

mkdrawf:

a utility for creating drawfile

User manual for version 3.10

1.	<u>Introduction</u>	3
1.1.	<u>About this manual</u>	3
1.2.	<u>Legal rubbish</u>	3
1.3.	<u>Who am I?</u>	3
1.4.	<u>How to read this manual</u>	3
1.5.	<u>What's new?</u>	3
2.	<u>How to use mkdrawf</u>	3
2.1.	<u>Normal use</u>	3
2.2.	<u>Errors</u>	4
3.	<u>How to get mkdrawf</u>	4
4.	<u>An introduction to drawfiles</u>	4
	Object type 1: <u>text</u>	4
	Object type 2: <u>path</u>	5
	Object type 5: <u>sprite</u>	5
	Object type 6: <u>group</u>	5
	Object type 7: <u>tagged</u>	5

	Object type 9: <u>text area</u>	5
	Object type 10: <u>text column</u>	5
	Object type 11: <u>options</u>	5
	Object type 12: <u>transformed text</u>	5
	Object type 13: <u>transformed sprite</u>	5
	Object type 16: <u>JPEG image</u>	5
5.	<u>The drawfile description language I</u>	6
5.1.	<u>General</u>	6
5.2.	<u>Tokens</u>	6
5.3.	<u>Numbers</u>	6
5.4.	<u>Comments</u>	6
5.5.	<u>Braces</u>	6
5.6.	<u>Points, dimensions and things</u>	6
5.7.	<u>Colours</u>	6
5.8.	<u>Keywords and special keywords</u>	7
5.9.	<u>Token lists</u>	7
6.	<u>The drawfile description language II</u>	7
6.1.	<u>General</u>	7
6.2.	<u>Global variables</u>	7
6.3.	<u>Macros</u>	8
6.4.	<u>Macro parameters and local variables</u>	8
6.5.	<u>Positional parameters</u>	8
6.6.	<u>Named positional parameters</u>	8
6.7.	<u>Undefined local variables</u>	8
6.8.	<u>Arithmetic and things</u>	9
6.9.	<u>String operations</u>	9
6.10.	<u>Conversions</u>	9
6.11.	<u>Conditionals</u>	9
6.12.	<u>Iteration</u>	9
6.13.	<u>File inclusion</u>	10
6.14.	<u>Units</u>	10
6.15.	<u>Infix expressions</u>	10
7.	<u>The drawfile description language III</u>	11
7.1.	<u>General</u>	11
7.2.	<u>The format of an object description</u>	11
7.3.	<u>Bounding boxes</u>	11
7.4.	<u>The Header pseudo-object</u>	11
7.5.	<u>The FontTable object</u>	11
7.6.	<u>The Text object</u>	11
7.7.	<u>The Path object</u>	12

7.8.	<u>The Sprite object</u>	12
7.9.	<u>The Group object</u>	13
7.10.	<u>The Tagged object</u>	13
7.11.	<u>The TextArea object</u>	13
7.12.	<u>The Options object</u>	14
7.13.	<u>The XfText object</u>	15
7.14.	<u>The XfSprite object</u>	15
7.15.	<u>The JPEG object</u>	15
8.	<u>Some sample drawfile descriptions</u>	15
8.1.	<u>The Koch snowflake curve</u>	15
8.2.	<u>A Lissajous figure</u>	16
8.3.	<u>A typical text area</u>	16
8.4.	<u>Circular arcs using Bezier curves</u>	17
8.5.	<u>A silly spiral</u>	17
9.	<u>Tagfiles</u>	18
10.	<u>Technical issues</u>	18
10.1.	<u>Memory management</u>	18
10.2.	<u>Variables</u>	18
10.3.	<u>Problems</u>	18
10.4.	<u>Arbitrary restrictions</u>	19
10.5.	<u>The future</u>	19
11.	<u>Revision history</u>	19
12.	<u>Decoding drawfiles</u>	19
12.1.	<u>Using decdrawf</u>	19
12.2.	<u>The output from decdrawf</u>	20
	<u>Symbols</u>	21

1. Introduction

1.1. About this manual

This is the user manual for **mkdrawf**, a program for creating drawfiles. Of course there are other programs for creating drawfiles, notably **!Draw**, but **mkdrawf** is unusual in that it allows you complete control over what goes into the drawfile. It takes as input a text file describing the objects that should go into the drawfile; the description language it uses is quite powerful, having variables, macros and some mathematical functions.

This manual describes the process of creating a drawfile using **mkdrawf**, and includes several sample **mkdrawf** input files and the resulting output. This manual also describes its much simpler companion program **decdrawf**, which performs the inverse process: it converts drawfiles to **mkdrawf** descriptions.

Before reading this manual you might want to read the **mkdrawf** tutorial. You might also want to look at the manual for **!Drawf**, a Wimp front end for **mkdrawf** and **decdrawf**.

1.2. Legal rubbish

1. These programs are not public domain; they are mine.
2. You are welcome to distribute them exactly unchanged (where “them” includes all the source code and all the documentation). You are also welcome to make changes to your own copy of the programs.
3. However, you are not permitted to distribute modified versions without my permission.
4. Furthermore, you are not permitted to charge money for copies of this software; it is free software, and I intend that it should remain free. And I certainly don’t want anyone other than me making money from it!

If you make improvements to the programs, I am very unlikely to object to your distributing the improved version provided it is made clear what changes you have made; the point is that I don’t want to be blamed for other people’s bad programming or credited with their good programming, and that I want the documentation for **mkdrawf** to be consistent with the program. So if you improve **mkdrawf**, get in touch with me so that I can give you an updated version of the documentation to go with your improved version.

If you have somehow acquired an incomplete or modified version of these programs, you may distribute it; but I’d prefer you to get in touch with me and get a complete, up-to-date version of the programs. At the time of writing, the current version is version 3.10.

1.3. *Who am I?*

If you want to get in touch (to moan about problems, to suggest improvements, to thank me for doing such a truly wonderful job, ...), you will need the following information.

I have an e-mail address and a snail-mail address. Both are valid as of September 1998, and both are likely to continue working for a year at least. E-mail is preferred, but I'll try to answer snail-mail too.

e-mail: gjm11@dpmms.cam.ac.uk

snail-mail: Gareth McCaughan
Peterhouse
Cambridge CB2 1RD

I would be really happy to hear from you. If you have complaints or suggestions, they may help me to make **mkdrawf** better; and if you have neither, presumably that's because **mkdrawf** is completely perfect for your needs... I'd be very glad to be told!

1.4. *How to read this manual*

Of course it's up to you; and it's not a very long manual, so you could probably read it in any order you like without causing yourself much trouble. However, I would suggest the following procedure:

- Have a look at the “sample drawfile descriptions”, to give you some idea about how **mkdrawf** works.
- If you don't have much idea of just what goes into a drawfile, read “An introduction to drawfiles”.
- Read the three chapters on “The drawfile description language”, skipping bits you don't understand. The second contains the most obscurities; you might well want to skim the third before reading the other two.
- Read “How to use **mkdrawf**”, and create a few files yourself, referring to the DDL chapters.

1.5. *What's new?*

This manual describes version 3.10 of **mkdrawf**. There have been a lot of changes since the last release; here's a brief summary of what has changed. There are no details in this section; you'll have to read the rest of the manual if you want those.

- Several colour names are predefined.
- There are some new mathematical operators.
- There's a new and more pleasant way of invoking macros, that makes them look more like functions in ordinary languages.

- Using JPEG objects is less unpleasant: you no longer have to specify so much redundant information.
- A new path element makes it easy to draw arcs of circles.
- Text areas can now include “expanded” text (not just literal text), and are no longer limited to 16 columns.
- There are now variants of the For loop whose syntax and semantics are less disgusting.
- There is a new infix parser that lets you enter mathematical expressions in something much more like their usual form.
- A number of bugs have been fixed. There are probably new ones to take their place.

2. How to use **mkdrawf**

2.1. Normal use

This is pretty easy. Just create a suitable input file, and type

```
mkdrawf <input-file-name> <output-file-name>
```

to make the output file. This will create the output file if it doesn't already exist, and will set its filetype to DrawFile. The only other way of invoking **mkdrawf** is to type

```
mkdrawf -v
```

which displays the version number and date of your copy of **mkdrawf**. This will be useful if I make further improvements. This version of the manual describes version 3.10.

Alternatively, you can use the !Drawf application to give you a Wimp front end to both **mkdrawf** and **decdrawf**.

2.2. Errors

mkdrawf produces three kinds of diagnostic: warnings, errors and fatal errors. I hope the distinction between them is clear. If any errors or fatal errors are produced, you should not rely on the resulting drawfile containing anything sensible, though in most cases it will at any rate be a valid drawfile.

mkdrawf tries hard to carry on reading your input file even if there are errors (because that may help you to spot several problems at once); but you should not rely on anything after the first error making sense. Just as with C, a single error can produce a cascade of error messages. After 100 errors, **mkdrawf** gives up. I'm afraid error messages are often not very useful; in particular, the line numbers are useless when the error is inside a macro. Sorry.

All this is an improvement on versions 2.10 and earlier, which simply stopped dead the first time they found anything they couldn't parse.

The command-line option **-e** will make **mkdrawf** omit the filename in error and warning messages. You are unlikely to need this.

3. How to get **mkdrawf**

If you're reading this manual, you probably already have a copy. But perhaps you want to be sure you have the latest version, or you've accidentally formatted your hard disc, or

something. Or perhaps you stole the manual. Anyway, you can get **mkdrawf**:

- from me, electronically or postally. If you want a copy sent to you by post, please send me either a floppy disc and whatever sort of packaging you think is suitable for the return journey, plus £1stg to cover postage and make me bother to send it; or £2stg, for which you will get an HD disc (or a DD disc if you request that), in what seems to me to be suitable packaging. If you add another £4 you can also have a nicely printed copy of the documentation.
- from some FTP sites. I know **mkdrawf** is available at micros.hensa.ac.uk (at least from academic sites in the UK), and I'm pretty sure it's also available at ftp.demon.co.uk. Please try any suitable FTP sites you can think of before trying to get a copy from me!

