Active TEX and the DOT Input Syntax

Jonathan Fine
Active TEX
203 Coldhams Lane
Cambridge, CB1 3HY
United Kingdom
fine@active-tex.demon.co.uk
http://www.active-tex.demon.co.uk

Abstract

The usual category codes give TEX its familiar backslash and braces input syntax. With Active TEX, all characters are active. This gives the macro programmer complete freedom in defining the input syntax. It also provides a powerful programming environment.

The DOT input syntax, like TROFF, uses a period at the start of the line as an escape character. However, its underlying element, attribute and content structure is based on SGML. It is both easy to use and easy to program for.

Conversion to other formats, such as SGML, HTML and XML, or to proprietary formats such as Word and RTF, will be straightforward. This is because the DOT syntax is rigorous. This new syntax will be described and demonstrated.

All manner of problems connected with TEX disappear when Active TEX packages are used. For example, all input errors can be detected and corrected before they can cause a TEX error message. This will make TEX accessible to many more users.

Visit http://www.active-tex.demon.co.uk for information and macros.

Introduction

Much has changed since the introduction of TEX in 1982. Computers have become cheaper, more plentiful and more powerful. The Internet has grown from a tool used largely by North American academics to become a mass medium subject to powerful commercial interests. And Microsoft, who supplied an operating system for IBM's first PC, has become a colossus.

Donald Knuth gave us a powerful and reliable typesetting system. Other systems may be easier to use and have all sorts of useful (and perhaps not so useful) features, but when it comes to the typographic quality of the resulting pages, TEX is still superior in many important respects to all of its competitors. No other software even comes close to matching it on its own home ground, which is technical books, articles and preprints that have large amounts of mathematical material.

Both individuals and publishers are now making information available on the Internet. This imposes new demands on the typesetting process. For many users, HTML (and perhaps soon its replacement, XML) is the preferred means for supplying and receiving textual material. Twenty years ago the typeset page was the principal result of the typesetting process. Today, users are wanting both typeset pages and HTML or similar pages. By typeset pages I mean both pages for printing in the usual way, and also pages for display, say in the Portable Document Format introduced by Adobe. (In principle, this term also includes the formatting of, for example, HTML, for display in a browser.)

Most TEX authors use a text editor (such as emacs) to prepare a computer file in the LATEX syntax, for example. Other authors will use a word processor to create a file that is stored in a proprietary format. Later down the line, these files will be typeset, converted into HTML and so forth.

By and large, the closer is the syntax of the author's file to being rigorous and compatible with the processing that will be applied to it, the better will be the outcome. Compromise may be necessary. With TEX each author became his or her own typesetter. Very often (LA)TEX files contain macro definitions, introduced for the author's own convenience. These definitions can be a great nuisance for those who have to deal with the file later,

particularly if they reside in an external file that becomes separated from the main manuscript.

This then is the present context for the use of TEX. Most TEX users now use the LATEX macro package, together with style files and additional packages. LATEX was developed in the early 1980s. The first edition of its manual was published in 1985, about a year after *The TEXbook*. It did a tremendous job of making the resources of TEX available to non-experts. Around 1990, however, its limitations became clear, and more than an inconvenience. One response was the birth of the LATEX3 project. In 1994 this group released LATEX2e. This helped to standardise the current situation.

Recently, Rahtz [11] described LATEX as "hugely powerful, but chaotic, and on the verge of becoming unmanageable." He also tells us that the CONTEXT macro package, due to Hans Hagen and Pragma, addresses this problem by incorporating into itself "all the facilities you need." It does away with document classes and user-contributed packages.

Plain TEX, LATEX and CONTEXT all use the familiar 'backslash and braces' input syntax. This can cause problems, because it is not rigorous. Translation to HTML, for example, requires that the source document be parsed. But LATEX, for instance, is in general the only program that can successfully parse LATEX documents. This tends to result in (LA)TEX living in a world of its own, isolated from the world of desktop publishing and word processing. For some communities of users, such as mathematicians, this may not be a hardship.

