Aspic: A Line-Art Processor

Philip Hazel

Aspic: A Line-Art Processor

Author: Philip Hazel

Revision 2.00   09 October 2022

Contents

1. Introduction to Aspic **1**

1.1 The aspic command 1

2. Simple Aspic examples **3**

3. General operation of Aspic **12**

3.1 Position of the coordinate origin 12
3.2 The bounding box 12

4. Aspic input **13**

4.1 Command format 13
4.2 File inclusion 13

5. Aspic variables **14**

6. Aspic macros **15**

7. Types of value used in commands **17**

7.1 <angle> 17
7.2 <boxpoint> 17
7.3 <colour> 17
7.4 <greylevel> 17
7.5 <integer> 17
7.6 <label> 17
7.7 <length> 17
7.8 <position> 17

8. Drawing objects and text **19**

8.1 arc 20
8.2 arcarrow 22
8.3 arrow 22
8.4 box 23
8.5 circle 24
8.6 curve 24
8.7 ellipse 24
8.8 iarc 25
8.9 ibox 25
8.10 icircle 25
8.11 icurve 25
8.12 iellipse 25
8.13 iline 25
8.14 line 25
8.15 text 26

9. Filled shapes **28**

10. Altering the 'last' item **29**

11. Changing the current direction **30**

12. Font control and character coding **31**

12.1 The bindfont command 31
12.2 The setfont command 31
12.3 The textdepth and fontdepth commands 31
12.4 Input character encoding 31
12.5 PostScript output character encoding 32
12.6 SVG output character encoding 33

13. Overall Aspic configuration **34**

13.1 boundingbox 34
13.2 magnify 34
13.3 resolution 34

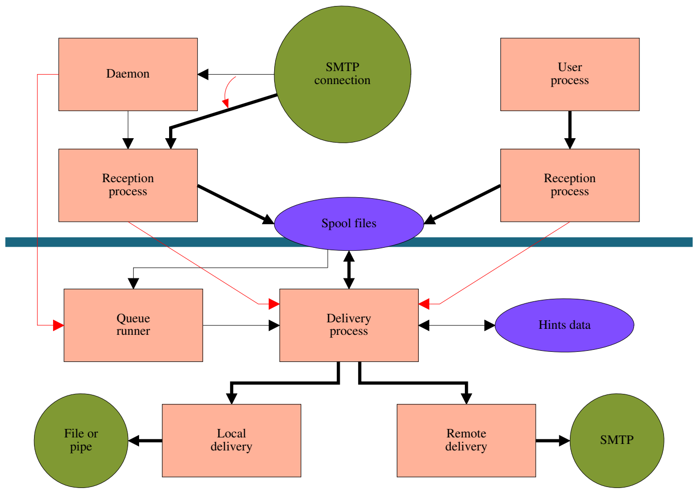
14. Saving and restoring the environment **35**

15. Changing environment parameters **36**

16. List of commands **38**

1. Introduction to Aspic

Aspic is a program that converts a textual description of a line drawing into instructions that can be processed by standard software in order to draw the picture. Usually this happens when the picture is included in some other document. This method of defining line art graphics is the same as that of the PIC system, described by Kernighan in *Software – Practice and Experience*, **12**, pages 1–21 (1982), though the details of the Aspic commands are quite different. Here is an example of the kind of picture Aspic can produce:



Aspic is written in C. It was originally written in the early 1990s, but the code has recently (2022) been tidied up and made more robust. It is developed on Linux and is supplied with a conventional *configure* command, but as it contains no system-specific code, it should be possible to run it in any environment that has a standard C compiler and library.

Aspic reads Unicode input in UTF-8 format. The default output format is Encapsulated PostScript (EPS), but there is also support for output in Scalable Vector Graphics (SVG) format. Both these formats are widely supported by browsers and text-processing software. Files generated by Aspic can be displayed on their own by commands such as *gv* (eps), *evince* (eps), *feh* (svg), *qiv* (svg) and the *display* command that is part of the ImageMagick suite (both formats).

Aspic supports the inclusion of text in drawings, but it does no text processing of its own. The implementation of operations such as text justification and centering is left to the back-end rendering processor. Aspic uses the font size to guess how much vertical space to leave between lines; this can be increased by the user if necessary. Character encoding is discussed in chapter 12.

1.1 The aspic command

The command to run Aspic is as follows:

```
aspic [options] [input [output]]
```

If no input or output is given, Aspic reads from the standard input and writes to the standard output. If an input file name is given without an output, an output file name is created by removing the input's extension (if any), and adding `.eps` or `.svg` as appropriate. The standard input or output may be explicitly referenced by specifying a single hyphen character. The options are as follows:

-help causes Aspic to display usage information on the standard output, and then exit.

-nv disables the use of Aspic variables. This means that dollar characters in the input file are no longer treated specially. The option is useful when there are dollar characters in an Aspic source that does not make use of Aspic variables.

-eps or **-ps** (the default) causes Aspic to generate Encapsulated PostScript.

-svg causes Aspic to generate Scalable Vector Graphics (SVG).

-tr causes Aspic to translate certain input characters; for example, a grave accent is translated into a typographic opening quote. Details are given in section 12.4.

-v or **--version** causes Aspic to display its version number on the standard output, and then exit.

There is also a **-testing** option that is used in testing to suppress the Aspic version number in the output, but is not intended for general use.

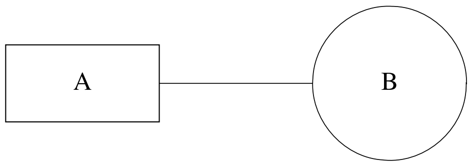
Error messages are written to the standard error stream. No output is generated if any errors are found in the input. A few errors are sufficiently serious that Aspic stops immediately, but for many it carries on and so may output more than one error message. The return code is one of the C values EXIT_SUCCESS or EXIT_FAILURE, which often equate to zero and one.

2. Simple Aspic examples

This chapter uses some simple examples to introduce the various facilities that are available in Aspic. Subsequent chapters contain reference material that explains things in more detail.

Aspic operates in a traditional Cartesian coordinate system, with the positive directions to the right and upwards. For PostScript output the units of length used in Aspic commands are printers' points. There are 72 points to an inch. SVG output contains the same dimensions without specifying a unit. The interpretation is left to the rendering software. On a screen, SVG dimensions are likely to be treated as numbers of pixels. The examples below all show Aspic source, followed by the resulting picture. We start with a simple diagram:

```
  box "A"; line; circle "B";
```



Each Aspic command is terminated by a semicolon, so there are three commands in this example:

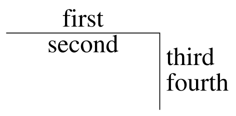
(1) The **box** command causes a box to be drawn, containing the text at its centre. The default box size is 72 by 36 (that is, 1" × 0.5" for PostScript output). There are commands to change the default – see chapter 15 – and the size of an individual box can of course be specified (see below). There is also a **magnify** command that affects the sizes of all shapes (but not the size of any text). In this document, all the examples are reduced by a factor of 0.8, so the size of the box above is actually 0.8" × 0.4".

(2) The **line** command draws a straight line; as nothing else is specified, the line is drawn in the current direction of motion, which defaults to the right. The length is the standard horizontal line length, which defaults to 72.

(3) The **circle** command draws a circle; as nothing was specified as to how it should join onto its predecessor, the 'obvious' joining position is chosen. The circle is drawn at a standard size. (Again, there are commands for changing this.) Aspic can draw ellipses as well as circles.

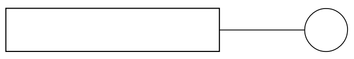
Each Aspic command that causes a shape to be drawn may have any number of text strings associated with it. In the above example, the box and the circle each have one associated string. For closed shapes such as these, the strings are centred in the shape. For horizontal lines, the strings are centred above and below the line, while for other kinds of line they are positioned near the middle of the line. For example:

```
  line "first" "second"; line down "third" "fourth";
```



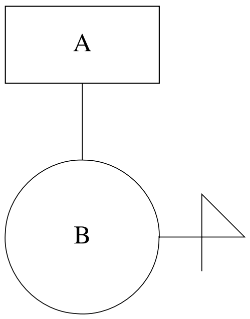
Strings may use different fonts, be of different sizes, and can be coloured. There is also support for rotated strings. For details, see chapter 8. The commands introduced above may be used with options that change the size of the shape that is drawn. For example:

```
  box width 100 depth 20; line right 40; circle radius 10;
```



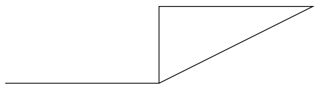
The current direction of motion can be changed by the commands **up**, **down**, **left**, and **right**. In addition, an individual line may be drawn in any direction and of any length (without changing the defaults) by means of appropriate options:

```
  down; box "A"; line; circle "B";
  line right 40; line left 20 up 20; line;
```



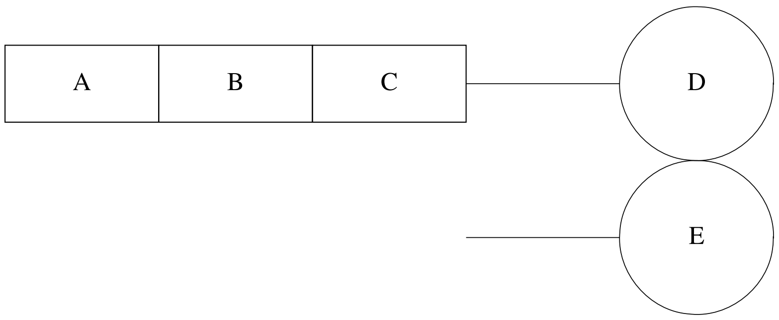
The length values for lines are interpreted as distances in the Cartesian coordinate directions rather than the actual length of the line drawn. There are separate standard values for the horizontal and vertical lengths, which are 72 and 36, respectively, by default.

```
  line; line up right; line left; line down;
```