4. An introduction to drawfiles

This chapter describes briefly what sort of stuff goes into a drawfile. If you already know all about that (for instance, if you've read the relevant section of the PRMs), you can skip it. If not, read on...

A drawfile consists of a short header followed by a sequence of objects. Each object consists, in turn, of a short header (similar but not identical in structure to the header of the whole drawfile) followed by some more interesting stuff. Exactly what that "more interesting stuff" is depends on the object; in some cases, it can include other objects (and so recursively...); see the sections on Group and Tagged objects.

There are 12 types of object; several are not understood by the version of !Draw distributed with RISC OS 2, and one is not understood by any version of !Draw earlier than 1.09 (distributed with RISC OS 3.6). There is no guarantee that yet more object types will not be added in later versions of !Draw. Each type has an associated number, by which objects of that type are identified in drawfiles. The numbers are not consecutive; the apparent gaps are not mistakes.

Object type 0: font table

When you use !Draw you never see a font table object. That's because it doesn't actually describe any particular part of the drawing. A drawfile may contain text in several different fonts; internally these fonts are referred to by magic numbers rather than by names, and the font table establishes a correspondence between names and numbers. I don't know for certain why it's done this way, but here are some possible reasons:

1. It saves a bit of space in the drawfile if one font is referred to many times: only one byte needs to be repeated each time, rather than a long font name.
2. It makes things easier for applications that have to render drawfiles: they can call `Font_FindFont` once for each font in the table, and then set up a table of font handles and use that.

3. It means that if you want to make wholesale changes to the fonts in a drawfile (say, replacing Trinity with Homerton throughout), you only have to change the font table.

Reason number 3 here is irrelevant if you're using !Draw, but if you're using **mkdrawf** or something like it it might be significant. My money is on reason number 2 being the main one, though.)

There may only be one font table in a drawfile; if there are any text objects, there must be exactly one font table, and it must appear before they do. Actually many versions of !Draw don't cope correctly if the font table is not the very first object in the file.

Object type 1: text

If you select the text tool in !Draw and type some text, you are creating a text object. A text object specifies, as well as the text to be contained in it,

- the colour in which the text is to appear;
- the background it's likely to be on;
- the font it is to appear in;
- the size and aspect ratio of the font;
- where the text is to start (that is, its bottom-left corner).

The background colour is just a hint. It doesn't actually cause anything to be drawn in that colour, but it helps the anti-aliasing code in (for instance) !Draw to improve the appearance of the text.

Object type 2: path

This is the commonest object type. A path is made up of a number of subpaths. Each subpath is drawn, and perhaps filled, separately (but the same options for things like path width, fill colour and so on are common to all the subpaths). A subpath consists of simple elements:

- move to a given place
- draw a line to a given place
- draw a Bezier curve to a given place, with given control points
- close the current subpath.

Each subpath starts with a move, and each move begins a new subpath. It's possible to specify the following things for a path object:

- the colour to use for filling it in (may be "none")
- the rule used to determine what gets filled and what doesn't

- the colour to use for drawing the outline (may be “none”)
- the width of the outline
- how to join up adjacent sections of path
- what to do at the ends of the path
- whether the path outline should be “dashed”, and what dash pattern to use

Object type 5: sprite

A sprite object simply consists of an object header and a sprite. (The object header determines exactly where the sprite will appear and how big it will be.)

Object type 6: group

A group object contains any number of other objects. (It’s possible to work out from the object header where in the drawfile the list of objects in the group ends.) The only other information in a group object is a name; the name has no effect on the rendition of the object, but it may be used for identification purposes by some applications.

Object type 7: tagged

A tagged object consists of a 4-byte identifier, an object and some quantity of extra data. A tagged object is rendered simply by rendering the object it contains; the other stuff may be used by certain applications. (The PRM says: “This allows specific programs to attach meaning to certain objects, while keeping the image renderable”.)

Object type 9: text area

A text area object consists of a number of columns and a quantity of text (and a little extra data). The text contains various escape sequences which control how it will be formatted. A text area object is rendered by formatting the text in the columns (using the columns in sequence); users of DTP packages may like to think of the columns as being linked text frames.

A complete list of the escape sequences and their meanings is contained in the next chapter of this manual.

Object type 10: text column

Text column objects appear only inside text area objects. They describe the columns into which the text will be placed.

Object type 11: options

An options object describes the following things, most of which are specific to !Draw.

- The size of paper on which the objects are supposed to appear
- Some information about “paper limits”
- The setup of the grid (shown? locked? what spacing? ...)
- The zoom ratio
- Whether the toolbox is to be shown or not
- What “entry mode” to use initially
- The size of !Draw’s undo buffer

If more than one options object appears, !Draw ignores all but the first. The RISC OS 2 version of !Draw ignores options objects (and doesn’t save them).

Object type 12: transformed text

A transformed text object is a text object with bells on. The bells specify:

- what affine transformation to apply to the text
- whether to apply kerning to the text
- whether to render the text right-to-left

An affine transformation is specified by a 2×2 matrix and a vector. More on this later.

Object type 13: transformed sprite

A transformed sprite object bears the same relationship to a sprite object as a transformed text object does to a text object, except that kerning and right-to-left aren’t issues. In other words, a transformed sprite object contains a transformation matrix (erm, and vector) and a sprite.

Object type 16: JPEG image

A JPEG image object consists of a small amount of header data and a JPEG image in JFIF format. (JPEG is a standard method of compressing images; JFIF is the name of the commonest file format for JPEG-compressed images. What is usually called a “JPEG file” should strictly be called a JFIF file.) Only the most recent versions of !Draw understand this object type.

5. The drawfile description language I

5.1. General_I

This chapter and the next two describe, in some detail, the language understood by

mkdrawf. To be more precise: this chapter is a brief description of the syntax of the language, the next deals with variables and macros and things, and the next describes what gets done with the tokens produced when all the macro expansion and variable substitution are finished. (So it's that chapter that actually discusses drawfile objects.)

5.2. Tokens

As **mkdrawf** reads an input file, it parses it into tokens. A token might be a keyword like `FontTable`, or a number like `1.23456`, or a colour like `r200g100b123`, or whatever. Except in funny circumstances – strings and things – tokens are delimited by whitespace and are not case-sensitive, so you can write `LINE` or `Line` or `line` or `lInE` or whatever you happen to prefer.

Here is a list of the token types understood by **mkdrawf**, with an example of each. Some of these are not explained until the next chapter.

- | | |
|-------------------|-----------------------------|
| • Number | <code>123.456e2</code> |
| • String | <code>"foo bar"</code> |
| • Colour | <code>r123g0b255</code> |
| • Keyword | <code>Width</code> |
| • Special keyword | <code>Define</code> |
| • Global variable | <code>\$myvar1</code> |
| • Local variable | <code>%length</code> |
| • Macro name | <code>Snowflake</code> |
| • Braces | <code>{, }</code> (2 types) |
| • Brackets | <code>[,]</code> (2 types) |
| • Parentheses | <code>(,)</code> (2 types) |

Internally, **mkdrawf** actually uses a couple of other keyword types, but they aren't things you can put into the file yourself.

Here's a brief account of how **mkdrawf** decides what type each token it sees has:

Opening and closing braces are easy. Anything that starts with a `$` is a global variable; anything that starts with a `%` is a local variable; anything that starts with a `"` is a string. Numbers are things that can be parsed successfully by the `strtod` function in the C library; colours are things matching the `scanf` format `r%dg%db%d`. Anything else is one of the other types. **mkdrawf** has a list of keywords and "specials"; anything else must be a macro name. Simple!

5.3. Numbers

You can enter a number either in a form that the C library function `strtod` can understand, or in the form `0x1234FEDC` (meaning hexadecimal). Both get dealt with in the program as doubles, should this be relevant.

5.4. Comments

If a token begins with a # then that token, and everything else up to the end of the line, is ignored by **mkdrawf**. This is useful for making your **mkdrawf** input files more comprehensible, and also for “commenting out” sections that you don’t want to use but don’t want to throw away completely either.

I suppose that strictly speaking this means there’s another type of token (comment start), but that’s not how **mkdrawf** thinks about it internally. Something that would be a token but begins with a # never makes it as far as tokenhood; the token-maker just reads a new input line and tries again.

5.5. Braces

The language understood by **mkdrawf** is a structured one; its structure reflects the structure of drawfiles. For instance, a Group object actually contains other objects; so the description of a Group object in a **mkdrawf** input file actually contains descriptions of other objects. Groupings of this sort are done with braces. Since braces are tokens, they must have whitespace on either side. (A brief note for the hitherto ignorant: “whitespace” includes newlines as well as spaces and tabs!)

Whenever it’s stated that something is delimited by braces, this means it’s delimited by matching braces. For instance, you can’t put unmatched braces into a macro definition.

5.6. Points, dimensions and things

A point (meaning a location in 2-dimensional space, rather than a pixel) is represented by two numbers. By convention, the point 0, 0 is at the bottom-left corner of the image. The unit for these numbers, and indeed for almost all dimensions, is by default the point; for our purposes a point is exactly 1/72 of an inch. (Historically the printer’s point is a slightly smaller unit than this; but the definition of the drawfile format says that its points are exactly this size. I think this accords with usage in PostScript and on the Apple Macintosh; it differs from that of TEX.) It’s possible to use different units; more on this later.

A bounding box is represented by two points, which should be the bottom-left and top-right coordinates of the box.

