[3] Documentation — bibliographic references — content, form and structure. ISO 690, International Organization for Standardization, 1987.

[4] Frank G. Bennett, Jr. LexiTEX: a LATEX macro package for lawyers. Document deposited in electronic archives, 1993.

[5] Judith Butcher. *Copy-editing.* Cambridge University Press, 3rd edition, 1992.

[6] *Publication Manual of the American Psychological Association.* American Psychological Association, 3rd edition, 1983. Obtainable from: American Psychological Association, P. O. Box 2710, Hyattsville, MD 20784.

[7] *The Chicago Manual of Style.* University of Chicago Press, 13th edition, 1982.

[8] Janet S. Dodd. *The ACS Style Guide.* American Chemical Society, 1986.

[9] *MHRA Style Book.* Modern Humanities Research Association, 4th edition, 1991.

[10] Joseph Gibaldi and Walter S. Achtert, editors. *MLA Handbook for Writers of Research Papers.* Modern Language Association of America, 3rd edition, 1988.

[11] International Committee of Medical Journal Editors. Uniform requirements for manuscripts submitted to biomedical journals. *British Medical Journal*, 302:340–341, February 1991. Note: This article was also published in the *New England Journal of Medicine* (7th Feb. 1991). It specifies the "Vancouver style" for manuscript-preparation, which is accepted by over 400 journals.

[12] Citing publications by bibliographic references. BS 5605, British Standards Institution, 1978.

[13] References to published materials. BS 1629, British Standards Institution, 1989.

[14] Citation of unpublished documents. BS 6371, British Standards Institution, 1983.

[15] Oren Patashnik. BibTeXing. Document deposited in electronic archives, January 1988.

[16] James C. Alexander. Tib: A TEX bibliographic preprocessor. Document deposited in electronic archives, 1989.

[17] Sue Stigleman. Bibliography formatting software: an update. *Database*, February 1993.

◇ David Rhead
Cripps Computing Centre
University of Nottingham
University Park
Nottingham NG7 2RD England,
U.K.
David_Rhead@vme.ccc.nottingham.ac.uk

Relative moves in LATEX pictures

Richard Bland

1 Introduction

In this note I hope to do three things:

1. Make a number of observations about why picture-drawing in LATEX, as described by Lamport, is so difficult and unpleasant.
2. Put forward a suggestion for a very simple mechanism to overcome at least some of these difficulties.
3. Show one way of implementing this suggestion, using the Unix utility `m4`. This particular implementation is presented only to demonstrate the simplicity of the underlying mechanism: no claim is made that it is an optimal implementation.

2 An example

Consider the simple picture in Figure 1. As is obvious, this picture has no meaning: it is just a collection of graphic elements such as labelled shapes, text strings, lines and arrows: but it does exemplify the kind of output which many users have in mind when they set out to draw a picture in LATEX. Such users want some form of diagrammatic representation in which different shapes are used to represent types of entity, lines and arrows are used to connect the entities, and labelling is used to give some domain-specific meaning. Often these pictures are conceptually quite simple.