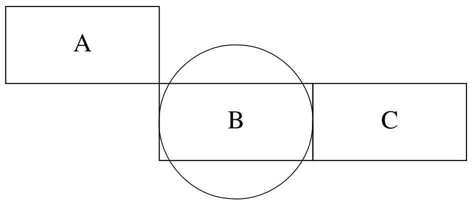
If a sequence of closed shapes occurs, the shapes are joined together according to the current direction of motion, but a closed shape following a line joins according to the direction of the line.

```
  box "A"; box "B"; box "C"; line;
  down; circle "D"; circle "E"; line left;
```



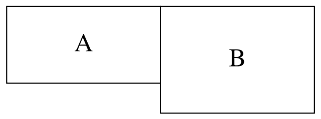
There is a **join** option for specifying where a closed shape joins its predecessor:

```
  box "A"; box join top left "B";
  circle join centre; box join left "C";
```



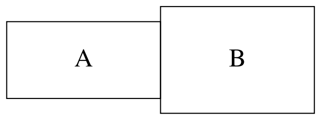
The argument for **join** specifies a point on the new shape that is to be joined to a point on the previous shape. Thus, in the above example, the top left-hand corner of the second box is the point which is joined to the first box. The joining point on the previous shape is the complementary position by default, but it can be also specified explicitly. For example:

```
  box "A"; box depth 50 join top left to top right "B";
```



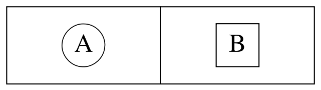
There are nine possible joining points – the four corners, the midpoints of the four edges, and the centre point. The midpoints of the edges are identified by the unqualified names of the edges. Thus, if the only joining information is an edge name, two boxes are joined with the midpoints of the edges aligned:

```
box "A"; box depth 50 join left "B";
```



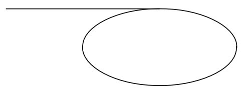
Joins can refer to items other than their immediate predecessors by the use of *labels*. For example:

```
BOXA: box "A";
BOXB: box "B";
      circle radius 10 join centre to centre of BOXA;
      box width 20 depth 20 join centre to centre of BOXB;
```



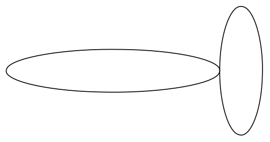
The **join** option may also be used for specifying how a closed shape joins onto a line:

```
line; ellipse join top;
```



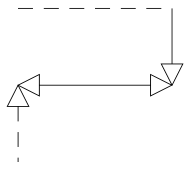
When a circle or an ellipse is being joined, the ‘corner’ joining points refer to points on the circumference, not the corners of the bounding box. The size of an ellipse is specified as a width and a depth, which determine the lengths of the horizontal and vertical axes. For example:

```
ellipse width 100 depth 20;
ellipse width 20 depth 60;
```



Lines may be drawn dashed, and, by using the **arrow** command, with arrowheads on one or both ends.

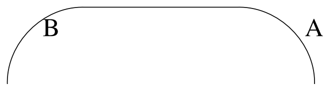
```
line dashed; arrow down; arrow left both;
arrow down back dashed;
```



The dash parameters (lengths of lines and gaps) are adjustable (see chapter 15). By default, **arrow** requests an arrowhead on the end of the line. The option **both** gives arrowheads on both ends, whereas **back** gives a backward-pointing arrowhead only.

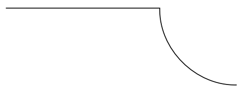
Circular arcs are another form of non-closed shape that Aspic supports. By default an arc is drawn in an anti-clockwise direction, for 90 degrees, and at a standard radius (default 36). If an arc follows a line or another arc, it continues in the same direction by default. If the very first shape is an arc, its initial direction is upwards.

```
arc "A"; line left; arc "B";
```



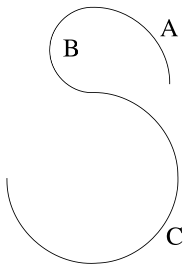
The options **up**, **down**, **left**, and **right** can be used to specify a different initial direction for the arc:

```
line; arc down;
```



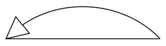
The angle subtended, the radius, and clockwise drawing can be specified:

```
arc "A"; arc angle 180 radius 20 "B";
arc clockwise angle 270 radius 40 "C";
```



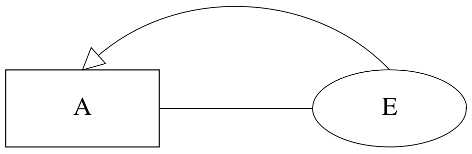
Arcs can be drawn from and to particular points, and, by using the **arcarrow** command, with arrowheads.

```
line; arcarrow from end to start;
```



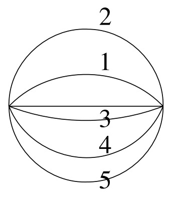
In this example, the positions ‘end’ and ‘start’ are taken to refer to the last-drawn shape. To refer to other shapes, labels are used:

```
BOXA: box "A"; line; ellipse "E";
arcarrow from top to top of BOXA;
```



When an arc of this type is drawn, *either* the radius *or* the angle subtended at the centre of the arc *or* the ‘depth’ of the arc *or* a point through which the arc is to pass may be specified, but only one of these. The ‘depth’ of an arc is the length of the line from the middle of the arc to the middle of the line joining the endpoints. If none of the above parameters is specified, a subtended angle of 90 degrees is used. Here is an example that shows the different ways of specifying arcs:

```
AA: line;
arc from end to start "1";
arc from end of AA to start of AA angle 180 "2";
arc clockwise from end of AA to start of AA radius 100 "3";
BB: arc to start depth 24 "4";
arc clockwise to start via middle of BB plus (10,-10) "5";
```

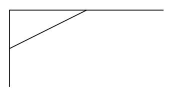


The fifth arc in this example passes through a point that is defined as

```
middle of BB plus (10,-10)
```

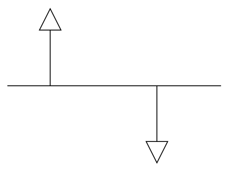
The ‘middle of BB’ is halfway along the fourth arc; the **plus** qualifier applies a relative offset that is 10 units to the right and 10 down from this midpoint. The starting and ending points of straight lines can also be specified explicitly; if both are specified, that determines the length of the line.

```
AA: line up; BB: line right;
line from middle of AA to middle of BB;
```



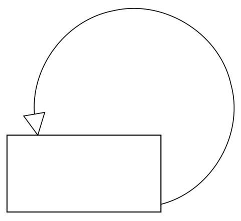
Positions on a line or arc may be specified as **start**, **middle**, or **end** (the word **centre** is reserved for the centre of a closed shape, or for the centre point of a circular arc). More precise positioning can be achieved by specifying a fraction of the way along the line from a named position:

```
AA: line right 100;
arrow up from 0.2 start of AA;
arrow down from 0.3 end of AA;
```



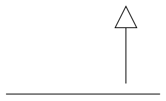
The upward arrow starts at a point that is 0.2 of the way along the line from the start, and the downward arrow starts at a point that is 0.3 of the way along the line from the end. This feature also applies to the edges of boxes:

```
AA: box;
arcarrow from 0.1 right of AA to 0.2 top of AA angle 270;
```



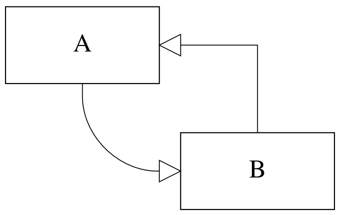
Positions along the top and bottom are measured from the left; positions on the sides are measured from the bottom. If these options are used with a circle or an ellipse, the positions used are those of the circumscribing box. Any position can be further modified by an explicit distance, specified as a vector enclosed in parentheses following **plus**. In the next example, the arrow starts at a position 20 to the right of the middle of the line, and 5 above it:

```
line; arrow up from middle plus (20,5);
```



There is one further positioning feature that is useful for horizontal or vertical lines. It allows the end of such a line to be aligned with a given point, which is often the easiest way to describe certain kinds of drawing:

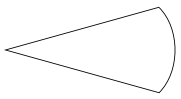
```
BOXA: box "A";
line down 5; arc; arrow right 10; box "B";
line up align centre of BOXA;
arrow to right of BOXA;
```



The **align** option is used in place of **to**; it defines a position, but only one of its coordinates is used as a coordinate of the end of the line. In the example above, a vertical line was specified, and so only the vertical coordinate of ‘`centre of BOXA`’ was used.

As well as circular arcs, Aspic supports cubic Bèzier curves. An end point must be supplied for the **curve** command; the start is deduced from the previous shape if not specified. Without any further parameters, the shape is almost the same as a circular arc. Like circular arcs, the default is to draw the curve anti-clockwise.

```
A: line down 20 left; line down 20 right; curve to start of A;
```

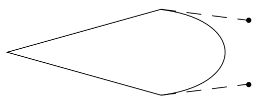


The shape of a cubic Bèzier curve is determined by two control points. The curve leaves the start in the direction of the first one, and approaches the end from the direction of the second, but passes through neither of them. Aspic creates two default control points as follows: in a rotated coordinate system where the line joining the end points is the x-axis, the first control point is a quarter of the way along the line and an equal distance from it. The other control point is in a matching position at the other end of the line.

