in the DocumentLayout package.

A display is a table in which the first column is blank:

preceding text
//@DispGap |@DispIndent *display*
//@DispGap
following text

Edge-to-edge is the appropriate spacing mode before and after displays, since the display could be a table or figure whose mark does not correspond to a baseline. Thus, 1v is a reasonable value for @DispGap.

The ordinary user cannot be expected to type the Lout source shown above; a more appropriate syntax is

preceding text
@IndentedDisplay { *display* }
following text

This presents a problem: if @IndentedDisplay is made a definition with a right parameter, its result will be an object separated from the surrounding text only by white space, hence part of the paragraph; while if it is a macro, the final //@DispGap cannot be included in it. The solution adopted in the DocumentLayout package uses a galley and a macro:

```
def @DispPlace { @Galley }
def @Disp into { @DispPlace&&preceding }
   right x
{
   @OneRow x
}

macro @IndentedDisplay
{
   //@DispGap  |@DispIndent  @DispPlace  |
   //@DispGap  //  @Disp
}
```

@DispPlace and @Disp are not exported, so there is no danger of a name clash with some other symbol. The ordinary user's syntax expands to

```
preceding text
//@DispGap  |@DispIndent  @DispPlace  |
//@DispGap  //  @Disp { display }
following text
```

and the @Disp galley appears at the preceding @DispPlace, being itself replaced by @Null. The // symbol protects the preceding //@DispGap from being deleted by this @Null when there is no following text.

An automatically numbered list could have an arbitrarily large number of items, so, by analogy with sequences of pages, we see immmediately that recursion must be involved:

```
def @List right num
{
   @DispIndent @Wide num.  |  @ItemPlace
   //@DispGap @List @Next num
}
```

Notice how the @Next symbol works in conjunction with the recursion to produce an ascending sequence of numbers; the result of @List 1 will be

```
1.  @ItemPlace
2.  @ItemPlace
3.  @ItemPlace
...
```

We can follow this with items which are galleys targeted to @ItemPlace&&preceding, and @List will expand just enough to accommodate them.

The usual problem with recursive-receptive symbols now arises: there is always one unexpanded @List, and until it can be removed the galley containing it will appear to be incomplete and will be prevented at that point from flushing into its parent (see page 29). We adopt the usual solution: a forcing galley into a later target will replace the last @List by @Null. This brings us to the definitions as they appear in DocumentLayout:

```
def @ItemPlace { @Galley }
def @ListItem into { @ItemPlace&&preceding }
    right x
{ x }

def @EndListPlace { @Galley }
def @EndList force into { @EndListPlace&&preceding }
{}

def @RawIndentedList
   named style right tag {}
   named indent { @DispIndent }
   named gap { @DispGap }
   named start { 1 }
{
    def @IList right num
    {
        indent @Wide {style num} | @ItemPlace
        //gap @IList @Next num
    }

    @IList start  //  @EndListPlace
}
```

Now given the input

```
@RawIndentedList
@ListItem { first item }
@ListItem { second item }
...
@ListItem { last item }
@EndList
```

@RawIndentedList will expand to receive the items, and will be closed off by @EndList.

The indent, gap, and start parameters are straightforward (note that the burden of typing 1 has been lifted from the ordinary user), but the style parameter has a parameter of its own (see page 16). It is used like this:

```
def @RawNumberedList { @RawIndentedList style { tag. } }
def @RawParenNumberedList { @RawIndentedList style { (tag) } }
```

In @RawNumberedList, style is given the value tag., where tag is its own right parameter, so the value of {style num} within @IList is num.; while in @RawParenNumberedList, {style num} is (num). In this way we achieve an unlimited variety of numbering formats without having to rewrite @RawIndentedList over and over.

These list symbols are objects without surrounding space, so macros similar to those used for displays are needed:

```
macro @NumberedList { //@DispGap @RawNumberedList //@DispGap }
macro @ParenNumberedList { //@DispGap @RawParenNumberedList //@DispGap }
```

and so on.