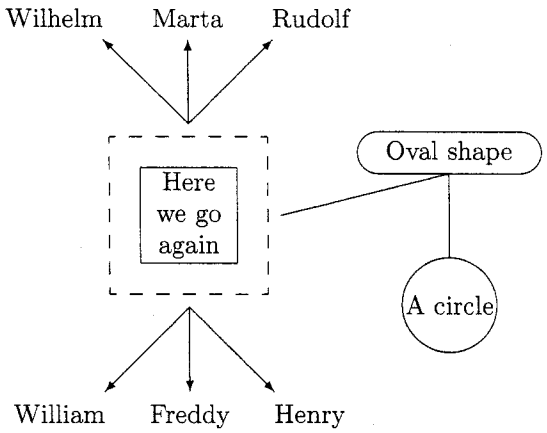

Figure 1: A LATEX picture

How does one produce pictures like this? Many people would suggest using an interactive drawing tool (on some suitable hardware) to produce an intermediate file which can be incorporated into the LaTeX source of a document (or added at some appropriate point downstream). I've never found this an agreeable approach, for two main reasons: first, as an occasional user I find it hard to come to grips with the supposedly intuitively-obvious interfaces which these tools present. After a certain point in one's career the fun of learning *another* system begins to diminish: the busy user who has learned one set of syntactic and semantic ideas (like those of LaTeX) would like to get results from those ideas rather than adding a new set. Second, in using an interactive drawing tool one often abandons or jeopardises some of the main reasons for using a markup system like LaTeX in the first place. These are, of course, portability, device-independence, and the ability to manipulate the source indefinitely with any number of the myriad tools which handle ASCII text. This last point is particularly important: because LaTeX source is just a character file it can be edited, cut, pasted, searched, burgled, extended, all without limit. This is certainly not the case with the behind-the-scenes formats of many drawing packages.

These considerations suggest that there are good reasons for trying to produce pictures with the LaTeX tools described by Lamport.

Now consider the commands which produced Figure 1. Slightly edited, they are as follows:

```
\begin{figure}[htb]
 \setlength{\unitlength}{1pt}
 \centering
 \begin{picture}(216,216)
  \put(48,78){\dashbox{5}(60,60){
     \begin{tabular}{|c|}
     \hline Here \\ we go \\ again \\
     \hline
     \end{tabular}}}
  \put(78,73){\vector(1,-1){32}}
  \put(110,36){\makebox(0,0)[tl]{Henry}}
  \put(78,73){\vector(-1,-1){32}}
  \put(46,36){\makebox(0,0)[tr]{William}}
  \put(78,73){\vector(0,-1){32}}
  \put(78,36){\makebox(0,0)[t]{Freddy}}
  \put(78,143){\vector(1,1){32}}
  \put(110,180){\makebox(0,0)[bl]{Rudolf}}
  \put(78,143){\vector(-1,1){32}}
  \put(46,180){\makebox(0,0)[br]{Wilhelm}}
  \put(78,143){\vector(0,1){32}}
  \put(78,180){\makebox(0,0)[b]{Marta}}
  \put(113,108){\line(4,1){64}}
  \put(177,132){\oval(70,16)}\put(177,132)
                {\makebox(0,0){Oval shape}}
  \put(177,124){\line(0,-1){32}}
  \put(177,74){\circle{36}}\put(177,74)
                {\makebox(0,0){A circle}}
 \end{picture}
 \caption{A \LaTeX\ picture
   \label{exampfig}}
\end{figure}
```

What can we say about this? Well, readers of *TUGboat* presumably have strong stomachs, but even those who read *The TEXbook* for fun will surely realise that these instructions are awful. Users who set out to produce pictures using this sort of apparatus will very soon become discouraged. In the next section we try to analyse the problem with these instructions.

3 The difficulties

Looking at the code above, we can draw three main conclusions. First, the *syntax* of the instructions is very complicated and very hard to remember, making the instructions extremely hard to write unless one has a model immediately to hand. Also, there seem to be inconsistencies. For example, the parameters for `\oval` are in round brackets while the parameter for `\circle` is in curly brackets, although they are semantically equivalent — in each case the parameter(s) give the size of the shape to be drawn.

Second, the code is stuffed full of literal numeric *constants*. This immediately makes users with a programming background uneasy. After all, one of the things we are always told (or are telling others) is *not* to use constants. Because they convey no semantic information they make code hard to read: because we have to change *every semantically-equivalent instance* of a constant in order to edit code, the code is hard to change without making mistakes. In this case there is the additional difficulty that we suspect that the author of the code must have sweated blood in order to work out what all these constants ought to be in the first place: in our mind's eye is an image of Lamport crouched over his quadrille paper, cursing.

Third, the picture is composed in terms of *absolute* positions rather than *relative* positions. We realise that if we were to try to move the components of the picture in relation to one another, it would be very hard to do so by editing the absolute positions (the pairs of values in all the `\put` instructions).