The default control points give an almost circular arc, but the **c1** and **c2** options can be used to move the control points. Their argument is a pair of parenthesized dimensions. The first moves parallel to the joining line, with a positive value moving further from the end point. The second number moves away from the line if positive, or nearer if negative. As a convenience, the **cs** option can be used to set both to the same values, thus preserving the symmetry of the curve. This example shows the positions of the control points:

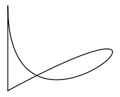
```
A: line down 20 left; B: line down 20 right;
curve to start of A cs (-5,30);
line dashed down 5 right 40 from start of A;
circle filled 0 radius 1;
line dashed up 5 right 40 from end of B;
```

```
circle filled 0 radius 1;
```



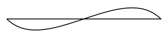
If **c1** or **c2** is used with **cs**, the result is cumulative.

```
A: line down 40; curve to start cs (50,0) c1 (0,100) c2 (25,-10);
```



If the **wavy** option is given, the default position of the second control point is moved to the opposite side of the curve.

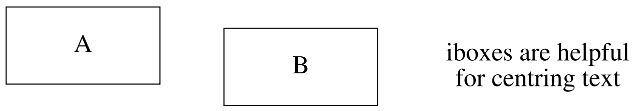
```
line; curve wavy to start;
```



Arrowheads are not supported for Bèzier curves. Because their shape can be so varied, the default position for any text is the middle of the line joining the end points.

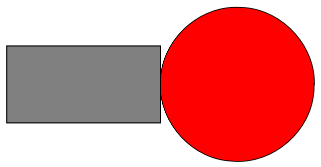
Aspic pictures are best specified as a sequence of shapes whose positions are inter-related. This makes the pictures easy to adjust as they are being created, and also easy to change subsequently. However, many pictures contain shapes that are not connected to other shapes in the picture. Aspic does allow absolute positioning for shapes, but it is often more useful to position these shapes in relation to the others. This can be done using *invisible* lines, boxes, arcs, and curves. If you use **iline**, **ibox**, **iarc**, or **icurve** instead of **line**, **box**, **arc** or **curve**, the relevant lines are not drawn. There are also **icircle** and **iellipse** commands.

```
box "A"; iline right 30 down 10; box "B";
ibox width 150 "iboxes are helpful" "for centring text";
```



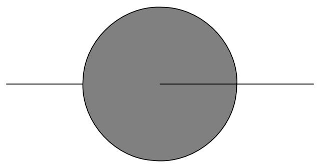
The shapes in the examples shown so far have all been just outlines, but Aspic also contains facilities for causing closed shapes to be filled with colour or shaded with grey. The commands that define closed shapes (**box**, **circle**, and **ellipse**) can be specified with a **filled** option. If it is followed by one number, that specifies a grey level. Otherwise, it must be followed by three numbers that specify a colour in terms of red, green, and blue levels. The numbers are separated by commas and/or spaces. In all cases, the numbers lie between 0 and 1. For example:

```
box filled 0.5; circle filled 1 0 0;
```



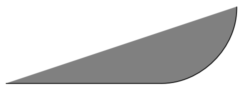
In this example, visible closed shapes are used, so their outlines are drawn. If an invisible shape is filled, no outline is drawn. Filling a shape obliterates items that are ‘beneath’ it. To make it easy to specify which filled shapes are ‘above’ others, there is a **level** option that can be used on any drawing command. The default level is zero; items with a higher value are drawn ‘above’ (later), whereas items with a lower (negative) level are drawn 'below' (earlier). The order in which the items are defined does not matter. For example:

```
A: circle filled 0.5;
line right from centre of A level 1;
line left from centre of A level -1;
```



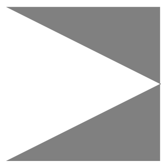
Arbitrary shapes can be filled by specifying the same **shapefilled** option on a sequence of lines and/or arcs/curves that are all at the same level and have the same thickness, colour, and dash characteristics. The end of the shape is marked by an item with different characteristics (or the end of the input). The shape is automatically closed, if necessary, by an invisible straight edge from the endpoint to the startpoint. For example:

```
line shapefilled 0.5; arc shapefilled 0.5;
```



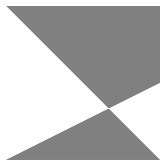
Sometimes it is necessary to supply a dummy item to terminate one shape when another with the same characteristics follows immediately afterwards. A line of zero length can be used for this. For example:

```
iline right shapefilled 0.5;
iline down shapefilled 0.5;
line left 0;
iline down left shapefilled 0.5;
iline right shapefilled 0.5;
```



Without the dummy, zero-length line, the result is:

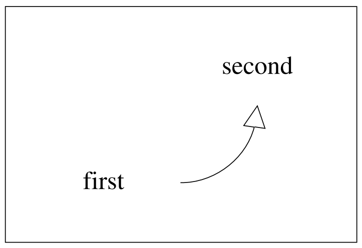
```
iline right shapefilled 0.5;
iline down shapefilled 0.5;
iline down left shapefilled 0.5;
iline right shapefilled 0.5;
```



When a long sequence of commands all have the same **shapefilled** value, you can save typing by using the **shapefill** command to set a default (see chapter 15).

Aspic can be requested to draw a frame round the picture, by means of the **boundingbox** command. A filled frame (see section 13.1) can be used to provide a background colour. The **boundingbox** command is followed by a dimension that specifies the space to be left between the bounding box and the frame. In this example the bounding box of the picture is determined by the invisible boxes that contain the text:

```
  boundingbox 10; ibox "first"; arcarrow; ibox "second";
```



This chapter has introduced many, but not all, of the features of Aspic. The following chapters specify the form of the input more rigorously, and list each command, together with its options.

3. General operation of Aspic

Aspic processes its input in order, interpreting commands that are instructions for moving about on the plane and causing shapes to be drawn and text to be output. There are many parameters for controlling the size and style of the shapes that are drawn; all of them have defaults, and most of these can be altered. Aspic builds up data structures in main memory that represent the final image. If it reaches the end of the input without finding any errors, it outputs a description of the picture in the appropriate output language.

For PostScript output, the units of length used by Aspic are printers' *points*, of which there are 72 to an inch. For SVG output, the units are interpreted by the SVG processor, and on screen displays, they are often taken as pixels unless a specific size is defined.

Aspic distinguishes between closed and open shapes. The closed shapes are boxes, circles, and ellipses, and the open shapes are lines, circular arcs, and Bézier curves. There are default sizes for everything, and text strings may be associated with each shape. Unless explicitly positioned, each shape is placed adjacent to its predecessor, in many cases taking note of the *current direction*, whose default is to the right. For example, the sequence:

```
  box; arrow; box;
```

places the three items in a horizontal row. There are commands to change the current direction, and, for the drawing commands, options to override it for individual items.

Only very simple pictures can be drawn as a series of shapes in which each shape is positioned relative to its predecessor. Aspic allows shapes to be labelled so that branches in the sequence of shapes may be constructed, and cross-references between different parts of the picture may be expressed. The previous chapter contains several examples.

3.1 Position of the coordinate origin

You can specify absolute positions on the drawing plane, but it is better to describe a picture in terms of relative positions between the shapes that comprise it, because such a description is much easier to adjust while you are creating the picture. If the first item in a picture is specified without an absolute position – this is normally the case – it is positioned as follows:

- A closed shape is placed with its centre at the origin.
- A circular arc is placed with the centre of the arc at the origin.
- A straight line or Bèzier curve starts at the origin.

However, for most pictures, it is not necessary to worry about absolute coordinates or the position of the origin.

3.2 The bounding box

Aspic computes a bounding box for the entire picture, and arranges that the bottom left of the bounding box is positioned at the bottom left of the picture's output. This means that the origin is not necessarily at the bottom left of the final picture. The coordinates of the bounding box are included in the output file and are used by programs that process it to determine the size of the image.

Invisible items that are not part of the boundary of a filled shape, and which have no associated text, are ignored when Aspic is computing the bounding box. The idea is that such items are assumed to be used for positioning purposes only. Occasionally you may want an invisible item to be included in the bounding box calculation. You can do this by providing it with an empty text string.

Because Aspic does not process text strings itself, it can only guess the size of a string when computing the bounding box. This matters only when a string extends beyond the box defined by the graphic shapes. A string's width is guessed as one half the font size times the number of characters in the string.

4. Aspic input

Aspic input consists of a sequence of commands, each of which must be terminated by a semicolon. Newlines and other white space may appear between the components of a command in the usual way. The maximum length of an input line is 255 bytes. If a sharp (or 'hash') character (#) is encountered when a command is expected, the remainder of the input line is ignored. This provides a facility for including comments in Aspic input. Each input line is processed for variable substitutions before any other processing takes place (see chapter 5 below). This action can be disabled by the **-nv** command line option.

4.1 Command format

An Aspic command consists of four components:

```
 label    command          options                strings
  A:        box    dashed width 100    "first" "second";
```

The case of letters is significant in all the components. Only the command name is mandatory.

- Commands that draw shapes may start with one or more labels, each terminated by a colon. A label consists of a sequence of letters and digits, starting with a letter. Upper case letters are commonly used in labels as it makes them stand out. Other commands may not be labelled. Labels must be unique. For commands inside macros, the special sequence `&$` can be used to ensure uniqueness (see 6).

- Many commands have option specifications that follow the command name. Each option consists of a keyword, possibly followed by one or more data items. The options may appear in any order, and except in a few special cases are all optional.

- Following the options, on commands that define lines or closed shapes, and on the **text** command, there may be any number of text strings, each enclosed in double quotes. The double-quote character itself may be included by doubling. There are some options that can follow a text string; these are described in chapter 8. The strings specify text that is to be output at an appropriate position, relative to the item that is drawn for all but the **text** command, which has special positioning arrangements. Details of text positioning are given below with each command. Strings may not extend over line boundaries in the input.

- Strings are interpreted as a sequence of Unicode characters. The inclusion of characters by name and by number is supported. Details of how the sequence of input bytes is decoded are given in section 12.4. Aspic does no typographic processing of strings. This means that any string-specific processing, such as measuring the string in order to centre it, has to take place in the backend processor. In PostScript output, PostScript operators are used to do this. In SVG output, an appropriate setting of the **text-anchor** attribute is generated.

