```
\def\specrm{\spectrue \selectspacing
   \aftergroup\selectspacing \specroman}
% Switch to large spacing and remember
% in \ifreset that we have to switch
% back after the group.
%
\def\setdimen{%
  \fontdimen2\specroman=\specialvalue
  \global\resettrue}
% Switch to normal spacing.
% If there is a call to
% \selectspacing after the group,
% there's no need to switch.
%
\def\resetdimen{%
 \fontdimen2\specroman=\savedvalue
 \global\resetfalse}
% This macro does two things:
%  1. If we have changed to larger spacing,
%     we switch back to normal spacing
%     (only if \resettrue).
%  2. If \ifspec is true for the
%     current group we switch to
%     larger spacing.  (The correct \font
%     change to \specroman is done
%     by TeX if this macro is called
%     after a group.)
%
\def\selectspacing{%
  \ifreset \resetdimen \fi
  \ifspec \setdimen \fi}

% A short test:
%
\obeylines
\rm n o r m a l
\specrm s p e c
{\specrm s p e c
 \rm n o r m a l
 \rm n o r m a l
 {\specrm s p e c}
 n o r m a l
 \specrm s p e c
}
s p e c
\specrm s p e c
\rm n o r m a l
```

◇ Alan Hoenig
17 Bay Avenue
Huntington, NY 11743
(516) 385–0736
ajhjj@cunyvm

Tutorials

Elementary Text Processing and Parsing in TEX

— the appreciation of tokens —

L. Siebenmann

Background

Token lists make up the material found in the upper digestive tract of TEX, and token list registers are very useful means to improve TEX's digestion. I begin this tutorial by showing how to do elementary 'text processing' with token lists. Then I apply this 'token list processing' to parsing of classical keyword syntax where the keys come in any order and their fields (or arguments) are terminated by nothing more than the next keyword. This processing and parsing are simple concepts that many TEXperts, not to mention beginners, have largely neglected. I find that TEX assimilates them well, and hope they will see wider use in the future.

I originally explored this parsing as a possible method to fix a subtle line-breaking bug in $\mathcal{AMS}$-TEX bibliographies that was pointed out by Barbara Beeton in 1990. This remains a convenient example to test methods; but in truth an academic one, since Michael Downes [Do] has successfully fixed the bug (for version 2.1 of July 1991) using a very different `\vbox` trick proposed by Don Knuth. The general subject of parsing in TEX language, to which this tutorial contributes two methods called (A) and (B) below, was introduced by W. Appelt in his book [App].

I want to thank Michael Downes, Victor Eijkhout, and Ron Whitney for contributing many helpful comments as this tutorial evolved. My ignorance and uncertainty about what all can or cannot be found in *The TEXbook* was a problem that delayed this tutorial; one remedy I enjoyed using is surely of interest to readers of *TUGboat*, namely string searches in an online version of *The TEXbook*.[1] Perhaps a "HyperTEX" soon will combine this brute force information processing with *The TEXbook*'s beauty and readability. It will

1) The `.tex` file for *The TEXbook* can for example be obtained by anonymous ftp from the archives

`labrea.stanford.edu`
`rusinfo.rus.uni-stuttgart.de`

It fits on a diskette and can conveniently be used on a microcomputer.

not be long before the mass of articles in *TUGboat* merits similar treatment.

Section 0. Token Lists and Registers

As TEX reads in a file, it builds,[2] from the incoming stream of characters (or octets), a closely corresponding stream of 'tokens', i.e., of control sequences and characters-with-category. For example, the ASCII characters `\TeX` — including spaces after `X` — become a single control sequence token representing the TEX logo, and an ordinary (English) word becomes its usual sequence of ASCII characters each with category 11 (= letter). The details (worth re-reading often!) are found in *The TEXbook*, particularly [Chapter 7].[3]

For our purposes, it is not too far from the truth to say that a control sequence is a token that one can specify in the input stream using a backslash followed by a finite sequence of letters (category 11) or a backslash followed by a single character of another category. However, once inside TEX, this control sequence name is, for efficiency, left in a cloakroom, and, in all internal activities, it is represented by a number of fixed length (four or five octets). This means that a control sequence with a long name is no harder for TEX to manipulate than one with a short name.

Control sequences come in many formally recognized varieties, somewhat like the professions of man. The command `\show\mycs` should make TEX tell you the 'profession' of `\mycs` along with some further details: perhaps `\mycs` is a macro, a token list register, a dimension register, a primitive, undefined, etc. We are most concerned with *macros* and *token list registers*. Both of these are 'white-collar workers' that would never get down to the dirty details of typesetting without help from typographic 'primitives' like `\char` and `\hbox`. Both have the same sort of information content, namely a token list, which means they are in some sense just containers holding other tokens! What makes macros and token lists different is their syntax and activity; for example, macros naturally expand while token list registers are fairly inert.

