CITY University

**Technical Report**

Department of Computer Science

School of Informatics

September 1993

# The Experience of TEX: Typesetting and Beyond

Jiří Zlatuška

TCU/CS/1993/3

Department of Computer Science, School of Informatics, City University, Northampton Square, London, EC1V 0HB, England.

**Abstract:** There are several reasons why Donald Knuth's typesetting system TeX merits the attention of a wider audience than it has hitherto attracted. Not only is it a superb state-of-the-art typesetting tool, the techniques used in its development as well as the high level of documentation and portability, provide an excellent case study in the design and implementation of such systems. This paper presents an overview of some of the points of general interest as an invitation for further study of it.

# 1 Introduction

TeX is not just a typesetting system designed and built to meet the need for fine printing of mathematical text, it is also a masterpiece as a program, as a software engineering product, and an excellent example of an open system design extensible to match a particular purpose. The purpose of the present paper is to examine in a tutorial style some aspects of TeX use which often remain hidden to the casual user who only uses the system as a formatting tool easily accessible on the computer. The present paper will discuss some of the features of the system which the author believes are of relevance to people interested in programming and software engineering in general; these include design of a typesetting system as a program interpreter, programming techniques and tools designed for portability and self-documentation. It is my conviction that the software-engineering aspects of TeX as a program system present a superb case study in program development, and that this case study is likely to be of value to a wider audience than the circle of people presently engaged in designing tools to enlarge the scope of TeX. If the reader is encouraged by this paper to examine and derive inspiration from the code Donald Knuth presents us as "a computer program of which a professor of computer science might be proud"[1], then its purpose will have been fulfilled. The use of TeX has also quite addictive power, and the *ability* to produce typographically superb results will sooner or later induce in the user an interest in issues of typographic design (hitherto the sole province of those professionally involved in the production of books), however little concerned with the æsthetics of the printed document he or she may be at first. This paper is an introductory part of a tutorial published in SOFSEM'93 proceedings, the remaining two papers in this series [20, 21] provide a discussion of the theory and practice of fine book design as well as advice on the successful application of TeX to achieve this end.

The bibliography at the end of this paper presents some basic titles from three areas: TeX as such, LaTeX (some familiarity with which will be useful to any reader of this paper who is not familiar with TeX per se) and general issues in typesetting and book design (the latter issues are addressed in the subsequent two papers in this series by Philip Taylor [20, 21]). Concerning TeX and LaTeX, quite a few texts aimed at the beginner have appeared within the last three or four years, but none of these is listed here.

[1]from the preface to *TeX: The Program.*

# 2 TeX as a program system

TeX as a program has been ported to almost all computer platforms and operating systems from ATARI micros to supercomputers. Book-size documents containing many fonts can be formatted on all of these with absolutely predictable and identical results (and with a precision far exceeding that of any of the top-range phototypesetters). Most of these implementations are in the public-domain, and have been produced by numerous people around the globe. Donald Knuth's code has certainly been difficult to surpass, and it was accompanied by a number of deliberate decisions to make portability as easy as possible.

## Programs discussed

Although TeX itself is the component of which the user is most aware, there are many programs other than just TeX the program which make the system complete, and we mostly refer to the complete set of programs when talking about a program *system*. TeX the program is the actual typesetting processor, transforming the user's input (`.tex`) into a device-independent (`.dvi`) file containing the typeset document. TeX itself knows nothing about either the particular glyphs the actual characters are composed of or of the characteristics of the actual device on which the formatted document will eventually be printed. The character definitions are (traditionally) produced by the METAFONT program (which, interestingly, is of nearly exactly the same size as TeX the program when measured in terms of lines of source code); METAFONT converts character definitions in a language of its own into rasters for a particular output device and resolution. METAFONT is essentially a solver for systems of equations, allowing for handling shapes defined as Bézier curves or as shape moves with special treatment of rasterization onto a particular grid and/or physical output device characteristics.[2] The output of METAFONT is (1) a *generic font* (`.gf`) format file, containing the raster definitions for the characters of the font, and (2) a *TEX font metric* (`.tfm`) file containing various font dimension characteristics, tables for character-pair alignments (*kerning*) and multiple character elision (*ligature programs*). Font metric files provide the only font information if which TEX the program is aware, and thee contain all the information needed for handling typesetting using a particular font of type. The device-independent output is further processed by a particular output *device driver*, which is actually an interpreter of the language used in `.dvi` files for a particular output device (phototypesetter, screen, or laser or dot-matrix printer). Each of the intermediate files has the form of a (binary encoded) abstract language, and programs allowing for the translation of these to/from human-readable form are also provided.

