TEX to HTML Translation via Tagged DVI Files [*]

Michael D. Sofka
Computing Information Services
Rensselaer Polytechnic Institute
Troy, New York 12180-3590
sofkam@rpi.edu
http://www.rpi.edu/~sofkam/

Abstract

This paper describes `dvihtml`, a program under development for translating a tagged DVI file into HTML. A common problem when translating TEX into another format is handling unexpected macros. Fortunately, TEX's macro language is flexible enough to pass markup information to the DVI file in the form of `\special`'s, fonts and small horizontal or vertical movements. Translating the resulting DVI file thus allows TEX itself to serve as the macro parser for translation. This technique can be extended for writing smarter DVI viewing programs, including viewers that can perform common layout editing.

A common typesetting request is the ability to place copies of books and articles on the Web in HTML, or to provide files in SGML or common word processor format. To aid in this task, many translators have been written that read TEX or LATEX files and write the appropriate output. Translators that read TEX files directly, however have the common limitation of not understanding TEX's macro language, or even being fooled by macros that simply redefine a common command already known to the translator. Add to this the inconsistency with which some authors (and typographers under the pressure of a deadline) code TEX files, and a uniform and universal translator seems a hopeless task.

TEX authors commonly write new macros that generate content or important layout not understood by the translator. In order to handle arbitrary macros the translator must be updated, or new translation tables supplied. Even then, a macro writer could fool the best translators by redefining the input syntax to better suit idiosyncratic work habits. For this reason most TEX translators have targeted specific input languages, usually LATEX. This is the method used by LATEX2html and Scientific Word, which are both discussed elsewhere in these proceedings (Deland, 1998, Moore, 1998). It is also the method used by IBM Techexplorer, which understands LATEX and a wide range of TEX's math primitives and plain TEX macros (Sutor and Dooley, 1998).

Alternatively, one could write a translator that understood TEX's primitives and macro language. This, however, is a daunting task given the many special cases embodied in TEX's expansion rules. Fortunately, a readily available TEX translator exists which is guaranteed to understand and correctly interpret any TEX file. The program is, of course, `initex`, the TEX executable itself. The only problem is that the output of TEX is a low-level DVI file in which most of the high-level document structure is lost.

Using the `\special` command and some other macro tricks, however, TEX can translate a document into a "tagged" DVI file. A tagged DVI file is a DVI file which encodes information about the higher-order coding which produced the lower-level DVI output. This tagging, along with the hierarchical structure of the DVI file, can be used to create HTML or other output according the user specifications. Depending on the specific restrictions required by the target language, the tagging need not even be complete. For example, HTML encodes headers as:

```
<H1>LaTeX and Postmodern Typesetting:
Hermeneutics and the Tyranny
of Documentclass Structure.</H1>
```

with no regard to specific font, size or line breaks. Indeed, this information should be left to the display program when standard HTML is the desired outcome. The only information required in the DVI

[*] I would like to thank Sebastian Rahtz and Eitan Gurari for their helpful comments on an early draft of this paper. I would also like to thank my managers at RPI, Gary Schwartz and Katherine Bursese, for the quiet time at work to finish this article, and for allowing me attend the Northeast TUG Conference.

file is a "tag" identifying which characters are in the header.[2]

Such is the flexibility of TeX's macro language, that the original author coding may not need to be modified. A LaTeX package file, for example, could redefine common commands to produce tags. The same package could further redefine primitives and definition commands so that all new macros will either be tagged, or will at the least not interfere with the translation process. Problem commands which do not generate content important for HTML display (such as running heads, page breaks, etc) can be disabled or tagged and ignored during translation.

The DVI File

While most TeX users are aware that the output of TeX is something called a "DVI" file, fewer have ever had the opportunity to study this file in detail. Indeed, this task is difficult since the file is binary and displays poorly in most editors. I suspect this is one reason various flavors of TeX input files have been the source language of choice for translation (the other being the lack of high-level information within the DVI file).

