Very High Level 2-dimensional Graphics with TEX and XY-pic

Kristoffer Høgsbro Rose
BRICS, University of Aarhus (DAIMI), Ny Munkegade building 540, 8000 Århus C, Denmark
krisrose@brics.dk
URL: http://www.brics.dk/~krisrose/

Abstract

A problem with using pictures in TEX and LATEX documents is that there is no natural universal notation encompassing all possible diagrams, flow-charts, *etc.*In this paper we argue why one should not attempt such generality but rather design *custom embedded graphic languages* for classes of similar pictures.

The main argument is the usual one for markup languages: using a specialised high-level notation means that the source captures the essential properties of the picture. Not only does this make it easier for the user, who can concentrate on contents rather than form, but it also makes it easier to abstract out inessential style issues such that the "picture style" can be varied without changing the source. The main concern is that implementing a large number of such languages is only feasible with access to a versatile and powerful *drawing library* such that the amount of hacking required for each language is minimal.

As an example we survey how one can include a small "directory tree" in a paper, and we design and implement an embedded tree drawing language useful for this purpose. We illustrate the generality of the notation by showing how the same source can be used to generate the tree and even to *grow* it (by animating it) following the structure information. We finally present the implementation in TEX using XY-pic to produce the actual graphics.

Introduction

A common reason why skilled professionals working in technical areas choose TEX (Knuth, 1988) is that TEX makes it is easy to produce high quality drafts and to introduce corrections based on comments into these, making the edit-publish-feedback loop a fast and smooth one. This is further encouraged by the principle of "logical markup" promoted by formats such as LATEX (Lamport, 1994): this makes it possible to work with manuscripts with the focus on contents rather than form. In particular the facility for mathematical typesetting based on the structure of formulae means that authors can work with notions as they think about them in manuscripts, using familiar notations and groupings, deferring fine points of the typesetting until the latest moment without compromising the quality of drafts seriously, and — more importantly — without compromising the quality of the final version at all. Often it is even possible to use the same source for typesetting and other purposes. This is a very safe way of ensuring that the formulae appearing in a technical report are, indeed, exactly the same used in computations.

However, in contrast to this pleasant situation for formulae with textual structures, the treatment of simple, illustrative *pictures* is presently seriously lacking both in convenience and in the quality of the result. Even on the Internet there is no universally useful standard for "logical" specification of illustrative diagrams.[1] The reason for this is probably that it is rather easy to just compose a "quick and dirty little picture" by throwing together some arrows, boxes, lines, *etc.*, using a small visual drawing tool. That the result is usually excessively ugly is largely ignored (that proper composition of pictures was an immensely complicated task in traditional typography is likely to play a role in this matter).

Presently the following statements are representative for the options for 2-dimensional graphics that can be used with "portable TEX source documents" such as submissions, *etc.*; the choices are listed in approximate order of frequency in the author's experience.

[1] The situation is acceptable for complicated 3-dimensional drawing, however, where several successful standards exist; one of these, VRML, is even becoming an Internet standard.

"Graphics is not portable!" A textual approximation will have to do.

"One can do everything with rules!" One can do wonderful things drawing just horizontal and vertical lines using TEX rules.

"PostScript is portable graphics!" PostScript can be used to create the picture as an encapsulated PostScript (Adobe, 1990) file distributed along with the article source.

"METAFONT is portable graphics!" Since TEX is bundled with METAFONT all TEX installations should have a METAFONT engine. So all we have to do is draw the graphics with METAFONT (Knuth, 1986) and generate a font containing the drawing on each platform.

"Use a custom notation!" Designing an embedded language targeted at expressing the desired graphics directly in the TEX source without any constraints as to how the picture is actually drawn, is the only truly portable form of picture. It is easy to provide a macro package that makes actual pictures using the style of the context as far as possible.

Below we will first survey the five options for a particular *example* before we summarise the *design principles* for custom embedded languages and show how the information contained about a drawing in a well-designed language can be used for other purposes such as *animation*.

Survey by Example

Consider the following: the staff of a computing facility writes a monthly article where tradition has it that the "current directory structure" is included. The article is distributed to a number of departemental newsletter editors that are all published with TEX (of course). The various newsletters are published using a variety of styles and fonts and printed on all sorts of equipment, so portability is a crucial issue.