## Literate Programming

The concept of *Literate Programming* is used for program code development and documentation in most of the programs used in conjunction with TEX. The programs should be capable of being read as literature, i.e. the flow of code should follow the flow of topics introduced within the program in a way natural to the human reader and program developer. Code sections are presented as sections of text, properly documented and formatted using typographic tools for pretty-printing the code and cross-references within the text.

**WEB:** The system of structured documentation developed by Donald Knuth for his TEX project is based on a subset of standard Pascal. The WEB system allows several things to be accomplished at once: (1) Although the basic assumed structure of the resulting program is that of a main program containing a single level of unnested procedures (essentially a C program structure), the code can be written in a logical sequence, and 'tangled' together into properly sequenced Pascal source code. (2) All of the program identifiers are converted into upper-case 7-byte long ones, so that the resulting code can be processed by any compiler modestly conforming to the Pascal 'standard'. (3) Simple macro facilities are provided, allowing for the definition either of simple parameterized replacement strings (not necessarily complete portions of text from the point of view of Pascal syntax), or numeric macros evaluated prior to compilation. (4) Pretty-printing of the code and explanatory comments is an integral part of WEB, and is in fact the only sensible means of looking through the program text. (5) Special facilities are provided for character and string handling. A sub-range of the integer data type is used to accomplish that: characters are converted into standardized ascii-based codes, while any string is represented by its index into an array stored as a "pool" and quite distinct from the program code itself. This allows complete independence of the character encoding system used at the system where the program is being compiled.

The user prepares the source program code as a Pascal text broken into sections consisting of an introductory comment part (plain text), macro definitions, and the actual Pascal code. Some 30+ two-character commands starting with '`@`' are provided, each of which may be used either to facilitate sectioning and macro extensions, or to aid pretty-printing when automatic heuristics fail to work properly. Both the documentation and executable source code are embodied in the WEB source. Two processors are used for source text processing: WEAVE and TANGLE.

**WEAVE:** The WEAVE processor generates TEX source code for pretty-printing the program. It performs limited syntactic analysis of the input, and recognizes occurrences of various syntactic units of Pascal code, nesting, etc. The resulting `.tex` file is then processed by plain TEX. WEAVE starts by inputting a standard set of macro definitions (`webmac.tex`) from the library of standard

[2]The underlying idea behind a METAFONT font definition is the specification of one set of basic definitions of character shapes which can be further tuned to a particular size, shape or version of the script, allowing for all the possible variations that this involves. This concept of *metaness* in fonts is not further discussed here, but the interested reader is referred to Volume E: Computer Modern Typefaces of Donald Knuth's *Computers & Typesetting* series showing METAFONT definitions for the Computer Modern family of fonts.

TeX input files, and produces typographically formatted source code output.

The following double-page spread shows an example of simple WEB input code, and both the actual Pascal code generated by TANGLE and the pretty-printed version generated by WEAVE and TeX. The input is a Pascal program divided into sections and augmented by comments as fragments of text (see the left-hand-side page).

The output prepared by WEAVE and processed by TeX yields the pretty-printed version – a standard version of a program listing used by a WEB programmer; this can be seen on the right-hand-side page of the example double page.

The sources published in the books for the TeX and METAFONT programs are merely the result of TeXing the output of WEAVE when applied to the publicly available sources of these programs, with the addition of local page indexes of symbols.

**TANGLE:** The second of the WEB-related processors is TANGLE. It takes the section-by-section input, strips out of it the textual comments, transforms the Pascal code into a standardized upper-case-only, seven-character-long-maximum identifier representation, and produces a file which is further processed by a Pascal compiler in order to produce executable code.

Sections need not appear contiguously in the WEB code. If a section is defined on several places within the code, TANGLE puts these pieces together. This is indeed a commonly used feature: any particular algorithm appearing in the code may need to add some type or variable declarations; these additions should be treated locally within the code, reflecting the logic of the program, not that of the syntax of the programming language.

The TANGLEed result of processing the example WEB source can be found on the two lines below the source WEB code example.

Note that as an artifact of the primary intent of efficient code representation, the code is *very* unreadable to humans. Of course, one is intended *never* to actually edit this standard Pascal source manually, but always to make changes through the medium of the core WEB source, thus keeping the code and its documentation entirely consistent.

**Efficiency:** Breaking a program into parts which reflect the logic of the problem frequently leads to a loss of run-time efficiency when such parts are subsequently implemented as separate procedures. Using WEB, on the other hand, the resulting program 'knit' together by TANGLE makes use of textual rearrangement of the code prior to compilation; hence the programmer can produce very efficient code without sacrificing its clarity, because little is lost on overheads or 'red tape' at run time.