DVI files, however, are really simple. As described in Knuth (1986b) they consist of a series of 1-byte commands and parameters which compactly describe how characters and rules should be placed on a page. There are 250 DVI commands in all, but most are for setting characters and changing fonts. In addition, many commands come in four flavors depending on if the parameter is 1, 2, 3 or 4-bytes long. Full details on the DVI file format, along with sample code for reading DVI files, can be found in `dvitype.web` (Stanford University, 1995).

Depending on how you group the commands there are about 11 categories of DVIoperation codes (or op-codes, as they are called). The entire set of op-codes is shown in Table 1.

Category	*op-codes*
Print Character	*set_char0* . . . *set_char127*, *set1*, *set2*, *set3*, *set4*, *put1*, *put2*, *put3*, *put4*
Select Font	*fnt_num0* . . . *fnt_num63*, *fnt1*, *fnt2*, *fnt3*, *fnt4*
Define Font	*fnt_def1* . . . *fnt_def4*
Print rule	*set_rule*, *put_rule*
Horizontal Movement	*right1* . . . *right4*, *w0*, *w1* . . . *w4*, *x0*, *x1* . . . *x4*
Vertical Movement	*down1* . . . *down4*, *y0*, *y1* . . . *y4*, *z0*, *z1* . . . *z4*
Header	*pre*, *post*, *post-post*
Page	*bop*, *eop*
Stack	*push*, *pop*
Special	*xxx1*, *xxx4*
Undefined/nop	*nop*, *250–255*

Table 1: DVI op-codes by category. Note that 136 commands are used to print characters, another 68 for fonts and 28 for moving within the DVI file.

There are a few items to note from Table 1. First, fully 136 of 250 DVI op-codes are used to print characters, and another 68 are used to select a font. Likewise, there are 14 horizontal and 14 vertical movement commands. Font definitions, which provide a mapping between an external font name and a DVI file font number, take another 4 bytes. This profligate consumption of op-codes for setting characters is done for efficiency. The letter 'G' in Computer Modern, for example, can be typeset with the DVI op-codes

fnt_num0 set_char71

which is only two bytes in the file. The word "Gentle" can be typeset using a total of 7 bytes, plus three bytes for a *right2* command (one for the command, and two for the parameter) which kerns between 'n' and 't'.

Second, there are a number of commands of the form *op*⟨n⟩ where ⟨*n*⟩ is the value 1, 2, 3 or 4. For example, *right1*, or *fnt2*. These are variations of a single op-codes which take a 1, 2, 3 or 4 byte parameter. TeX tends towards using the more efficient op-code to represent a value.

Third, the movement parameters *w1–4*, *x1–4*, *y1–4* and *z1–4* are register commands. They move the given distance and set the value of the corresponding *w*, *x*, *y* or *z* register. These register values can then be recalled using the one byte *w0*, *z0*, *y0* or *z0* commands. TeX tends to use the horizontal registers for word spaces and kerns, and the vertical registers for movement between lines and paragraphs.

Fourth, the *push* and *pop* commands store and retrieve the current values of the *w*, *x*, *y* or *z* registers and the current horizontal and vertical position on the DVI page. TeX uses these to slightly optimize parameter setting. More important for translating tagged DVI files, TeX outputs *push*/*pop* pairs which correspond to boxes in the original TeX file. This

[2] There are additional issues such as handling simple math in a header, and finding correct word boundaries. These are addressed below.

correspondence is not 100%. Particularly, TEX optimizes the output of lines from paragraphs so that most boxes are removed from common baselines. But in math-mode and tables most of the boxes remain.

Finally, the *xxx1* and *xxx4* are how TEX outputs `\special`'s to the DVI file. The literal (macro expanded) text of the `\special` is placed in the file. The single parameter of the *xxx* command is the length of the special. It is entirely left to the DVI translator program to interpret what a `\special` means, and the macro writer to be sure the contents of a `\special` are correct, and correctly located within the DVI file.