Let us get down to specifics. Given, for example, the token list produced by Plain TEX reading `{\TeX} is useful`, as every TEX programmer knows, we can define a macro called `\mymacro` whose content or 'expansion' is this token list, by typing

```
\def\mymacro{{\TeX} is useful}
```

Check this by executing `\show\mymacro`; there are eight alphabetical characters, two space tokens, one control sequence `\TeX`, and two brace characters.

But, we can also allocate a token list register `\mytoks` by typing `\newtoks\mytoks` and give it the same contents by typing

```
\mytoks={{\TeX} is useful}
```

Here the equal sign is optional; we will often omit it. One checks the contents by executing `\showthe\mytoks`.

There are exactly 256 token list registers `\toks0`,...,`\toks255` and `\mytoks` has been made to stand for one of these by use of a primitive `\toksdef` which is called by the macro `\newtoks` above. This limited number of registers is fixed by the structure and documentation of TEX, whereas the number of control sequences (= hash size) is either flexible or decided by the programmer who compiled your TEX. OzTEX for example has a configuration file letting you set hash size (up to 6500) along with many other parameters.

There is a clear distinction between a token list register and the token list it contains — analogous to the distinction between the wine bottle and the wine. Thus it is an 'abuse' of language (in the benign sense of N. Bourbaki) when one nevertheless talks of 'a token list `\mytoks`'. The word 'toks' will often be used in what follows as an informal abbreviation for 'token list'.

The contents of `\mymacro` can be transferred to `\mytoks` and the other way around as (1) and (2) indicate.

```
\mytoks=\expandafter{\mymacro}                (1)
\expandafter\def\expandafter\mymacro
  \expandafter{\the\mytoks}                   (2)
```

To understand these formulas, recall that the primitive `\expandafter` serves to modify TEX's reasonably 'straight-ahead' expansion procedure by expanding the token next-but-one to the right. Thus, in (1), it causes `\mymacro` to be replaced by its expansion token list before the token list register `\mytoks` has its value assigned. In (2), the first `\expandafter` acts on the second which then acts on the third which acts on `\the` to replace `\the\mytoks` by the token list in `\mytoks` to give the intermediate result

```
\def\mymacro{⟨the toks in \mytoks⟩}
```

2) With its 'lips', to use Knuth's helpful digestive tract analogy. Token list manipulation is done in TEX's 'mouth' and so could be called 'mastication'.

3) In the absence of more explicit indications, citations in square brackets refer to *The TEXbook*.

Further uses of \expandafter will occur below.

Try now the following less well known alternative to formula (2):

```
\edef\mymacro{\the\mytoks}          (2*)
```

An alert reader may wish to protest at this point that this formula will fail whenever the token lists in \mytoks would itself admit expansion by \edef. Wrongly! In fact, although \edef *usually* does a maximum of the 'formal' expansions, it does just a single expansion of anything of the form \the⟨*token register*⟩; see [p. 216 (top)]—a very convenient exception.

Speed as well as elegance argues for using formula (2*) rather than (2). I was surprised to find that (2*) runs at over twice the speed of (2) or of (1). (In principle, speed ratios could vary with the implementation of TEX.)

It is probably because of this 'material equivalence' of macros (simple ones without parameters) and token list registers, that most TEX users and programmers very much neglect token list registers. Notwithstanding, I hope to gradually convince the reader that token list registers are helpful, both conceptually and practically, and deserve a place on every TEXpert's workbench.

Some pitfalls involving token lists

> *Exposition in physics should be as simple as possible. But not simpler.*
>
> *A. Einstein*

(1) Where token list registers are concerned, we should always restrict ourselves to token lists that are *balanced* in the usual sense that the grouping symbols { and } balance. For example {} and {{}{}} are balanced while }{ and }{ are not. Knuth assures us [p. 375 (bottom)] that it is impossible to put an unbalanced token list into a token register.

Note that there is absolutely no requirement that a token list in a toks register be balanced with respect to other standard grouping pairs such as \bgroup, \egroup and \begingroup, \endgroup.

(2) Be prepared for some mind-boggling distinctions among the three grouping pairs just met. For example, in the token assignment \mytoks{...}, the { can be replaced by \bgroup but not by \begingroup. On the other hand } cannot be replaced at all! This is carefully documented on page 276 of *The TEXbook*.

(3) To put one sharp character #, with its usual category (6=Parameter), into the token list that is the expansion text of a macro \mymacro requires one to input two sharps ##. Thus \def\mymacro{##} makes the expansion a single sharp. The single sharp in macro definition input is reserved for macro parameters. In token list register input, this complication does not exist: \mytoks={#} puts one sharp into \mytoks. Many (all?) output functions to screen or file double each (category 6) sharp, notably \show and \showthe; thus \mytoks={#}\showthe\mytoks yields ##. The reader will have to be aware of doubling phenomena for # to understand the formulas for parsing in the sidebar of section 2. See [pages 203–204, 216, 228].

