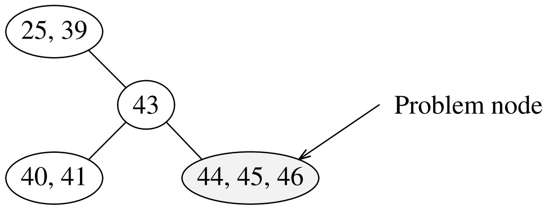
Fig – A Lout Package for Drawing Figures

Jeffrey H. Kingston

Basser Department of Computer Science
The University of Sydney 2006
Australia

ABSTRACT

This report describes the use of Fig, a package of definitions for use with the Lout document formatter. Fig draws, colours, and positions arbitrary shapes made from straight lines, circular and elliptical arcs, and Bezier curves:



The graphical objects may be rotated and scaled; they may enclose, and be enclosed by arbitrary Lout objects (text, equations, tables, other graphical objects, etc.) without restriction. A convenient algebra of points and a method of labelling points assist positioning.

22 December, 1992

Fig – A Lout Package for Drawing Figures

Jeffrey H. Kingston

Basser Department of Computer Science
The University of Sydney 2006
Australia

1. Introduction

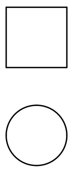
Fig is a package of Lout definitions for drawing and filling in arbitrary shapes made from straight lines, circular and elliptical arcs, and Bezier curves. Its features are smoothly integrated with the rest of Lout: one can rotate and concatenate objects created by Fig, draw a box to fit neatly around any object, etc. The design of Fig is based entirely on Brian W. Kernighan's PIC language [2]. The implementation of Fig makes good use of the PostScript[1] page description language [1], which was designed by John Warnock and others. Lout was designed and implemented by Jeffrey H. Kingston [3].

To use Fig within a Lout document, first ensure that its definition is included, either by putting @SysInclude { fig } at the start of the document, or -ifig on the command line. Then, anywhere at all within the document, write

```
@Fig {  ...  }
```

and the symbols of Fig will be available between the braces, which may enclose an arbitrary Lout object. Throughout this report we will show the Lout text on the left and the corresponding result on the right, like this:

```
@Fig {
  @Square
  //0.5c
  @Circle
}
```

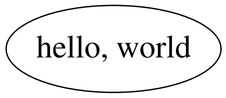


Subsequent examples will omit the enclosing @Fig.

2. Basic Shapes

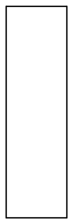
Fig has a repertoire of basic shapes, whose size varies depending on what they enclose:

```
@Ellipse { hello, world }
```



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

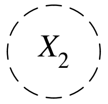
```
@Box { 2c @High }
```



There are six such shapes: @Box, @Square, @Diamond, @Polygon, @Ellipse, and @Circle; the result in each case is the right parameter, enclosed in a small margin, with the shape around it.

There are options for changing the appearance of these shapes. The boundary line's style may be solid, dashed, cdashed, dotted, or noline (that is, no line is drawn), and the length of the dashes may be changed:

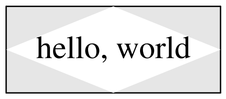
```
@Circle
  linestyle { cdashed }
  dashlength { 0.2 cm }
@Eq { X sub 2 }
```



If the line style is not mentioned, it becomes solid by default. The dashed option makes all dashes the same length; cdashed halves the length of the first and last dash on each segment, which looks better in some cases. The distance between dashes or dots will be at most dashlength, reduced to make the dashes or dots fit evenly.

Shapes may be painted black, dark, grey or gray, light, white, or nopaint (the default):

```
@Box
  margin { 0c }
  paint { grey }
@Diamond
  linestyle { noline }
  paint { white }
{ hello, world }
```

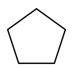


Here, the right parameter of @Box is a diamond containing hello, world. There is no limit to the amount of this sort of nesting; the right parameter may be any Lout object.

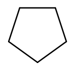
When painting it is important to know what order things are done in, because anything put down earlier will disappear under the paint. This is why nopaint and white are different. Painting is done first, then boundaries, and finally the right parameter.

The @Polygon shape has a sides option for specifying the number of sides, and an angle option for specifying what angle anticlockwise from vertically beneath the centre the first corner will appear at:

```
@Polygon
  sides { 5 }
{}
```



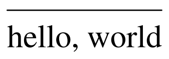
```
@Polygon
  sides { 5 }
  angle { 0 dg }
{}
```



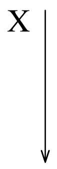
The defaults are 3 sides and the angle that gives the polygon a horizontal base (i.e. 180 degrees divided by the number of sides). Thus the two cases with symmetry about a vertical axis are obtained by the default angle and 0 dg respectively, which is convenient.