Tagging a DVI File

How can information in the DVI file be used to recover high-level coding? The trick is to use TEX's superlative macro language to send markup information, embedded in the DVI file, to the translator. The markup information can be indicated in at least three ways: distance, fonts and `\special`'s. Further, much of the marking can be accomplished by redefining existing TEX macros and primitives, reducing intervention into the authors coding.

Tagging using distance. One source of tagging information in a DVI file is the size of horizontal and vertical movements. TEX use the *w* and *x* registers for movement between words, but the amount of a move will vary from line to line. Likewise, movement between lines and paragraphs is accomplished with the *y* and *z* register commands. A typical DVI sequence (simplified) representing two lines in a paragraph is:

```
push
    right3⟨n1⟩fnt_num0
    set_char71 set_char101 set_char110 right2⟨n2⟩
    set_char116 set_char108 set_char101 w3⟨n3⟩
    set_char114 set_char101 set_char97 set_char100
    set_char101 set_char114 set_char115 w0
...
pop
y3⟨m⟩
push
    set_charn...
pop
```

That is, each line is nested in a *push*/*pop* pair. Within this pair the *w* register is used for inter-word spacing, while a *right* or the *x* register is used for kerns. Each line is separated by a *y* register command. In addition, paragraphs are usually separated by a *z* register command if `\parskip` is non-zero.

The problem is that while words are *typically* separated by *w* register commands, not all *w* commands are the result of word spaces. When generating the DVI file, TEX will optimize horizontal and vertical movements within boxes by using the *w*, *x*, *y* and *z* registers. A kern might be a *right*, or it could be a *x* command if a kern of the same amount appears later in the same line (a frequent occurrence). The details of this optimization are in Knuth, 1986b.

Fortunately, TEX's macro language can help us out. Consider the following TEX code.

```
\spaceskip=1sp
\xspaceskip=1sp
\hsize=\maxdimen

\baselineskip=1sp
\lineskip=0pt
\lineskiplimit=-16383pt

\parskip=0pt
```

The first two lines set the value of word spaces to one scaled point (sp). A scaled point is 1/65536th of a point, and is the smallest unit that TEX can move. Under normal circumstances there are no distances of 1 sp in a DVI file. Typical distances actually found are measured in at least $^1/_{10}$ of a point units.

The third line sets the width of a paragraph to the value of `\maxdimen`, which is 16383.9999 pt or about 18.9 ft—longer than a typical paragraph. The combined effect is to turn off line breaks making each paragraph a single line, and move exactly one scaled point between each word.

The next three lines adjusts TEX's vertical list building so that one scaled point is placed between each line (each paragraph) of text. This is accomplished by first setting `\baselineskip` to 1 sp then turning off other interline glue by forcing TEX to never use `\lineskip` glue.

Finally, `\parskip` is set to 0 pt so that no additional glue is added between paragraphs. The same overall effect could be accomplished by setting:

```
\cs{baselineskip=0pt}
\cs{parskip=1sp}
```

The sum effect is that one can be reasonably sure that all 1 sp horizontal movement in the DVI file represent word spaces, and all 1 sp vertical movement represents paragraphs. All other movement can be ignored, unless it to is being used for tagging.[3]

[3] Variations on the above allow for normal hyphenation and justification, but mark lines and paragraphs with one and two scaled point vertical movements. Recovering exact

word boundaries would be more difficult, but the resulting paragraphs would be legible and formatted by TEX.

A potential problem remains in that a later macro might be expected to set the `\baselineskip`, `\parskip` or other values, or even restore `\hsize` something under 8 inches. Fortunately this can be prevented with the following commands.

```
\newskip\junkskip
\let\spaceskip=\junkskip
\let\xspaceskip=\junkskip
\let\baselineskip=\junkskip
\let\lineskip=\junkskip
\let\parskip=\junkskip

\newdimen\junkdim
\let\lineskiplimit=\junkdim
\let\hsize=\junkdim
```

To be thorough we should also disable vertical and horizontal movement commands such as `\vskip` and `\hskip`. Care must be taken, however, to ensure the semantics of such commands otherwise remains the same.