Active TEX is a new way of using TEX. It allows us to either avoid or solve many of our problems. For the technical, its key idea is that each character is active, and is defined to be a macro. For example, the active letter '`a`' is a macro that expands to the control sequence `lcletter`, followed by an active '`a`'. Uppercase letters, digits and visible symbols are treated in a similar manner. By manipulating these definitions, we can make TEX do whatever we want. In particular, we can choose our input syntax. Both TEX and the system macro programmer work harder, to ease the life of both the user and the application programmer.

We will consider the problems relating to macros under three heads, namely Input Syntax, Macro Programming, and the Processing of Text. The final section gives the history and prospects of Active TEX. This article is somewhat informal, and should not be read as a definitive or legally binding statement. The software is still under development.

The DOT input syntax

There are two aspects to an input syntax, namely the concrete and the abstract. The abstract syntax is the structure or organisation that the syntax provides. The concrete syntax is a means of expressing objects so organised. Provided they have the same abstract syntax, translation from one concrete syntax to another will be a routine matter. The parsing process starts with a concrete instance of the structure, and produces from it events that characterise its abstract structure.

In SGML the concept of the content model provides a large part of the abstract syntax. It might say, for example, that an article such as this one consists of front matter, sections and end matter. Each section would be a sequence of paragraphs, together with figures and tables. The end matter might consist of appendices and a bibliography. The latter would be a sequence of bibliographic items.

In LATEX one would write

```
\section{Input syntax}
```

to start a section. In SGML one might write

```
<section title="Input syntax">
```

to start a section. This gives two examples of a concrete syntax. In SGML the `title` is an *attribute* of the `section` tag. In LATEX, `Input syntax` is a *parameter* of `\section`.

The abstract syntax provided by SGML is solid and well-understood. It is already widely used in data processing. The concrete syntax, however, tends to be somewhat verbose and difficult to use without dedicated software. This has been an obstacle to its widespread use. In the author's view, with XML this problem will become more acute.

Twenty years ago or so, the text formatting programs `troff` and `nroff` were developed, as part of UNIX. In these systems, a dot at the start of a line is an escape character, which can be used to call a macro. For example

```
.SH  2.1  "Section heading"
```

might introduce a section.

The author has developed a syntax whose concrete form is similar to the dot syntax of `troff` and `nroff`, but whose abstract syntax is modelled on SGML. This syntax we call the DOT syntax. As in SGML, a tag name can contain digits, period and hyphen as well as letters. As a section is, say a second-level head, one could write

```
.h2 Input syntax
```

to start a section.

In LATEX one might write

```
\documentclass{article}
```

```
\author{Jonathan Fine}
\title{Active \TeX\ and input syntax}
\date{20 January 1999}
```

to start an article. In SGML terms, the author, title and date are all attributes of the article element.

As in SGML, the DOT syntax allows start tags to have attributes. One might write

```
.article  Active &TeX and input syntax
..author  Jonathan Fine
..date    20 January 1999
```

to specify the same information. This double dot notation for attributes is similar to the leading dots notation that TEX the program uses to show the content of boxes [8, page 66]. LATEX does not really have a concept of attributes.

An end tag in the DOT syntax is like so:

```
./article   This is a comment
```

but, as in SGML, end tags can often be implied by the context. For example, if a section cannot contain a section, the start of a new section implies the end of the current one.

SGML has the useful concept of a short reference. In the DOT syntax the start of a line, the end of a line, white space at the start of a line and a blank line are the possible short reference events. One can set matters up so that ordinary lines start paragraphs, blank lines end paragraphs and indented lines commence math mode. Thus the fragment

```
Einstein's famous equation
    E = m c ^ 2
expresses the equivalence of
matter and energy.
```

might be equivalent to

```
Einstein's famous equation
.eq
    E = m c ^ 2
./eq
expresses the equivalence of
matter and energy.
```

but the former is easier both to type and to read.