The simple global structure of the resulting Pascal code also allows for subsequent translation into other programming language. For TeX systems running on Unix platforms, this is actually the most usual way of producing executable code from original sources: the target language is C, not Pascal, and a simple converter web2c performs translation of the WEB source into C source at the textual level of the source code. The result is eventually processed by a C compiler.

**Portability:** We have already mentioned the astonishing number of ports of the TeX system onto various platforms, and we have touched on some of the support given to that by Donald Knuth's choice of a carefully selected subset of Pascal. This would not, by itself, be enough: there are always variations in the interface to the operating system, limitations of memory models or sizes, etc. Modifications to the standard sources for a particular site are solved via *change files* specifying 'patches' to the sources for a particular system. Both TANGLE and WEAVE read as input a pair of files, the actual WEB input, and the appropriate difference or 'Change' file, and perform an update of the source on the fly. Developers are therefore encouraged to prepare their own change files (presenting changes as drastic as, e.g., virtualization of the memory arrays used in the sources), and are actually forbidden to make any changes to the original canonical sources.[3]

**Self-documentation:** Both TANGLE and WEAVE are written and distributed as WEB sources, so the technique of change files can be used for bootstrapping the system onto any suitable platform. The WEB system has been adapted to many other languages (C,

[3]Nowadays hardly anybody uses the canonical source *as is*, as it is written and maintained in a Pascal dialect for DEC10.

```
@* This is an example of three sections of a |WEB| source. The
first one shows a section consisting entirely of a commentary.

@ Here we define a simple piece of code consisting of
reference to two other sections.
@p  type
    @<Type definition@>
    var
    @<Declarations@>
    begin
    @<Body code@>
    end.

@ Character constants are automatically converted into numbers in
ASCII code. The following defines the appropriate type in order
to be used later.
@<Type definition@>=
    ASCII_code=0..127;

@ Identifier |max| is defined as numeric macro. When used in the
context of arithmetic expressions, these get evaluated allowing
constructions such as follows:
@d max=30           {numeric macro}
@d area(#)==a[#+1]  {replacement macro with a parameter}
@<Declarations@>=
    a: array [1..max] of ASCII_code;
    i: 0..max-1;

@ References to sections may have trimmed names which
automatically get completed.
@<Body ...@>=
    for i:=0 to max-1 do
      area(i):="A";
```

**Example WEB source.**

```
{2:}TYPE{3:}ASCIICODE=0..127;{:3}VAR{4:}A:ARRAY[1..30]OF ASCIICODE;
I:0..29;{:4}BEGIN{5:}FOR I:=0 TO 29 DO A[I+1]:=65;{:5}END.{:2}
```

**Standard Pascal code generated by TANGLE.**

**1. This is an example of three sections of a *WEB* source.** The first one shows a section consisting entirely of a commentary.

**2.** Here we define a simple piece of code consisting of reference to two other sections.

**type** ⟨ Type definition 3 ⟩
**var** ⟨ Declarations 4 ⟩
  **begin** ⟨ Body code 5 ⟩
  **end**.

**3.** Character constants are automatically converted into numbers in ASCII code. The following defines the appropriate type in order to be used later.

⟨ Type definition 3 ⟩ ≡
  *ASCII_code* = 0 .. 127;

This code is used in section 2.

**4.** Identifier *max* is defined as numeric macro. When used in the context of arithmetic expressions, these get evaluated allowing constructions such as follows:

  **define** *max* = 30  { numeric macro }
  **define** *area*(#) ≡ *a*[# + 1]  { replacement macro with a parameter }

⟨ Declarations 4 ⟩ ≡
*a*: **array** [1 .. *max*] **of** *ASCII_code*;
*i*: 0 .. *max* − 1;

This code is used in section 2.

**5.** References to sections may have trimmed names which automatically get completed.

⟨ Body code 5 ⟩ ≡
  **for** *i* ← 0 **to** *max* − 1 **do** *area*(*i*) ← "A";

This code is used in section 2.



**Formatted source listing produced by WEAVE.**

Modula-2, Ada, and FORTRAN, for example, as well as TEX itself), and the sources are available from the major network archives.

### Language support

TEX prior to version 3, as defined in the *TEXbook* printings before 1990, would allow the typesetting of accented characters, but that was all in terms of multi-lingual support. Michael Ferguson provided an adapted version from Knuth's source, called 'MLTEX', and some of the changes he proposed became w.e.f. version 3 an official part of TEX per se. There are three extensions of special significance: firstly, different hyphenation patterns can be used at different points during a single TEX run, thus permitting the correct hyphenation for each embedded language to appear in the formatted text; secondly, full 8-bit code is used for internal character representation (it has always been available for fonts), so that it is now easier to include directly most of the European languages; thirdly, support for virtual fonts was added.