Lists numbered by Roman numerals present a problem, because @Next will not increment Roman numerals. Instead, they must be stored in a database:

```
def @Roman
   left @Tag
   right @Val
{ @Val }

@SysDatabase @Roman { standard }
```

@SysDatabase is preferred over @Database here because this database should be kept in a standard place and shared by everyone. The database itself, a file called standard.ld in Basser Lout, contains invocations of @Roman, each enclosed in braces:

```
{ 1 @Roman i }
{ 2 @Roman ii }
...
{ 100 @Roman c }
```

Then @Roman&&12 for example has value xii, and

```
def @RawRomanList { @RawIndentedList style { {@Roman&&tag}. } }
```

produces a list numbered by Roman numerals. The counting still proceeds in Arabic, but each Arabic numeral is converted to Roman by the cross reference. Since arbitrary objects may be stored in databases, arbitrary finite sequences of objects may be 'counted' in this way.

4.3. Page layout

The page layout definitions given in Section 1.2, although correct, are very basic. In this section we present the definitions used by the DocumentLayout package for laying out the pages of books, including running page headers and footers, different formats for odd and even pages, and so on. The present document is produced with these definitions.

We begin with a few definitions which permit the user to create cross references of the 'see page 27' variety which will be kept up to date automatically. The user marks the target page by placing @PageMark intro, for example, at the point of interest, and refers to the marked page as @PageOf intro elsewhere:

```
export @Tag
def @PageMarker right @Tag { @Null }

def @PageMark right x
{
   @PageMarker&&preceding @Tagged x
```

```
}

def @PageOf right x
{
   @PageMarker&&x @Open { @Tag }
}
```

We will see below that an invocation of @PageMarker appears before each page, with @Tag parameter equal to the page number. Suppose that @PageMark intro, which expands to

```
@PageMarker&&preceding @Tagged intro
```

happens to fall on page 27 of the final printed document (of course, its value is @Null which makes it invisible). Then the effect of @Tagged is to attach intro as an extra tag to the first invocation of @PageMarker preceding that final point, and this must be @PageMarker 27. Therefore the expression

```
@PageMarker&&intro @Open { @Tag }
```

will open the invocation @PageMarker 27 and yield the value of its @Tag parameter, 27. Thus, @PageOf intro appearing anywhere in the document yields 27.

Next we have some little definitions for various parts of the page. @FullPlace will be the target of full-width body text:

```
def @FullPlace { @Galley }
```

@ColPlace will be the target of body text within one column:

```
def @ColPlace { @Galley }
```

@TopList will be the target of figures and tables:

```
export @Tag
def @TopList right @Tag
{
   @Galley
   //@TopGap @TopList @Next @Tag
}
```

We have taken a shortcut here, avoiding an unnecessary @TopPlace symbol. @FootList and @FootSect define a sequence of full-width targets at the foot of the page for footnotes, preceded by a short horizontal line:

```
export @Tag
def @FootList right @Tag
{
   @Galley
   //@FootGap  @FootList @Next @Tag
}
```

```
def @FootSect
{
   @FootLen @Wide @HLine
   //@FootGap  @FootList 1  ||@FootLen
}
```

Similarly, @ColFootList and @ColFootSect provide a sequence of targets for footnotes within one column:

```
export @Tag
def @ColFootList right @Tag
{
   @Galley
   //@FootGap  @ColFootList @Next @Tag
}

def @ColFootSect
{
   @ColFootLen @Wide @HLine
   //@FootGap  @ColFootList 1  ||@ColFootLen
}
```

The next definition provides a horizontal sequence of one or more columns:

```
def @ColList right col
{
   def @Column
   {  @VExpand { @ColPlace //1rt @OneRow { //@MidGap @ColFootSect } }  }

   col @Case {
      Single @Yield @Column
      Double @Yield { @DoubleColWidth @Wide @Column  ||@ColGap  @ColList col }
      Multi  @Yield { @MultiColWidth @Wide @Column  ||@ColGap  @ColList col }
   }
}
```

Each column consists of a @ColPlace at the top and a @FootSect at the foot. The @VExpand symbol ensures that whenever a column comes into existence, it will expand vertically so that the bottom-justification //1rt has as much space as possible to work within. The col parameter determines whether the result has a single column, double columns, or multiple columns.