Avoiding graphics. The staff can use the first choice easily, *e.g.*, through the ouput of some standard tool. The UNIX *tree* command, *e.g.*, produces output as shown in figure 1. While this solution is ugly it is certainly completely portable, and in fact used more often than not.

```
xy-3.4
|-- doc
|   `-- xyguide-html
|-- mfinputs
|-- pkfonts
|   |-- ljfour600
|   |-- ljfour657
|   |-- ljfour720
|   `-- ljfour864
|-- ps
|-- psfonts
|-- src
|-- texfonts
`-- texinputs
```

Figure 1: UNIX *tree* output.

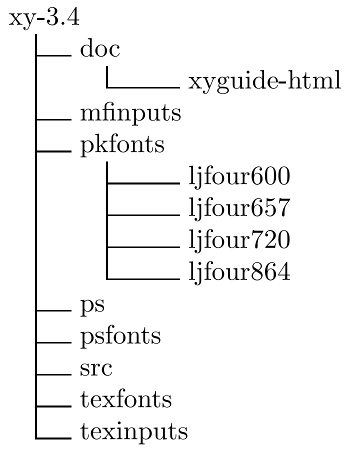


Figure 2: UNIX *tree* output with substitutions.

Using standard TEX rules. The second choice is almost as easy to realise and only requires a bit of hacking. Figure 2 shows what one can produce easily by substituting parts of the textual form with appropriate rules and spaces in a TEX `\halign`s construction. Such substitution essentially means that we translate the text directly into an appropriate TEX source representation which means that we need to know how to interpret the text in order to make a meaningful translation.

PostScript. The (encapsulated) PostScript choice is often the most practical, and is almost portable—only a few platforms cannot print PostScript and a few more cannot preview files with PostScript well. With PostScript the example tree might look as shown in figure 3. While the size of the figure can be changed by scaling it remains difficult to change the "Times" (or whatever) look of the figure, however, with an "embedded PostScript" package, such as the very powerful PSTricks (van Zandt, 1996), one can reduce this problem somewhat.

METAFONT. This choice is interesting in that all TEX installations are supposed to have METAFONT and thus compatibility is not a problem in principle.

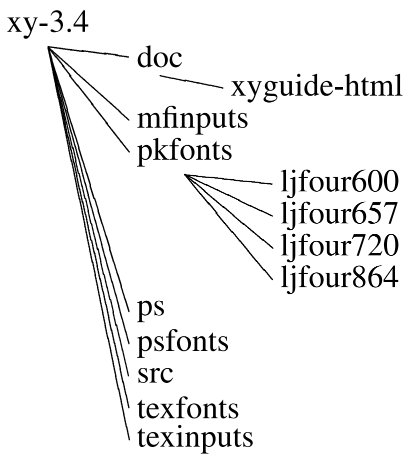


Figure 3: PostScript picture.

```
\tree{
  {xy-3.4} (
    {doc} (
      {xyguide-html} ()
    )
    {mfinputs} ()
    {pkfonts} (
      {ljfour600} ()
      {ljfour657} ()
      {ljfour720} ()
      {ljfour864} ()
    )
    {ps} ()
    {psfonts} ()
    {src} ()
    {texfonts} ()
    {texinputs} ()
  )
}
```

Figure 4: Directory as abstract tree.

However, it turns out that not all TEX installations can handle dynamic changes to the repertoire of fonts gracefully, thus in practice this is less of an option than one would hope. Two alternatives exist: using META O T instead relaxes the dynamic font problem somewhat at the penalty of requiring the use of PostScript, and using the mfpic package (Leathrum and Tobin, 1994) makes it possible to mix METAFONT-generated drawing with TEX-produced text. These are options that need to be further invesitigated. In general TEX would benefit greatly from a *component model* permitting interaction between the various graphic forms — but this is beyond the scope of this presentation.

Designing a custom embedded graphic language. The fifth and last option is obviously ideal — once a dedicated language exists for the kind of graphics in question, that is. The most obvious way to think of the directory structure is as a *tree* with a node for each directory (as hinted at by the UNIX command name). This leaves us with the problem of coming up with a good textual notation for trees. One possible such notation has been used in figure 4 using parentheses to express the directory nesting structure.

In this paper we will argue that designing such very high level drawing languages for *embedding* picture descriptions directly in the manuscript is an option worthwhile pursuing in many cases, even considering the initial cost of designing and implementing the language.

An Embedded Language

We first explain the principal properties of embedded languages and how this is reflected by our toy directory tree sample language; we then describe the implementation of it in TEX.