**Virtual fonts:** This is actually not a change concerning TEX itself, but rather a change to the functionality of output drivers: a virtual font is a font defined in terms of `dvi` "macros" – any operation allowed in a `dvi` file is now allowed to appear in a virtual font definition (except for the "new page" operation, but including font switching). TEX need not be aware of this, as the `tfm` interface remains unchanged: it is just the output device driver which checks for a presence of a virtual font definition (`.vf` file), and if found, it expands the character definitions therein into the output. (Similarly to the original `tfm` interface, two conversion programs exist, `vftovp` and `vptovf` allowing conversion to and from human-readable form.) The distribution of the most popular `dvi` driver for PostScript printers (`dvips` by Tom Rokicki) actually contains a conversion procedure for ADOBE-encoded font information into virtual fonts for TEX, including support for a multilingual European font table (the so-called 'DC-encoding', aimed at putting all of the characters appearing in European languages into a single font table, and a part of a wider multilingual support initiative supported by TUG)[4].

[4]TUG is the international TEX Users Group organization; quite a few local TEX users groups exist around the world, such as, e.g., CSTUG, the Czecho-Slovak TEX Users Group, publishing four issues a year of its *TEX bulletin*.

## 3 TEX as a typesetting engine

The approach taken by TEX to typesetting from a stream of input text is based on handling pieces of the typeset material in the form of *boxes*, of which nothing except their dimensions and baseline position is known to the processor; these boxes are put together either tight or with the use of *glue*, allowing for some flexibility before the positions are fixed when forming hierarchically higher boxes. This *box and glue* paradigm provides a powerful, yet quite simple, unifying concept for handling various portions of the typeset material.

### Boxes and glue

**Boxes:** The material to be typeset is assembled into boxes – units defined by a rectangular shape with a baseline-positioned reference point, a width (distance to the right of the reference point), a height, and a depth. The measurements of the box have no direct relationship with the contents of the box, and in particular they need not necessarily confine the contents of the box within its boundaries. TEX works with two types of boxes, vertical and horizontal: i.e. boxes within which the enclosed material is either connected in a vertical direction (e.g., lines of text on a page), or in a horizontal direction (e.g., letters in a line of text). The boxes are constructed from lists of boxes of lower level (characters of a font, in the limiting case), and from glue, which is an element allowing for the adjustment of a list of boxes to fit into a predefined dimension.

**Glue:** When a box is being made from a list of other boxes, glue is the element which allows adjustment of the relative positions of boxes within the hierarchically higher unit. There are three basic components which define glue's behaviour: its natural space, its shrink component, and its stretch component. When free of other constraints, boxes are set at the distance corresponding to the natural space of the glue occurring between them; however, it is often the case that the dimension of the resulting box is fixed in advance (e.g., line width or page height), and in such cases some proportion of the stretch or shrink components is taken from each of the participating glue elements in the same ration as their stretchability or shrinkability. Three degrees of 'infinite glue' are provided (for first-level infinite stretchability these are `\vfil` and `\hfil` for the vertical and horizontal directions respectively, these being the direct equivalents of `\vskip` or `\hskip` with `0pt plus 1fil` glue specification, where '`fil`' is the 'infinite' component and '`plus`' is the specification of stretchability) in order to supersede effects of the lower-level (finite) stretch or shrink components.

The behaviour of glue allows various effects concerned with the justification of pieces of text on the page, and the specification of proportionate properties of the white space. There is one drawback to it, however, because there has never been any equivalent of this in traditional typographic practice, which has always worked with fillings of certain discrete dimensions. As a consequence, certain requirements of traditional typography are sometimes difficult to meet, as discussed in [21].

Another data type known to TEX is that of 'dimension', which is similar to glue except that there is neither stretch nor shrink component in it. A glue quantity can be 'coerced' to a dimension by throwing away all but the natural width of the glue. LATEX users know glue under the name of "rubber length", thereby pretending that it is just a special kind of length specification. Glue is actually fixed when inside a (completed) box, and hence cannot contribute to the elasticity of the boxes being built outside of it.

## Modes

The basic internal activity of TEX is the perpetual building of the 'main vertical list', i.e. a galley of horizontal boxes (lines) which is trimmed into pages by the page builder and the (asynchronously invoked) output routine. In general, all of the box-building process is performed by processing a list of elements within certain *modes* of work. Mode switching is usually done automatically, but awareness of the process is needed, if the user needs to understand details of unexpected behaviour.

There are six modes in which TEX performs its box-building activities from input vertical or horizontal lists.