In summary, the DOT syntax combines the power of SGML with the simple concrete syntax of `troff` and `nroff`. It provides a concrete syntax that ordinary authors can use, whose abstract form is equivalent to that of SGML.

Macro programming

This section is particularly for the TEXnically minded. In Active TEX all characters are active. This is both a problem and an opportunity for the macro programmer. Ordinarily a line in a TEX file such as

```
\def \hello {\message{Hello world!}}
```

would define a macro `hello`, whose execution issues a greeting. This relies on the customary or plain category codes being in force. In Active TEX another approach must be taken.

Ordinarily, control sequences are formed using TEX's eyes. Thus, `\def` in the source file produces the control sequence `def`.

Active TEX uses the mouth of TEX, or more exactly `csname` and `endcsname`, to form control sequences. Macro definitions will be built up using aftergroup accumulation. The plain code line

```
\expandafter \aftergroup
\csname def\endcsname
```

contributes the control sequence `def` aftergroup.

Similarly, the lines

```
\aftergroup {\iffalse}\fi
\iffalse{\fi \aftergroup}
```

contribute left and right braces respectively. Finally, if the macro

```
\def \agchar #1{\expandafter
  \aftergroup \string #1}
```

is passed a character as an argument, it will contribute aftergroup the inert form of this character.

This mechanism allows us to define macros without making use of the ordinary category codes. For example, if we call `begingroup`, then `aftergroup` commands as detailed above, and then `endgroup`, the result could give exactly the same definition of `hello` as at the beginning of this section.

To store such definitions in a file, a syntax is required. Active TEX has been set up so that in a *compiled TEX code* (`ctc`) file, a line such as

```
def hello {message
  {'H'e'l'l'o' 'w'o'r'l'd'!}};
```

has exactly the same effect as the previous definition. Within a `ctc` file, a letter takes itself and all visible characters that follow, and uses `csname`, `endcsname` and `aftergroup` to form and contribute a control sequence. Similarly, active `{` and `}` contribute explicit (or ordinary) begin- and end- group characters `{` and `}` aftergroup. Active right quote `'` is as `agchar` above. Finally, the semicolon `;` closes the existing accumulation group and opens a new one.

This technique of aftergroup accumulation is enormously powerful. It allows arbitrary control sequences and character tokens to be placed into macro definitions. One can even do calculations or pick up values from an external file, as the definitions are being made. Tools are required to make full use of this power.

Suitable content in ctc files allows arbitrary macros to be defined. Active TEX has a development environment, which produces ctc files from suitable source code files. For example,

```
def hello
  { message { "Hello world!" } } ;
```

will when compiled produce the ctc code exhibited above.

Here is another example:

```
def ctc.letter
{
  begingroup ;
    string.visible.chars ;
    let SP endcs ; let TAB SP ;
    let RE suspend.RE ;
    let suspend endcs ;
  xa endgroup xa ag cs ;
}
```

This macro is used within ctc files to produce control sequences aftergroup.

Some comments are in order. Any visible characters can appear in control sequence names. This power should not be abused. We rely on the definitions

```
def string.visible.chars
{
  let lcletter string ;
  let ucletter string ;
  let digit string ;
  let symbol string ;
}
def suspend.RE { suspend ; RE } ;
```

being made already. The tokens SP, TAB and RE in the source file produce (and here it is a mouthful) characters in the ctc file that in turn produce active space, tab and end-of-line characters aftergroup. The tokens xa, ag, cs and endcs in the source file are shorthand for expandafter, aftergroup, csname and endcsname respectively. It is the latter strings that are written to the ctc file by the compiler. Semicolons in macro definitions are for punctuation only. They are ignored. Outside macro definitions they trigger renewal of the aftergroup accumulation group.

This process, of defining macros via ctc files, allows many of the basic problems in TEX macro development to be solved. For example, one can insist that identifiers (tokens in the source file) be declared before they can be used. No more misspelt identifier names! One can also apply a prefix to chosen identifiers, thus segmenting the name space. This will allow a module to control access to its identifiers. No more name clashes!