In a drawfile, most dimensions are stored as integer multiples of 1/640 of a point; so giving dimensions to 10000 significant figures is probably not sensible.

As of version 3.10, anywhere you can give a number you can instead give an explicit dimension; that is, a number followed immediately (no space) by one of the following: sp os pt mm cm in. This is interpreted as “whatever number would, if given as a bare number, yield the specified dimension”; thus, typically 5pt means

exactly the same as 5. Note that if the unit is changed from its default of points (see later) then the interpretation of these dimensions changes accordingly, and that (2) this applies even when the number is going to be used as something other than a dimension.

If you don't know what all those units are, look under Units in the next chapter.

5.7. Colours

A colour is specified by giving its red, green and blue intensities. The format is as described above; the numbers should be between 0 and 255 inclusive. There is also a special colour, called **None** or **Transparent** (these are synonyms), which means just what you would expect it to.

Alternatively, wherever a colour is required you may instead give three integers in the correct range. They will be used as the red, green and blue intensities of the colour. This means you can say things like:

```
for $x from 0 to 255 {
  path {
    outlinecolour none
    fillcolour $x 0 ( 255 - $x )
    move ( 100 + $x ) 100
    rline 1 0   rline 0 100   rline -1 0 close
  }
}
```

(which produces a nice shaded effect).

As of version 3.10, a number of symbolic colour names are defined, so that you can use Black or Red as a colour. The defined names are: **Black White Red Green Blue** and **Wimp0...Wimp15** (you can surely guess what these are). The colours **Wimp8** onwards have synonyms: **Wimp_DarkBlue Wimp_Yellow Wimp_LightGreen Wimp_Red Wimp_Beige Wimp_DarkGreen Wimp_Orange Wimp_LightBlue**.

5.8. Keywords and special keywords

Keywords are bits of drawfile object specification. Some of them name object types; some name parameters of objects; some are “flags” (their presence or absence determines something about an object); some are possible parameter values, when those values form a finite set. Examples (in order): Path, OutlineColour, Kerned, Mitred. Special keywords do not directly describe drawfile objects. They are dealt with earlier, by the macro-and-variable machinery described in the next chapter. They are used in defining macros, doing arithmetic calculations, and so on.

5.9. Token lists

I'm not sure whether I should count these as tokens. They are treated as such in the program. A token list is, erm, a list of tokens; the expansion text of a macro is a token list, and the value of a variable can be too. The only thing you really need to know about these is that there's an implementation limit on the depth of nesting of token lists, namely 256 levels of nesting.

You get a token list into a variable by doing, say,

```
Set %womble { 1 2 3 4 }
```

which gives the local variable `womble` the token list `1 2 3 4` as its value. (What's a local variable? See the next chapter.)

Note: the body of a For loop is not actually a token list, although it looks like one; nor are the brace-delimited bits of drawfile objects.

6. The drawfile description language II

6.1. General_II

The drawfile format, unlike (say) PostScript, has no “language” features at all. A drawfile cannot use variables, looping constructs and so on. **mkdrawf** allows you to use some (fairly weak) programming features of this kind in a way that fits quite well with its underlying language; for instance, you can give values to variables and define macros.

mkdrawf splits your input file into tokens, as discussed in the previous chapter. However, most of the time it isn't just reading tokens; it's expanding them too. This means that if it sees an expandable token (I'll explain this in a moment) it replaces it with its expansion, which is usually another token or a whole sequence of tokens.

The tokens resulting from expansion are then expanded again, and so on; thus, what eventually emerges is a stream of unexpandable (i.e., simple) tokens. What happens to those is the subject of the next chapter.

Here's a list of expandable token types, together with what happens to them on expansion. There's more detailed information in the next few sections.

- A variable (local or global) is replaced by its value.
- A macro, together with its parameters, is replaced by its text; while that's being processed, local variables are set up corresponding to the parameter values given.
- A token list is replaced by the tokens in the list, in order.

- A For, together with its parameters, is replaced by several copies of the loop body.
- An If vanishes, together with any conditional stuff in the “wrong” arm of the If.
- A Define or Set vanishes, together with the thing being defined and the value it’s being given; of course the defining or setting happens too.
- An Include, together with the filename following it, is replaced by the contents of that file. (This is slightly inaccurate: see below.)
- An operator (like Plus or Sin), together with one or more tokens following it, is replaced by the result of applying that operator to the appropriate values.

6.2. Global variables

A global variable is named by a token that starts with a dollar sign \$. When **mkdrawf** sees a global-variable token in its input stream it replaces it with the current value associated with that variable. Of course this isn’t quite true, since it must be possible to set the variable in the first place; when **mkdrawf** sees the special keyword Set it doesn’t expand the following token, but treats it as a variable name. It then sets the variable’s value to be the token that follows, or (if that’s an open-brace token) the token-list that follows.

The value of a variable can be any single token, or a token list. Note that if you do

```
Set $foo { 1 2 3 }
```

the value of \$foo is the list

```
1 2 3
```

rather than the list

```
{ 1 2 3 }.
```

When you set a variable to a token list, the tokens in the list are not expanded as they are being read. (But of course they will be expanded later when the variable name is read and expanded.) See the comments below on the same feature for macro parameters if you want to know why.

mkdrawf sets up some useful global variables for you when it starts. Some of them are described in the section on units; here is as good a place as any to mention that it also defines variables \$pi, \$2pi and \$pi/2 with the obvious values; this can be useful for drawing arcs of circles.

6.3. Macros

A macro's name can be any token that doesn't already have some other significance. So you can't name a macro `Width` or `123` or `$foo` or `%zog`, but you could call it `Fred` or `2a` or `^!"£@`. A macro expands to a list of tokens. You call a macro by giving its name followed by `{ }`. (Why? See the next section.) Macros are defined with the `Define` special keyword, which behaves very much like the `Set` special keyword: give it the name of the macro, followed by the text of the macro enclosed in braces.

6.4. Macro parameters and local variables

There's more, though: macros can take parameters, and can use local variables. I'm lumping these together because they have the same syntax and use the same internal machinery. A parameter or local variable is named by a token that starts with a percent sign `%`. It has a value local to the macro containing it; local, in fact, to the macro invocation containing it. (The point is that you can call a macro many times; it's a different variable each time.)

Local variables get their values in two ways. The first is by the `Set` special keyword, just as for global variables; the second is by being used as parameters when the macro is called. To call a macro with parameters, put some stuff between those braces. The stuff should look like this:

```
Fred { %foo 123 %aardvark Width %%% %foo }
```

which means that macro `Fred` should be called, and three of its local variables given values. The local variables are called `%foo`, `%aardvark` and `%%%`; the values are, respectively, the number `123`, the keyword `Width`, and whatever the value of local variable `%foo` is when `Fred` is called. (The procedure is: read a token without expansion, check it's a local variable name, then read a token with expansion and assign it to the appropriate local variable for the macro about to be called. Warning: versions of **mkdrawf** between 3.00 and 3.09 inclusive had a bug in this mechanism, so that e.g.

```
Foo { %x %y %y %x }
```

would set `Foo`'s local `%x` to the current value of `%y` and then `Foo`'s local `%y` not to the current value of `%x` but to `Foo`'s newly acquired `%x`. This is fixed in version 3.10.)

A local variable (or macro parameter) can have a token list as its value. To do this with a macro parameter, you say (for instance):

```
Fred { %foo 123
      %bar { Width 1 OutlineColour r0g0b0 }
      %baz "Hello" }
```

which means that `%foo` gets the value `123` and `%baz` the value `"Hello"`, as before, but `%bar` gets that list as its value. Within the macro body, `%bar` will expand to `Width 1 OutlineColour r0g0b0`. Note again that whatever's inside the braces doesn't get expanded until `%bar` is expanded.

This allows you to do something very like anonymous functions (!), and if you need something variable in there you can always do

```
Fred { %list { Set %zog Plus %zog %foo } %foo $zoo }
}
```

since the value given to %foo will be expanded at invocation time.

The fact that a local variable is local to a macro invocation rather than to a macro definition makes recursion possible. See the snowflake curve example later.

Note that the following will not work:

```
Define Foo { FontTable }
Foo {
  1 "Trinity.Medium"
  2 "Homerton.Bold"
}
```

but the following will:

```
Define Foo { FontTable }
Foo { } {
  1 "Trinity.Medium"
  2 "Homerton.Bold"
}
```

And so will:

```
Set $Foo { FontTable }      # or, Set $Foo FontTable
Foo {
  1 "Trinity.Medium"
  2 "Homerton.Bold"
}
```

6.5. Positional parameters

Sometimes you need or want a less cumbersome way of getting parameters into macros. Macro parameters of the form %number are special: %1 is the first token after the macro's parameter list, %2 is the next and so on. As many of these are read as are needed, up to a maximum of 9. (This number may be different for different invocations of the same macro, please note.)

This allows you, for instance, to produce your own functions with syntax the same as that for the arithmetic functions described below.

Warning: this way of passing parameters to functions is deprecated as of version 3.10; there is now a better way of achieving the same results. In a later version, positional parameters may well disappear.

6.6. Named positional parameters

In version 3.10, **mkdrawf** has finally acquired a sensible way of passing parameters to macros. If you define a macro like this

```
Define Foo [ %x %y ] { Set $foo-x %x Set $foo-y %y
}
```

then you can call it like this:

```
Foo [ 123 %x ]
```

and the right thing will happen. If there are more values in the invocation than variables in the definition, the spares will be put into local variables with names %_1, %_2 and so on (and there is no restriction on the number of such variables). If there are more values in the definition than in the invocation, the excess variables simply aren't set.