(4) *About* \edef *and its cohorts.* Each macro has an expansion to a token list. It is tempting to believe that, analogously, (balanced) token lists have an 'immediate expansion' provided by \edef. To expand the token list in \mymacro execute

```
\edef\mymacro{\mymacro}
```

Use \show\mymacro before and after to see the effect; the expansion is in some sense complete and immediate.

Alas, this 'complete expansion' is not always defined, and when defined may be utter nonsense; for example, if the token expansion for \mymacro is \def\aaa{AAA} where \aaa is not already defined then TEX will balk, while if \aaa is defined to be aaa then one gets \def aaa{AAA}!

TEX also has a surprise in store for you if you believe that, when you change an occurrence of \def to \edef, the (unexpanded) definition text read in will necessarily be the same for each; see [Exercise 20.17].

The rules for \edef are carefully laid out in *The TEXbook* [p. 215 (bottom) and p. 216 (top)]. The double bends there are justified by the subtlety of \edef, not by its rarity or lack of importance! The rules are all the more worth learning because they apply with only minor modification to \mark{...}, \message{...}, \errmessage{...}, \special{...}, and \write{...}; see [p. 216 below 20.16]. Roughly speaking, \edef and these 'cohorts' do all the formal expansion that is possible subject to an overriding condition that this expansion process should change nothing in the TEX environment other than the ultimate expansion token list for the macro being defined. It in fact does slightly less than that because of the important single expansion rule [p. 216 (top)] for \the⟨*token register*⟩ that we have already encountered.

Always keep in mind that \edef and its cohorts can only be used when the programmer has such intimate knowledge of the toks to be expanded that he can guarantee the results are well-defined and

suitable for his purposes. (In other cases, simpler tools such as `\expandafter` and `\noexpand` may prove useful.) Since the `\edef` primitive is powerful, and can often do more for us in less time and with less programming effort than competing tools, its (prudent!) use is to be encouraged.

The single expansion rule above for `\the`⟨*token register*⟩ with respect to `\edef` and its cohorts offers the only way I know to efficiently suppress expansion of a long list of tokens; the primitive `\noexpand` applies to only a single token.

Section 1. Elementary 'text processing' with Token Lists

It is well known that TEX can dabble in computer graphics (LATEX does) and even in number theory [p. 218], so it should come as no surprise that it can master the rudiments of classical text processing. But although this ability is obviously relevant to TEX's main purpose, typesetting, it seems little attention has been paid to it.

The most basic operations of text processing on a list of characters (or more generally of tokens) are:

(a) *copying.*
(b) *concatenating* two lists x and y to form a composed list xy.
(c) *searching* for one list x in another z (is x a sublist of z?).
(d) *splitting* a token list z at a sublist x (known to be present) into parts a, x, and b, so that z is the concatenation axb.

The problems these token list processing operations pose for us are practical problems of coaxing TEX to perform these useful operations efficiently. It turns out that most of them are a bit tricky to define, but reasonably compact and efficient once defined. To keep the formulas simple, I often do not give the operations a catch-all syntax, as might be desirable in a large macro package. That can be left to the programmer.

One can at first imagine that the token lists are segments of English prose, but in general there are control sequence tokens as well as character tokens. The situation is somewhat analogous in computer printer scripts of the 1970's and in some wordprocessor files that represent changes of font style, etc., as tokens intermixed with the ordinary characters.

TEX forces on us a very stringent notion of equivalence for token lists, namely one-to-one order preserving correspondence of the tokens in the lists so that corresponding tokens are *identical* (not just `\let`-equal or identical-after-expansion). Coarser notions are probably best approached by doing some preliminary macro expansion. Assuming two toks are the expansions of `\mymacro` and `\thymacro` respectively, the standard test for equivalence uses `\ifx` as in

```
\ifx\mymacro\thymacro\message{EQUIVALENT}
 \else\message{INEQUIVALENT}\fi
```

We assume below that `\xtoks`, `\ytoks`, `\ztoks`, `\atoks`, `\btoks`, are allocated token list registers, cf. section 0.

Copying token lists

To copy the toks in register `\atoks` into the toks register `\btoks` is a simple matter:

```
\btoks=\atoks
```

This is analogous to `\let\b=\a`; speed is great and independent of the contents of the register `\atoks`. Quite the opposite can be said of the alternative formula `\btoks=\expandafter{\the\atoks}`.