In the same way, one can use named rather than numbered parameters in macro definitions. For example, instead of

```
\def \agchar #1{\expandafter
  \aftergroup \string #1}
```

as above, one could write

```
def ag.char Char { xa ag str Char } ;
```

where Char has been previously declared to be a macro parameter place holder.

Although this process is somewhat indirect, it does not cause performance to suffer. The compilation process, to produce the ctc files, needs to be done only once, by the macro developer. With modern machines, this does not take long. Similarly, most files will be loaded only once, in the process of making a preloaded format file.

In fact, Active TEX gives two performance benefits. The first is that macro programmers no longer need to resort to tricks, to obtain access to unusual control sequences or character tokens. Thus, more efficient code can be written. The second is that ctc files are generally quite compact. This compression allows them to be retrieved from the hard disc (or network) more rapidly. The DOT syntax gives the same benefits.

Tools for macro programmers are under development. For example, short references will cause indentation to indicate code lines. This section has given examples of what has been done already, and a taste of what lies in the future. The author invites comments.

Processing text

We now turn to the raison d'être of TEX, which is of course typesetting. In §2 we saw how the DOT input syntax allows a document to be broken down into elements with attributes. In §3 we saw some examples of how Active TEX can be programmed. This section is concerned with the content of the document or, more exactly, with the text and the attribute values.

Typesetting plain text, such as

```
This is plain text.
```

is straightforward. Each visible character produces itself, and spaces give interword spaces. Thus the values

```
let lcletter string ;
let ucletter string ;
let digit string ;
let symbol string ;
def SP { unskip ; ~ } ;
```

will, to a first approximation, suffice. (The `unskip` is present so that multiple space characters will count as one. The `~` is Active TEX's way of calling for an ordinary space character.)

Occasionally, the user will want to add *emphasis*. In SGML one uses tags

```
<em>emphasis</em>
```

like so, while in LATEX one uses a macro

```
\emph{emphasis}
```

but in Active TEX one might use

```
Plain text with {emphasis}.
```

for example. Because `{` and `}` are active, they can be programmed to open and close an emphasised group.

This brings us to perhaps the most important concept of this section, which is that of a *data content notation* (DCN). Roughly speaking, such tells us how text is to be processed. For example, the plain text above already has a DCN, namely that it is in English. This is very important if we are using a spell checker or a search engine. Computer programming languages are more formal examples of a data content notation. Mathematics encoded in either plain or LATEX is a third example.

The DOT syntax will be set up so that a data content notation can be associated to the text in each element, and to the text in each attribute value. The DCN will, in a more or less formal manner, tell us what is admissible and how it should be processed. The specification of a DCN is not, however, a matter for the DOT syntax. Rather, it is for the users and experts in the area. Many countries have official bodies that attempt to regulate and bring order to the use, at least in printed form, of a language.

The author suggests as a first step that for plain text a DCN along the following lines be used. For emphasis use `{` and `}`. For bold use `+` and `+`, and for math use `$` and `$`. For verbatim use `|` and `|`. Certain nestings will be prohibited. The following

```
Plain text, {emphasis},
|verbatim| and +bold+,
with some elementary
$2+2=4$ mathematics.
```

is an example of its use.

In math mode, new rules will be required. The author suggests that ordinary TEX but without the backslashes, like so

```
sin ^2 theta + cos ^2 theta = 1
```

as a first approximation. This is only a beginning. Building up a complete system that is capable of handling the complexity of modern mathematical notation, whilst retaining both rigour and ease of use, is not going to be an easy matter. Gaining general acceptance and support of the user community is as much a problem as the formulation and solution of the technical problems.