Principles. There are the two principal properties that embedded languages should opt for:

Use generic abstract structures. Each custom embedded language is unique. Chances are, however, that your users will see several of your languages. Therefore try to use generic abstract structures as the "glue" of the language: this eases both the design and implementation task, and makes it easier for users to learn and later remember several embedded languages without despairing.

Use conservative notation. A custom embedded language should fit smoothly in with its "host language." Use the host language's notations whenever possible.

The "abstract structure" of a directory tree is a tree. Thus we can use standard prefix notation for trees and write each "branch" as

label (*subtree* … *subtree*)

where *label* is the text associated to each node (for a directory this will be its name) and each *subtree* is an entire tree rooted below the present node (corresponding to subdirectories and files). Branches with no subtrees are often called "leaves" but we do not need to distinguish: a leaf can be written

label ()

This constitutes the "glue." For the nodes we should try not to extend the TeX notation too much. A nice and conservative approach is to use the TeX argument notation, *i.e.*, write the labels as

`{`*text*`}`

to indicate that *text* should be interpreted as TeX source text. In fact this is the notation we used in the directory tree example in figure 4.

Implementation. We will base our implementation on XY-pic (Rose and Moore, 1997) since this is a generic platform for 2-dimensional graphics that works with TeX and can do what we wish to illustrate without compromising the quality of the typeset drawings. However, the technique used to produce the actual graphics is not essential as long as the "library" of available graphics functions is sufficiently easy to use.

Embedded languages are implemented by writing a small *interpreter* that parses the language and performs the appropriate actions, in this case calls the appropriate XY-pic drawing primitives. In order to write such an interpreter we should write the BNF[2] of the language. This looks as follows:

$$\begin{array}{rcl} \langle\mathit{tree}\rangle & ::= & \texttt{\{}\ \langle\mathit{text}\rangle\ \texttt{\}}\quad \texttt{(}\quad \langle\mathit{subtrees}\rangle\quad \texttt{)} \\ \langle\mathit{subtrees}\rangle & ::= & \langle\mathit{empty}\rangle \\ & | & \langle\mathit{tree}\rangle\quad \langle\mathit{subtrees}\rangle \end{array}$$

The interpreter is then a parser that recognises that this format is followed and performs an appropriate *action* for each recognised symbol.

This implementation will emulate the layout of the *tree* command graphically: each label should be indented relative to its parent and connected to it, furthermore it should be below the previous label. This can be described by actions associated to each symbol as it is encountered: these are shown in figure 5. This is implemented by the small (plain) TeX file `tree.tex` shown in figure 6: the `\parser` macro selects the appropriate action based on the current symbol; each of the `\...action` macros implements the appropriate action from figure 5 using XY-pic with the 'arrow' extension (for details on how to use XY-pic refer to the reference manual, Rose and Moore, 1997). Running this on the source in figure 4 produces the tree shown in figure 7. The macros make use of the general font style of the program, *i.e.*, `\baselineskip` is used for distances to fit with the line skips used. Furthermore, some components of the state change have been isolated as definitions — something that is possible with a generic macro language as TeX; a production version of the tree language the layout style of the tree should also be extracted into definitions. In fact, the "PostScript" sample of figure 3 was created using the `times` package with the redefinition

```
\def\branch{\save;s0!CD**@{-}\restore}
```

which tells XY-pic to make a plain line from the center bottom of the "parent" to the "child."

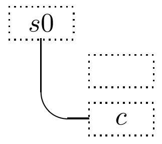
Symbol	Before	⇒	After
`{`*text*`}`	*empty stack*	⇒	*text*
	s0 *c*	⇒	
`(`	*c*	⇒	*s0* *c* *+ grow stack*
`)`	*empty stack*	⇒	*error!*
	s0 *c*	⇒	*c* *+ shrink stack*

Figure 5: Tree interpreter actions.