4 Existing remedies

Some of these problems can, of course, be dealt with by sensible use of existing LaTeX facilities. We can make the literal constants into symbolic constants (using `\newcommand`), we can tinker a bit

with the syntax of repeated constructions (also using `\newcommand`) and we can modularise the picture (using nested `\picture` environments) and move the modules in relation to one another using offsets.

These solutions only go so far, however. Defining symbolic constants is fine: but one soon needs a facility for *arithmetic* in these definitions, which LaTeX lacks. For example, if one has a symbolic constant for the width of a box, you may need one for half the width as well. There's no easy way (that I know of) of defining one constant as a function of a previously-defined constant, so you must define them both literally: once again this makes changes difficult. Also, the scope for simplifying the syntax is quite limited because each command is still quite complicated semantically: 'at the point (177,132) draw an oval of size (70,16) and within it centre the string "Oval shape"' could certainly be more simply expressed, but not very much more simply. Finally, the method of modularising the picture by nesting `\picture` environments is useful, but has to be set out *very* carefully if the human reader is not to become hopelessly lost about the scope of the environments and hence about the offset to be applied to any particular position.

5 New remedies

There are two remedies which I wish to propose: one minor and one major. The minor one is to *make it easier to define symbolic constants as functions of other constants.* The major one is to *remove the 'position' information from the drawing commands.* The minor remedy really needs no further discussion at this stage: the only question to be settled is the method of implementation. The idea of taking the 'position' information out of the drawing commands is more complex.

The basic notion is to introduce the idea of a *current position* at which the next drawing action is to be done. Using macros, we re-package all the drawing operations which we wish to use, so as to

- Make them all draw at the current position. The re-packaged commands can now be simpler, because they no longer need position parameters.
- Give each of them a defined effect on the current position. For entity-representing shapes (boxes, ovals, circles, strings) the command will leave the current position where it is: for connectors (lines, arrows) the command will start drawing the line or arrow at the current position and end by moving the current position to the other end of the line or arrow.

Obviously we also need to add new commands to manipulate the current position: these will include

- An absolute `jump`, to move the current position to some new point.
- A relative `move`, to move the current position by an offset (it turns out to be convenient to have a family of these: four single-parameter moves, `up`, `down`, `left` and `right`, as well as a full two-parameter move).
- A method of 'remembering' the current position, and of resetting the current position to some remembered point.

One way of thinking about these commands, and of implementing the 'remembering' mechanism, is that we have introduced *position variables.* There's a behind-the-scenes position variable, the current position, which is global to all commands, and as many explicitly-named position variables as the user wishes. The only defined operations (so far, anyway) are those of assigning from a user-declared position variable to the current position and vice-versa.

The payoff turns out to be quite considerable. Our repackaged commands can be much simpler (for example, the command for an oval has three parameters instead of five). More importantly, the whole business of absolute positions (which gives the user so much difficulty) has now disappeared and been replaced by a much more natural idea of drawing one thing, moving relative to that thing and drawing another thing. This is what we do when we sketch naturally, on the backs of envelopes: we certainly don't work as Lamport recommends, "first pick[ing] the slopes of all lines, then ... *calculat[ing]* the position of each object before drawing it on the graph paper" ([3], page 110). The naturalness of this new approach is particularly obvious when the graph of the entities and connectives is a tree: in this case the new approach makes the picture simple to draw and very easy to change.

An example is needed here, but before we can present one, an implementation is needed. This is discussed in the next section.

6 Implementation

No doubt in an ideal world I would now present an implementation in TeX macros. In fact I shall not do this. For many years I have used the Unix macroprocessor `m4` ([1, 2]), which comes free with Unix and is in the public domain for MS-DOS. It has the facilities which we need (including arithmetic) and I know how to use it. Unfortunately I don't know how to write TeX macros.

Is this a problem? I believe not. My intention in this note is not to advertise a product but to discuss an approach. Although I shall of course be happy to share my few lines of code with anyone who wants them, my purpose here is to demonstrate that a particular approach can be made to work very easily and can greatly simplify a particular task. I hope that readers will be stimulated to suggest better or fuller implementations of the idea.