Although lines and arrows do not enclose things in the way that boxes and circles do, Fig treats them as it does the other shapes. The line or arrow is drawn along the mark of the right parameter, either horizontally or vertically:

```
@HLine { //0.2c hello, world }
```

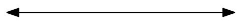


```
@VArrow { 2c @High X ^|0.2c }
```

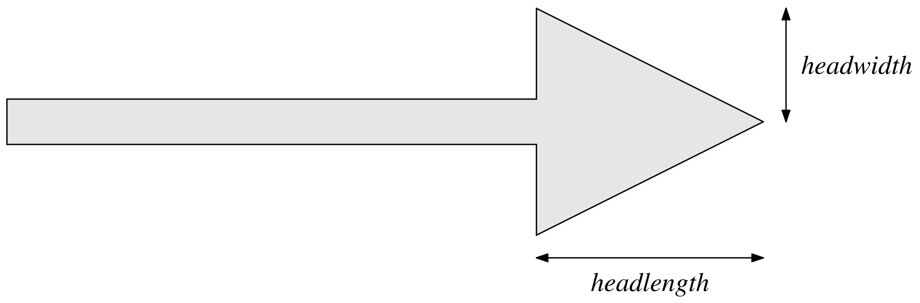


The usual line style options are available; the default margin is 0c. Arrows can be forward (the default), back, both, or noarrow (which just draws a line); the style of the arrowhead can be open (the default), halfopen, or closed:

```
@HArrow
  arrow { both }
  headstyle { closed }
{ 3c @Wide }
```



It is also possible to change the shape of the arrowhead, using the headwidth and headlength options:

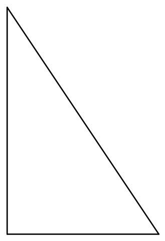


Their default values are 0.05 cm and 0.15 cm respectively.

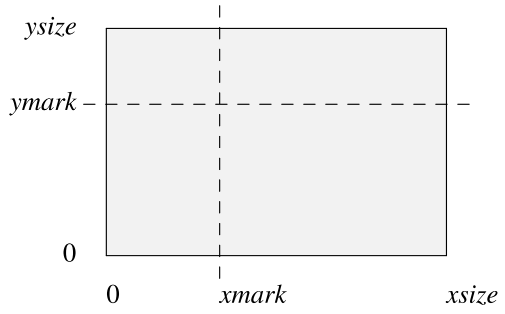
3. Creating New Shapes

If the needed shape is not provided by Fig, it can be created using the @Figure symbol. @Figure takes all the options we have already seen, plus another one called shape. For example,

```
@Figure
  shape {
    0 0  xsize 0
    0 ysize  0 0
  }
{ 3c @High 2c @Wide }
```



The pairs of numbers define points in a coordinate system whose origin is the lower left corner of the right parameter; the upper right corner is xsize ysize, and the point where the right parameter's marks cross is xmark ymark:



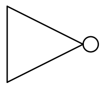
This arrangement is identical with that for the @Graphic symbol of basic Lout. A sequence of points defines a shape, like the triangle above. Arrowheads are drawn pointing forwards from the last segment and backwards from the first, as requested; the margin option has default value 0c.

Normally, the points are connected by straight lines to form the shape, which is then painted and drawn in the usual way, depending on the other options.

If two points in the shape are separated by [], no line will be drawn between them. This permits a shape to consist of two or more disconnected parts.

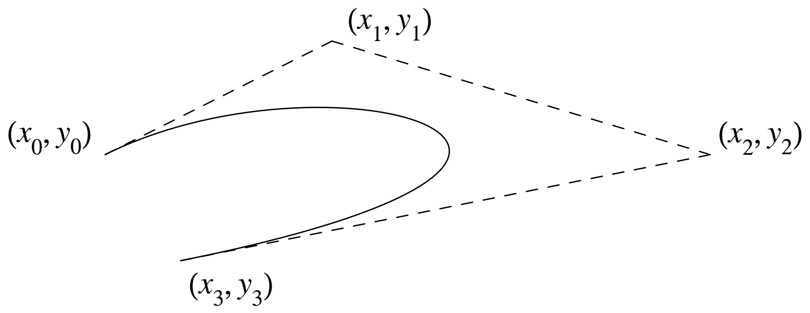
If two points in the shape are separated by [*x y*], where *x* and *y* are numbers, the two points will be joined by an anticlockwise arc whose centre is the point (x, y). This arc will be circular if possible, otherwise it will be part of an ellipse whose axes are oriented horizontally and vertically. The notation [*x y* clockwise] makes the arc go clockwise. For example,

```
@Figure
  shape {
    0 -0.5 cm
    0 0.5 cm
    1 cm 0  0 -0.5 cm []
    1 cm 0 [ 1.1 cm 0 ]
    1 cm 0
  }
{}
```



We have recklessly disregarded the size of the right parameter when drawing this shape, a dangerous thing to do since Lout thinks that the figure is the same size as its right parameter.