Exploiting the structured notation. Being able to vary the style this way is useful, of course. However, a common mistake when implementing packages such as `tree` is, in the author's opinion, to implement an interpreter that is too general. It is better to think of separate tasks as requiring separate embedded languages, implemented by separate interpreters, even if they happen to have the same syntax. One can go even further with *non-standard interpretation* of the information in the embedded language. Say that we wish to interpret the notion of "a tree" in a different way. For example, one could wish to show how a directory tree like our sample can be *grown*. This is slightly more complicated in that it requires our graphic library to include animation. This is possible in a (not yet published) module for XY-pic called `movie`. With this, animations are composed of "scenes" within which something varies from a starting point to an ending point. We can show growth by having a scene with just the root, then one level of branches, *etc.*; at a finer level we can let the branches grow gradually for greater

[2] BNF is the notation for "meta-linguistic formulae" first used by (Naur et al., 1960) to describe the syntax of the Algol programming language. We use it with the conventions of the TeXbook (Knuth, 1988): "::=" is read "is defined to be", "|" is read "or", and "⟨*empty*⟩" denotes "nothing."

```
% tree.tex: Print \tree{ <tree> } as directory tree.

% Use Xy-pic, including ‘stack empty’ primitive.
\input xy
{\catcode‘\@=11 \global\let\sempty=\sempty@}
\xyoption{arrow}

% Idioms.
\def\FN{\futurelet\next}
\def\DN{\def\next}
\def\SP.{\futurelet\SP\relax}\SP. %

% <tree> parser.
\def\tree#1{\xy \beginaction \FN\parser#1\relax \endaction \endxy}
\def\parser{%
  \ifx\SP\next \expandafter\DN\space{\FN\parser}%
  \else\ifx\bgroup\next \DN##1{\textaction{##1}\FN\parser}%
  \else\ifx(\next       \DN({\openaction \FN\parser}%
  \else\ifx)\next       \DN){\closeaction\FN\parser}%
  \else\ifx\relax\next  \DN\relax{}%
  \else \DN{%
    \errmessage{<tree> build from (, ), and {text} only: not \meaning\next}}%
  \fi\fi\fi\fi\fi \next}

% Initial action : start fresh stack frame.
\def\beginaction{\POS @( }

% Interpretation action for {text} : typeset node and its branch!
\def\textaction#1{\node{#1}\if\sempty\else \branch \fi}
\def\node#1{\drop+!L\txt{#1}}
\def\branch{\ar @{-} ‘l/\jot s0+DC="s0" "s0" }

% Interpretation action for ( : move left and down!
\def\openaction{\POS @+c +R+/r1em/ +/d\baselineskip/ }

% Interpretation action for ) : move back below parent!
\def\closeaction{\if\sempty \errmessage{too many )s in <tree>}%
  \else \POS {;p+/r/:s0;p+/d/,x}@-c \fi}

% Final action : obliterate stack frame.
\def\endaction{\if\sempty\else \errmessage{missing )s in <tree>}\fi
  \POS @) }
```

Figure 6: The `tree.tex` macros.

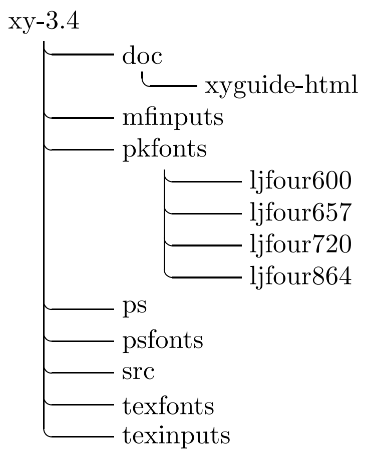


Figure 7: Generated directory tree.

effect. Using the movie class of XY-pic[3] one can produce an animation of the same tree, based on the same source, by modifying the actions for each component to draw it in a manner dependent on the time. The resulting animation can be found in the electronic version of this paper (Rose, 1997); here we can merely reproduce the (also automatically generated) "storyboard" of the animation, shown in figure 8. The source of the movie is shown in figure 9: the actions have been enriched with conditions for hiding leaves until the `\level` counter gets higher than their `\nesting` value, permitting them to appear. Some extra tricks make this happen gradually, using the `\F` construction of the movie class.

Conclusions

We hope to have shown that TEX is quite naturally extended with embedded languages and that this can be a convenient way of

- getting nice pictures and diagrams in papers,
- permitting aesthetic integration of text and diagrams, and
- ensuring that the information in the pictures can be exploited in alternate ways.

Acknowledgements. The author wishes to thank TUG '97 and BRICS for jointly funding my participation in the conference.