Using m4 means that the source file (a mixture of m4 statements and LaTeX statements) must be run through m4 before processing by LaTeX, but this step is easily arranged and has a negligible penalty in processing time.

In the following account I shall not show the full details of the implementation in m4 (although this is only a few dozen lines): I shall concentrate instead on explaining the commands which a user would need to know in order to draw the picture of Figure 1. In this account, I shall show macro names defined by me in capital letters, for clarity, and will follow the m4 convention of describing macro parameters as $1, $2, etc,, rather than the TeX convention of #1, #2, etc. I shall not attempt to give a rigorous account of m4, which is completely defined in [2]. As a working label, I'll refer to the set of macros written by me as the Macro Library for LaTeX Pictures MLLP, although this perhaps conveys an undue air of importance for a very few lines of code.

We begin by noting that in m4 we define a macro using the `define` macro, which takes two arguments, as in `define(HEIGHT,216)` which sets up the symbolic constant `HEIGHT` to be 216. This is the intended height of the picture — 216 points (which is about three inches). We also define other useful dimensions for the picture, whose meanings should be obvious: `WIDTH`, `BOXHEIGHT`, `BOXWIDTH`, `CIRCLEDIAM`, `OVALHEIGHT`, `OVALWIDTH` and `XARROWLEN`. This is done in the same way as for `HEIGHT` (but with different values, of course, the exact values of which aren't important for the exposition). We also define a useful quantity `SEPARATION`, which is defined as 5 (points) and is used as a general spacing parameter in the picture.

We can now write

```
\begin{picture}(WIDTH,HEIGHT)
```

as the start of the environment. The first thing we should like to do is draw the most significant element of the picture, the dashed box, slightly to the left of the midpoint of the picture. We can calculate this using the m4 built-in macro `eval`, which takes a conventionally-formed arithmetic expression as its argument and replaces it by an integer, the result of evaluating the expression. Before we do the sum, we first the constant `LEFTABIT` to be (say) 20 points, to move the box off-centre, and note that boxes are usually drawn with their bottom-left corner as the reference point: this means that we must jump to a point half a box-width to the left of, and half a box-height below, the chosen centre point of the box.[1] All of this is rather a mouthful. However, we can now set the current point to its starting position:

```
JUMP(eval((WIDTH-BOXHEIGHT)/2-LEFTABIT),
     eval((HEIGHT-BOXWIDTH)/2))
```

Now the dashed box. MLLP includes a three-parameter macro which draws a dashed box of size $1 by $2, with the (optional) $3 centered within it. This operation does not affect the current position. We can now write

```
DASHBOX(BOXHEIGHT,BOXWIDTH,
  `\begin{tabular}{|c|} \hline Here \\
   we go \\ again \\ \hline
   \end{tabular}')
```

demonstrating in rather a flashy way that the third parameter of `DASHBOX`, the object to be centered within it, can be a complicated LaTeX object. This is not exclusive to `DASHBOX`: the other macros in the set can also have complicated picture objects as parameters. Notice that to be on the safe side the parameter is wrapped in paired left and right single-quotes: this protects it from any unwanted processing by m4.

We now wish (say) to draw the cluster of arrows, and the associated strings, under the box. First we move the current point from the bottom-left corner of the box: in doing so we use another macro from MLLP, `HALF`, whose effect is obvious. Once we've arrived, we want to remember this position because it will be the base for three arrows, so we shall use the MLLP macro `SET` to hold the position.

```
RIGHT(HALF(BOXWIDTH))
DOWN(SEPARATION)
SET(`arrowbase1')
```

The string `arrowbase1` is the name of an MLLP position variable, as described above. It can be any identifier which won't interfere with LaTeX or m4. When acting as the parameter to `SET`, it needs to be in paired left and right single-quotes: this is for reasons internal to m4.