Tagging using fonts. A second method of sending tagging information to the DVI file is by fonts. There are two ways a font can be used to indicate output format: name and size. For example, in a particular document the font Palatino at 16 point might only be used in one-heads. This is a clear indication that during translation all 16 point Palatino and intervening rules should be set within `<H1>/<H2>`.

What if the design includes a three head in 10 point Optima, but 10 point Optima is also used for figure captions. How can the two be distinguished based only on fonts? One method would be to increase the font size by one scaled point. The difference between Palatino at 655360 sp and Palatino at 655361 sp is will have no discernable affect on appearance, but they will be two different fonts in the DVI file.

There are two drawbacks to using fonts to tag markup information. First, it uses more fonts. TEX has a limit of 256 fonts per DVI file, so any method that makes extensive use of fonts will need to carefully select which fonts are actually loaded and used. Second, each font can only carry one tag. Setting, for example, `\it\bf` will result in only the boldfaced font being used.

Tagging using specials. Nearly any tagging information can be included in a DVI file by using TEX's `\special` command. The `\special` command causes TEX to out insert the literal, macro expanded argument, into the DVI file as an *xxx1* or *xxx4* command, depending on the length of the string. This is among the more heavily used and abused features of TEX since specials are used for all rotation, color, figure inclusion and PostScript commands.

The major disadvantage of specials is that they require DVI interpreters which understand the specific specials used—interpretation of specials is outside the purview of TEX. As a result, there appeared a number of drivers which understood only specific sets of specials. Some of these drivers were commercial or were used internally by typography companies, and made use of `\special`'s which were not in general use. Others were freely available, but as a result lagged behind in the special sets accepted.

In 1997 Tom Rokicki (Rokicki, 1994) proposed a set of specials to be supported by his `dvips` program. This was recomended with modification by the TUG Technical Working Group on DVI Driver Implementation and Standardization Issues(Rokicki, 1995). While the proposed standard has inherent flexibility, it cannot be used for all `\special` needs. Specifically, it doesn't cover markup tagging, and its stack scheme doesn't allow for DVI file re-writing (as described below). It does, however, propose a standard method of writing non-standard macros which will be followed in `dvihtml`. See Sofka (1995) for more details of the standardization process.

Delimited tags. In principle, markup via the `\special` primitive is easy. To mark a section, for example, would require:[4]

```
\catcode`\@=11

\let\t@gsection=\section
\def\section#1{%
   \special{::tag begin(section)}%
   \t@gsection{#1}%
   \special{::tag end(section)}}

\catcode`\@=12
```

assuming the macro `\section` had previously been defined.

The `\let` primitive is used to preserve the true definition of `\section`. The new definition is same as the old, except `\special` places tags around it.

The `::` identifies the special as being experimental according to the draft standard. The type of special is a "tag", which means it is providing high-level information for an interpreter. The `begin` and `end` indicate that the high-level element is delimited by two specials. `section` is the name of the tag.

[4] My examples are in plain TEX to keep them simple. The same can be done in LATEX by suitably redefining basic generator macros such as `\@startsection`, `\new@command`, etc.

Block scoped tags. Not all tag-able elements can be delineated using begin and end markers. Sometimes the the range of an element is implicit in the coding, but not explicitly marked. For example, when processing:

```
$$ABCE\over DEFG$$
```

"ABCD" is in the numerator, while "DEFG" is in the denominator. It would be awkward to require plain TEX users type

```
$$\special{::tag begin(numerator)}
     ABCE
  \special{::tag end(denominator)}
\over
  \special{::tag begin(denominator)}
     DEFG
  \special{::tag end(denominator)}$$
```

when inputing math—even if suitable shorthand tagging macros were defined. However, a tag can be inserted into the scope of the numerator and denominator by redefining the `\over` primitive as:

```
\def\tag#1{\special{::tag block(#1)}}

\catcode`\@=11
\let\t@gover=\over
\def\over{\tag{num}\t@gover\tag{den}}
\catcode`\@=12

$${ABCE \over EFGH}$$
```