**Vertical mode:** This is the mode in which TEX starts life, and in which the galley list (i.e. the *main vertical list*) is formed. Pages are then trimmed by the page builder and *output routine*, and output to `.dvi` file by the `\shipout` primitive (for an example of the use of `\shipout`, see the book design paper in this volume).

**Internal vertical mode:** This mode is used for the explicit construction of vertical boxes within the user's input code. No automatic page trimming occurs within it.

**Horizontal mode:** This is the mode in which the horizontal boxes containing the lines of a paragraph are formed. It can be entered automatically (from vertical mode) by the occurrence of (for example) a character to be typeset, or can be entered explicitly using any of `\leavevmode`, `\indent` or `\noindent`; LATEX users can also switch into this mode by creating `\parbox` or `minipage` boxes – the main issue is that automatic cutting of the input into separate lines is performed by the paragraph building procedure built into the internals of TEX.

**Restricted horizontal mode:** Used for explicit building of horizontal lists under the user's control. No line breaking is performed, and the paragraph builder does not participate here. LATEX users enter this mode when using, e.g., `\mbox` or `\makebox`.

**Math mode:** A special mode for building mathematical formulas for inclusion into a horizontal list. This is the mode TEX in which TEX demonstrates most clearly its power to take typesetting decisions without requiring explicit guidance from the user.

**Display math mode:** One-line math formula building when interrupting temporarily current paragraph building. As with restricted horizontal mode, no line breaking is performed in it.

## Paragraph building

The operation of breaking the horizontal list prepared in horizontal mode presents the most intriguing operation built into TEX. Without abusing TEX, the user lacks the tools for writing procedural code to perform the line breaking operations which form a paragraph out of a horizontal list. Instead, an optimization procedure is supplied, adjustable by changes of value of several parameters, affecting paragraph shape, feasibility of line breaking at either particular places, or as a function of certain structural properties of the paragraph formed (such as, e.g., occurrences of consecutive lines with hyphenated words, etc.). TeX works with several numerical quantities, the *badness* defined by the deviation of a constructed box from its "natural" dimension when no stretch or shrink glue components are used within it, the *penalty* values inserted into the paragraph body to indicate (un)desirability of a break at that particular place, and the *demerits* value, derived as a formula from explicit demerits, penalties and badnesses. The actual paragraph breaking is then an optimization process trying to find a way of breaking the horizontal list into lines so that the badness of none of them exceeds a predefined valued of `\tolerance`; furthermore, the way of minimizing the total demerits is selected from the solutions satisfying the `\tolerance` constraint.[5] There are three phases during which a paragraph is consecutively tried to be formed: first, a solution is tried without hyphenating words and within a `\pretolerance` limit; if not successful, all possible hyphenation points within the paragraph are determined, and another try is given within `\tolerance`. W.e.f. TeX version 3, yet another attempt is made using the quantity `\emergencystretch`, which is a dimension added as auxiliary stretchability to each line in order to distribute evenly additional stretch among the words on each line. When setting text with inadequate break points and within narrow margins, it is usually better to set it with a greater value of this quantity (some 1 em should be both necessary and sufficient), than to raise the tolerance value, because that tends to concentrate all the badness into smallest possible number of lines, most usually just into one of them, which then looks extremely ugly. (This paper has been typeset with `\emergencystretch=1cm`).

[5]One way of preventing hyphenation following a suggestion given in the TeX book by Knuth is setting `\pretolerance=10000`. With the advent of TeX version 3, it is also possible to say, e.g., `\lefthyphenmin=\maxdimen`. In this case, the third pass is still attempted, if necessary.

### Page breaking

While line breaking is a global minimization operation within a paragraph, page breaking is handled is a rather less satisfactory manner, mainly because at the time of writing TeX, Knuth obviously didn't consider it feasible to run page breaking optimization for whole chapters. Again, penalties are associated with particular page breaks, but now this is just step-by-step incremental operation searching for a local minimum, instead a global one: when the point of filling the current page by text and other material is reached, the partially constructed vertical list is sent to the (asynchronously invoked) `\output` routine, which prepares the actual page to be sent out by `\shipout`. Note that once a page has been shipped out, this is a one-way process, no backtracking is done, and hence no guarantee of optimal chapter shape is given. Quite a few problems are caused by this approach when actual typographical design is imposed on a document. The tricks to be used make writing output routines perhaps the most difficult part of writing code for TeX.

## 4 TeX as a language

Typesetting using TeX is controlled using a programming language, which is actually a computationally complete language.

### Levels of operation

There are three levels of operation, for which TeX provides programming tools: input, macro expansion, and execution.