Now we draw an arrow and the string at the end of it. MLLP includes a three-parameter macro `ARROW`, which is just a packaging of Lamport's `vector`. The first two parameters give the slope and the third the length, just as described by Lamport

[1] Alternatively, one could make the centre of the box the reference point: but if you work it through this doesn't simplify things.

([3], page 106). The arrow is drawn from the current point *and the current point is moved to the head of the arrow.* There are two variants, ARROWUP and ARROWDOWN, which move the current point slightly away from the end of the arrow, either up or down: the length of the move is given by SEPARATION. The string at the end of the arrow is written using PUT, which is just a packaging of Lamport's put. The first argument is the string to be written. The (optional) second argument gives the relative position of the string with respect to the current point. The default is to centre the string round the current point, horizontally and vertically, but this can be changed by using the second parameter. Just as in Lamport, $2 can be 0, 1 or 2 of the letters t, b, l or r. These determine where the current point is with reference to the text. For example, tl means that the current point is at the top left of the text. PUT does not move the current point. So:

```
ARROWDOWN(1,-1,XARROWLEN)
PUT(Henry,tl)
```

The remaining two arrows in the cluster can be drawn easily once we note that JUMP will accept a position variable as its (single) argument. This of course resets the current point to the position stored in the position variable. Off we go:

```
JUMP(arrowbase1)
  ARROWDOWN(-1,-1,XARROWLEN)
  PUT(William,tr)
JUMP(arrowbase1)
  ARROWDOWN(0,-1,XARROWLEN)
  PUT(Freddy,t)
```

Drawing the top set of arrows doesn't require any new techniques: we move to the top of the box, establish a new arrow-base and draw the cluster.

```
JUMP(arrowbase1)
UP(eval(BOXWIDTH+2*SEPARATION))
SET('arrowbase2')
  ARROWUP(1,1,XARROWLEN)
  PUT(Rudolf,bl)
JUMP(arrowbase2)
  ARROWUP(-1,1,XARROWLEN)
  PUT(Wilhelm,br)
JUMP(arrowbase2)
  ARROWUP(0,1,XARROWLEN)
  PUT(Marta,b)
```

Given that our aim here is not to produce a reference manual for MLLP, or anything like it, it will perhaps be enough to leave the reader to infer from the code the properties of the remaining macros to be used, LINE, VLINE, OVAL and CIRCLE, given the information that OVAL and CIRCLE are drawn centred on the current point. We first move round to the right-hand side of the box, then draw the rest of the picture:

```
JUMP(arrowbase2)
  RIGHT(eval(BOXHEIGHT/2+SEPARATION))
  DOWN(eval(BOXWIDTH/2+SEPARATION))
LINE(4,1,eval(XARROWLEN*2))
%
UP(HALF(OVALWIDTH))
OVAL(OVALHEIGHT,OVALWIDTH,Oval shape)
DOWN(HALF(OVALWIDTH))
%
VLINE(-XARROWLEN)
DOWN(HALF(CIRCLEDIAM))
CIRCLE(CIRCLEDIAM,A circle)
\end{picture}
```

7 Conclusion

This note has attempted to identify a number of factors which make LaTeX picture-drawing a frustrating and error-prone business, and to suggest a simple approach which ameliorates those difficulties, and which can be implemented without much difficulty. An example has been presented: the code of this example is, I believe, strikingly easier to understand and to change than the original LaTeX code. Practical experience with a number of drawings has reinforced the belief that the approach presented here is simple and effective.

No claim is made that the implementation of these ideas in m4 is particularly elegant, or that the MLLP set of macros (which is larger than that shown above) is optimal or complete. I have, however, found it to be effective for my purposes. I should be very grateful for suggestions or comments on these points.

References

[1] Brian W Kernighan and P J Plauger. *Software Tools.* Addison-Wesley, Reading, Mass, 1976.

[2] Brian W Kernighan and Dennis M Ritchie. The m4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1977.

[3] Leslie Lamport. *LaTeX: a document preparation system.* Addison-Wesley, Reading, Mass, 1986.

◇ Richard Bland
Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA
Scotland