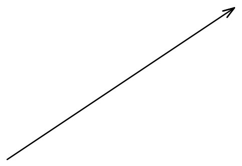
Finally, two points may be separated by [x_1 y_1 x_2 y_2], which requests that a Bezier curve be drawn between them with control points (x_1, y_1) and (x_2, y_2):



The curve is attracted toward the control points, without reaching them; it is tangent to the straight line from the start point to the first control point, and from the second control point to the finishing point, and it lies wholly inside the quadrilateral formed by the four points. Owing to the author's laziness, dashes and dots do not fit as neatly onto Bezier curves as they do onto lines and arcs. @Figure should be general enough to draw most shapes; for example, all the other shapes (@Box, @Circle, etc.) are just instances of @Figure. When it is inadequate, one can fall back to the standard @Graphic symbol.

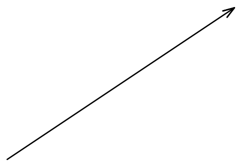
Lines, arrows and arcs at any angle can be produced using @Figure:

```
@Figure
  shape { 0 0  xsize ysize }
  arrow { forward }
{ 2c @High 3c @Wide }
```



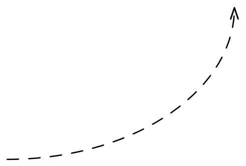
However, for convenience there are symbols @Line and @Arrow which have two options, from and to, for specifying the endpoints:

```
@Arrow
  from { 0 0 }
  to { xsize ysize }
{ 2c @High 3c @Wide }
```



There is also an @Arc symbol, which draws a circular or elliptical arc from one point to another about a given centre, with the usual options:

```
@Arc
  from { 0 0 }
  to { xsize ysize }
  ctr { 0 ysize }
  direction { anticlockwise }
  arrow { forward }
  linestyle { dashed }
{ 2c @High 3c @Wide }
```



The arc goes either clockwise (the default) or anticlockwise about the centre, depending on the direction option. Any arrowhead will point in a direction tangent to the arc.

4. Colour

An arbitrary Lout object may be printed in colour, like this:

```
red @Colour hello
```
hello

@Colour may also be spelt @Color. Of course, a colour printing device is needed to see the effect. The @Colour symbol is intended to provide a fixed palette of colours indicated by names, including white, grey, gray, black, red, green, and blue at least.

An unlimited range of colours can be obtained with the @RGBColour (or @RGBColor) symbol, which is given three numbers in the range 0 to 1 specifying the required intensity of red, green and blue colour in that order. For example,

```
{1.0 0.0 0.0} @RGBColour hello
```

is equivalent to red @Colour hello. There is also @HSBColour (or @HSBColor) for specifying colour using the hue-saturation-brightness model (see [1], Section 4.8).

5. Lengths, angles, and points

We already know that two lengths placed side by side define a point. This is only the simplest of a number of such geometrical combinations.

The symbol @Distance returns the length of the line joining two points:

```
{0 0} @Distance {3 cm  4 cm}
```

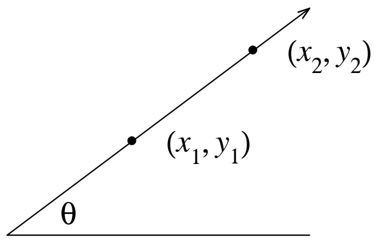
is equivalent to the length 5 cm. The result of @Distance is never negative. Notice that braces must enclose the two points. The symbol @XDistance returns the distance in the *x* direction from one point to another:

{x_1 y_1} @XDistance {x_2 y_2}

has for its result the length $x_2 - x_1$, and so may be negative. There is an analogous @YDistance symbol.

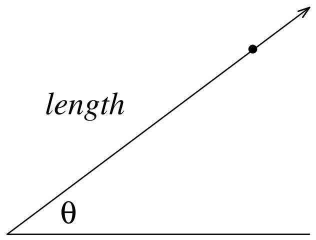
The symbol @Angle returns the angle θ from one point to another:

{x_1 y_1} @Angle {x_2 y_2}



The result will be 0 if the two points are equal. The symbol << returns the point at a given distance and angle from (0, 0):

{*length* << θ}



For example, { 5 cm << 45 dg } is the point 5 cm from (0, 0) at 45 degrees.