This is output in the DVI file roughly as:

push
    set_char65...
    xxx1⟨*16*⟩::tag block(num)
pop
right4⟨*n*⟩
down3⟨*m*⟩
putrule⟨*a*⟩⟨*b*⟩
down3⟨*m*⟩
push
    xxx1⟨*16*⟩::tag block(den)
    set_char69...
pop

Note that the contents of the numerator and denominator are each contained within a *push*/*pop* pair. The `block` type of `::tag` affects the entire block within which it is contained.[5]

Nested tags. The `::tag` specials can be nested. For example, a tag for italic text (assuming this were not indicated using a font) might be nested within the tag for a section. There is an ambiguity, however, when a delimited tag and a block tag interact. How, for example, should the following be interpreted?

push
    xxx1⟨*17*⟩::tag begin(list)
    set_char71 set_char101...
    xxx1⟨*18*⟩::tag block(quote)
    set_char108 set_char111...
    xxx1⟨*15*⟩::tag end(list)
pop

Is the quote contained within the list, or the list within the quote? When the order of application matters, the resulting output will be different for each interpretation.

By default this will be resolved by assuming that block tags are delimited by begin/end tags, as well as *push*/*pop* pairs. That is, internally, `dvihtml` or other `::tag` aware translator should convert the above into:

push
    xxx1⟨*17*⟩::tag begin(list)
    xxx1⟨*18*⟩::tag begin(quote)
    set_char71 set_char101...
    set_char108 set_char111...
    xxx1⟨*16*⟩::tag end(quote)
    xxx1⟨*15*⟩::tag end(list)
pop

Why does the `end()` tag specify the element being ended? Wouldn't a simple `end` with no argument be enough to end the current tag? Unfortunately no. The problem is TEX's asynchronous output routine. This means that in the middle of a paragraph of quoted material you may suddenly find yourself in the middle of page layout. The result is the following sequence in the DVI file:

push
    xxx1⟨*18*⟩::tag begin(quote)
    set_char$n_1 \ldots n_x$
pop
xxx1⟨*15*⟩::tag end(page)
pop
eop
bop⟨$c_0, \ldots, c_9, p$⟩
right3⟨*4736286*⟩
push
xxx1⟨*17*⟩::tag begin(page)
push
    set_char$n_{x+1} \ldots n_z$
    xxx1⟨*16*⟩::tag end(quote)
pop

[5] There is an annoying rule between the two tags in this example. If rules are being translated, this one can be removed by redefining `\over` using the `\atop`. If the rules used in `\over` need special treatment they can be set with a 1 sp width using `\above`.

If the output routine were also tagging elements (e.g., top of columns, crop-marks, running head, and so on), they would all appear between and interlaced with the `quote`. Explicite `end` statements with matching parameters helps the above be rewritten as:

```
push
    xxx1⟨18⟩::tag begin(quote)
    set_char n_1 ... n_x
    xxx1⟨16⟩::tag end(quote)
pop
xxx1⟨15⟩::tag end(page)
pop
eop
bop⟨c_0, ..., c_9, p⟩
right3⟨4736286⟩
push
xxx1⟨17⟩::tag begin(page)
push
    xxx1⟨18⟩::tag begin(quote)
    set_char n_{x+1} ... n_z
    xxx1⟨16⟩::tag end(quote)
pop
```

The problem is knowing exactly where in the DVI file to insert matching `begin` and `end` tags. There are at least three ways to resolve this. The first is the method shown above, which uses explicit tags in the output routine to delimit pages and columns. All that is necessary for correct rewrite is inserting
textend tags at the same nesting level as the matching `begin`, but before the end of page is marked. Likewise for `begin` tags at the top of the page. The assumption is that all `begin`/`end` pairs should perfectly nest in the rewritten DVI file.

A second method of resolving this problem, applicable only to a translator, is to redefine macros so that page breaks do not occur at inopportune times. For example, setting spacing and paragraph parameters as given above guarantees that page breaks will not occur in the middle of a paragraph. By further defining `\output` to be simply `\` other interrupted tags can be reconstructed. Alternatively, the techniques discussed in Appendix D of Knuth (1986a) can be used to signal the output routine about bad break points.