A number of command words and option names use the British English spellings 'centre', 'grey' and 'colour'. In all cases the American spellings 'center', 'gray' and 'color' are also accepted.

4.2 File inclusion

The **include** command can be used to insert the contents of a given file into the sequence of Aspic commands. This can be used, for example, to include a standard header file (which might define fonts or give names to colours) in a number of different pictures. The command name is followed by a file name, which is not quoted. For example:

```
include /home/me/MyAspicHeader;
```

The **include** command may not appear inside a macro (see chapter 6), but included files may contain further inclusions.

5. Aspic variables

Aspic supports simple variables, which can be used to save repetition in the input. This feature can, however, be disabled by use of the **-nv** command line option. If you are not using Aspic variables, but are making use of dollar ($) characters in strings, you should use **-nv**, because otherwise the dollars will be misinterpreted by Aspic.

When variables are not disabled, a dollar character in an input line introduces a variable substitution. There is, however, one exception: the special sequence `&$` that is used in Aspic macros – see chapter 6. In all other cases, a dollar character must either be followed by another dollar (indicating a single literal dollar character), or be followed by a variable name, optionally enclosed in braces (curly brackets). Variable names start with a letter and contain letters and digits. Braces are required if the character that follows the variable name is a letter or a digit.

When each input line is read, the values of any variables that are mentioned are substituted before any other processing takes place. A variable must be defined before it is used. The contents of a macro (see chapter 6) are processed for variable substitutions when the macro is defined, not when it is called.

Variables are given values by means of the **set** command, which is followed by a variable name (without a dollar) and a string value. The value of a variable can be changed as many times as you like during the course of a picture description. For example, this command defines the variable **red** to contain the three colour values for the colour red:

```
  set red "1,0,0";
```

Later in the input file, the variable could be used like this:

```
  box filled $red;
```

At the start of an Aspic run, the variable **$date** is initialized to contain the date and time, and the variables **$creator** and **$title** are each set to the string ‘Unknown’. These three variables are used to create comments at the start of the output, but otherwise they are treated like any other variable, and you can change them as required. For example, you might like to set **$date** to the date on which the picture was defined.

6. Aspic macros

To save a lot of command repetition, Aspic contains a simple macro facility that allows you to define compound commands. A macro is defined by the command **macro**, which is followed by a name and a macro body. The name must not be the same as the name of any inbuilt command. If it is the same as a previously defined macro, the new macro overrides. The body consists of *either* all the following text up to the first non-quoted semicolon, *or*, if the first character after the name is an opening brace, all the text up to the next non-quoted closing brace, which must be followed by a semicolon to terminate the **macro** command. For example:

```
  macro bigbox box width 100 depth 100;
  macro box2 { box; box; };
```

In the first example, the terminating semicolon is not part of the macro body. A macro definition may extend over more than one line. Variables are substituted into the body of a macro when it is defined; there is no re-substitution when the macro is called. If you need such a facility, it can be obtained by passing variables as arguments to macro calls.

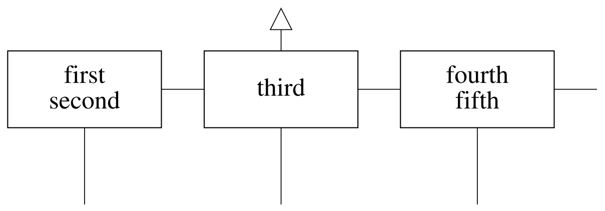
Macros are called by using their names as command names. A macro body may contain calls to other macros, but recursion is not allowed. Macros can be called with arguments, which are treated as character strings. White space is used to delimit macro arguments, unless they are enclosed in either single or double quotes. If double quotes are used, they are retained when the contents of an argument are substituted into the macro body.

Macro arguments are referenced in the macro body by items of the form `&1`, `&2`, etc. These references are replaced by the actual argument values each time the macro is called. If the character `&` is required for another purpose in a macro body, it must be doubled. If the special string `&$` appears in a macro body, it is replaced throughout by a sequence number that is incremented for each macro call. This can be used to generate unique labels for shapes that are drawn in macro bodies. A nested macro call will have a different value for `&$`.

The following example starts with the definition of a macro that draws a box containing text given as one or two arguments, with two lines attached to it. This macro is then used to generate an array of boxes. Because Aspic allows multiple labels on shapes, these compound items can themselves be labelled, as shown in this example:

```
  macro item {
    B&$: box &1 &2; line down;
    line right 20 from right of B&$;
    };

  item "first" "second";
  MID: item "third";
  item "fourth" "fifth";
  arrow up 20 from MID;
```



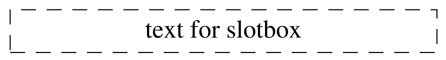
Note the use of `&$` to generate a unique label within the macro. When an macro call is labelled, the first command inside the macro must be a command for which a label is allowed, that is, it must draw something.

Because the macro in the example above was not defined with double quotes surrounding the argument references, double quotes had to be used when calling it in order to supply strings to the **box** command. If the quotes had been present in the definition, they could not be have been used in the calls, but single quotes could have been used if the arguments contained spaces.

If an argument that has not been supplied is referenced, nothing is substituted; thus the second call of **item** above expands into a call to **box** with only a single string argument. If too many arguments are supplied, the surplus ones are left in the input following the substituted text.

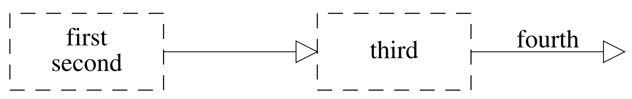
It is not necessary to include a semicolon before a terminating brace when defining a macro. If the semicolon is present, it is included in the replacement text when the macro is called. Sometimes it is useful to be able to set up a macro that generates part of a command, so that additional options can be added on each call. This can be done by omitting the terminating semicolon. In this example, any text following the macro name is added to the command:

```
macro slotbox { box width 200 depth 20 };
slotbox dashed "text for slotbox";
```



When such additional text is required, and also not all the arguments of a macro are to be supplied, the vertical bar character can be used to mark the end of the arguments. For example:

```
macro dashbox { box dashed &1 &2; arrow };
dashbox "first" "second";
dashbox "third" | "fourth";
```



In the first call, the two strings are taken as arguments of the macro; in the second call, the second string is added onto the end of the replacement text, and therefore goes with the **arrow** command.

7. Types of value used in commands

Unless explicitly stated to be an integer, a number may always be specified with an optional decimal point and fractional part. Negative numbers are preceded by a minus sign. Non-integer numbers are held in a fixed-point format to three decimal places. In the descriptions of the commands that follow, the following types of value are used:

7.1 <angle>

A non-negative number, specifying an angle in degrees.

7.2 <boxpoint>

One of the phrases *top, bottom, left, right, centre, bottom left, bottom right, top left*, or *top right*. The first four refer to the midpoints of the respective sides of a box; the last four refer to the corners.

7.3 <colour>

Three numbers in the range 0.0 to 1.0, separated by spaces and/or commas. They specify a colour by the amount of red, green, and blue, respectively, with 0.0 meaning none and 1.0 the maximum amount of each component.

7.4 <greylevel>

A number in the range 0.0 to 1.0, where 0.0 is black and 1.0 is white.

7.5 <integer>

A positive or negative integer.

7.6 <label>

A label that identifies an existing drawing item, that is, one whose definition falls earlier in the input file.

7.7 <length>

A non-negative number, specifying a length.

7.8 <position>

A <*position*> identifies a point in the plane. It is either an absolute position, specified as a pair of x-y coordinates, separated by a comma and enclosed in parentheses, or a relative position, specified as follows:

[<*fraction*>] <*point*> `[of` <*label*>`]` `[plus` <*vector*>`]`

where all but <*point*> are optional. The position is computed relative to the shape whose <*label*> is given, or if no <*label*> is mentioned, relative to the previous drawn shape.

- If the referenced shape is a box, circle, or ellipse, <*point*> must be a <*boxpoint*>. The 'corner' points of a circle or ellipse are the intersections of the shape with the diagonals of the bounding box.
- If the referenced shape is an arc, curve, or line, <*point*> is one of the words *start, end*, or *middle*. These refer to positions along the arc, curve, or line. In addition, for an arc, *centre*, meaning the centre of the circle of which the arc is part, may be specified.
- If there is no referenced shape, this form of <*position*> causes an error.

<*fraction*> is a number between 0.0 and 1.0, specified either as a decimal fraction (for example, 0.5) or as two numbers (usually, but not necessarily integers) separated by a slash (for example, 1/3). It specifies a position part-way along a line, arc, or curve. If <*fraction*> is present, there are some additional constraints on <*point*>:

- If the referenced object is closed, then <*point*> must be one of *top, bottom, left*, or *right*, and the line to which <*fraction*> refers is the appropriate side of the bounding box of the object. The fraction is measured from the left of horizontal lines, and from the bottom of vertical lines.

- If the referenced object is not closed, then <*point*> must be *start* or *end* – the fraction is taken from that end of the line, arc, or curve.

In effect, the presence of <*fraction*> changes the meaning of *top, bottom, left*, or *right* as a <*point*>. With no <*fraction*>, these words refer to the midpoints of the respective sides of the bounding box; when <*fraction*> is present, they refer to the sides themselves.

The final optional component of a <*position*> is the word `plus` followed by a <*vector*>, which is two numbers separated by a comma and enclosed in parentheses. It specifies a Cartesian adjustment to the position defined by the remainder of the <*position*>. Here are some examples of <*position*> specifications:

```
(45,67)
top
top plus (10,0)
centre of A
bottom right of B plus (0,-5)
1/3 top of C
middle
end of line1
0.25 start of line3 plus (0,7)
```

8. Drawing objects and text

All the commands that cause something to be drawn and/or text to be output are described in this chapter. Each command is summarised by listing its options and the type of value that must follow each option keyword, where relevant. A value type that is followed by an asterisk is optional, and a vertical bar is used to separate alternative value types. Some combinations of options are mutually exclusive, and these are noted in the description of each command below.