6.7. Undefined local variables

Since version 3.00, a local variable that isn't set explicitly inherits its value from whatever context it was called from; thus, in

```
Define First {
  Set %x "blah"
  Second { }
}

Define Second {
  Text { ... Text %x }
}
```

the Text object will say “blah”. This hasn't really been documented before, but despite my feeling that it's a bug I'm not changing it — it might be useful. However, all this does mean that the IfExists test may not do what you think it does, so in version 3.10 there is a new test called IfExistsHere which tests whether a global variable exists at all, and whether a local variable has a value belonging to the current macro invocation (i.e., set in this macro invocation or passed to it as a parameter). If you've used IfExists anywhere, you should think carefully about whether you actually want IfExistsHere.

6.8. Arithmetic and things

You can do some (very limited) mathematical operations on numbers. This is all done by the token-expansion machinery: for instance, expanding the Plus special keyword reads two more tokens (with expansion), checks that they're numbers, and yields a numerical token whose value is the sum of their values. The special keywords

that do this sort of thing are:

**Plus Minus Times Over ToThe Sqrt Sin Cos Tan Arcsin
Arccos Arctan Arctan2 Floor Ceiling Abs Sign**; it's probably obvious what most of these do, apart from

Floor gives the greatest integer \leq the number you give it;

Ceiling gives the greatest integer \geq the number you give it;

Arctan2 reads two numerical values a and b , and returns the angle between the positive x -axis and the vector (a,b) ;

Abs gives the absolute value of the number you give it (i.e., whichever of x , $-x$ is non-negative);

Sign gives $+1$, 0 or -1 depending on the sign of the number you give it.

I'm afraid that mathematical expressions look a bit ugly when written like this; for instance, what you might like to write as " $\sin(2(x+3))$ " becomes

```
Sin Times 2 Plus $x 3
```

which I agree is nasty. Those used to LISP will probably find it a bit more manageable than most...

As of version 3.10, several of these operators have shorter names: $+$, $-$, $*$, $/$, $^$. Furthermore, you no longer have to use prefix notation: see the section later about the new infix feature.

There is also **Random**, which reads no more tokens and expands to a random number between 0 (inclusive) and 1 (exclusive). Unless you are very unlucky you will get a different sequence of random numbers every time you run **mkdrawf**.

6.9. String operations

It's possible to do some things with strings, too. Two things, in fact. Firstly, you can append them to form new strings.

```
Append 3 "foo" "bar " "baz"
```

expands to "foobar baz". Secondly, you can feed them to the `OS_GSTrans` SWI, to get variable substitution and things. (Try making a text object including the line `Text GSTrans "<Sys$Time>".`)

6.10. Conversions

It's sometimes useful to be able to convert strings to numbers, and vice versa. There are three operators for doing this. **Str2Num** and **Num2Str** do just what you would expect them to (**Str2Num** will give an error and a result of zero if given a string that

wouldn't be recognised as a number by the token-parsing routines). Font "foo" expands to the number of the font whose name is "foo", if there is any such in the font table. If there is a font table and it doesn't contain a font called foo then the expansion is 0 and an error message is given; if no font table at all has been given, **mkdrawf** will insert the font name into an internal table and remember to build a font table for you when it finishes.

6.11. Conditionals

Expanding the token **IfExists** does the following: First it reads one token, without expansion. If this is the name of a variable that exists (i.e., has had a value assigned to it with Set or as a macro parameter), the next bunch of tokens will be read from whatever follows it, until an **EndIf** or **Else** token is seen. If an **Else** token is seen first, everything from there to the matching **EndIf** will be skipped. On the other hand, if what it reads is not the name of an existing variable, everything up to the matching **EndIf** or **Else** is skipped.

This is easier than it sounds; the behaviour is just the same as that of an If in BASIC or C or whatever. Tokens are not expanded while skipping; they are expanded while not skipping. This means that if you define a macro with unbalanced **If** and **Else** and **EndIf** things in it, you can cause chaos. So don't.

As well as **IfExists**, there are two other sorts of **If**: **IfLess** reads two tokens, with expansion; they must be numbers, and it does what you think it does; and **IfEqual** reads two tokens, with expansion, and again does just what you think it does. If there's a need for other sorts of conditional, I can put them in pretty easily.

6.12. Iteration

Finally, there is a **For** construct that behaves a little like the similarly-named things in other languages. Remember, though, that this works by macro-expansion; it doesn't "compile" to a block of stuff followed by a test and a conditional jump. All the testing and jumping happens in the macro-expansion mechanism.

As of version 3.10, there are three different syntaxes for **For**. The only one that existed in older versions is now deprecated; it was horrible.

For variable from initial to limit [step increment] [do] { stuff }

is very much like BASIC's FOR I = 1 TO 100 STEP 3: stuff: NEXT loop. The variable, which may be local or global, is initially given the value initial. Then stuff is spliced into the token stream. When the end of the stuff is reached, the variable is increased by step (or, if that wasn't specified, by 1). If that value is at most limit (if step was positive), or at least limit (if step was negative), then stuff is spliced in again, and it all repeats; otherwise we're finished. There is one error in that description: in fact we do the termination test before the first splicing too, so that it's possible for the stuff never to get spliced in.

For variable in token-list [do] { stuff }

does stuff with variable having each value in the token-list. If the token-list is empty then it does nothing, except perhaps to corrupt variable.

For variable initial limit { stuff }

is the old, deprecated syntax. It is similar to the first variant, except that it doesn't allow an increment other than 1, is harder to read, and always does the stuff at least once. And if the variable reaches limit exactly, then this variant doesn't do another iteration of stuff whereas the new version does.

Relevant facts: the variable is of course not expanded; the stuff is not expanded before it's spliced into the token stream, so it can refer to variable and have that expand differently each time round the loop; it is legal to change the value of variable within the stuff, and this behaves in the way you would expect. If you include do, it's ignored; its only purpose is to make your code easier to read.

6.13. File inclusion

This is not really a “language feature”, but it seems to belong here more than it does anywhere else. If your **mkdrawf** code contains a line like

```
Include "$.Zog.Wombat"
```

the contents of that file will effectively be included at that point in the input file. You can nest these inclusions; **mkdrawf** can cope with having up to 17 input files open at once (although the Shared C Library or the operating system may impose other restrictions).

This allows you to have a “header file” containing frequently used variable and macro definitions, or to have a mostly-unchanging file with a few variable bits read from other files.

I advise you, for reasons of clarity, to put each **Include** on a line by itself. If you don't, the precise behaviour is as follows: The remainder of the line on which the Include (with its associated string) finishes is read, and then input switches to the named file until that is exhausted.

You might also want to know that **Include** doesn't do anything if it's read at a time when tokens aren't being expanded. The most notable such case is when the a macro definition is being read. So you can, if you must, have a macro called **Include** (although that would stop you using the Include feature from then onwards). And you can define a macro that includes a file every time it is expanded; but you can't read the contents of a file into the definition of a macro.

6.14. Units

It's possible, as of version 3.07, to use units other than points. If **mkdrawf** sees Units 123, it interprets this as "From now on, treat all dimensions as being given in units each of which is 123 times the size of a point". To make this friendlier, **mkdrawf** predefines a few variables for you: for instance, \$Inches is 72, so that Units \$Inches will allow you to write all your dimensions in inches. The complete list of such variables is, in ascending order of size: **\$ScaledPoints**, **\$OSUnits**, **\$Points**, **\$Millimetres**, **\$Centimetres**, **\$Inches**. (A scaled point is 1/640 of a point, the smallest possible dimension in most contexts in a drawfile; an OS unit is 1/180 of an inch. More precisely, **!Draw** assumes that an OS unit is 1/180 of an inch, although this is probably not true of your screen in your favourite screen mode.) The change in treatment of dimensions happens when the Units token is expanded, just as file inclusion happens when the Include token is expanded; so you can put unit changes into a macro.

There is also a **Unit token**; this expands to a number which is the number of points in the current unit — thus **Units Unit** is guaranteed to do nothing. This can be useful if you want to change units temporarily:

```
Set $OldUnit Unit
do something
Units $OldUnit
```

Warning: changing the **Units** affects all dimensions, including things you might not think of like dash patterns and widths of paths.

6.15. Infix expressions

As of version 3.10 of **mkdrawf**, it's no longer necessary to use the horrible prefix notation all the time. A (token introduces an infix expression, which lets you say things like

```
Move ( $x + 2 * cos ( $r * %theta ) )
      ( $y - 3 * sin ( $r * %theta ) )
```

and have the Right Thing happen.

The way this works is that when the token-reader sees a left parenthesis it calls a recursive-descent parser that collects tokens until the matching right parenthesis into a token list; then that token list is spliced into the token stream (as happens for the body of a macro, for instance).

If you use an operator (like **Append**, for instance) that takes more than one argument, you should enclose those arguments in parentheses. (Using *macros* with several arguments doesn't need any such special treatment, because their arguments are already enclosed in {...} or [...].)

Many things don't work inside infix expressions; for instance, the **If**s don't. If you

stick to things that really are *expressions* you're probably safe.

One non-obvious thing you can do with infix expressions: Set has been made an infix operator. Saying (**\$x set 1234**) wouldn't be very pleasant, but **:=** has been made a synonym for **Set**, so that you can say (**\$x := 1234**).

The division of your input into tokens is still done in exactly the same way as it always has been, so spaces are still needed: if you write (**\$x+\$y/3**) then **mkdrawf** will think you're referring to a variable named **\$x+\$y/3**.

If you want to write more than one expression in infix notation, you can do it quite neatly by putting the whole thing in two sets of parentheses. (This is a side-effect of the way in which operators with more than one argument work.)

7. The drawfile description language III

7.1. *General_III*

This chapter (finally!) describes what happens to the “simple” tokens that are spat out by the macro-expansion stuff. Here the correspondence between your input file and the final output file is extremely close. After expansion, your input file should look like a succession of *object descriptions*. An object description describes a single drawfile object; when one object contains another, its object description will contain the object description for that other object.