Finally, it is possible to reconstruct the original nesting of the `begin`/`end` pair by merging all intervening *push*/*pop* pairs nested at the same level as the interrupted tags. This method works, however, only if it is assumed that *push*/*pop* pairs and `begin`/`end` perfectly nest—a condition that requires careful macro writing since TEX has no way of enforcing the rule.

All three of the methods are used in `dvihtml`. Macro and simplification will be used when possible, tag nesting will be encouraged and nesting rewrites will be used whenever it can simplify the coding. The goal is a minimal re-write of author macros, so the translator must make use of all the information available in the DVI file.

Overriding scope. There are times when it may be necessary to override the default scope of a `::tag` special (for example, if a `block` tag should be moved outside of a delimited tag. This can be done using the `scope()` option, which takes a single parameter indicating what the scope for the current tag should be. There are special cases for `global` scope and `page` scope, to affect the entire DVI file or the page on which the tag appears. `stack` specifies the current *push*/*pop* pair. Otherwise, the parameter should be label of a delimited tag which encloses the new tag at any level.

What about alignments? The alignments commands used by TEX present a mixed bag of difficulties. Redefining & and `\cr` to provide block-level tagging is trivial, but this breaks the `\halign` alignment template. While scanning the alignment template TEX is expecting category 4 characters to indicate tabs, and a real `\cr` (or `\endline`, which is defined in `virtex`) to end the template. So, while pre-defined math alignments such as `\eqalign` can be handled via:

```
\def\tag#1{\special{::tag block #1}}

\catcode`\&=\active
{\catcode`|=4\gdef&{\tag{AMP}|}}

\catcode`\==\active
\def={\tag{EQ}\char`\=}

\def\cr{\tag{CR}\endline}

$$\eqalign{A&=B\cr
          B&=D}$$
```

This same code breaks any future `\halign` attempts. Tagging alignment entries requires something slightly more convoluted. An example of how to do this is in figure 1, which redefines `\halign` so that & is a tab character while the template is being scanned, but is an active character while the body of the alignment is being read. The active character inserts tag specials.

Dvihtml and Tagged DVI Files

An outline of the proposed tagging `\specials` is in figure 2.

```
\catcode`\@=11
\def\tag#1{\special{::tag block #1}}

\def\makebraceother{\catcode`\{=12 }
\def\makebracenormal{\catcode`\{=1 }

\def\maketabactive{\catcode`\&=\active}
\def\maketabtab{\catcode`\&=4 }
{\maketabactive \catcode`|=4\gdef&{\tag{lamp}|\tag{ramp}}}

\let\t@ghalign=\halign

% Remove the { from \halign
{\makebraceother \catcode`[=1 \catcode`]=2
   \gdef\@halign{[\makebracenormal\@@halign]}

% Collect alignment template and call halign primitive
\def\@@halign#1\cr{\t@ghalign\bgroup#1\cr\global\maketabactive}

% set catcodes and start halign
\def\halign{\makebraceother\maketabtab\@halign}
\catcode`\@=12
```

Figure 1: Redefining `\halign` so that & is category code 4 (tab) while the alignment template is being read, but active characters while the body of the `\halign` is read. The above introduces a potential problem in that & remains active between `\halign`'s. This is okay for most macros built using `\halign` because the alignment template was read when the macro was defined. This macro also breaks plain TeX's tabbing macros.

The `dvihtml` translator understands these specials, and uses them to re-write the DVI file so that hierarchical information is preserved, and tagging applied to the appropriate elements. It will optionally write out a new DVI file, or a translated tagged output file (HTML by default). Translation is guided by a configuration file specifying conversions for horizontal and vertical movements, fonts and `::tag` specials. By default, tags labels will be converted verbatim so that in the absence of additional information the ASCII output file will have intelligible markup. A sample `dvihtml` configuration file is in figure 3.