When a command does not specify an option, a default value is used. There are separate sets of defaults for boxes, circles, ellipses, and lines (both straight and curved). Many of these defaults can be changed by the commands that are described in chapter 15 (*Changing environment parameters*).

The **thickness**, **dashed**, **colour**, and **grey** options are common to all these commands except **text**. The first specifies the thickness of lines that are drawn. If **dashed** is specified, lines are drawn dashed, with the lengths of dashes and spaces controlled by the relevent setting in the current environment (see chapter 15). The colour of lines is specified either by **colour**, which must be followed by three numbers (for red, blue, and green components), or by **grey**, which must be followed by a single number. This is a shorthand for **colour** followed by three identical numbers. The obsolete synonym **greyness** is also supported. Colour numbers lie in the range 0.0. to 1.0 inclusive. They specify the amount of colour to be used. In the case of **grey**, a value of 0.0 is black and 1.0 is white. If no colour option is present, default values are used.

The **level** option is common to all these commands (including **text**). It is useful when filled shapes are being drawn, because filling a shape obliterates items that are drawn 'below' it. The default level is zero; items on levels greater than zero are drawn 'above' and items with levels less than zero are drawn 'below'. The default can be changed by the **level** command (see chapter 15).

Many of the commands also have **filled** and **shapefilled** options. For closed shapes (**box**, **circle**, **ellipse**), **filled** specifies a colour with which to fill the shape. It can be followed either by three numbers to specify red, green, and blue components, or by a single number, to specify a grey level. For lines or arcs that start or end with arrowheads, **filled** specifies the colour with which the arrow-head is filled.

The **shapefilled** option also takes either one or three numbers as its argument. It is available on commands for drawing lines, arcs, and curves. A sequence of such commands with the same **shapefilled** values, and also the same thickness, dash, colour, and level parameters, is interpreted as a closed shape that is to be filled with the appropriate colour. If the lines do not define a closed shape, an invisible straight line from the end to the start bounds the area that is filled. If one such closed shape immediately follows another with the same characteristics, you may need to insert a dummy command without a **shapefilled** option, in order to terminate the first shape. A line of length zero can be used for this.

Options may be specified in any order, and may be followed by any number of strings enclosed in double quotes (with doubling for any double quotes within the string), followed optionally by a <*vector*> (as described above in the definition of <*position*>) and one of `/l` (the letter ell), `/r`, or `/c`. If present, the <*vector*> must come first. For example:

```
"the quick brown fox"
"move this up"(0,20)
"justify right"/r
"move left and centre"(-20,0)/c
```

The presence of a <*vector*> causes the position at which the string is output to be modified by the value of the <*vector*>. The `/l`, `/r`, and `/c` options specify left, right, or centre justification respect-ively, with respect to this position. The default justification depends on the shape with which the string is associated, and is documented below. Other options may also be present, separated by slashes:

- An sequence of digits is interpreted as a font number for this string. A default font, 12-point Times-Roman, is provided, numbered zero, The **bindfont** command can be used to define additional fonts, and the **setfont** command can be used to change the default font. For further details of these commands, see chapter 12, and for details of character codes, see section 12.4.

- A number preceded by a plus or a minus sign is interpreted as an angle of rotation, in degrees. Positive rotation is anticlockwise. The centre of rotation is the defining position for the string, on the baseline. For left-justified strings it is at the left, for centred strings in the middle, and for right-justified strings at the right.

- A single unsigned number that contains a decimal point is interpreted as a grey level for the text. The value must not be greater than 1.0.

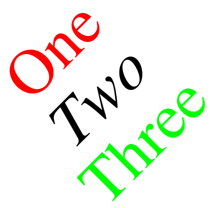
- Three comma-separated numbers are interpreted as the red, green, and blue components of a colour for the text. The values must not be greater than 1.0.

Here are some examples:

```
"the quick (font 2)"/2 "brown fox (font 5)"/5
"move left and centre (font 3)"(-20,0)/c/3
"coloured and rotated"/1,0.5,0.4/+45
```

When a command is followed by more than one string, each successive string is placed ‘below’ the previous one. In the case of a rotated string, ‘below’ takes the rotation into account. Strings that have no rotation specified adopt the rotation of their predecessor.

```
bindfont 1 "Times-Roman" 30;
bindfont 2 "Times-Italic" 30;
text "One"/1,0,0/+45/1 "Two"/2 "Three"/1/0,1,0;
```



8.1 arc

angle	<*angle*>
clockwise	
colour	<*colour*>
dashed	
down	
depth	<*length*>
from	<*position*> | <*label*>
grey	<*greylevel*>
left	
level	<*integer*>
radius	<*length*>
right	
shapefilled	<*colour*> | <*greylevel*>
thickness	<*length*>
to	<*position*>
up	
via	<*position*>

A circular arc is drawn in an anti-clockwise direction from its starting point to its ending point, unless the **clockwise** option is present, in which case the arc is drawn clockwise. The position, size, and orientation of an arc may be specified in one of four different ways:

(1) If neither **from** nor **to** is specified, the arc is positioned according to the previously drawn shape. The **depth** and **via** options may not be given. The user may supply either or both of a radius and

an angle. If no radius is supplied, the default arc radius is used; if no angle is supplied, an arc of 90 degrees is drawn.

The initial direction of the arc can be specified by **up**, **down**, **left**, or **right**. If none of these are present, and the previous item was a line or arc, the new arc starts by continuing in the same direction. When an arc follows a closed shape (box, circle, or ellipse), the current direction is used. The direction determines the position on the closed shape from which the arc starts (**up** starts from the middle of the top of a box, and so on). If the first item in the input is an arc without an explicit direction, it is drawn upwards.

(2) If **from** is supplied without **to**, either or both of a radius and an angle may be supplied. If no radius is supplied, the default arc radius is used; if no angle is supplied, an arc of 90 degrees is drawn. The **depth** and **via** options may not be given. If the initial direction of the arc is not specified, the current direction is used.

(3) If **to** is given without **from**, a starting point is determined from the previous shape, and then the action is as described in the following paragraph. If the previous shape is a line, arc, or curve, its end point is used; otherwise the direction (explicit or implicit) is used to decide on which side of the bounding box of the previous closed shape to place the starting point, but not for any other purpose. The midpoint of the appropriate side is used.

(4) If both **from** and **to** are given (or if **to** is given and **from** is determined from the previous shape as just described), there are four mutually exclusive ways in which the size of the arc can be specified:

(a) The **radius** option can be used to give an explicit radius; this must not be less than half the distance between the end points.

(b) The **angle** option can be used to specify the angle subtended at the centre of the arc.

(c) The **depth** option can be used to specify the distance between the midpoint of the line joining the end points and the midpoint of the arc. If the **depth** option specifies a distance that is more than half the distance between the end points, an arc of more than 180 degrees is drawn.

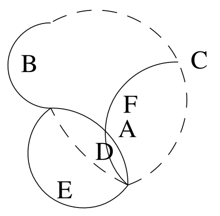
(d) The **via** option can be used to specify a third point through which the arc is to pass. This point must not be be on the line joining the end points, and it must also be on the appropriate side of that line. If it is not suitable, an error message is output.

If none of these options is given, an arc that subtends 90 degrees at its centre is drawn. If more than one of these options is given, an error message is generated.

If the **from** option specifies the label of a closed shape without further qualification, the actual starting point on that shape is determined by the initial direction of the arc.

Texts are output near the midpoint of the arc, and are left-justified by default. The following example illustrates various types of arc:

```
A: arc "A";
B: arc clockwise radius 20 angle 180 "B";
C: arc dashed from start of A to end of B angle 190 "C";
arc clockwise dashed from start of A to end of A radius 75 "D";
arc to start depth 30 "E";
arc clockwise to middle of C via middle of A "F";
```

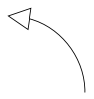


8.2 arcarrow

angle	<*angle*>
back	
both	
clockwise	
colour	<*colour*>
dashed	
depth	<*length*>
down	
filled	<*colour*> | <*greylevel*>
from	<*position*> | <*label*>
grey	<*greylevel*>
left	
level	<*integer*>
radius	<*length*>
right	
shapefilled	<*colour*> | <*greylevel*>
thickness	<*length*>
to	<*position*>
up	
via	<*position*>

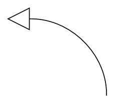
The options for **arcarrow** are exactly as for **arc**, but with the addition of **filled** (specifying a filled arrowhead), **both** (which specifies a double-headed arrow) and **back** (which specifies a backwards-pointing arrow). If neither **both** nor **back** is given, an arrowhead is drawn at the end of the arc. Arrowheads are drawn within the length of the arc so if, for example, a 90-degree arc is drawn from absolute angle zero, the arrow head is not horizontal:

```
  arcarrow;
```



A horizontal arrowhead can be drawn by adding a short linear arrow afterwards, but note that this extends beyond the end of the arc:

```
  arc; arrow left 10;
```



8.3 arrow

align	<*position*>
back	
both	
colour	<*colour*>
dashed	
down	<*length*>*
filled	<*colour*> | <*greylevel*>
from	<*position*> | <*label*>
grey	<*greylevel*>
left	<*length*>*
level	<*integer*>
right	<*length*>*
shapefilled	<*colour*> | <*greylevel*>

thickness	<*length*>
to	<*position*>
up	<*length*>*

The **arrow** command has exactly the same options as **line** (see section 8.14 below), but with the addition of **filled** (specifying a filled arrowhead), **both** (specifying a double-headed arrow), and **back** (specifying a backwards-pointing arrow). If neither **both** nor **back** is given, an arrowhead is drawn at the end of the line.