When SGML is used for markup, there is a tendency to use it for everything, regardless of size. This causes enormous problems to users who either do not have the software tools required, or who prefer to work with plain text files. For example, in the C programming language the `&` operator gives the address of a variable. Code fragments such as

```
ptr = &i ;
```

are common. But in SGML, `&` followed by a letter gives an entity reference, so for an SGML parser to produce the above as output, it must be given

```
ptr = &amp;i ;
```

as input. Something similar happens if one wishes to produce `a<b` as parser output, for the `<b` must not be recognised as a start tag.

Part of the philosophy of the DOT syntax is that it deals with the big things (and also some of the medium sized) while the data content notation deals with the little things. The DOT syntax will have its parser, and each DCN will have its parser. They take turns in processing the input stream.

Let us now return to typesetting. Most of the time, when TEX is typesetting, it is forming either a horizontal list or a math list. Each DCN will, as it processes characters, add items to the current list. Special characters (such as `$`, `{` and `}`) will change the mode in some way. From time to time, say at the end of a title, the current list will be closed and material will be added to the main vertical list, for example. From here on the main difference between plain TEX, LATEX, CONTEXT and Active TEX will be in the libraries of macros used for page makeup, output and so forth.

Typesetting is the purpose of a TEX macro package. TEX was developed to allow typesetting of the highest quality. However, not until the basic functions of Active TEX have been met will it be possible to move on to the typesetting (composition, hyphenation and justification, galleys and page makeup) aspects of the process. Put another way, macro packages such as plain and LATEX have typesetting as their main purpose. Rigour, power and ease of use are the main goals of Active TEX, at least in this stage of its development. A fourth goal is to provide an enduring fixed point for document source files.

History and prospects

Although the basic concept of Active TEX is quite simple — all characters are active — it is surprising just how long it has taken for this idea to emerge. A brief history follows.

In plain TEX the tilde ~ is an active character, which produces an unbreakable interword space, and in math mode apostrophe ' is effectively an active character, used for putting primes on symbols, as in $f'(x)$. Technically, a prime is a superscript. In addition space and end-of-line could be made active, to achieve special results such as verbatim listing of files. In 1987 Knuth [9] wrote about some macros he had written for his wife, in which many of the symbols are active.

In 1990 Knuth froze the development of TEX. In his announcement [10] he wrote:

> Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

In 1992 Fine [2] produced the `\noname` macro development environment, which, like Active TEX, is based on aftergroup accumulation. This solves a major technical problem, namely how to define exactly the macros we wish to define, when the category codes are against us. The `\asts` problem [8, page 373] at the start of Appendix D (Dirty tricks) is solved using aftergroup accumulation.

In 1993 Fine presented a paper [3] to the 1993 AGM of the UK TEX Users Group that contains in embryonic form most if not all of the ideas in this paper. For example, he wrote

> Let us solve all category code problems once and for all by insisting that *the document be read throughout with fixed category codes.*

and then described how '\', for example, could be an active character that parses control sequences, in much the same way as `ctc.letter` does. The paper also contains other prospects that have not been discussed here, such as visual or WYSIWYG TEX (see [7] for a more recent presentation.)

In 1994 Fine [4] argued that the deficiencies in TEX the program had been exaggerated, and that "It would be nice if both TEX and its successor shared at least one syntax for compuscripts that are to be processed into documents" (p. 385). This syntax would have to be rigorous.

In 1994–95 Fine went the whole way, and made all characters active. Using this, he produced a prototype TEX macro package that was able to typeset directly from SGML document files. Due to lack of both sponsorship and commercial interest, the project was left unfinished. This work was presented at the Bridewell meeting (January 1995) on Portable Documents, and published both in Baskerville [5] and MAPS [6], but regrettably not in the special *TUGboat* issue **16** (2) on TEX and SGML, published later that year.

In late 1997 the project was revived, and in May 1998 Fine spoke on Active TEX and input syntax at a meeting of the UK TEX Users Group. Since then a proof-of-concept version of the macros has been available to all those who ask.