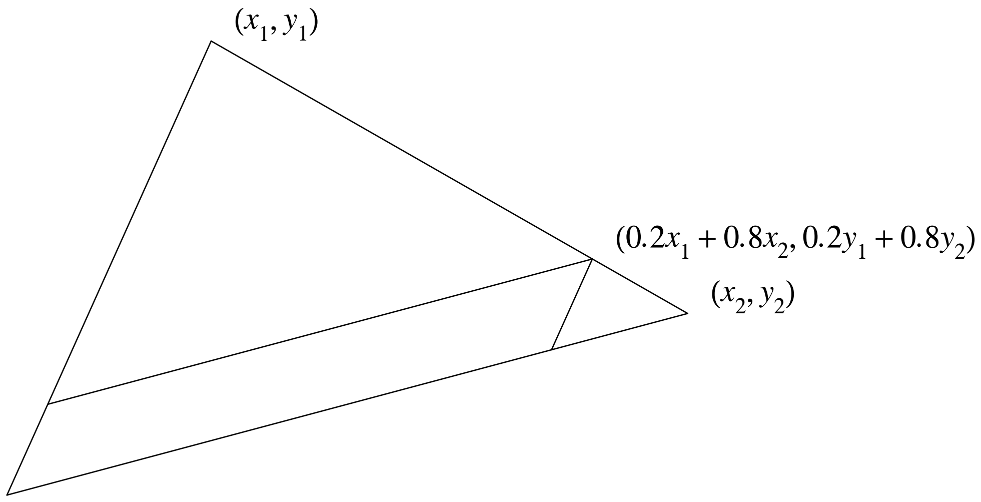
Points may be added, subtracted, and multiplied by a number:

$\{x_1\ y_1\}$ ++ $\{x_2\ y_2\}$	is	$(x_1 + x_2, y_1 + y_2)$
$\{x_1\ y_1\}$ -- $\{x_2\ y_2\}$	is	$(x_1 - x_2, y_1 - y_2)$
$\{x_1\ y_1\}$ ** k	is	$(x_1 k, y_1 k)$

For example,

$\{x_1\ y_1\}$ ** 0.2 ++ $\{x_2\ y_2\}$ ** 0.8

is the point eight tenths of the way from (x_1, y_1) to (x_2, y_2) on the line joining them:



When using **, the point must be on the left and the number on the right. It would be more convenient to name these symbols +, -, and *, but these names are often taken by equation formatters, and - appears in lengths, so we don't. There are @Max and @Min symbols:

$\{x_1\ y_1\}$ @Max $\{x_2\ y_2\}$	is	$(\max(x_1, x_2), \max(y_1, y_2))$
$\{x_1\ y_1\}$ @Min $\{x_2\ y_2\}$	is	$(\min(x_1, x_2), \min(y_1, y_2))$

Note carefully that these apply to points, not to numbers.

The result of adding two points together depends on where the origin is at the time, as well as on the points themselves. This can lead to unexpected results, as the author has found to his cost more than once. Within the shape option of @Figure, the origin is the lower left corner of the result of the @Figure. In cases like the example on page 12, where points are added outside of any @Figure symbol, the origin is usually at the bottom left corner of the figure as a whole.

A label always denotes a particular point on the printed page, regardless of where the origin happens to be.

The symbol @Prev within the shape option of @Figure denotes the previous point of the shape, ignoring points within []. This makes it easy to specify each point relative to the previous one:

```
shape {
  0 0
  { 2 cm << 30 }
  @Prev ++ { 2 cm << 90 }
  @Prev ++ { 2 cm << 150 }
  @Prev ++ { 2 cm << 210 }
  @Prev ++ { 2 cm << 270 }
  0 0
}
```

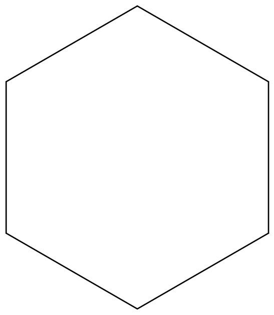
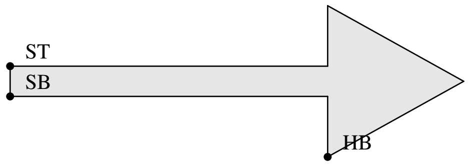


Fig provides a @Label symbol for attaching a label to a point in a shape, like this:

```
{xsize ysize} ** 0.5 @Label CTR
```

The point may then be referred to more concisely by its label, CTR. For example, the large arrow appearing in Section 2 was built like this:

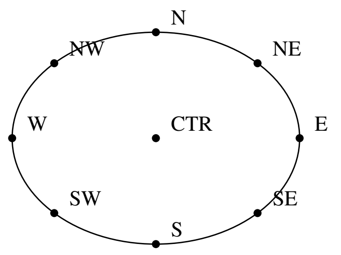
```
@Figure
  shape {
    {0 ysize} ** 0.4 @Label SB
    {0 ysize} ** 0.6 @Label ST
    {xsize 0} ** 0.7 @Label HB
    SB
    SB ++ HB
    HB
    {xsize 0} ++ {0 ysize} ** 0.5
    HB ++ {0 ysize}
    HB ++ ST
    ST
    SB
  }
  paint { grey }
{ 6c @Wide 2c @High }
```



Incidentally, the labels of a figure can be displayed as above by putting the symbol @ShowLabels at the end of the figure. Labels can save a lot of effort. They should contain only digits, upper-case letters and @, because Fig and Lout use labels of their own made from lower-case letters.

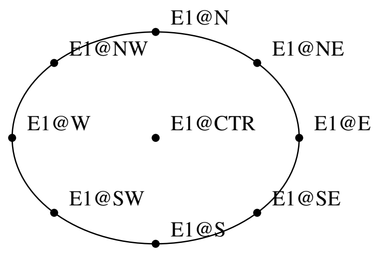
The standard shapes have standard labels; for example, the labels of @Ellipse are

```
@Ellipse
{ 3c @Wide 2c @High }
```



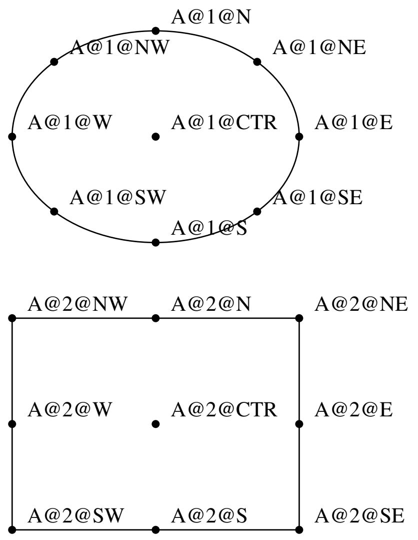
There is a symbol, ::, for *relabelling* an object. Each label in the right parameter is relabelled in the following way:

```
E1:: @Ellipse
{ 3c @Wide 2c @High }
```



Within the right parameter of :: the original names hold sway; but afterwards the names are changed by prefixing the label and @ to them. These composite labels may be used exactly like other labels. Relabelling can be nested to arbitrary depth:

```
A::
{   1:: @Ellipse
    { 3c @Wide 2c @High }
    //1c
    2:: @Box
    { 3c @Wide 2c @High }
}
```



The right parameter of :: may be any object.

The six standard shapes (@Box, @Square, @Diamond, @Polygon, @Ellipse, and @Circle) have a special CIRCUM label, not displayed by @ShowLabels. The expression

angle CIRCUM

yields the point on the boundary of the shape at the given angle from the centre, in a coordinate system with the centre for origin. Thus, given a labelled standard shape such as

```
A :: @Ellipse ...
```

the point on its boundary at an angle of 45 degrees from the centre is

```
A@CTR ++ {45 A@CIRCUM}
```

The braces must be present. Regrettably, there is no way to produce a CIRCUM label for shapes defined by the user in any reasonable time.

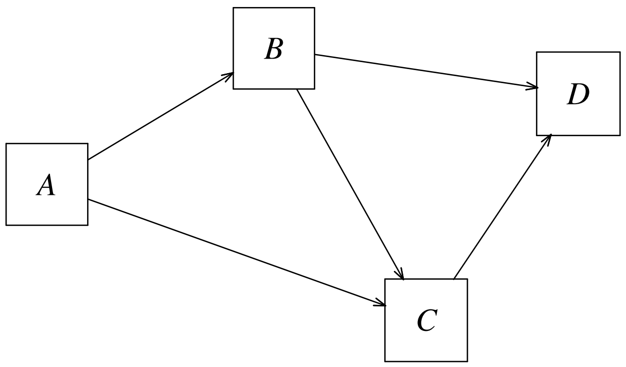
If the same label is used twice, as is inevitable with unlabelled standard shapes, only the most recent value is remembered. There is a limit on the maximum number of distinct labels in any one figure, which can be changed by means of an option to the @Fig symbol:

```
@Fig
  maxlabels { 500 }
{ ... }
```

The default value is 200. Large values could cause the printing device to run out of memory. Memory is reclaimed at the end of each figure.