8.4 box

at	<*position*>
colour	<*colour*>
dashed	
depth	<*length*>
filled	<*colour*> | <*greylevel*>
grey	<*greylevel*>
level	<*integer*>
join	<*boxpoint*> to <*position*> | to <*label*>
thickness	<*length*>
width	<*length*>

This command causes a rectangular closed box to be drawn. The **width** and **depth** options specify the horizontal and vertical dimensions of the box. The **at** option specifies the position of the centre of the box; if not given, the centre point is computed by reference to the previous shape. In the absence of a **join** option, if the previous shape was a line or arc, the midpoint of an appropriate side of the box is joined onto its end. If the previous item was a Bèzier curve, the midpoint of the box is positioned at the end of the curve. If the previous item was a closed shape, the side which is abutted depends on the current direction. If there was no previous shape, the centre is placed at the origin of the coordinate system.

The **join** option specifies how the box is to be joined to a previous shape, and is mutually exclusive with **at**. This option takes three different forms:

(1) If <*position*> is given, the given point on the box is placed at the given position. For example:

```
box join bottom right to centre of C;
box join top left to start plus (10,15);
```

As with all <*positions*>, if ‘of <*label*>’ is omitted, the preceding shape is implied.

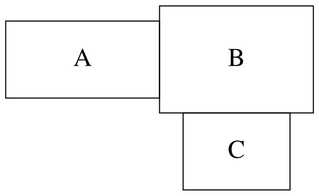
(2) If no <*position*> is given, and the previous shape (or the named shape if ‘to <*label*>’ is present) is a closed shape, the given point is joined to the complementary point of the referenced shape. For example:

```
box join top;
box join top left to A;
```

(3) If no <*position*> is given, and the previous shape (or the named shape if ‘to <*label*>’ is present) is not a closed shape, the given point is joined to its end.

Note that if boxes of different dimensions are joined by naming their edges, the middle points of the edges are made coincident:

```
box "A"; box depth 50 "B";
box width 50 join top "C";
```



Any text items are centred at the centre of the box. Because Aspic does not process the text itself, it cannot tell whether the text will actually fit into the box.

8.5 circle

at	<*position*>
colour	<*colour*>
dashed	
filled	<*colour*> | <*greylevel*>
grey	<*greylevel*>
level	<*integer*>
join	<*boxpoint*> to <*position*> | to <*label*>
radius	<*length*>
thickness	<*length*>

The **at** option specifies the position of the centre of the circle; it is mutually exclusive with the **join** option, which specifies how the circle is to be joined to the previous shape, exactly as for boxes (see above). Text items are centred at the centre of the circle.

8.6 curve

c1	<*vector*>
c2	<*vector*>
clockwise	
colour	<*colour*>
cs	<*vector*>
dashed	
from	<*position*> | <*label*>
grey	<*greylevel*>
level	<*integer*>
shapefilled	<*colour*> | <*greylevel*>
thickness	<*length*>
to	<*position*>
wavy	

This command draws a Bèzier curve; the **to** option is mandatory. If **from** is not given, it is deduced from the previous item. The **c1** and **c2** options define adjustments to the curve's control points. The first number moves parallel to the line joining the endpoints, positive values moving away from the relevant end. The second number moves in a perpendicular direction, positive values moving further away from the joining line. The **cs** option applies to both control points and is cumulative with **c1** and **c2**. The **wavy** option moves the second control point to an equal distance on the opposite side of the base line. Text items are placed at the middle of the base line.

8.7 ellipse

at	<*position*>
colour	<*colour*>
dashed	
depth	<*length*>
filled	<*colour*> | <*greylevel*>
grey	<*greylevel*>
level	<*integer*>
join	<*boxpoint*> to <*position*> | to <*label*>
thickness	<*length*>
width	<*length*>

The options for **ellipse** are the same as for **circle**, except that **radius** is replaced by **width** and **depth**, which specify the lengths of the horizontal and vertical axes. That is, they specify the size of the bounding box.

8.8 iarc

<*as arc*>

The **iarc** command defines an invisible arc. Its options are the same as for the **arc** command. Although the arc is not actually drawn, any text supplied is output, and the invisible arc can form part of a shape that is filled. The arc counts towards the bounding box only if there is text, or if it is part of a filled shape.

8.9 ibox

<*as box*>

The **ibox** command defines an invisible box. Its options are the same as for the **box** command. Although the box is not actually drawn, any text supplied is output, and the shape is filled if **filled** is specified. The box counts towards the bounding box only if there is text, or if it is filled.

8.10 icircle

<*as circle*>

The **icircle** command defines an invisible circle. Its options are the same as for the **circle** command. Although the circle is not actually drawn, any text supplied is output, and the shape is filled if **filled** is specified. The circle counts towards the bounding box only if there is text, or if it is filled.

8.11 icurve

<*as curve*>

The **icurve** command defines an invisible Bèzier curve. Its options are the same as for the **curve** command. Although the curve is not actually drawn, any text supplied is output, and the invisible curve can form part of a shape that is filled. The curve counts towards the bounding box only if there is text, or if it is part of a filled shape.

8.12 iellipse

<*as ellipse*>

The **iellipse** command defines an invisible ellipse. Its options are the same as for the **ellipse** command. Although the ellipse is not actually drawn, any text supplied is output, and the shape is filled if **filled** is specified. The ellipse counts towards the bounding box only if there is text, or if it is filled.

8.13 iline

<*as line*>

The **iline** command defines an invisible line. Its options are the same as for the **line** command. Although the line is not actually drawn, any text supplied is output, and the invisible line can form part of a shape that is filled. The line counts towards the bounding box only if there is text, or if it is part of a filled shape.

8.14 line

align	<*position*>
colour	<*colour*>
dashed	
down	<*length*>*
from	<*position*> | <*label*>
grey	<*greylevel*>
left	<*length*>*
level	<*integer*>

right	<*length*>*
shapefilled	<*colour*> | <*greylevel*>
thickness	<*length*>
to	<*position*>
up	<*length*>*

This command draws a straight line. The start is given by the **from** option; if it specifies a label only, the starting point on the referenced shape is its end point if it is a straight line, arc, or curve. Otherwise the starting point is determined by the direction of this line. For example, a line to the right from a box starts at the midpoint of the right-hand edge. The **to** option can be used to specify the end point by reference to the previous or some other shape. For example:

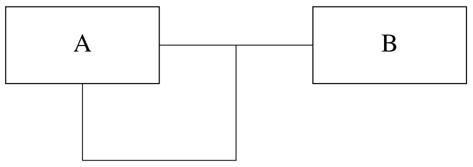
```
  box; line from bottom right to 0.25 top;
```

When the **to** option is not used, a horizontal and/or a vertical distance (that is, Cartesian coordinates relative to the start) can be specified by the options *up*, *down*, *left*, or *right*. If none of these are present, the line is drawn in the current direction. In this case, and also if a direction is given without a length, the current default horizontal or vertical length (as appropriate) is used unless the **align** option applies (see below). Here are some examples:

```
  line;
  line right;
  line up right;
  line down 20 left;
  line down 40 right 60;
```

The **align** option can be used only with horizontal or vertical lines. The vertical or horizontal coordinate of the <*position*> specified by **align** is used, as appropriate, overriding any other value. For example:

```
  A: box "A";
  L: line right;
  B: box "B";
     line down from bottom of A;
     line right align middle of L;
     line up align right of A;
```



As always, a reference to an unqualified box edge implies the midpoint of that edge.

For a horizontal line, text is positioned at the midpoint of the line, centred by default. If there is more than one string, they are positioned both above and below the line. For non-horizontal lines, text is left-justified (by default) close to the midpoint of the line.

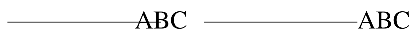
8.15 text

at	<*position*>
level	<*integer*>

This command does nothing unless it ends with at least one optional text string. It provides a means of outputting text without drawing a graphic shape. It may not be labelled, and whatever is set as the previous shape is unchanged. The text is centre-justified by default, both horizontally and vertically. If no **at** option is given, the text is positioned with reference to the previous shape. For a closed shape it is put at the centre of the shape, for a straight line at the midpoint of the line, for a circular arc at the centre of the circle. and for a Bèzier curve at the midpoint of the base line.

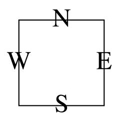
String options can be used to make fine adjustments to the position of the text. For example, if text is placed to the right of a horizontal line, it should normally be specified as left-justified.

```
line; text at end "ABC";
iline right 20;
line; text at end "ABC"/l;
```



In the first of these two examples, the default centre-justification has caused the text to be output on top of the line. The **text** command is often the easiest way of positioning strings that are not near the default string position for a shape.

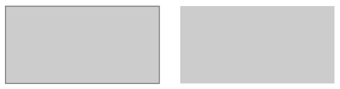
```
box width 40 depth 40; text at top "N"; text at bottom "S";
text at left "W"; text at right "E";
```



9. Filled shapes

Boxes, circles, and ellipses are filled when they have the **filled** option set, or if the **boxfill**, **circlefill**, or **ellipsefill** commands have been used to set filling as a default (see chapter 15). If the shape is invisible, no outline is drawn, and only the filling is shown. Otherwise, the outline is drawn in the line-drawing colour. For example:

```
box grey 0.5 filled 0.8; iline right 10; ibox filled 0.8;
```

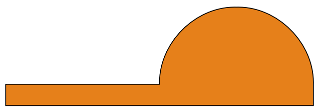


A sequence of lines, arcs, and Bèzier curves with the same **shapefilled** values and the same general characteristics (line width, colour, dash parameters, and level) is turned into a closed outline (if necessary) and filled. The lines and arcs themselves are drawn as normal, unless they are invisible. The automatically supplied line that closes such a shape is not drawn. A dummy command such as:

```
line left 0;
```

is sometimes needed to separate two successive shapes that have the same characteristics. You can set a **shapefilled** value as a default, to save having to repeat it for all the different constituents of a shape. The ability to save and restore the environment (see chapter 14) can be helpful here:

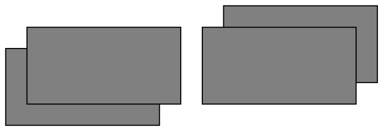
```
push; shapefill 0.9 0.5 0.1;
arc; arc; line left; line down 10;
line right; line right; line up 10;
pop;
```



Arrowheads on lines and arcs are filled if the **filled** option is set on the appropriate drawing command. Alternatively, the **arrowfill** command can be used to specify a default filling colour (see chapter 14).