**Input:** During this phase, characters are read, and each of them assigned one of 16 *categories*, classifying tokens according to their function within the system, of which only characters of two categories ('*letter*' and '*other*') get eventually printed. From these characters *tokens* are formed (possibly compound identifiers), which are further processed by expansion or execution.

The input proceeds in a rather intricate way: its behaviour depends on classification of character codes, but this comes from executing certain parts of the input first.[6] Also,

[6]Run-time change of input character classification occurs for various purposes: the most common one

in order to complete reading of certain parts of the input, lookahead is needed, which in turn depends on expansion. In other cases, matching tokens are sought even though the input is just skipped over (false branches of conditionals), but certain tokens may cause confusion, because TEX would look for them in vain. The syntax description itself is thus rather heavily execution-based, and far from an ordinary programming language syntax presentation...

**Expansion:** The process of expanding macro definitions is a standard one, with primarily call-by-name parameter passing (although this can be circumvented using `\expandafter`[7] and similar constructs to change the effective order of expansion). Up to nine parameters can be specified, and their scope can even be made call-template determined. Several modes of macro definitions[8] are made possible, even controlling the level of expansion within the body.

**Execution:** A fixed number of primitive commands actually get *executed* instead of expanded. These are either invocations of the built-in TEX algorithms (e.g., `\par`), or actions over internal data structures. The difference between execution and expansion is often neglected by the user, and indeed the control sequences referring to actions can be redefined as macros. In some contexts the difference is quite essential and it actually influences the way in which the input is processed at the highest level (e.g., scanning numbers, glue specifications, etc.).

## Structure

TEX uses the standard stack-related concept of scopes for visibility of its entities, extended to include visibility of any change of parameters which are not explicitly prefixed as `\global`. Hence not only local macro definitions are local to 'blocks', called *groups* in TEX, but also the effect of an assignment is lost after leaving the block in which it occurred.

Both static and dynamic nesting is employed. The former allows for easier detection of *begin-end* pairing errors, while the latter makes it possible to embed parts of the matching structures into separate macro definitions.[9]

## Formats

Very few of the constructions that a TEX typesetter would regularly need are included as commands known to TEX the program itself. Most of them are definitions or macros which have to be read in first. In order to speed up this process and avoid re-processing vast amounts of code at the beginning of each run, Knuth came up with the idea of providing a mechanism for dumping out an image of TEX's memory into a *format file*, (`.fmt`), which can be downloaded again at the beginning of each run without actually executing the declarations needed. Two versions of TEX thus exist: one called `initex`, capable of dumping out a format file (and of processing definitions of hyphenation patterns), the other one is called `virtex` ("virgin TEX") capable of loading the format file and performing the actual typesetting.

**Plain and lplain:** Two format files are widespread and well-known: `plain` and `lplain`. The first of them represents the set of TEX macros as defined by the TEXbook, the other one is an extended variant of these (regrettably not a true superset) which adds the macros needed for structural document markup – this is known as LATEX. On Unix systems, a symbolic link to the name of the format makes `virtex` download the proper format automatically, and hence quite a few

is to switch TEX into "verbatim" mode, forcing it to replicate certain or all elements of the input into the output. Another application is represented by recent self-documenting styles used in LATEX: when used as a document to be printed, they switch the meaning of the '%' comment character so that formatted commented source for the style file is produced, very much in the style of WEB programming. When used as a style (in `\documentstyle` declaration), their code is normally executed and '%' acts as a comment delimiter. The mode in which the file is used is also detected by testing the category of a character, which is normally different during the style-reading and document-formatting phases of a LATEX run.

[7] expanding first the second token in the input stream (possibly containing arguments), inserting its result of expansion into the input, and then returning to expand the token which follows.

[8] `\def` is the basic one, hidden to LATEX users inside of `\newcommand`; `\edef` expands the body of the definition; `\noexpand` can be used to prevent expansion in certain stages, and thus to affect the resulting order of expansion.

[9] e.g., constructions like `\begin{...}` and `\end{...}` in LATEX are realized in this way.

users get the impression that TEX and LATEX are different programs.

### Isn't WYSIWYG better?

TEX gets often criticized for not providing so-called WYSIWYG ("what you see is what you get") facilities, i.e. there are no fancy windows, no use of mouse, none of those things which reviewers from the popular magazines like so much. One cannot just play around with the screen, and instead is forced to *think* (and, boy, it hurts!) about how the result is supposed to look like as a whole, and provide some *consistent* (God forbid!) definition of it.

On the side of WYSIWYG programs the problem is not only that they are simply at present *not* suited for *typesetting*, but merely DTP stuff, but in quite a few cases they are very far from really obtaining what one sees on the screen – if not for anything else then for lack of independence of the output device.[10] With increased processing power of computers, it's more and more often possible to get the preview on the screen in the time comparable to the time required by screen-oriented DTP programs to update the formatting.