There have been other developments that make extensive use of active characters. Michael Downes [1] has done important work on the typographic breaking of equations. He writes (p. 182):

> Some of the changes are radical enough that it would be more natural to do them in LATEX3 than in LATEX2e — e.g., for LATEX3 there is a standing proposal to have nearly all nonalphanumeric characters active by default; having ^ and _ active in this way would have eased some implementation problems.

Werner Lemberg [12] describes the CJK (Chinese, Japanese and Korean) package. This package makes the extended ASCII or eight-bit characters active. He notes (p. 215):

> It's difficult to input Big 5 and SJIS encoding directly into TEX since some of the values used for the encodings' second bytes are reserved for control characters: '{', '}' and '\'. Redefining them breaks a lot of things in LATEX; to avoid this, preprocessors are normally used [...].

Active TEX has been designed from the ground up so that it can go the whole way, and allow problems such as these to be given completely satisfactory solutions, without unnecessary difficulties. The only real price seems to be performance. Because it does much more, it is not as quick as plain TEX or LATEX. This could be a problem for those who use a 286, but on a 486 or better, disk input/output is the real bottleneck.

One of the great things about TEX the program is that it is a fixed point. I would like Active TEX to become a similar fixed point, upon which users can build style files and the like. I would also like the DOT syntax to become a fixed point.

When TEX was developed, Donald Knuth had [8, page vii] the active interest and support of the

American Mathematical Society, the National Science Foundation, the Office of Naval Research, the IBM Corporation, the System Development Foundation, as well as hundreds of more or less ordinary users, many of whom went on to play an active part in the life of the TEX Users Group, and the community generally, and some of whom are still with us.

I firmly believe that Active TEX is a worthwhile idea whose time has come. Please give it your support.

References

[1] Michael Downes. "Breaking equations." *TUGboat* **18**(3), 182–194 (1997).

[2] Jonathan Fine. "The `\noname` macros — A technical report." *TUGboat* **13**(4), 505–509 (1992).

[3] ———. "New perspectives on TEX macros." *Baskerville* **3**(2), 17–19 (1993).

[4] ———. "Documents, compuscripts, programs and macros." *TUGboat* **15**(3), 381–385 (1994).

[5] ———. "Formatting SGML manuscripts." *Baskerville* **5**(2), 4–7 (1995).

[6] ———. "Formatting SGML manuscripts." *MAPS* **14**, 49–52 (1995).

[7] ———. "Editing `.dvi` files, or Visual TEX." *TUGboat* **17**(3), 255–259 (1996).

[8] Donald E. Knuth. *The TEXbook*. Addison-Wesley (1984).

[9] ———. "Macros for Jill." *TUGboat* **8**(3), 309–314 (1987).

[10] ———. "The future of TEX and METAFONT." *TUGboat* **11**(4), 489 (1990).

[11] Sebastian Rahtz. Editorial. *Baskerville* **8**(4/5), 1 (1998).

[12] Werner Lemberg. "The CJK package for LATEX 2_ε: Multilingual support beyond `babel`." *TUGboat* **18**(3), 214–224 (1997).

(untitled)

Oh, what a tangled web is TEX,
what you need to reach success.
To make sure you don't get a reject,
why not get some help from TEX.

—Peggy Kempker

(untitled)

Breathes there the man with the eye so blind
Who never for himself can find
The *cos*sine, 'c' times 'o' times 's'
The `\acronym` run on to the rest
The sentence ending at *et al.*
Although no verb shall yet befall
Until a phrase two lines below
The endquotes where the quotes should go?

If such there be, away go he
To Delaware for a Ph.D.!
The thesis clerk no more shall check;
The only folks to judge his TEX
Are senior faculty, by norm
Concerned with content, not with form
His approval page they'll surely sign;
The publisher, with wit sublime,
His words in XML encases;
And, should he be obscure in places,
With sentence structure ill-prepared,
His authorship will now be shared:

Credit will to the copy editor belong
Grammatically correct, but scientifically wrong.

—Stephen Fulling