6. Putting it all together

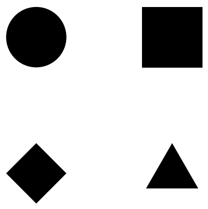
In this section we consider the problem of linking individual shapes together to form complex diagrams like this one:



We already have several aids to hand: the standard Lout symbols, especially horizontal and vertical concatenation, rotation and scaling; the ability to nest text, equations, and other figures (in fact arbitrary Lout objects) within our shapes; and the standard Lout definition mechanism.

The default values of the various options – solid for linestyle, noarrow for arrow, and so on – may be changed by giving options to the @Fig symbol:

```
@Fig
  linestyle { noline }
  paint { black }
{
  @Circle |1c @Square
  /1c @Diamond | @Polygon
}
```



A complete list of options is given in the next section.

Fig provides an additional aid: the symbols @BaseOf and @MarkOf. The right parameter of @BaseOf is an arbitrary object, and its left parameter is a point. As far as Lout is concerned, the result of @BaseOf is always an empty object; but the right parameter will appear on the page with the bottom left-hand corner of its base at the point denoted by the left parameter. We stress that Lout has no idea that this is happening, and so cannot prevent the shifted object from writing over other objects or even going off the edge of the page. Of course, this lack of discipline is just what is needed very often in diagrams.

The @MarkOf symbol works in a similar way, except that the point where the object's marks cross (rather than its bottom left-hand corner) will appear on the page at the point denoted by the left parameter.

We can set up a coordinate system for a figure:

```
@Figure shape { xsize 0 @Label X  0 ysize @Label Y }
{ 10c @Wide 6c @High }
```

In fact, Fig contains this shape under the name @Frame, so we need only write

```
@Frame { 10c @Wide 6c @High }
```

Of course, the right parameter may contain an arbitrary Lout object.

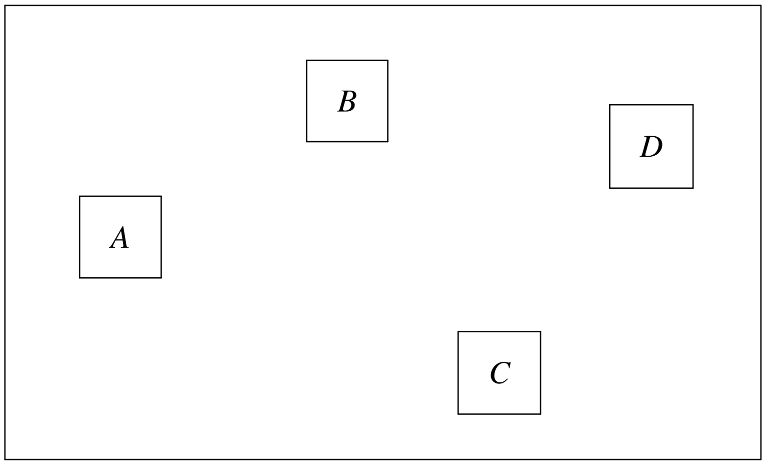
Once the frame is set up, we can specify points by their X and Y coordinates, as fractions of the total width and height:

```
X ** 0.5  ++  Y ** 0.8
```

To place the squares in the diagram above, we can use

```
//  X**0.1 ++ Y**0.4  @BaseOf  A:: @Square { @I A }
//  X**0.4 ++ Y**0.7  @BaseOf  B:: @Square { @I B }
//  X**0.6 ++ Y**0.1  @BaseOf  C:: @Square { @I C }
//  X**0.8 ++ Y**0.6  @BaseOf  D:: @Square { @I D }
```

The symbols' precedences are chosen so that this very common situation requires no braces. The result of this is



where we have drawn a box with margin 0 around the frame to make its extent clear.

Now the arrow from *A* to *B* starts on the boundary of *A* at the angle of a line drawn between the centres of *A* and *B*:

```
A@CTR ++ { {A@CTR @Angle B@CTR} A@CIRCUM }
```

and a similar expression will yield the endpoint of the arrow. Such expressions should be placed into definitions if they are to be used often:

```
import @Fig
def @JoinFigures
  left A
  named linestyle { solid }
  named dashlength { 0.15 cm }
  named arrow { noarrow }
  named linewidth { 0.5 pt }
  right B
{  @Arrow
     from { A"@CTR" ++ {{A"@CTR" @Angle B"@CTR"} A"@CIRCUM"} }
     to   { B"@CTR" ++ {{B"@CTR" @Angle A"@CTR"} B"@CIRCUM"} }
     linestyle { linestyle }
     dashlength { dashlength }
     arrow { arrow }
     linewidth { linewidth }
  {}
}
```

Definitions preceded by import @Fig may use the symbols of Fig within them; they may themselves be used only within @Fig { ... }. Now, to the figure above we can add

```
// A @JoinFigures arrow { forward } B
// A @JoinFigures arrow { forward } C
// B @JoinFigures arrow { forward } C
// B @JoinFigures arrow { forward } D
// C @JoinFigures arrow { forward } D
```

to obtain the diagram as it appears at the beginning of this section. Definitions are the best means of managing complexity in diagrams, and serious users maintain a file of definitions which is included automatically by an @Include command.