7.2. *The format of an object description*

An object description consists of a keyword naming the object type, followed by a left brace, followed by some stuff describing the object, followed by a right brace. The details of the “stuff” depend, of course, on just what sort of object it is; in broad terms, though, it's very much like what goes into the parameter list for a macro invocation, but instead of local variable names you put appropriate keywords. I shall refer to the keyword/value pairs as items.

In what follows, a “complete” (in the sense of showing all the possible keywords and the sort of values they take) description for each object type is given.

7.3. *Bounding boxes*

Almost any object can contain an item of the form

```
BoundingBox 0 0 100 200
```

which overrides the bounding box **mkdrawf** would otherwise have given it. This is not usually a good thing to do, since **mkdrawf** calculates its bounding boxes very

sensibly. Sprite and transformed sprite objects need a bounding box (see below); font table and options objects have no bounding boxes. You also can't specify explicitly the bounding box of a text area object, because that is absolutely determined, with no room for disagreement, by its text columns.

7.4. The **Header** pseudo-object

There is one sort of object description that doesn't actually describe a real object. Rather, it describes what goes into the header at the start of the drawfile. Here's a "complete" **Header** object description.

```
Header {  
    Version 201 0  
    Creator "Aardvark"  
    BoundingBox 0 0 500 500  
}
```

The **Version** item takes two integers, the major and minor version numbers. Changes to the major number indicate incompatible changes in the file format, so you shouldn't give a major number other than 201 unless you have also modified **mkdrawf** to make it produce the new format. The default is 201 0.

The **Creator** item takes a string, which must be at most 12 characters long. If it's less than that, it will be padded with spaces. It's usual to make it no more than 8 characters. The default is **mkdrawf3** (this is the third version of the **mkdrawf** program. Until version 3.10 this field was set by default to **mkdrawf2**; I just forgot to update it).

The **BoundingBox** item takes four numbers; their significance was explained two chapters ago. You don't have to give this; generally **mkdrawf** does a good job of working out bounding boxes. You might want to change it for special effects, though, but you should be warned that applications are perfectly within their rights to misbehave horribly if given dishonest bounding boxes!

7.5. The **FontTable** object

Here's a complete **FontTable** object description.

```
FontTable {  
    1 "Trinity.Medium.Italic"  
    2 "Helvetica.Medium.Exploding.Strawberry"  
    3 "Corpus.Oblique"  
}
```

Font number 0 is always the bitmapped system font; you should not try to redefine it. Also, font numbers are only 1 byte long. In other words, the numbers you give must

be integers between 1 and 255, inclusive.

Incidentally, I have lost my only copy of the **Helvetica.Medium.Exploding.Strawberry** font; if you know where I might find another, please let me know.

If you don't put a **FontTable** object in your drawfile description, **mkdrawf** will construct one for you. However, if you do have a **FontTable** it must contain all the fonts you use in **Text** and **XfText** objects.

7.6. The **Text** object

Here's a complete **Text** object description.

```
Text {
  Colour r0g0b0          # black
  Background white       # remember this is just a
hint
  Style 1                 # in other words, font
number 1
  Size 10 12             # x-size 10pt, y-size 12pt
  StartAt 100 100        # start at (100,100)
  Text "This is a text object"
}
```

If you find **Style** unpleasant in this context, let me know. It's what the description in the **PRM** uses. It might be relevant that the "style" here is a 4- byte word; one byte is used for the font number, and the rest is reserved. So perhaps one day it will mean something other than just "font"...

You can specify the placing of the text in another way besides **StartAt**: if you replace it with an item

```
CentreIn 100 100 500 300
```

(say), then the text will be placed so that its bounding box has its centre at the centre of that box. Alternatively you can give both

```
StartAt point
```

and

```
HCentreIn box
```

and **mkdrawf** will compute the horizontal position from the box and the vertical position from the point. Since these are horribly verbose, you can also say

```
CentreAt 300 200
```

or

```
HCentreOn 100 500 200;
```

the former means what you think it does, the latter takes two x-coordinates and a y-coordinate.

Probably a better option is to use the `PlaceText` macro defined in the `TextBBox` example source file.

As of version 3.08, when a **Text** or **XfText** object (we'll meet **XfText** objects later) is made, **mkdrawf** sets the variables `$_x0`, `$_y0`, `$_x1`, `$_y1` to the bounding box of the object it has just created. This makes it possible to tell **mkdrawf** to put two pieces of text so that the gap between them is of some desired size. It's sometimes useful to be able to get at the bounding box of a piece of text before you typeset it; version 3.08 also introduces the **Virtual** keyword which allows you to do this. A **Text** or **XfText** object which is **Virtual** doesn't actually get put into the drawfile. The bounding box variables, however, do get set. Thus you can find out the vital statistics of a text object before you actually commit yourself to putting it in a particular place. One other thing about **Text** and **XfText** objects has changed in 3.08: most attributes of a piece of text now default to whatever values they had for the last **Text** or **XfText** object (even if it was virtual). This should save typing.

7.7. The Path object

This one is rather complicated; glance at the description in the drawfile summary earlier to see why ! Anyway, here's a complete **Path** object description.

```
Path {
  FillColour None
  OutlineColour r0g0b255
  Width 1      # points. 0 = as thin as possible
  Style {
    Mitred      # join style.
                # Options: Mitred Round Bevelled
  EndCap Round      # Options: Butt Round
                    #      Square Triangular
  StartCap Triangular
  WindingRule NonZero #determines how filling is
done.
                    # Alternative: EvenOdd
  CapWidth 32      # for triangular caps only.
                    # 0..255, in 16ths of line
width
  CapLength 64      # ditto
  Dash {
    1 3 2 5 3      # if you have no idea what
```

```

    this
        Offset 2 }          # means, experiment!
    }                      # ditt
# Now the path itself
    Move 100 100
    Line 200 100
    Curve 250 100 200 300 400 400 # Bezier curve.
                                   # Two ctl points,
end
    Close                    # this subpath
    Move 200 200
    Line 300 200
    Move 500 500             # this begins a new subpath
too,                          # even though no Close
before it
    Line 0 0
}

```

If the **style** item specifies a dash pattern, it must come before any path elements. This is a consequence of the way in which the information is arranged in the drawfile: the dash pattern, if present, comes before the path elements, and it's of variable size.

There are also “relative motion” keywords **RMove**, **RLine** and **RCurve**. These are just like the R-less versions, except that the numbers they mention are relative to the last point visited: thus they behave exactly like the similarly-named PostScript operators.

As of version 3.10 (purely as a concession to human laziness) you can specify a Bezier curve segment approximating to a circular arc with the keyword **Arc**. That takes three parameters: the two coordinates of the centre of the circle, and an angle indicating how far around the circle the arc is to go. (So an angle of zero will produce a curve of length zero.) This is precisely equivalent to a carefully chosen Curve element.

The arc starts from the previous point.

Warning: the **CapWidth** and **CapHeight** values are treated a bit oddly. Usually they are interpreted as 16ths of the line width; but if you give an explicit dimension (with units) then **mkdrawf** will do the best it can to accommodate you. This doesn't work if the dimension isn't given literally; if you say **CapWidth \$x** then **\$x** will be taken as a multiple of $\text{\$i}\{\text{width}\}/16$. (Which won't make any sense at all if you specified **\$x** as a dimension!)

7.8. The Sprite object

I'm afraid the way you specify a sprite is pretty nasty; this is more or less inevitable, but it's possibly nastier than it ought to be. A sprite object description contains a **BoundingBox** item (like that in the drawfile header), and a load of numbers. I strongly advise you to specify the numbers in hexadecimal; don't forget that if you do this the 0x on the front is not optional!

The bounding box is not optional. It specifies where the sprite is to be placed, and how big it is to be.

The numbers are simply the words that will be placed into the file, in order. See the PRMs for details of the sprite format.

Here's an example.

```
Sprite {
  BoundingBox 100 100 200 200
  0x000002D4 0x745F6369 0x6C706D65 0x00006465
  several lines of hexadecimal rubbish deleted
  0x00000077
}
```

As of version 2.20, there is another way of specifying a sprite: the BoundingBox item is still needed, but instead of the load of numbers you write `FromFile` followed by two strings. The first should be the name of a sprite file, and the second the name of a sprite in that file. The resulting behaviour should be obvious. I recommend that you use the old method where possible, because this keeps all the information necessary to recreate the drawfile in one place.

7.9. The Group object

Here's a typical Group object description.

```
Group {
  Name "ThisIsAGroup"      # maximum 12 characters
  here
  Text {
    Size 10 10
    StartAt 100 100
    Style 2
    Text "Foo"
  }
  Group {                  # one group may contain
  another
    Path {                 # All the path style
```

```

options
    # have sensible defaults
    Move 100 100
    Line 200 100
}
Path {
    FillColour r200g200b200
    Move 200 200
    Line 300 300
    Line 100 300
    Close
}
}
name # a group doesn't need to be given a
}

```

7.10. The Tagged object

A tagged object, remember, consists of a 4-byte tag identifier, an object, and some other (optional, word-aligned) data. Here's what it looks like:

```

Tagged {
    Identifier 0x12345678
    Text {
        blah blah blah
    }
    OtherData 0x9ABCDEF0
    OtherData 0x11111111
    OtherData 12345 # this is decimal 12345
}

```

Strictly speaking, you should register tag identifiers with Acorn. Unless you're planning to use them in a commercial product, there's not much need to bother, since they aren't used much. Just pick a random number and hope it doesn't clash with anything.