The @Page symbol places its parameter in a page of fixed width, height, and margins:

```
def @Page right x
{
   @PageWidth @Wide @PageHeight @High {
      //@PageMargin  ||@PageMargin
      @HExpand @VExpand x
```

```
        ||@PageMargin  //@PageMargin
    }
}
```

@HExpand and @VExpand ensure that the right parameter occupies all the available space; this is important when the right parameter is unusually small. The @High symbol gives the page a single row mark, ensuring that it will be printed on a single sheet of paper (page 29).

Next we have @OnePage, defining a typical page of a book or other document:

```
def @OnePage
    named @Columns {}
    named @PageTop {}
    named @PageFoot {}
{
    @Page {
        @PageTop
        //@MidGap  @TopList
        //@MidGap  @FullPlace
        //@MidGap  @ColList @Columns
        // //1rt   @OneRow { //@MidGap @FootSect //@MidGap @PageFoot }
    }
}
```

The page top and page foot, and the number of columns, are parameters that will be given later when @OnePage is invoked. The body of the page is a straightforward combination of previous definitions. The // symbol protects the following //1rt from deletion in the unlikely event that all the preceding symbols are replaced by @Null. The following object is enclosed in @OneRow to ensure that all of it is bottom-justified, not just its first component.

Before presenting the definition of a sequence of pages, we must detour to describe how running page headers and footers (like those in the present document) are produced. These are based on the @Runner symbol:

```
export @TopOdd @TopEven @FootOdd @FootEven
def @Runner
    named @TopOdd right @PageNum { @Null }
    named @TopEven right @PageNum { @Null }
    named @FootOdd right @PageNum { @Null }
    named @FootEven right @PageNum { @Null }
    named @Tag {}
{ @Null }
```

The four parameters control the format of running headers and footers on odd and even pages respectively. Invocations of @Runner, for example

```
@Runner
    @TopEven { @B @PageNum |1rt @I { Chapter 4 } }
    @TopOdd  { @I { Examples }   |1rt @B @PageNum }
```

will be embedded in the body text of the document, and, as we will see in a moment, are accessed by @Runner&&following cross references on the pages. Notice how the @PageNum parameter of each parameter allows the format of the running header to be specified while leaving the page number to be substituted later.

We may now define @OddPageList, whose result is a sequence of pages beginning with an odd-numbered page:

```
def @OddPageList
   named @Columns {}
   right @PageNum
{
   def @EvenPageList ...

      @PageMarker @PageNum
   // @Runner&&following @Open {
         @OnePage
            @Columns { @Columns }
            @PageTop { @TopOdd @PageNum }
            @PageFoot { @FootOdd @PageNum }
      }
   // @EvenPageList
         @Columns { @Columns }
         @Next @PageNum
}
```

Ignoring @EvenPageList for the moment, notice first that the invocation of @OnePage is enclosed in @Runner&&following @Open. Since @Runner&&following refers to the first invocation of @Runner appearing after itself in the final printed document, the symbols @TopOdd and @FootOdd will take their value from the first invocation of @Runner following the top of the page, even though @FootOdd appears at the foot of the page. Their @PageNum parameters are replaced by @PageNum, the actual page number parameter of @OddPageList.

After producing the odd-numbered page, @OddPageList invokes @EvenPageList:

```
def @EvenPageList
   named @Columns {}
   right @PageNum
{
      @PageMarker @PageNum
   // @Runner&&following @Open {
         @OnePage
            @Columns { @Columns }
            @PageTop { @TopEven @PageNum }
            @PageFoot { @FootEven @PageNum }
      }
   // @OddPageList
         @Columns { @Columns }
```

```
        @Next @PageNum
    }
```

This produces an even-numbered page, then passes the ball back to @OddPageList – a delightful example of what computer scientists call mutual recursion. The two page types differ only in their running headers and footers, but other changes could easily be made.

It was foreshadowed earlier that an invocation of @PageMarker would precede each page, and this has been done. Although this @PageMarker is a component of the root galley, it will not cause a page to be printed, because Basser Lout skips components of height zero.