There is yet another advantage in actually using a programming language for defining the resulting layout: formatting database data[11] is of the same order of difficulty as formatting a few pages of text, and I have already seen quite a few successful applications of this feature; during the past two years, some 10 multilingual catalogues for the Brno trade fair were produced using TEX (some of them in 5 different languages including Russian, some over 1000 pages of text produced within less than one day after the database of the exhibitors was completed), timetables for municipal transport, or dictionaries (dictionary of computer terms, and Czech-English dictionary).

One of the funnier applications I regularly use is the use of TEX for generating individual test sheets for final tests for my students: you start by defining a few test problems with three versions each, and then using a couple of nested loops ensure that no neighbours have any of the variants identical, and that no pair of students within the class have the same set of problems (to make it even more fun, some parts of the problems can be made macros which change themselves out of phase with the rest of the loops) — see the example code for this shown in the Appendix. Then replace the problem definitions by their solutions, and here you print individual answer sets for each of the students.

## 5 A bit of self-reference

At last, a few words concerning how this paper has been typeset. The typeface used is the LucidaBright family of fonts designed by Bigelow and Holmes and produced as Type 1 fonts by Y&Y. The document style is the standard LATEX article style with `twocolumn` option, with fixed baseline distance and a modification of `\section` and `\subsection` commands for typesetting on the grid using the construction of "pseudobox" from [21] (with added white space distribution and compensation for `\topskip` when on top of a column).[12] For inserting double-column portions of text into standard '`twocolumn`' LATEX option, a simple `\header`-based macro was written which assumes manual insertion of `\vskip` in the other column – due to the grid, this is fairly easy, and remarkably stable

[10]Recently I saw a user of one of these programs struggle to coerce his DTP system into not using uneven spacing, to forbid all single-letter prepositions at ends of lines (something mandatory in Czech typography), etc., when editing a 600+ page long multi-author book. After that, he decided to produce the proof copies for the authors on a dot-matrix printer, to cut the costs. After going through all of this, polishing the result into perfection, the real printing was done on a 400 dpi laser printer. But merely changing the output device had an unpleasant side-effect: the whole of the book got reformatted, and all the typographic editing had to be done again from the scratch.

[11]For more on this type of TEX use, see also P. Sojka, et. al.: TEX for database publishing, in: Proc. EuroTEX'92, Prague.

[12]As the first attempt, a change of parameters for typesetting on a grid had been tried in the editor-supplied style file, but this broke down the `multicol` package on which that particular is based. This is a fairly typical result of using multi-layered systems of sufficient complexity when an unexpected combination of parameters is selected. Another manifestation of that caused by the use of LATEX with the so-called "new font selection scheme" (NFSS) interferes with `\ignorespaces` within a macro defined using `\aftergroup` so that it ceases to ignore spaces after its use: NFSS invokes its own macro `\glb@settings` after each group, and this unfortunately terminates the effect of `\ignorespaces` (not using NFSS got the problem disappeared).

even when text keeps changing (it doesn't appear in this paper, but it was used for [21]). The `\looseness` setting was used occasionally for fine-tuning of the output. The input source was nonetheless a LaTeX source, so this may be thought of as a constructive proof that even within LaTeX it is not necessarily impossible to produce a decently looking output.

## Bibliography


[1] Donald E. Knuth: *The TEXbook. Computers & Typesetting*, Volume A, Addison-Wesley, Reading, MA, 1986.

[2] Donald E. Knuth: *TEX: The Program. Computers & Typesetting*, Volume B, Addison-Wesley, Reading, MA, 1986.

[3] Donald E. Knuth: *The METAFONTbook. Computers & Typesetting*, Volume C, Addison-Wesley, Reading, MA, 1986.

[4] Donald E. Knuth: *METAFONT: The Program. Computers & Typesetting*, Volume D, Addison-Wesley, Reading, MA, 1986.

[5] Donald E. Knuth: *Computer Modern Typefaces. Computers & Typesetting*, Volume E, Addison-Wesley, Reading, MA, 1986.

[6] Donald E. Knuth, and David E. Fuchs: *TEXware*. CSR 86-1097, Stanford University, Stanford, 1986.

[7] Donald E. Knuth, and Michael F Plass: Breaking paragraphs into lines. *Software Practice and Experience*, **11**, 1981, 1119-1184.

[8] Donald E. Knuth: Virtual fonts: more fun for grand wizards. *TUGboat*, **11**, 1, 13-23.

[9] Donald E. Knuth: The Errors of TEX. *Software Practice and Experience*, **19**, 1990, 607-681.