7.11. The TextArea object

Here's a fairly complete text area object description. The only incompleteness is in the fact that there should really be some escape sequences in the text.

```

TextArea {
    Column 100 400 200 500
    Column 220 400 320 500
    Colour r0g0b0
    Background r255g255b255
}

```

```

Text {
    Really there should be some escape sequences at
the start
    of this; too bad. This text will be formatted
in the
    2 columns specified at the start of the obj
description,
    or would be if the escape sequences were there.
}
}

```

Note that the text is *not quoted*. The processing of text in a text area is an exception to the normal way in which input is processed: it is just read character by character, and terminated by the first line containing nothing other than a closing brace and whitespace. The extra space at the starts of lines in the example above isn't a problem, because multiple whitespace characters are treated as single spaces in text areas (unless there's more than one consecutive newline; see later).

This unusual processing has certain unfortunate consequences: you cannot get anything into the text of a text area using macros or variables, and comments don't work. Also, anything following the open-brace token after Text and on the same line is simply ignored.

As of version 3.10, that last paragraph is no longer quite true because there's another way of specifying the text to go in a text area: the **XText** keyword, followed by a list (in braces) of strings. These strings need not be literal; they can, for instance, be the result of using **Num2Str** or **GSTrans**.

All the columns must come before the text.

Multiple spaces and tabs are treated as single spaces. A single newline is treated as a single space (including absorption into multiple spaces); extra newlines insert extra vertical space (for paragraphs). Newlines occurring before any printing text are considered to be multiple newlines for this purpose.

Here's a list of escape sequences. Each starts with a backslash; the escape sequences themselves are case-sensitive, but their arguments aren't. Any argument may be terminated with a /, and variable-length ones must be (except that a newline will do instead). Dimensions should be given in points.

\! <version number>

This must appear at the start of the text, and may not appear anywhere else. The version number must be 1, and must be followed by / or a newline.

\A<code>

Determines alignment. <code> is L, R, C or D, indicating left, right, centred or double (i.e., L and R). Default is L.

\B<colour>

Set background colour hint for following text. <colour> might be 200 3 150; range is 0...255, at least one space is needed between successive numbers, and a / or newline is needed at the end.

\C<colour>

Set foreground colour for following text.

\D<number>

Indicates number of columns. This must be given, before any printing text, unless the number is 1.

\F<fontspec>

Defines a font reference number. Typical use: \F 1 Trinity.Medium 12 means number 1 is Trinity Medium, 12 point. Font numbers are 1 or 2 digits, and are entirely separate from the numbers used for text objects. You may specify a font width as well: \F 1 <name> 12 10 means 12pt high, 10pt wide. Don't forget the slash or newline. (This last goes for \C, \D, and most other escape sequences. I shall now stop mentioning it.)

\<number>

Means "now use font number ...".

\L<leading>

Defines the leading; i.e., the vertical distance between baselines of successive lines of text. The default is 10pt; specify it in points.

\M<lt><spaces><rt>

Margin sizes; must be positive; default is 1 (point).

\P<leading>

Paragraph leading; i.e., the amount of extra space left from each newline after the first in any consecutive sequence. Default is 10pt.

\U<ulspec>

Switches on underlining. \U 10 2 means 10 units above (or is it below?) the baseline, 2 units thick. One unit is 1/256 of the font size. Ranges - 128...127, 0...255 respectively. Switch off either by giving a thickness of 0 or by using "\U.".

\V[-]<digit>

Vertical move. In points.

\-

Allow a line break here, inserting a hyphen if one happens (and nothing otherwise).

\<newline>

Force a line break here.

\;

Everything from here to next newline is ignored.

Note: there is no such thing as a TextColumn object descriptor. Text column objects only occur inside text area objects, and are produced automatically by **mkdrawf**.

7.12. The Options object

Here's an options object description illustrating just about everything. Almost all of this is specific information for **!Draw** telling it how to behave when it has the file loaded.

```
Options {
  PaperSize 4                      # meaning: A4
  Limits {
    Shown                          # not sure what this
does.

    Landscape                      # Omit for portrait
    NonDefault                    # not sure what this
does.

  }
  Grid {
    Spacing 1.2345                # cm, unless "Inches"
given;
                                # see below
    Divisions 5
    Isometric                     # omit for rectangular
    AutoAdjust                   # Omit to turn this off
    Shown                        # Omit for invisible
grid
    Lock                         # Omit for grid lock off
    Inches                      # Omit for centimetres
  }
  Zoom {
    Ratio 3 2                    # means 3:2. Both
numbers
                                # should be in range
1..8
    Lock                        # to powers of 2. !Draw
seems
                                # to ignore it anyway
  }
  NoToolbox                      # omit to have toolbox
shown

  Mode Line                      # when file loaded
                                # initial entry mode.
                                # Possibilities are
                                # Line,ClosedLine,Curve,
                                # ClosedCurve,Rectangle,
                                # Ellipse,Text,Select.
```

```

UndoSize 1234           # size of undo buffer,in
bytes
}

```

7.13. The *XfText* object

That is, transformed text. An **XfText** object description is just like a **Text** object description, but it has up to three extra items. The most important is one that looks like

```
Matrix 0.5 1 2 -0.1 30 0
```

which means that the text should be transformed by the 2x2 matrix given as the first 4 numbers, and then translated by the vector given as the last 2. To be more precise, the transformation done is

$$\begin{pmatrix} x & y \end{pmatrix} \rightarrow \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} x & y \end{pmatrix} .$$

The other two items available are “flags”: **Kerned** and **RightToLeft**. You can probably work out what they do. Incidentally, I think text in text objects is never kerned, and text in text area objects is always kerned.

The position of the text in an **XfText** object may be specified using **CentreIn** or **CentreAt**; as of version 3.09 you can even use **HCentreIn** or **HCentreOn**, but the results aren’t guaranteed to be sensible if the Matrix is non-trivial.

7.14. The *XfSprite* object

An **XfSprite** object is just like a sprite object except that it can (and should) also contain a transformation matrix, exactly like those used for **XfText** objects. The sprite itself can (as for sprite objects) be given either directly or in terms of a sprite file and a sprite name.

The placing and size of a transformed sprite are not determined by the bounding box, but by the transformation matrix and the contents of the sprite itself. However, **mkdrawf** doesn’t understand sprites (it just treats a sprite as a lump of data), so it can’t make any sort of attempt at computing a bounding box. So give it one.

For the record, an identity matrix with zero translation puts the sprite’s bottom-left corner at (0,0) and its top right corner at (w,ex,h,ey) where w,h are the width and height of the sprite in pixels and ex,ey are the “eigenfactors” for the mode in which the sprite was defined.

7.15. The *JPEG* object

A **JPEG** object is in several ways rather like a transformed sprite object. Here is a “complete” **JPEG** object description:

```
JPEG {
  BoundingBox 100 100 300 200
  Size 200 100                # this is needed
  DPI 90 45                   # x,y. Default is
90,90
  Matrix 2 0 0 1 100 0
  FromFile "a_jpeg"
}
```

The **Size** is the size of the image before the transformation matrix is applied. (This could in principle be computed from the image size in pixels, embedded in the **JPEG** data, and the **DPI** values; the point is to avoid having to look inside the **JPEG** data.)

If you give a **BoundingBox** but no **Matrix**, then **mkdrawf** will provide a suitable matrix, scaling and translating the image so that it exactly fits the bounding box. If you give a **Matrix** but no **BoundingBox**, then **mkdrawf** calculates the bounding box. (In fact, this still happens if you give neither. In that case, the matrix used is the identity.)

Instead of giving a filename, you can in principle include the contents of the **JPEG** file in the object description (as with sprites and transformed sprites), but **JPEG**s are usually big enough that you really don’t want to do this. If you do do this, you need to provide an item like

```
Length 12345
```

giving the exact length of the **JPEG** image in bytes; **mkdrawf** will not try to guess from the number of integers you give, because the size of a **JPEG** image need not be a whole number of words.

A reminder: most versions of **!Draw** don’t understand **JPEG** image objects.

8. Some sample drawfile descriptions

Here are some examples of what can be done with **mkdrawf**. Each diagram was produced by running **mkdrawf** on the appropriate bit of code and dropping the drawfile into Impression. (If you have a version of this manual without illustrations, you can probably find them as drawfiles in the same directory as this file. If you’re reading a printed version of the manual without illustrations, you’ll need to use your imagination.)

8.1. The Koch snowflake curve

The fractal everybody knows. Dead easy. This takes just under 8.4 seconds on my machine, by the way.

The fractal everybody knows. Dead easy. This takes just under 8.4 seconds on my machine, by the way.

```
# Snowflake curve
Set $r3/2 Over Sqrt 3 2
# Side { %x0 .. %y0 .. %x1 .. %y1 .. %n .. }
# this should be inserted into a Path
# The assumption is that we're
# already "at" (x0,y0).
Define Side {
  IfLess %n 1
    Line %x1 %y1
  Else
    Set %dx Over Minus %x1 %x0 3
    Set %dy Over Minus %y1 %y0 3
    Set %xa Plus %x0 %dx
    Set %ya Plus %y0 %dy
    Set %xb Minus %x1 %dx
    Set %yb Minus %y1 %dy
    Set %xh Over Plus %x0 %x1 2
    Set %yh Over Plus %y0 %y1 2
    Set %xt Plus %xh Times %dy $r3/2
    Set %yt Minus %yh Times %dx $r3/2
    Set %m Minus %n 1
    Side { %x0 %x0 %y0 %y0 %x1 %xa %y1 %ya %n %m
  }
    Side { %x0 %xa %y0 %ya %x1 %xt %y1 %yt %n %m
  }
    Side { %x0 %xt %y0 %yt %x1 %xb %y1 %yb %n %m
  }
    Side { %x0 %xb %y0 %yb %x1 %x1 %y1 %y1 %n %m
  }
  EndIf
}
Set $x0 100 Set $y0 100
Set $x1 400 Set $y1 100
Set $x2 250 Set $y2 Plus 100 Times 300 $r3/2
Path {
  Move $x0 $y0
  Side { %x0 $x0 %y0 $y0 %x1 $x1 %y1 $y1 %n 5 }
  Side { %x0 $x1 %y0 $y1 %x1 $x2 %y1 $y2 %n 5 }
```

```

    Side { %x0 $x2 %y0 $y2 %x1 $x0 %y1 $y0 %n 5 }
    Close      # not really needed unless you want to
fill it
}