In the case of LaTeX files, a package can be written which redefines the standard commands to produce tagged output. Plain TeX is, of course, trickier since there is no way of knowing in advance what an author will call a macro. Adding a couple `\special` calls, however, is relatively easy and by default the translation will pick up changes in font size, paragraphs, simple math, etc, without needing to know the individual macros which produced the DVI file.

Smart DVI Viewers

The approach of translating a tagged DVI file was been used in at least two private translators (Rahtz, 1995, Sofka, 1993). It is also the approach used by TeX4ht (Gurari, 1997b, Gurari, 1997a), which is used to author hypertext documents. The method is robust, and it is hoped that a pseudo-standard set of tagging `\special`'s will encourage macro writers to voluntarily pre-tag their code.

Once a DVI file is tagged, however, a number of additional translation possibilities arise. For example, complex page layout is notoriously difficult using TeX. Usually, by the time a book is printed, the source code is filled with hard-coded page-breaks, `\vskip`'s to balance columns, and so on. For some designs, all glue stretch is removed to prevent TeX from “fixing” layout attempts. This is tedium at it's worst.

On the other hand, the actual task—moving a block of text a couple points up or down, or cutting and pasting a figure—are trivial in WYSIWYG environment. The typographer knows exactly what he or she wants to do, the difficulty is conveying that information to TeX. What if the DVI viewer knew how to edit TeX files? What if there were a way to go from the image on the screen to the source file that generated the image?

::tag := ⟨*tag*⟩ [scope(⟨*scope*⟩)]
  | line: ⟨*file:lineno*⟩
⟨*tag*⟩ := begin(⟨*label*⟩) ⟨*op-codes*⟩ end(⟨*label*⟩)
  | block(⟨*label*⟩)
⟨*label*⟩ := [_,a-z,A-Z,0-9] | ⟨*quoted-string*⟩
⟨*scope*⟩ := global | page | stack | ⟨*label*⟩
⟨*quoted-string*⟩ := "⟨*printable ASCII*⟩"
⟨*op-codes*⟩ := ⟨*any DVI op-codes*⟩

Figure 2: Specials recognized by `dvihtml`for tagging a document.

```
# Translate fonts to bold, italic, etc.
font cmit10:  scope(<I>, </I>);
font cmb10:   scope(<B>, </B>);

font cmr17 at 28pt:
   insert(header,
          scope(<TITLE>, </TITLE>)),
   scope(<H1>, </H2>);
...

hdimen 1sp:   translate(" ");
vdimen 1sp:   translate(<P>);
...

tag  section:      scope(<H1>, </H1>);
tag  subsection:   scope(<H1>, </H1>);

tag  enumerate:    begin(<OL>);
tag  enumerate:    end(</OL>);

tag  list_item:   translate(<LI>);
...
```

Figure 3: Sample `dvihtml` configuration file. The elements are choose to display the range of translation possibilities.

```
\nopagenumbers
\def\sb#1{\special{before #1}}
\def\sa#1{\special{after #1}}

\gdef\numberlines{\special{line: \jobname:\number\inputlineno}%
        \immediate\write-1{line: \jobname:\number\inputlineno} }

{\catcode`\^^M=\active%
  \gdef\startnumbering{\catcode`\^^M\active \let^^M=\numberlines}%
  \global\let^^M=\numberlines} % this is in case ^^M appears in a \write

\startnumbering

Misc paragraph: This is a normal line ending, while
this line ends with the macro \TeX
and this one ends with a  hyphenated-
word broken across lines.  This last line%
ends with a \%.
```

Figure 4: Macro to number input lines in the DVI file. Note that this macro modifies TEX's end-of-line semantics slightly.

This style of editing has been dubbed "two-view" by Kenneth Brooks (Brooks, 1988). In a two-view editor both the source language and the WYSIWYG image can be modified with changes being reflected in both views. This approach is used in Lilac (Brooks, 1991), which uses a non-TEX boxes-n-glue language to typeseting (short) documents. Brooks' choice of language was to avoid TEX global scoping, lack of key-words, and modifiable syntax. Contrast Lilac with Blue-Sky's Lightning TEXtures(Hampson and Smith, 1992), which repeatedly reads the entire TEX file from the beginning while the user types. Inbetween these two extremes, Chen, Harrison, and Minakata (1988) and Harrison (1989) have discussed some of the problems associated with incremental formatting in the VorTEX project.