4.4. Chapters and sections

The definitions of chapters and sections from the DocumentSetup package of Version 2 (in Version 3, the BookSetup extension of DocumentSetup) form the subject of this section. They allow a chapter to be entered like this:

```
@Chapter
   @Title { ... }
   @Tag { ... }
@Begin
   ...
@End @Chapter
```

Within the chapter a sequence of sections may be included by writing

```
@BeginSections
@Section { ... }
...
@Section { ... }
@EndSections
```

These are numbered automatically, and an entry is made for each in a table of contents.

The user of the DocumentSetup package can find the number of the chapter or section with a given tag by writing @NumberOf tag at any point in the document. This feature is based on the following definitions:

```
export @Tag
def @NumberMarker right @Tag { @Null }

def @NumberOf right x
{  @NumberMarker&&x @Open { @Tag }  }
```

Each chapter and section will contain one invocation of @NumberMarker; a full explanation will be given later.

A sequence of places for receiving chapters is easily defined:

```
export @Tag
```

```
def @ChapterList right @Tag
{
   @Galley
   //@ChapterGap  @ChapterList @Next @Tag
}
```

@ChapterGap will usually be 1.1b, ensuring that each chapter begins on a new page. The @Chapter galley itself is defined as follows:

```
export @FootNote @BeginSections @EndSections @Section
def @Chapter force into { @ChapterList&&preceding }
   named @Tag {}
   named @Title {}
   named @RunningTitle { dft }
   body @Body
{
   def @FootNote right x { @ColFootNote x }

   def @BeginSections ...
   def @EndSections ...
   def @Section ...

   def @ChapterTitle
   {
      @ChapterNumbers @Case {
         {Yes yes}  @Yield  { Chapter {@NumberOf @Tag}.  |2s  @Title }
         else  @Yield  @Title
      }
   }

   def @ChapterNum
   {
      @ChapterNumbers @Case {
         {Yes yes}  @Yield  { Chapter {@NumberOf @Tag} }
         else  @Yield @Null
      }
   }

      ragged @Break @BookTitleFormat @ChapterTitle
   //  @NumberMarker {
         @ChapterList&&@Tag @Open { @Tag }
      }
   //  @ChapterList&&preceding  @Tagged @Tag
   //  @NumberMarker&&preceding @Tagged @Tag
   //  @PageMarker&&preceding   @Tagged @Tag
   //  { @ChapterTitle } @MajorContentsEntry {@PageOf @Tag}
   //  @Runner
```

```
            @FootEven { |0.5rt 0.8f @Font @B @PageNum }
            @FootOdd  { |0.5rt 0.8f @Font @B @PageNum }
      // @Body
      //@SectionGap @ChapRefSection
      // @Runner
            @TopEven { @B @PageNum |1rt @I @ChapterNum }
            @TopOdd  { @I {@RunningTitle @OrElse @Title} |1rt @B @PageNum }
   }
```

We will see the symbols for sections shortly. Notice how their use has been restricted to within the right parameter of @Chapter, by nesting them and using a body parameter.

The meaning of @FootNote within @Chapter has been set to @ColFootNote, which produces a footnote targeted to @ColFootList (see Section 4.3). In other words, footnotes within chapters go at the foot of the column, not at the foot of the page. (Of course, in single-column books this distinction is insignificant.) @ChapterTitle and @ChapterNum are trivial definitions which vary depending on whether the user has requested numbered chapters or not.

Each invocation of @Chapter has its own unique @Tag, either supplied by the user or else inserted automatically by Lout. We now trace the cross referencing of chapter numbers on a hypothetical third chapter whose tag is euclid.

@ChapterList&&preceding @Tagged euclid attaches euclid as an extra tag to the first invocation of @ChapterList preceding itself in the final printed document. But this @ChapterList must be the target of the chapter, and so

```
@ChapterList&&euclid @Open { @Tag }
```

is 3, the number of the chapter (@Tag refers to the parameter of @ChapterList, not the parameter of @Chapter). Consequently the invocation of @NumberMarker within the chapter is equal to @NumberMarker 3.