```

That example was written in the bad old days, before named positional parameters and infix expressions. Here's how I'd write it now:

```

# Snowflake curve
( $r3/2 := ( sqrt 3 ) / 2 )
# Side [ x0 y0 x1 y1 n ]
# this should be inserted into a Path
# The assumption is that we're
# already "at" (x0,y0).
define Side [ %x0 %y0 %x1 %y1 %n ] {
  IfLess %n 1
    Line %x1 %y1
  Else ( (
    %dx := ( %x1 - %x0 ) / 3    %dy := ( %y1 - %y0 )
  / 3
    %xa := %x0 + %dx           %ya := %y0 + %dy
    %xb := %x1 - %dx           %yb := %y1 - %dy
    %xh := ( %x0 + %x1 ) / 2   %yh := ( %y0 + %y1 )
  / 2
    %xt := %xh + $r3/2 * %dy   %yt := %yh - $r3/2 *
%dx
    %m := %n - 1
    Side [ %x0 %y0 %xa %ya %m ]
    Side [ %xa %ya %xt %yt %m ]
    Side [ %xt %yt %xb %yb %m ]
    Side [ %xb %yb %x1 %y1 %m ]
  ) ) EndIf
}
( ( $x0 := 100  $y0 := 100
    $x1 := 400  $y1 := 100
    $x2 := 250  $y2 := 100 + 300 * $r3/2 ) )
Path {
  Move $x0 $y0
  Side [ $x0 $y0 $x1 $y1 5 ]
  Side [ $x1 $y1 $x2 $y2 5 ]
  Side [ $x2 $y2 $x0 $y0 5 ]
  Close      # not really needed unless you want to
fill it
}

```

8.2. A Lissajous figure

You might have produced things slightly like this at school by connecting a couple of signal generators to an oscilloscope. (Except I bet it didn't do the pretty chequered pattern.) The drawfile this produces could certainly be made smaller by using Bezier curves instead of straight lines, but it's not very big anyway. **mkdrawf** takes about 3.1 seconds to process this file on my machine.

```
( $factor := $pi / 400 )
Path {
  Fillcolour r255g0b0
  Style { WindingRule EvenOdd }
  for $i from 0 to 800 {
    ( (
      $j := $i * $factor
      $p := cos ( 9 * $j )
      $q := sin ( 13 * $j )
    ) )
    ifequal $i 0 Move else Line endif
    ( 400 + 300 * $p )
    ( 400 + 300 * $q )
  }
  Close
}
```

8.3. A typical text area

This drawfile contains nothing but a text area. I apologise for the inanity of the text in it; this was typed in at random one evening while I was testing an earlier version of the program, and I just wanted a certain amount of text.

```
TextArea {
  Column 100 400 200 500
  Column 220 400 320 500
  Colour r0g0b0
  Text {
    \! 1
    \F 1 Trinity.Medium.Italic 12
    \F 2 Trinity.Medium 12
    \1
    \AD
    \D2
    \L12
    This is some text I'm putting in a text area.
    I have no idea how it will look, nor indeed
    whether
  }
}
```

```

        it will work at all. For all I know \2mkdrawf\1
will
        just choke utterly on it, or corrupt my file,
or
        cause demons to fly out of the monitor.
        This should be a new paragraph; it will still
be
        in italics. \2 Now we should be in roman type.
        (Isn't this fun, boys and girls?)

        Apparently the 1998 World Cup will be decided,
        in the event of a draw, by a sudden-death
playoff
        instead of by a penalty shootout. How
interesting.
    }
}

```

8.4. Circular arcs using Bezier curves

The following code does a little more than the new Arc pseudo-path-element, much less elegantly. That is, It produces (an approximation to) an arc of a circle using Bezier curves. I don't know whether the particular approximation it uses is the same as the one used in **!Draw**, but it gives good results. This takes about 0.5 seconds on my machine, by the way.

I'm leaving the code in its ghastly pre-3.10 form (prefix notation, no named positional parameters). Aren't you glad you don't have to write code like this any more?

```

# %x0,%y0 is centre
# %r      is radius
# %a0,%a1 are starting and ending
#          angles in radians
# %n      if set, indicates how many
#          arcs to use
Define Arc1 {                                # can't call it Arc any
more!
    Set %da Minus %a1 %a0
    IfExists %n Else
        Set %n Floor Plus .99 Over %da .75
    EndIf
    Set %da Over %da %n
    Set %t Over Minus 1 Cos Over %da 2
        Times .75 Sin Over %da 2
    Move Plus %x0 Times %r Cos %a0

```

```

        Plus %y0 Times %r Sin %a0
Set %aa %a0
Set %cc Cos %aa
Set %ss Sin %aa
For %k 0 %n {
    Set %a %aa
    Set %aa Plus %a %da
    Set %c %cc          Set %s %ss
    Set %cc Cos %aa     Set %ss Sin %aa
    Curve Plus %x0 Times %r Minus %c Times %t %s
        Plus %y0 Times %r Plus %s Times %t %c
        Plus %x0 Times %r Plus %cc Times %t %ss
        Plus %y0 Times %r Minus %ss Times %t %cc
        Plus %x0 Times %r %cc
        Plus %y0 Times %r %ss
    }
}
Set $Pi 3.141592653589793 # unnecessary now
Path {
    Arc1 { %x0 300 %y0 300 %r 200 %a0 0 %a1 Times 2
$Pi %n 2 }
    # very inaccurate: only 2 arcs
    Close
}
Path {
    Arc1 { %x0 300 %y0 300 %r 200 %a0 0 %a1 Times 2
$Pi }
    # I think this does 9 arcs
    Close
}

```

8.5. A silly spiral

I first drew pictures like this on an old Hewlett-Packard desktop computer, back in 1982 or so. I still think they rate quite highly for prettiness-to-effort ratio. This is really an excuse for demonstrating **RLine**. Oh, and it takes about 2s.

```

( $conv := $2pi / 360 )
define Spiral {
    ( ( %theta := 0 %r := %dr ) )
    For %i from 0 to ( %n - 0.5 ) {
        RLine ( %r * cos ( $conv * %theta ) )
            ( %r * sin ( $conv * %theta ) )
        ( ( %r := %r + %dr %theta := %theta + %dt ) )
        IfLess 360 %theta
            ( %theta := %theta - 360 )
    }
}

```



```
        EndIf
    }
}
Path {
    Outlinecolour r0g0b0  Width 0
    Move 250 250
    Spiral { %dr .5 %dt 65 %n 500 }
}
```

9. Tagfiles

Note: This describes an optional feature of **mkdrawf**, which is not included in the ready-compiled version of **mkdrawf** distributed with this manual. You'll have to recompile if you want to use it. This is because I expect most people not to need it — it was added because one particular user asked for it. If you don't use it, why should it take up space in your executable? To make a version of **mkdrawf** with tagfiles enabled, compile **mkdrawf** with the preprocessor symbol **TAGS** defined.

Suppose you have a **mkdrawf** “script” which gets used repeatedly, but with slight changes made to some of the strings. (If you don't, you don't need tagfiles.) You could go through and change the file every time, but that would be tedious. You could have the strings put into variables by an Included file, and use different files on different occasions, but that would be a waste of variables. The tagfile feature provides an alternative.

A tagfile consists of a number of lines, each containing a tag and an associated value. There is a **mkdrawf** operator which takes a string, looks for a matching tag and replaces it with the corresponding value. There are also **mkdrawf** “specials” for opening and closing tagfiles. This is getting terribly abstract: here's a concrete example.

A tagfile:

```
One: un
Two:deux
# This is a comment; it will be ignored.
Saturday:samedi # ceci n'est pas une remarque
ThoughtPolice:Academie Francaise
A mkdrawf file:
TagOpen "tagfile"
...
Text { ...
    Text Append 3 "The "
                        TagLookup "ThoughtPolice:FBI"
                        " will get you!"
}
...
TagClose
```

This will behave exactly as if you'd had instead a line saying

```
Text "The Academie Francaise will get you!".
```

If your tagfile contained a line

```
ThoughtPolice:Child Support Agency
```

instead of the one above, then the effect would be as if the Text item was

Text "The Child Support Agency will get you!".

If it didn't contain any ThoughtPolice line at all, you'd get

Text "The FBI will get you!".

You are allowed to miss off the default (":**FBI**" above) in a **TagLookup**; if the tag is not found you will get an error message and the expansion of the **TagLookup** will be the empty string.

You may only have one tagfile "open" at a time. (In fact it is only really open while the **TagOpen** is doing its work; its contents are read into memory, and **TagClose** means "free the memory used for tagfile data".) Its lines are, in effect, searched in order when looking for a tag.

The similarity between tagfiles and RISC OS message files is deliberate. However, tagfiles don't provide any of the fancy wildcarding features provided by **MessageTrans**. They may do in the future.

If you put "-t foo" on the **mkdrawf** command line, the effect will be as if the very first line of the file had been **TagOpen "foo"**. This can be useful if you want to run something with several different tagfiles without changing the **mkdrawf** file itself; for instance, in makefiles.

10. Technical issues

If you're interested in **mkdrawf**, you're quite likely to be interested in its internals. (Wimps use !Draw. Real Men use **mkdrawf**. Or something.) So here is some information about how it works, with particular emphasis on things that need improving. (After all, you might be able to make it better...)

10.1. Memory management

Oh dear.