A tagged DVI file offers another intermediate approach. Tags can be inserted into the DVI file to aid two-view editing. For an extreme example, consider the macro in figure 4, which inserts a `\special` into the DVI file at the end of each **input** line. A two-view editor could count input lines to find the TEX code that produced the DVI output. An example of the viability of this approach can be seen in Asher (1992), who used specials to mark pagination points within a DVI file, and the *push/pop* structure of the DVI file to find good breakpoints within paragraphs. The resulting file was processed, paged and printed automatically.

The problem of efficiently parsing TEX's input, however, will require the cooperation of macro writers and users. It would be nice, for example, if in LATEX3 all the relevant state information could be inferred by the environment nesting, and commands which altered expansion or redefined control-sequences were unavailable to the user. This would greatly reduce the amount of processing required by a two-view TEX editor. The goal for `dvihtml`, beyond document conversion, is to serve as a testbed for using tagged DVI files in smarter, if not true two-view, TEX editing systems.

References

Asher, Graham. "Inside Type & Set". *TUGboat* **13**(1), 13–22, 1992.

Brooks, Kenneth P. *A Two-View Document Editor with User-Definable Document Structure*. Ph.D. dissertation, Stanford University, 1988.

Brooks, Kenneth P. "A Two-View Document Editor". *Computer* **24**(6), 7–19, 1991.

Chen, Pehong, M. A. Harrison, and I. Minakata. "Incremental Document Formatting". In *Proceedings of the ACM Conference on Document Processing*, page 93–100. ACM, NY, 1988.

Deland, Donald. "WYSIWYG LATEX" 1998. workshop presented at TEXNorthEast conference, March 1998.

Gurari, Eitan M. "A Demonstration of TeX4ht". 1997a. URL: `http://www.cis.ohio-state.edu/~gurari/tug97/tug97-h.html`.

Gurari, Eitan M. "TeX4ht: TeX and LaTeX for Hypertext". 1997b. URL: `http://www.cis.ohio-state.edu/~gurari/TeX4ht/mn.html`.

Hampson, Steve and B. Smith. "A High Performance TEX for the Motorola 68000 Processor Family". *TUGboat* **13**(3), 269–271, 1992.

Harrison, Michael A. "News from the VorTEX Project". *TUGboat* **10**(1), 11–15, 1989.

Knuth, Donald E. *The TEX Book*. Addison-Wesley, Reading, MA, 1986a. TEX version 3.0, 1994, 14th printing.

Knuth, Donald E. *TEX: The Program*. Addison-Wesley, Reading, MA, 1986b. Reprinted with corrections May, 1988.

Moore, Ross. "Making Web Sites using LATEX2HTML" 1998. Workshop presented at TEXNorthEast conference, March 1998.

Rahtz, Sebastian. "Another Look at LATEX to SGML Conversion". *TUGboat* **16**(3), 315–324, 1995.

Rokicki, Tomas G. "Driver Support for Color in TEX: Proposal and Implementation". *TUGboat* **15**(3), 205–212, 1994.

Rokicki, Tomas G. "A Proposed Standard for Specials". *TUGboat* **16**(5), 395–401, 1995.

Sofka, Michael D. "`dvitag` Users Guide". 1993. Internal document, Publication Services, Inc.

Sofka, Michael D. "DVI Driver Implementation and Standardization Issues.". 1995. URL: `http://www.rpi.edu/~sofkam/dvi.html`.

Stanford University. *The DVIype processor*. Stanford University, 1995.

Sutor, Robert S. and S. S. Dooley. "TEX and LATEX on the Web Via IBM Techexplorer". *TUGboat* **19**(2), 157–161, 1998.