The concept of 'levels' is important when filled shapes are being drawn, because filling a shape obliterates anything that is underneath it, even if the filling colour is lighter than what was there before. It is like using opaque paint. By specifying different levels for different components of a drawing, you can control the order in which they are output, and therefore which parts are obliterated by other parts. The default level is zero; items on higher levels are output later ('above'), whereas items on lower levels are output earlier ('below'). Items at the same level are output in the order in which they are defined. Consider this example:

```
box filled 0.5; box at centre plus (10,10) filled 0.5;
iline right 10;
box filled 0.5; box at centre plus (10,10) filled 0.5 level -1;
```



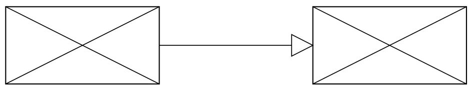
The second box is 'above' the first box, but because of the **level** specification, the fourth box is 'below' the third box. A default level can be set by means of the **level** command (see chapter 15).

10. Altering the ‘last’ item

Occasionally it is useful to be able to change which shape is considered to be the previous shape when the next shape is drawn. The **goto** command, which is followed by a label, is used to do this. It can be useful in macros. Consider a macro that draws a box with diagonals. When this macro is called, you may want subsequent shapes to be positioned relative to the box, not to the diagonal lines. This example shows how this can be done:

```
macro diag {
A&$: box;
line from top right to bottom left;
line from top left of A&$ to bottom right of A&$;
goto A&$; };

diag; arrow; diag;
```



The final **goto** in the macro causes any subsequent shapes to be positioned with reference to the box. Without the final **goto** in the macro, the ‘last item’ after a macro call would be the second diagonal line.

If **goto** is followed by an asterisk instead of a label, it unsets the last item. Aspic then treats the next item as if it were the first. This feature is mostly useful for testing.

11. Changing the current direction

The current direction defaults to ‘right’. It can be changed by the commands **up**, **down**, **left**, and **right**, which have no arguments. The current direction is used in the following circumstances:

(1) When drawing a line, if no direction is specified;

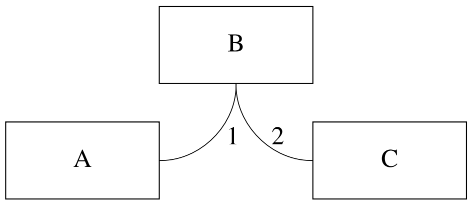
(2) When drawing an arc, if no direction is specified and a start point is given without an end point;

(3) When drawing an arc after a closed shape, if neither a direction nor a start point is given.

(4) When one closed shape follows another, to determine their relative positions if no **join** option is given;

Note that the current direction is *not* used when a closed shape follows a line or arc; the position of the closed shape in this case is determined by the direction of the end of the line or arc. For example,

```
right; box "A"; arc "1"; box "B";
down; arc "2"; box "C";
```



The initial direction of the arcs in this example is determined by the current direction, but the boxes that follow them are positioned with reference to the ending direction of the arc. (This can be changed by using the **join** option of the **box** command.)

12. Font control and character coding

Text is output by default in a 12-point Times-Roman font, though certain special characters in PostScript output may use other fonts, as described in section 12.5 below. Additional fonts can be specified by the **bindfont** command, and individual strings can be output in any of the bound fonts.

12.1 The bindfont command

The **bindfont** command is used to define additional fonts, at specified sizes. The **magnify** command (see section13.2) does not affect the size of text. The syntax of **bindfont** is as follows:

```
  bindfont <number> <fontname> <font size>
```

For example:

```
  bindfont 1 "Times-Italic" 12;
  bindfont 2 "Times-Bold" 16;
```

The font number must be greater than zero (the default font has the number zero). Once a font has been bound, it may be referenced in a **setfont** command to make it the default font, or it may be specified for an individual text string. For example:

```
  setfont 2; box "this is font 2" "this is font 1"/1;
```

For PostScript output, the font name is used verbatim. For SVG output, if the name contains a hyphen, it is split into two parts. The first part (or the whole name if there is no hyphen) is output as the **font-family** parameter for text strings. The second part is used to control the **font-style** and **font-weight** parameters. Aspic recognizes the suffixes ‘Italic’, ‘Bold’, and ‘BoldItalic’.

12.2 The setfont command

The **setfont** command changes the default font for subsequent text strings. It must be followed by a non-negative font number.

12.3 The textdepth and fontdepth commands

When multiple text items are specified with a drawing command, they are output one below the other. The default vertical separation is computed from the sizes of the fonts. This can be increased (but not decreased) by the **textdepth** command, which sets a minimum vertical separation for subsequent items.

Aspic also needs to know the approximate height of letters when positioning text vertically, for example, when centring a single line of text within a box. Since it does not itself do any text processing, it guesses a height from the font size. This can be increased (but not decreased) by the **fontdepth** command, though this should rarely be needed.

12.4 Input character encoding

Aspic assumes that text strings specify characters in Unicode. Using escapes, it is possible to encode all possible characters using only ASCII input. Characters may also be encoded as UTF-8 sequences or (for backwards compatibilty) as single bytes. The input byte sequence is handled as follows:

- Bytes with values less than 128, with the exception of ampersand, are interpreted as single-byte Unicode code points – these are of course identical to ASCII. If the **-tr** command line option is given, the following translations are then performed:
  - A single grave accent character (`) is translated to an opening typographic quote (‘) using code point U+2018.
  - Two grave accents in succession are translated to a double typographic opening quote (“) using code point U+201C.

  - A single quote character (`'`) is translated to a closing typographic quote (’), which is the same as an apostrophe, using code point U+2019.
  - Two single quotes in succession are translated to a double closing typographic quote (”) using code point is U+201C.
  - Two hyphens in succession are translated to an en-dash (–) using code point U+2013.

  If any of the translated characters are required when the **-tr** option is on, they can be specified using numerical escapes. It is only the literal characters that are translated.

- If an ampersand character is encountered, the following bytes are inspected:
  - An ampersand followed by a sharp (hash) sign and a sequence of digits terminated by a semicolon represents the code point defined by the decimal number. For example, `&#169;` specifies a copyright symbol.
  - An ampersand followed by a sharp (hash) sign, an x, and a sequence of hexadecimal digits terminated by a semicolon represents the code point defined by the hexadecimal number. For example, `&#xa9;` is another way of specifying a copyright symbol.
  - If an ampersand is followed by a letter and then a sequence of alphanumeric characters terminated by a semicolon, it is treated as a named entity reference. Aspic contains a table of named entities taken from the DocBook documentation. For example, `&copy;` is a third way of specifying a copyright symbol.
  - If an ampersand is not followed by one of the above forms, or if an entity name is not found in Aspic’s inbuilt list, the ampersand character is treated as a literal.
- When a byte with a value of 128 or above is encountered, it and the following bytes are inspected to see if they form a valid UTF-8 sequence. If they do, the code point that it encodes is used. If they do not, the value of the single byte is taken as the code point. This means that isolated high-value bytes in an otherwise ASCII source are treated as ISO 8859 characters. Several such bytes in succession might accidentally form a valid UTF-8 sequence, so Aspic cannot be guaranteed to handle every possible ISO 8859 input document.

12.5 PostScript output character encoding

PostScript output consists entirely of ASCII characters. In strings, parentheses, backslashes, and code points greater than 127 are escaped using the normal PostScript backslash escape mechanism.

When generating PostScript output, Aspic handles fonts that are defined with PostScript’s *standard encoding* as their default in a special way. A font may contain more than 256 characters, though only 256 are accessible in any font ‘binding’, via an encoding vector that translates character numbers to names. The encoding can be changed when the font is bound.

The PostScript standard encoding is not the same as Unicode, and in any case, Aspic needs to access more than 256 characters in these fonts. It does this by binding two versions of a font, and re-encoding both of them. The first is encoded with the first 256 Unicode code points. The second is encoded with characters 0–127 corresponding to Unicode code points U+0100 to U+017F and characters with codes greater than 127 corresponding to those Unicode characters with code points greater than U+017F that are available in the font (for example, typographic quotes). This use of two fonts is handled automatically.

If a character is not available in a standardly encoded font, Aspic checks to see if it can be found in the Symbol or Dingbats fonts. The former contains Greek and mathematical characters, and the latter contains various special symbols such as ✓. If a character is not available in any of these fonts, it is output as the currency symbol ¤.

The result of this special processing is that, for a font that is standardly encoded by default, Unicode code points can be used for all the characters in that font, as well as characters in the Symbol and Dingbats fonts. You do not need to set up any separate special fonts.

If, on the other hand, you specify a font that does not use the standard encoding by default, Aspic makes no changes to it. Character values in the range 0–255 will correspond to its default encoding. The behaviour of other character values is undefined.

12.6 SVG output character encoding

SVG output consists entirely of ASCII characters. In text strings, angle brackets and ampersands are converted to the named XML entities `&gt;`, `&lt;`, and `&amp;`, respectively. Characters whose code points are greater than 127 are output as hexadecimal numerical escapes. For example, the copyright character is output as `&#xa9;`.