@NumberMarker&&preceding @Tagged euclid attaches euclid to @NumberMarker 3 as an extra tag, and so @NumberOf euclid, which expands to

```
@NumberMarker&&euclid @Open { @Tag }
```

must be equal to 3, as required. This scheme could be simplified by placing the invocation of @NumberMarker within @ChapterList rather than within @Chapter, but it turns out that that scheme does not generalize well to sections and subsections.

There is a trap for the unwary in the use of preceding and following. Suppose that the invocation of @NumberMarker within @Chapter is replaced by the seemingly equivalent

```
@NumberMarker { @ChapterList&&preceding @Open { @Tag } }
```

Now suppose that @NumberOf euclid appears somewhere within Chapter 7. It will expand to

```
@NumberMarker&&euclid @Open { @Tag }
```

which would now be equal to

```
@ChapterList&&preceding @Open { @Tag }
```

whose value, evaluated as it is within Chapter 7, is 7, not 3. Use of preceding or following within the parameter of a symbol, rather than within the body, is likely to be erroneous.

Much of the remainder of the definition of @Chapter is fairly self-explanatory: there is a heading, a tag sent to mark the page on which the chapter begins, a @ContentsEntry galley sent to the table of contents, galleys for the figures and tables of the chapter to collect in, @Body where the body of the chapter goes, and @ChapRefSection to hold a concluding list of references. This leaves only the two invocations of @Runner to explain.

The first @Runner is just below the heading. It will be the target of the @Runner&&following cross reference at the beginning of the first page of the chapter (see Section 4.3), which consequently will have null running headers and the given footers.

The second @Runner appears at the very end of the chapter, hence on its last page. Since no invocations of @Runner lie between it and the first @Runner, it will be the target of @Runner&&following on every page from the second page of the chapter to the last, inclusive, and will supply the format of their headers and footers.

The interested reader might care to predict the outcome in unusual cases, such as when the heading occupies two pages, or when a chapter occupies only one, or (assuming a change to the gap between chapters) when a chapter starts halfway down a page. Such predictions can be made with great confidence.

The expression @RunningTitle @OrElse @Title appearing in the second @Runner returns the value of the @RunningTitle parameter of @Chapter if this is not equal to the default value dft, or @Title otherwise:

```
def @OrElse
   left x
   right y
{
   x @Case {
      dft  @Yield  y
      else  @Yield  x
   }
}
```

This produces the effect of

```
named  @RunningTitle  {  @Title  }
```

which unfortunately is not permissible as it stands, because @Title is not visible within the default value of @RunningTitle.

Finally, the definitions for sections omitted earlier are as follows:

```
def @EndSectionsPlace { @Galley }
def @EndSections force into { @EndSectionsPlace&&preceding } {}
macro @BeginSections { //@SectionGap @SectionList 1 // @EndSectionsPlace // }
```

```
def @Section force into { @SectionList&&preceding }
   named @Tag {}
   named @Title {}
   named @RunningTitle { dft }
   body @Body
{
   def @SectionTitle
   {
      @SectionNumbers @Case {
         {Yes yes} @Yield { {@NumberOf @Tag}.  |2s  @Title }
         else      @Yield @Title
      }
   }

      @Heading @Protect @SectionTitle
   // @NumberMarker {
         {@ChapterList&&@Tag @Open { @Tag }}.{
          @SectionList&&@Tag @Open { @Tag }}
      }
   // @ChapterList&&preceding  @Tagged @Tag
   // @SectionList&&preceding  @Tagged @Tag
   // @NumberMarker&&preceding @Tagged @Tag
   // @PageMarker&&preceding   @Tagged @Tag
   // { &3f @SectionTitle } @ContentsEntry {@PageOf @Tag}
   //0io @Body
}
```

The @BeginSections macro invokes @SectionList, preceded by the appropriate gap and followed by an @EndSectsPlace for closing the list of sections when the @EndSections symbol is found. @Section itself is just a copy of @Chapter with slight changes to the format. The parameter of @NumberMarker is a simple generalization of the one within @Chapter. Notice that we have taken care that the value of this parameter be a juxtaposition of simple words: although

```
{@ChapterList&&@Tag @Open { @Tag }}. &
{@SectionList&&@Tag @Open { @Tag }}
```

is formally equivalent, & was not permitted within a @Tag parameter until recently.