7. Errors

Lout normally produces an output file that will print without mishap on any PostScript device. However, some of the options of Fig's symbols are passed through Lout to the output file without checking, including anything containing Fig lengths, angles, points, and labels. Any errors in these options will not be detected until the file is printed.

The most likely errors are *syntax errors*, as in shape { 0 0 [0 xsize } for example, in which a] is missing; *type errors*, as in 0 0 @Distance 45 where the right parameter is not a point; and *undefined errors*, arising from labels misspelt or used before being defined. Less commonly, the options may all be correct but the figure is too large in some way: too many labels, too deeply nested, etc.

When an error is detected, Fig arranges for the offending page to be printed up to the point where the error occurred, with a message nearby describing the error. Printing of the document is then aborted.

8. A Concise Reference for Fig

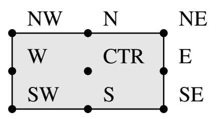
The options to the @Fig symbol, and their default values, are as follows. The complete range of options is shown at right:

```
@Fig
  maxlabels  { 200     }     any whole number
  linestyle  { solid   }     solid dashed cdashed dotted noline
  linewidth  { 0.5 pt  }     any Fig length (see below)
  linecap    { round   }     butt round project
  dashlength { 0.15 cm }     any Fig length
  paint      { nopaint }     nopaint white light grey gray dark black
  margin     { 0.4c    }     any Lout length
  arrow      { noarrow }     noarrow forward back both
  headstyle  { open    }     open halfopen closed
  headwidth  { 0.05 cm }     any Fig length
  headlength { 0.15 cm }     any Fig length
```

The linecap option determines the shape of the ends of lines: round puts a round cap on them, which is the most useful in Fig; butt is a square end; and project is a square end projecting half a line width past the end of the line; it is useful for getting sharp corners on rectangles and square dots in dotted lines.[1]

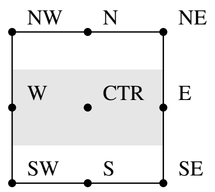
The following standard shapes take the same options as @Fig, except that they do not have maxlabels or the last four (arrow-drawing) options, and occasionally they have other options. In most cases the default values of these options are taken from the enclosing @Fig. Where there are extra options, or where a different default value is used, the option and its default value are shown. The list also shows the shape's labels, and how it is superimposed on its right parameter (shown as a grey rectangle). A larger margin will enlarge the right parameter and hence the shape as well. Squares, polygons and circles have a diameter equal to the larger of xsize and ysize.

@Box

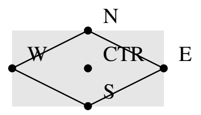


[1]The line joining options of PostScript are not reflected in Fig options because Fig strokes paths segment by segment, not all at once, and so there are no line joins in the PostScript sense. This was done to improve the appearance of dashed and dotted lines.

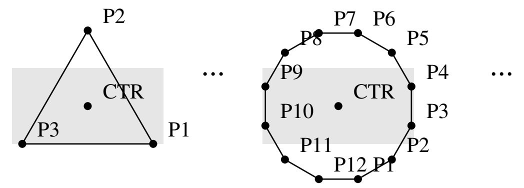
@Square



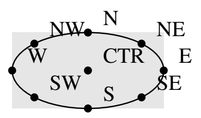
@Diamond



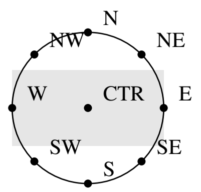
@Polygon
  sides { 3 }
  angle { 180/sides }



@Ellipse

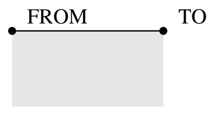


@Circle

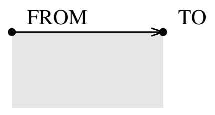


The following standard shapes have the same options as @Fig, including the four arrow-drawing options, and occasionally they have others. In each case the only difference between the line and arrow symbols is the default value of arrow, which lines take from @Fig and arrows set to forward. The first four draw a line along the mark of the right parameter, which is not necessarily the same as its left or top edge.