13. Overall Aspic configuration

This chapter describes commands that affect the overall appearance of the picture.

13.1 boundingbox

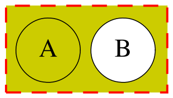
This command causes Aspic to draw a frame round the picture. It must be followed by a length, specifying the width of gap between the actual bounding box of the picture, and the frame. For example,

```
boundingbox 20;
```

If a value of zero is given, the frame that is drawn is the actual bounding box. Because Aspic does not process text items itself, it has to guess a bounding box for them, and so under some circumstances the computed bounding box for a picture may not be strictly accurate.

The gap width may optionally be followed by any of the following options of the **box** command (see section 8.4): **colour**, **dashed**, **filled**, **grey**, **thickness**. For example:

```
boundingbox 5 dashed filled 0.8, 0.8, 0 thickness 1 colour 1,0,0;
circleradius 15; circle "A";
iline right 5; circle filled 1,1,1 "B";
```



If **dashed** is specified, the dash parameters are taken from the **boxdash** setting in the current environment at the time that **boundingbox** is obeyed. If **boundingbox** appears more than once, the last set of parameters is used.

13.2 magnify

This command specifies overall magnification of the graphic items in a picture. It must be followed by a single number. For example

```
magnify 0.8;
magnify 1.5;
```

Magnification applies to those shapes that follow this command. Subsequent appearances of **magnify** apply cumulatively to the previous magnfication. Magnification does *not* apply to text. If smaller or larger text is required, suitable fonts must be set up and used.

13.3 resolution

This command sets the resolution of the output; it must be followed by a single fixed-point number. All output dimensions are rounded to this resolution. The default depends on the output style. For PostScript it is 0.12, which corresponds to 600 dpi; for SVG output it is 0.001, which disables rounding. If **resolution** appears more than once, the last value is used.

14. Saving and restoring the environment

The Aspic *environment* consists of a number of parameters that control the way items are drawn. They are listed in the following table, together with their initial values:

arc radius	36.0
arrowhead filling	no filling
arrowhead length	10.0
arrowhead width	10.0
box dash parameters	7.0 5.0
box depth	36.0
box edge colour	0.0 0.0 0.0
box edge thickness	0.5
box filling	no filling
box width	72.0
circle dash parameters	7.0 5.0
circle edge colour	0.0 0.0 0.0
circle edge thickness	0.4
circle filling	no filling
circle radius	36.0
current direction	right
ellipse dash parameters	7.0 5.0
ellipse depth	36.0
ellipse edge colour	0.0 0.0 0.0
ellipse edge thickness	0.4
ellipse filling	no filling
ellipse width	72.0
level	0
line dash parameters	7.0 5.0
line colour	0.0 0.0 0.0
line horizontal length	72.0
line thickness	0.4
line vertical length	36.0
magnification	1.0
shape filling	no filling
text colour	0.0 0.0 0.0
text line depth	12.0
text font	0
text font depth	6.0

Many of these values can be overridden for a single item by the use of options on the drawing command. There are also commands for dynamically changing these values, so they become the defaults for commands that do not specify the relevant options. The **magnify** command is described in section 13.2, and commands to change the current direction are described in chapter 11. Commands to change the remaining values are described in the next chapter.

It is often useful to be able to save the current state of the environment and restore it later. The **push** and **pop** commands are provided to do this. The **push** command puts a copy of the current environment onto a stack, and **pop** restores the environment from the top item on the stack.

15. Changing environment parameters

In addition to the commands for changing the current direction and the **magnify** command, which are specified above in chapter 11 and section 13.2, respectively, the following commands are provided for changing the values of environmental parameters. The changed value applies to subsequently defined items, and a value may be changed as often as necessary. The entire environment can be saved and restored by means of the **push** and **pop** commands.

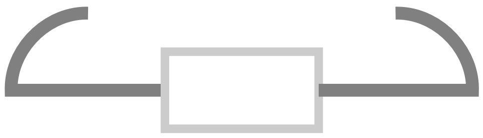
Most of these commands take a single numerical argument. The exceptions are those that set dashed line parameters, and those that specify a colour. The former take two arguments, specifying the length of dashes and the length of gaps, respectively, and the latter take three arguments, specifying the red, green, and blue components of the colour. Setting a grey value is equivalent to setting a colour with three identical values.

`arcradius`	default radius for arcs
`arrowfill`	fill colour for arrowheads
`arrowlength`	length of arrowheads
`arrowwidth`	width of arrowheads
`boxcolour`	colour of box edges
`boxdash`	dash parameters for boxes
`boxdepth`	default depth of boxes
`boxfill`	colour of box interiors
`boxgrey`	greyness of box edges
`boxthickness`	thickness of box edges
`boxwidth`	default width of boxes
`circlecolour`	colour of circle edges
`circledash`	dash parameters for circles
`circlefill`	colour of circle interiors
`circlegrey`	greyness of circle edges
`circleradius`	default radius of circles
`circlethickness`	thickness of circle edges
`ellipsecolour`	colour of ellipse edges
`ellipsedash`	dash parameters for ellipses
`ellipsedepth`	default depth of ellipses
`ellipsefill`	colour of ellipse interiors
`ellipsegrey`	greyness of ellipse edges
`ellipsethickness`	thickness of ellipse edges
`ellipsewidth`	default width of ellipses
`fontdepth`	minimal character depth
`hlinelength`	default horizontal length for lines
`level`	default item level
`linecolour`	colour of lines
`linedash`	dash parameters for lines
`linegrey`	greyness of lines
`linethickness`	thickness of lines
`setfont`	the current font
`shapefill`	colour for shapes defined by lines/arcs
`textcolour`	colour for text
`textdepth`	minimal vertical text separation
`vlinelength`	default vertical length for lines

Each kind of closed shape has its own set of parameters for controlling the default thickness and colour of the lines used to draw it, and the appearance of dashed lines. Setting the dash parameters does not of itself cause dashed lines to be drawn; the **dashed** option must be given with the drawing command. The default thickness, dashedness, and colour of arcs and curves is the same as that for straight lines; hence there are no separate commands. For example:

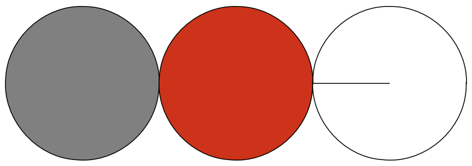
```
linethickness 6; linegrey 0.5;
boxthickness 4; boxgrey 0.8;
```

```
  arc left; line; box; line; arc up;
```



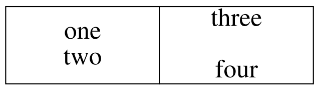
Those commands that specify filling can be followed either by a single number, to specify a shade of grey, or by three numbers for a general colour. To turn off filling, a single negative number should be given. For example:

```
  circlefill 0.5; circle; circlefill 0.8 0.2 0.1;
  C: circle; line right 36;
  circlefill -1; circle join left to right of C;
```



Note that turning off filling is not the same as filling with white. If the third circle above is filled with white, it would hide the horizontal line. The **textdepth** parameter controls the minimal vertical separation of multiple text items. For example:

```
  box "one" "two"; textdepth 24; box "three" "four";
```



The **fontdepth** parameter can be set to the approximate height of letters in a font. It is used when positioning text vertically, for example, when centring a single line of text within a box. However, this parameter is used only when it is greater than the depth computed from the size of the font.

16. List of commands

This is a complete list of all Aspic commands.

arc	draw a circular arc
arcarrow	draw a circular arc with arrowhead(s)
arcradius	set default arc radius
arrow	draw a straight line with arrowhead(s)
arrowfill	set arrowhead fill colour
arrowlength	set length of arrowheads
arrowwidth	set width of arrowheads
bindfont	bind a new font
boundingbox	enclose picture in frame
box	draw a box
boxcolour	set default colour for boxes
boxdash	set dash parameters for boxes
boxdepth	set default depth for boxes
boxfill	set box fill colour
boxgreyness	set default greyness for boxes
boxthickness	set default line thickness for boxes
boxwidth	set default width for boxes
circle	draw a circle
circlecolour	set default colour for circles
circledash	set dash parameters for circles
circlefill	set circle fill colour
circlegrey	set greyness for circles
circleradius	set default radius for circles
circlethickness	set thickness of lines for circles
curve	draw a Bèzier curve
down	set current direction
ellipse	draw an ellipse
ellipsecolour	set default colour for ellipses
ellipsedash	set dash parameters for ellipses
ellipsedepth	set depth of ellipses
ellipsefill	set ellipse fill colour
ellipsegrey	set greyness for ellipses
ellipsethickness	set line thickness for ellipses
ellipsewidth	set width of ellipses
fontdepth	set minimal height of letters
goto	set named shape as previous
hlinelength	set default horizontal line length
iarc	draw an invisible arc
ibox	draw an invisible box
icircle	draw an invisible circle
icurve	draw an invisible Bèzier curve
iellipse	draw an invisible ellipse
iline	draw an invisible line
include	include a file's contents
left	set current direction
level	set default level
line	draw a line
linecolour	set colour for lines (and arcs and curves)
linedash	set dash parameters for lines (and arcs and curves)
linegrey	set greyness for lines (and arcs and curves)
linethickness	set thickness of lines (and arcs and curves)
magnify	magnify or reduce the picture

macro	define an Aspic macro
pop	restore environment from the stack
push	push environment onto the stack
resolution	set output resolution
right	set current direction
set	set value of variable
setfont	set current font
shapefill	set drawn shape fill colour
text	output text at given position
textcolour	set text colour
textdepth	set minimal separation of text items
up	set current direction
vlinelength	set default vertical line length