The DocumentSetup package also contains definitions for subsections in the same style. They raise the question of whether Lout is capable of producing subsections should the user place @BeginSections, @Section, and @EndSections within a *section*, and whether such nesting could proceed to arbitrary depth. Arbitrary nesting of sections within sections is available now, although the numbering would of course be wrong. The author has worked out definitions which provide correct numbering to arbitrary depth, with an arbitrary format for each level. These were not incorporated into DocumentSetup because the author considers sub-subsections to be poor style, and he prefers separate names for the symbols at each level.

4.5. Bibliographies

The first step in the production of a bibliography is to create a database of references based on the definition

```
export @Type @Author @Title @Institution @Number @Publisher
    @Year @Proceedings @Journal @Volume @Pages @Comment

def @Reference
  named @Tag              { TAG? }
  named @Type             { TYPE? }
  named @Author           { AUTHOR? }
  named @Title            { TITLE? }
  named @Institution      { INSTITUTION? }
  named @Number           { NUMBER? }
  named @Publisher        { PUBLISHER? }
  named @Year             { YEAR? }
  named @Proceedings      { PROCEEDINGS? }
  named @Journal          { JOURNAL? }
  named @Volume           { VOLUME? }
  named @Pages            { PAGES? }
  named @Comment          { @Null }
{ @Null }
```

For example, the database might contain

```
{ @Reference
   @Tag { strunk1979style }
   @Type { Book }
   @Author { Strunk, William and White, E. B. }
   @Title { The Elements of Style }
   @Publisher { MacMillan, third edition }
   @Year { 1979 }
}

{ @Reference
   @Tag { kingston92 }
   @Type { TechReport }
   @Author { Kingston, Jeffrey H. }
   @Title { Document Formatting with Lout (Second Edition) }
   @Number { 449 }
   @Institution { Basser Department of Computer
Science F09, University of Sydney 2006, Australia }
   @Year { 1992 }
}
```

Since named parameters are optional, we have one for every conceivable type of attribute, and simply leave out those that do not apply in any particular reference. We can print a reference by using the @Open symbol to get at its attributes:

```
@Reference&&strunk1979style @Open
{  @Author,  {Slope @Font @Title}.  @Publisher, @Year.  }
```

The right parameter of @Open may use the exported parameters of the left, and so the result is

William Strunk and E. B. White, *The Elements of Style.* Macmillan, 1979.

Incidentally, we are not limited to just one database of references; several @Database symbols can nominate the same symbol, and invocations of that symbol can appear in the document itself as well if we wish.

The second step is to create a database of print styles for the various types of reference (Book, TechReport, etc.), based on the following definition:

```
export @Style
def @RefStyle
  left @Tag
  named @Style right reftag {}
{}
```

Notice that the named parameter @Style has a right parameter reftag. The style database has one entry for each type of reference:

```
{ Book @RefStyle @Style
  {  @Reference&&reftag @Open
     { @Author,  {Slope @Font @Title}.  @Publisher, @Year.  @Comment }
  }
}

{ TechReport @RefStyle @Style
  {  @Reference&&reftag @Open
     { @Author,  {Slope @Font @Title}.  Tech. Rep. @Number (@Year),
@Institution.  @Comment }
  }
}
```

and so on. The following prints the reference whose tag is strunk1979style in the Book style:

```
@RefStyle&&Book @Open { @Style strunk1979style }
```

It has result

William Strunk and E. B. White. *The Elements of Style.* Macmillan. Third Edition, 1979.

Notice how the @Style parameter of @RefStyle is given the parameter strunk1979style, which it uses to open the appropriate reference.

We can consult the @Type attribute of a reference to find out its style, which brings us to the following definition for printing out a reference in the style appropriate to it:

```
def @RefPrint
  right reftag
{  @RefStyle&&{ @Reference&&reftag @Open { @Type } }
   @Open { @Style reftag }
}
```

For example, to evaluate `@RefPrint strunk1979style`, Lout first evaluates

```
@Reference&&strunk1979style  @Open  { @Type }
```

whose result is `Book`, and then evaluates

```
@RefStyle&&Book @Open { @Style strunk1979style }
```

as before. Complicated as this is, with its two databases and clever passing about of tags, the advantages of separating references from printing styles are considerable: printing styles may be changed easily, and non-expert users need never see them.