This is a real mess, basically because it's seldom trivial to know when a piece of memory is finished with and can be thrown away. Every time a string is created, a new block of memory is **malloced**; these blocks are never freed. (Once upon a time this was true for numbers as well!) There are some places in the program where we could free things, but there are so many memory leaks that it is probably more sensible to wait until I have the time and energy, and fix it all properly.

The real problem is that there is no guarantee that any particular token will not get included in a token list. If that happens, we can't afford to throw it away. So we can't, for instance, always free the memory used by an old variable value.

One solution to this problem would be to make sure that we always copy tokens when we use them, but this would be terribly time-inefficient. The best solution is

probably to implement a garbage-collector. Hmmm.

Fortunately, most drawfiles are quite small; and the amount of memory that gets wasted at any one time is also small. There's only likely to be a practical problem here if you do complicated recursive things. At the moment, that snowflake curve program needs about 280k (the **mkdrawf** program itself being 94k), which I feel is rather excessive. I'll work on it. Incidentally, the corresponding figure in version 2 of **mkdrawf** was more like 500k.

10.2. Variables

... are stored in a hash table. The way in which all this works has changed dramatically between version 2 and version 3; in version 2 there was a fairly big global hash table and each macro invocation had its own local hash table, whereas now there is a very big global hash table (which, by the way, accounts for the increase in size since version 2) and local variables are saved and restored as needed.

The hash scheme used is called the "method of coalescing lists". I stole it from Knuth, and stole his choices of hashing parameters too. In version 2 of **mkdrawf** there were two sorts of local variable token, "resolved" and "unresolved". This distinction no longer exists; in effect, all variables are now resolved.

10.3. Problems

This is quite a complicated program, and I'm not a very reliable programmer. There are bound to be a few bugs left, though I think I'd have spotted anything really big. Fortunately, I am both a fast bug-fixer and a helpful person; so if you spot any bugs and let me know, I can probably let you have a fixed version of **mkdrawf** pretty quickly.

Similar remarks apply if you have features to suggest. Of course I don't guarantee to implement every (or any!) feature that is suggested to me, but I am always open to suggestions.

10.4. Arbitrary restrictions

The following aren't bugs, but do represent limitations of which you should be aware. I don't guarantee robustness in the event of your violating these...

- Token lists cannot be nested more than 256 deep; nor (independently) can macros; nor (independently) can Fors; nor (independently) can Ifs.
- There must not be more than 2100 global things-with-names. That means keywords and specials (there are about 100 of these); global variables; (names of) local variables; and macros. Before version 3 of **mkdrawf**, there was a limit of 677 global things and 61 local variables per macro.
- The drawfile produced by **mkdrawf** must be at most 4 megabytes long.

- A macro may not use more than 9 positional parameters.
- Include files cannot be nested more than 17 deep.

10.5. The future

Here are some of the things I want to do to **mkdrawf** if and when I have the time.

- Better conditionals.

It is ludicrous to have several different kinds of **if**. Instead, we should have a single **if** token, and several predicates to use with it. It would then become possible to combine them with boolean operators, etc.

It would be nice to have **elseif** too, but there are some slightly non-trivial syntactic issues. Specifically, the parser needs to be able to recognise the end of the condition following **elseif** while skipping tokens. Maybe we should have a **then** token, serving only to identify the end of a condition.

- Better error handling.

At the moment, when anything goes wrong with a **mkdrawf** script it's terribly painful to fix because **mkdrawf's** error reporting is so very primitive. It would be nice to fix that.

In the longer term, it might be nice to stick the facilities of **mkdrawf** inside a better language. Python would be a pretty good choice, though it's rather slow. I have built most of a language system of my own for this purpose, and it's pleasant enough, but I'm not really convinced the world really needs yet another programming language just like all the others.

11. Revision history

mkdrawf has been changed a number of times, and will doubtless change more in the future. In case you're curious about what has happened to it (or want to see what's changed since the last version you saw), here is a brief account of the changes introduced with each new version. Not all of these versions actually correspond to public releases of **mkdrawf**. (There was a version 1, but it was very different to **mkdrawf** as it is now.)

2.00 First "new" version of **mkdrawf**; many incompatible changes in input format; programming features added; manual written.

2.10 Sprites and transformed sprites added.

2.20 Sprites and transformed sprites can be loaded from sprite files; Include feature; much improved error handling; several minor code changes.

2.30 Bounding boxes of paths computed using **Draw_ProcessPath** instead of by computing bbox of points on path; optional tagfile features; **Append**, **GSTrans** features; almost all objects can have explicit

BoundingBoxes; numerous bugfixes; code made rather cleaner; **decdrawf** -s and -v.

2.35 **Font**, **Str2Num**, **Num2Str**, **HCentreIn**, **CentreIn**, **Random** features; implicit font tables; colours as triples of integers; **JPEG** objects; several minor bugfixes and code changes; tutorial written; **!Drawf** written, and some minor changes made to **mkdrawf** to support it.

3.00 Complete rewrite of macro-expansion code, improving speed and memory use and fixing a serious bug with nested macros. General reorganisation of the source code (now in several files).

3.10 Units (via Units and suffixes on numbers); symbolic colour names. Virtual text objects and text bounding-box variables. Various new mathematical operators. New syntax for macro invocation. Clarification of scoping rules;

IfExistsHere. **decdrawf -x**. Relaxed limit on text area columns. Friendlier handling of **JPEG** objects. Circular arcs in paths. **XText** in text areas. New kinds of For loop. Infix notation.

12. Decoding drawfiles

With **mkdrawf** comes another program, called **decdrawf**, which turns drawfiles into descriptions of the sort understood by **mkdrawf**. This is useful if you've lost the **mkdrawf** source code for a drawfile, or if you want to sketch something roughly and then tweak it so that all the points are in just the right places.

12.1. Using *decdrawf*

... is trivial, even easier than using **mkdrawf**. Just type, say,

```
decdrawf foo
```

to get the contents of the drawfile named foo displayed. If you want the output sent to a file, type

```
decdrawf foo > bar
```

and the output will be put in a file called bar.

As with **mkdrawf**, you can say

```
decdrawf -v
```

to get a version number (which will keep in step with the version number of **mkdrawf**, even though one program may change without any change in the other). More importantly, you can also request that **decdrawf** put sprites in a separate sprite file rather than “inline” in hexadecimal, by putting **-s <filename>** on the command line. If there are no sprites in the drawfile, the sprite file will not be created.

Warning: A drawfile may contain several copies of one sprite, or (worse) several different sprites with the same name. **decdrawf** will not output multiple copies of duplicated sprites to a sprite file; where different sprites have the same name, **decdrawf** will write them all to the sprite file as they occur, but will emit warning messages indicating that this is happening.

12.2. The output from *decdrawf*

is a file suitable for feeding into **mkdrawf**. It doesn't use any macros, variables etc, of course. Here and there in the output are comments, which are intended to make clearer the correspondence between the data in the drawfile and its textual description, or (in, for instance, the case of the Options object) to remind you what alterations you could make.

If you give the **-s <filename>** option to **decdrawf**, you will (if there actually are any sprites in the drawfile) get two output files. If you change the name of the sprite file, you will need to change the relevant **FromFiles** before feeding the result back to **mkdrawf**. Also, when you use **-s** you are not actually guaranteed that feeding the output into **mkdrawf** will produce the right results; see the warning above.

If there are JPEG image objects in a drawfile fed to **decdrawf** it will by default include them (in hexadecimal) in the decoded drawfile. This is inefficient, and **JPEG** images are usually rather large; so, instead, put **-j <prefix>** on the command line; this will make **decdrawf** put **JPEG** images in separate files named **prefix01** and so on. **decdrawf** makes no attempt to avoid overwriting existing files; one consequence of this is that you should be careful of truncation problems if a drawfile contains several **JPEG** images.

As with **mkdrawf**, you can get **decdrawf** not to tell you the filename it's working on in warnings and errors by giving it the option **-e**.

You can make **decdrawf** produce output in terms of units other than points, by putting **-u <unit>** on the command-line. Here unit is one of **sp**, **os**, **pt**, **mm**, **cm**, **in** (meaning scaled points, OS units, points, millimetres, centimetres, inches). The short names are used here to save typing; the long ones are used by **mkdrawf** to avoid clashing with potentially useful short variable names.

If you're grovelling through a drawfile by hand as well as looking at **decdrawf** output, you will want to see all the dimensions in their full 32-bit hexadecimal glory. To do this, put **-x** on the command line. This implies **-u sp** (and if you specify any other units explicitly, disaster may ensue).

Symbols

!Draw 4, 9, 23

\$Centimetres 23

\$Millimetres 23

\$OSUnits 23

\$Points 23

\$ScaledPoints 23

A

Abs 20

Academie Francaise 42

affine transformation 11

alignment 32

anonymous 18

anti-aliasing 9

Append 24

Apple Macintosh 14

Arc 29, 39

arc 6, 17

Arccos 20

Arcsin 20

Arctan 20

Arctan2 20

B

background colour 9

background colour hint 32

bells 11

Bezier curve 10

Bounding box 25

bounding box 14

bounding box variables 27

bounding boxes 26

BoundingBox 25, 29

brace 25

Braces 13

bugs 44

C

CapHeight 29

CapWidth 28

Ceiling 20

CentreAt 27

CentreIn 27

Colour 14

columns 10, 32

Comments 13

complaints 5

Conditionals 21

Conversions 21

Cos 20

Creator 25

D

dash pattern 28

decdrawf 4

Define 16

diagnostic 7

dimensions 13

Draw 12

drawfile 8

E

eigenfactors 35

Else 21

EndIf 21

entry mode 11

Errors 7

escape sequences 32

expandable token 16

expansion 16

explicit dimension 14

F

Floor 20

Font 21

FontTable 26

S

Sign 20

V

virtual 27

Virtual 27, 46