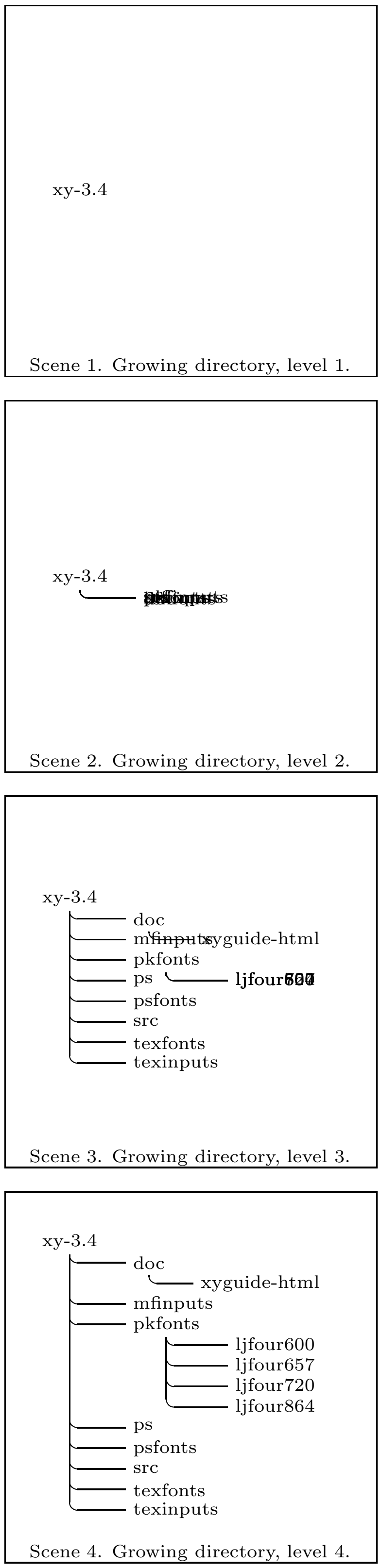


Figure 8: Growing the directory tree.

[3] Available in an experimental version with XY-pic 3.4.

```
\MovieSetup{height=18em,width=18em}% -*-LaTeX-*- animation of directory tree

\newcount\level
\newcount\nesting

\def\node#1{%
  \ifnum \level<\nesting \drop i+!L\txt{#1}%
  \else \drop+!L\txt{#1}\fi}

\def\branch{%
  \ifnum \level<\nesting
  \else \ar @{-} `l/\jot s0+DC="x" "x" \fi \relax}

\def\openaction{%
  \POS @+c +R+/r1em/ \F\down
  \global\advance\nesting by +1 \relax}

\def\down(#1){%
  \ifnum \level>\nesting \POS+/d\baselineskip/*{}%
  \else\ifnum \level=\nesting \POS+/d#1\baselineskip/*{}\fi\fi \relax}

\def\closeaction{\global\advance\nesting by -1 %
  \if\sempty \errmessage{too many )s in <tree>}%
  \else  \POS {;p+/r/:s0;p+/d/,x}@-c \fi \relax}

\level=0 \loop
  \advance\level 1 %
  \nesting=1 %
  \scene{%
    \input{dirtree.tree}%
    \caption{Growing directory, level \the\level.}%
  }%
\ifnum \level<4 \repeat
```

Figure 9: The `dirtree.texmovie` movie.

References

Adobe. *PostScript Language Reference Manual*. Addison-Wesley, second edition, 1990.

Goossens, Michel, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesley, 1997.

Knuth, Donald. *The TeXbook*. Addison-Wesley, second edition, 1988.

Knuth, Donald E. *The METAFONTbook*. Addison-Wesley, 1986.

Lamport, Leslie. *LaTeX—A Document Preparation System*. Addison-Wesley, second edition, 1994.

Leathrum, Thomas and G. Tobin. "The mfpic package". Available from CTAN: `graphics/mfpic`, 1994.

Naur, Peter et al.. "Report on the Algorithmic Language ALGOL 60". *Communications of the ACM* **3**, 299–314, 1960.

Rose, Kristoffer H. "Very High Level 2-dimensional Graphics with TeX and XY-pic". Available from `http://www.brics.dk/~krisrose/Xy-pic/tug97/`, 1997. Electronic version of TUG97 paper.

Rose, Kristoffer H. and R. R. Moore. "XY-pic release 3.4". Available from CTAN: `macros/generic/diagrams/xypic`, 1997. See also chapter 5 of (Goossens, Rahtz, and Mittelbach).

van Zandt, Timothy. "The PSTricks package". Available from CTAN: `graphics/pstricks`, 1996. See also chapter 4 of (Goossens, Rahtz, and Mittelbach).