Finally, we come to the problem of printing out a numbered list of references, and referring to them by number in the body of the document. The first step is to create a numbered list of places that galleys containing references may attach to:

```
def @ReferenceSection
   named @Tag {}
   named @Title { References }
   named @RunningTitle { dft }
   named style right tag { tag. }
   named headstyle right @Title { @Heading @Title }
   named indent { @DispIndent }
   named gap { @DispGap }
   named start { 1 }
{
   def @RefList right num
   {
      @NumberMarker num  &  indent @Wide {style num}  |  @RefPlace
      //gap @RefList @Next num
   }

      @Protect headstyle @Title
   //  @PageMarker&&preceding @Tagged @Tag
   //  @Title @MajorContentsEntry {@PageOf @Tag}
   //  @Runner
          @FootEven { |0.5rt 0.8f @Font @B @PageNum }
          @FootOdd  { |0.5rt 0.8f @Font @B @PageNum }
   //@DispGap  @RefList start
   //  @Runner
          @TopEven { @B @PageNum }
          @TopOdd  { @I {@RunningTitle @OrElse @Title} |1rt @B @PageNum }
}
```

We place the expression @ReferenceSection at the point where we want the list of references to appear; its value is something like

```
1.  @RefPlace
2.  @RefPlace
3.  @RefPlace
...
```

where @RefPlace is @Galley as usual. We can scatter multiple lists of references through the document if we wish (at the end of each chapter, for example), simply by placing @ReferenceSection at each point.

Our task is completed by the following definition:

```
def @Ref right x
{
   def sendref into { @RefPlace&&following }
      right @Key
   {
      @NumberMarker&&preceding @Tagged x  &
      @PageMarker&&preceding @Tagged x  &
      @RefPrint x
   }

   @NumberMarker&&x @Open { @Tag } sendref x
}
```

Given this definition, the invocation @Ref strunk1979style has result

```
@NumberMarker&&strunk1979style @Open { @Tag }
```

plus the galley sendref strunk1979style. We first follow what happens to the galley.

According to its into clause, the galley will replace a @RefPlace in the nearest following @ReferenceSection. If every such galley is a sorted galley whose key is the reference's tag, as this one is, they will appear sorted by tag. The galley's object is

```
@NumberMarker&&preceding @Tagged strunk1979style  &
@PageMarker&&preceding @Tagged strunk1979style  &
@RefPrint strunk1979style
```

The result of the @Tagged symbol is always @Null, so this prints the strunk1979style reference in the appropriate style at the @RefPlace, as desired.

Now @NumberMarker&&preceding is the nearest preceding invocation of @NumberMarker in the final document. This must be the invocation of @NumberMarker just before the @RefPlace that received the galley, and so this invocation of @NumberMarker is given strunk1979style as an additional tag by the @Tagged symbol. Its original tag was the number of the reference place, which means that

```
@NumberMarker&&strunk1979style @Open { @Tag }
```

has for its result the number of the reference place that received the strunk1979style galley, and this is the desired result of @Ref strunk1979style.

It might seem that if we refer to the strunk1979style reference twice, two copies will be sent to the reference list and it will appear twice. However, when more than one sorted galley with the same key is sent to the same place, only one of them is printed (Section 1.4); so provided that sorted galleys are used there is no problem.

References

[1] Adobe Systems, Inc.. *PostScript Language Reference Manual, Second Edition.* Addison-Wesley, 1990.

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

[3] Jeffrey H. Kingston. *The Basser Lout Document Formatting System (Version 3).* Computer program, 1995. 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, and by electronic mail from the author. All enquiries to *jeff@cs.su.oz.au.*

[4] Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System (Version 3).* Basser Department of Computer Science, University of Sydney, 1995.

[5] Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.

[6] Brian K. Reid. 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.

[7] William Strunk and E. B. White. *The Elements of Style.* Macmillan. Third Edition, 1979.

Index