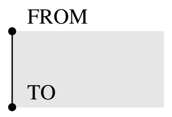
@HLine
  margin { 0c }



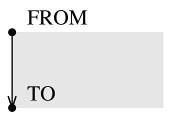
@HArrow
  margin { 0c }
  arrow { forward }



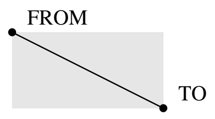
@VLine
  margin { 0c }



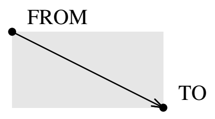
```
@VArrow
  margin { 0c }
  arrow { forward }
```



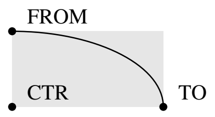
```
@Line
  from { 0 ysize }
  to { xsize 0 }
  margin { 0c }
```



```
@Arrow
  from { 0 ysize }
  to { xsize 0 }
  margin { 0c }
  arrow { forward }
```



```
@Arc
  from { 0 ysize }
  to { xsize 0 }
  ctr { 0 0 }
  direction { clockwise }
  margin { 0c }
```



More generally, the @Figure symbol takes all the options of @Fig except maxlabels, together with a shape option containing a sequence of points, and it connects each pair of points by a line or curve as specified by the following:

point point	Straight line
point [] *point*	Draw nothing
point [*point*] *point*	Anticlockwise circular or elliptical arc
point [*point* clockwise] *point*	Clockwise circular or elliptical arc
point [*point point*] *point*	Bezier curve with two control points

The remaining symbols do not draw shapes. They are

colour @Colour *object*
colour @Color *object*

Print *object* in *colour*, which may be white, grey, gray, black, red, green, or blue at least.

{ *number number number* } @RGBColour *object*
{ *number number number* } @RGBColor *object*
{ *number number number* } @HSBColour *object*
{ *number number number* } @HSBColor *object*

Print *object* in the colour defined by the three numbers, using the red-green-blue or hue-saturation-brightness colour models.

point @Label *label*

Within a shape, makes *label* stand for the point on the page denoted by *point*, which is not made part of the shape. The label applies from here onwards until some other point is given this label, a relabelling occurs, or the figure ends.

label :: *object*

Relabel every labelled point in the right parameter (which may be an arbitrary Lout object), by adding *label*@ to the front of each label.

@Frame *object*

Equivalent to @Figure shape {xsize 0 @Label X 0 ysize @Label Y} *object*.

point @BaseOf *object*

Translate *object* so that its bottom left-hand corner appears at *point*. Lout thinks that the result is an empty object.

point @MarkOf *object*

Translate *object* so that the point where its marks cross appears at *point*. Lout thinks that the result is an empty object.

@ShowLabels

Display all the labels of the figure created up to this point.

The following lists define all the ways to specify lengths, angles and points.[1] Brief explanations appear to the right, with the symbols' precedences in parentheses where appropriate.

length

0	zero
xmark	distance to column mark
ymark	distance to row mark
xsize	distance to right boundary
ysize	distance to top boundary
number in	*number* inches (39)
number cm	*number* centimetres (39)
number pt	*number* points (39)

[1]A length is represented in PostScript by a single number on the operand stack; so is an angle. A point is represented by two numbers on the stack. Those familiar with PostScript and willing to sacrifice portability and increase their risk of error can therefore write, for example, *length* sqrt within a shape, to obtain a length which is the square root of another length, or *point* exch to obtain the reflection of a point about the main diagonal, and so on.

number em	*number* ems (39)
number sp	1 sp is the current width of a space (39)
number vs	1 vs is the current inter-line space (39)
number ft	1 ft is the size of the current font (39)
point @Distance *point*	distance between two points (35)
point @XDistance *point*	horizontal distance between two points (35)
point @YDistance *point*	vertical distance between two points (35)

angle

number dg	*number* degrees (39)
number	*number* degrees (dg is optional)
point @Angle *point*	angle from first point to second (35)

point

length length	*x* and *y* distance from origin (5)
length << *angle*	distance and angle from origin (38)
point ++ *point*	vector sum of two points (36)
point -- *point*	vector difference of two points (36)
point @Max *point*	vector maximum of two points (36)
point @Min *point*	vector minimum of two points (36)
point ** *number*	multiplication of point by number (37)
label	a previously defined label
@Prev	the previous point in a shape

References

1. Adobe Systems, Inc., *PostScript Language Reference Manual, Second Edition*. Addison-Wesley, 1990.

2. Kernighan, Brian W., PIC – A language for typesetting graphics. *Software Practice and Experience* **12**, 1–21 (1982).

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