[10] Donald E. Knuth: Literate Programming. *Computer Journal*, **27**, 1984, 97-111.

[11] Wayne Sewel: Weaving a Program: *Literate Programming in WEB*. Van Nostrand Reinhold, New York, 1989.

[12] *TUGboat*. TEX Users Group, Santa Barbara, CA.

[13] Leslie Lamport: LaTeX*: A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, 1985.

[14] Michael Goosens, Frank Mittelbach, and Alexander Samarin: *The* LaTeX *Companion*. Addison-Wesley, 1993.

[15] *Hart's Rules for Compositors and Readers at the Oxford University Press*. Oxford University Press, Oxford, 1983 (39th edition).

[16] Adrian Wilson: *The Design of Books*. Chronicle Books, 1993.

[17] Geoffrey Dowding: *Finer Points in the Spacing and Management of Type*. Wace, London, 1954.

[18] Sean Jennet: *The making of books*. Faber, London, 1964.

[19] Theodore Low De Vinne: *Modern Methods of Book Composition*. Century, New York, 1904.

[20] Philip Taylor: Book design for TEX users. Part 1: Theory. in: *Proceedings SOFSEM'93*, J. Staudek, ed., 1993, 293-305.

[21] Philip Taylor: Book design for TEX users. Part 2: Practice. in: *Proceedings SOFSEM'93*, J. Staudek, ed., 1993, 307-322.

## Appendix

The following is an example of a simple test-generating LaTeX source:

```
\documentstyle[a4wide]{article}
\pagestyle{empty}
\begin{document}
\parindent=0pt \hbadness=10000\emergencystretch=10cm\raggedright
\def\bigskip{\vspace*{\fill}}
\newcounter{number}\newcounter{test}\setcounter{number}{1}
\def\u#1{\mbox{\underline{{\fontshape{n}\tt#1\vphantom{p}}}}}
\def\p#1#2{\ifnum\value{number}=1 \clearpage
        {\Large{\bf Test sheet}}
        \hspace*{\fill} Student name: \makebox[6cm]{\dotfill}\\
        {\hspace*{\fill}\addtocounter{test}{1}\tiny\fbox{\thetest}}\\[10pt]\fi
        \ifnum\value{number}=3 \clearpage
        \hspace*{\fill} Student name: \makebox[6cm]{\dotfill}\\
        {\hspace*{\fill}\tiny\fbox{\thetest}}\\[10pt]\fi
        \par
        {\bf Problem \thenumber\ \addtocounter{number}{1}%
        \ifnum\value{number}>4\setcounter{number}{1}\fi
        (#2~points):}\hspace{\fill}\\#1}
%% Define your problems and number of points for their solution
%% as twelve definitions of the form (for problems to solve, three
%% versions for each of the problem):
%%      \def\one{{Text of the problem}{number-of-points}}
%% to
%%      \def\twelve{{Text of the problem}{number-of-points}}
%% For example:
\def\one{{Compute $1+1$}{12}}
%%  ... (and so on)
% The following is the program which generates sets of test sheets,
% four problems each, two problems per page, based on \p macro definition.
\newcount\a\newcount\b\newcount\c\newcount\d
\newcount\aa\newcount\bb\newcount\cc\aa=0\bb=0\cc=0
\newcount\A\newcount\B\newcount\C
\def\1#1#2#3{\expandafter\p#1\bigskip}
\def\2#1#2#3{\expandafter\p#2\bigskip}
\def\3#1#2#3{\expandafter\p#3\bigskip}
\a=1
\loop{
   \b=1
   \loop{
       \c=1
       \loop{
           \d=1
           \loop
                \A=\a \advance\A by \aa
                \ifnum\A>3 \advance\A by -3\fi
                \B=\b \advance\B by \bb
                \ifnum\B>3 \advance\B by -3\fi
                \C=\c \advance\C by \cc
                \ifnum\C>3 \advance\C by -3\fi\relax
```

```
                \csname\the\A\endcsname\one\two\three
                \csname\the\B\endcsname\four\five\six
                \csname\the\C\endcsname\seven\eight\nine
                \csname\the\d\endcsname\ten\eleven\twelve
                \advance\aa by 1
                \ifnum\aa>3\advance\aa by-3\fi
                \advance\bb by 1
                \ifnum\bb>3\advance\bb by-3\fi
                \advance\cc by 1
                \ifnum\cc>3\advance\cc by-3\fi
           \advance\d by 1
           \ifnum\d<4\repeat
       }\advance\c by 1
       \ifnum\c<4\repeat
   }\advance\b by 1
   \ifnum\b<4\repeat
}\advance\a by 1
\ifnum\a<4\repeat
\end{document}
```