There is another form of copying: macro arguments, written `#1`, `#2`, etc., represent token lists too and, in the definition of a macro with arguments [Chap. 20], they can be stuffed directly into a token list register or a macro expansion. See the splitting macro `\SPLITT@` below for a simple example.

The `\read` primitive provides still another form of copying: it reads in a line from an open file `\myfile` thus:

```
\read\myfile\mymacro
```

converting it to the expansion toks of the macro `\mymacro`. The inverse operation can be accomplished[1)] by

```
\mytoks=\expandafter{\mymacro}
\immediate\write\myfile{\the\mytoks}
```

Recall that `\write` is one of the cohorts of `\edef`; this is another use of the 'single expansion' phenomenon. Beware that because of category codes

1) Ron Whitney [Wh] has shown how to do this inverse operation using `\meaning` in place of a toks register. His approach is preferable for non-immediate writes which are often used in index construction; the difficulty with the toks register approach is revealed by executing

```
\mytoks={aaa}\write\myfile{\the\mytoks}
\mytoks={bbb}\write\myfile{\the\mytoks}
```

Whitney's approach is much simpler and not less effective than an earlier one of Todd Allen [p. 377].

and TEX's reading conventions these two operations may not be strictly inverse one to the other.

Concatenating

We propose to concatenate \xtoks and \ytoks and put the result in \ztoks.

The following simple formula gives the right idea but fails dismally

```
\ztoks{\the\xtoks\the\ytoks}          (1x)
```

because of the distinction between wine bottle and wine. It is well known that cunning use of the primitive \expandafter can correct this. *We assume* \let\e=\expandafter *henceforth.* The most usual formula is impressive

```
\e\ztoks\e\e\e
  {\e\the\e\xtoks\the\ytoks}          (1a)
```

and also fun to expand: to begin, the five odd-numbered tokens from the left (all \expandafter's) go off in sequence like a long fuse and detonate the last \the to produce an intermediate form:

```
\ztoks\e{\the\xtoks⟨the toks in \ytoks⟩}
```

From this point, a short fuse consisting of just one \e similarly detonates the first \the to produce a second intermediate result

```
\ztoks{⟨the toks in \xtoks⟩%
  ⟨the toks in \ytoks⟩}
```

which is then normally executed to give the desired result.

Do not bother to memorize intimidating formulas like (1a)! You just have to remember the intermediate stages and work backwards stringing out your fuse lines of \e's.

And do not go out of your way to use them in serious programming! They often execute more slowly than than alternatives. In this case there is an alternative that *entirely* avoids \expandafter, exploiting \edef instead:

```
\edef\dummy{\ztoks={%
 \the\xtoks\the\ytoks}}\dummy          (1b)
```

It executes 15% faster than (1a). There are many less elegant solutions that execute as quickly, e.g.

```
\edef\dummy{\the\xtoks\the\ytoks}
\ztoks=\e{\dummy}
```

Concatenation can also be done directly for the toks of macro expansions; the trickery is much the same. Indeed, given \x and \y, we can define \z as follows

```
\e\e\e\def\e\e\e\z\e\e\e{\e\x\y}          (2a)
```

or by

```
\toks0=\e{\x} \toks2=\e{\y}
\edef\z{\the\toks0 \the\toks2}          (2b)
```

In (2b), we have used two of the five local 'scratch' toks registers, numbers 0, 2, 4, 6, 8, that TEX reserves for temporary storage [p. 346]; this merely avoids allocating special registers for the purpose, using \newtoks. Caution: Many technicalities arise in using explicit registers. For one, the odd registers 1, 3, 5, 7, 9 are reserved for global definitions; see [p. 346]. For another, space after the second \toks0 above is obligatory. Indeed, without it (or some alternative like \relax), TEX expands \the\toks2 in the process of assimilating \the\toks0 and then a full expansion of \the\toks2 is attempted, which is not what we want here.

Searching for one token list in another

Our goal is to decide whether a toks ⟨*toks sought*⟩ is equivalent to a sublist of another toks ⟨*toks to be searched*⟩.

The notion of a sublist of a (balanced!) token list that we shall use is restricted to balanced sublists occurring at nesting level zero for the TEX grouping symbols { and }. Such sublists of a balanced list z are precisely those sublists x inducing a splitting $z = axb$ with all three of a, x, and b balanced. Call such sublists *admissible*. For example, the sublist st in the seven token list r{st}uv is a balanced but *inadmissible* sublist, being at brace level 1. On the other hand, {st}u is a balanced and admissible sublist. (If this notion is not to your liking, see [p. 376 (middle)].)

The tool we use for searching is the full TEX macro mechanism including parameters and match text. As Knuth treats search macros in a highly condensed fashion in the dirty tricks chapter [Appendix D], a motivated discussion will be given here.

To get the main idea, observe that a definition

```
\def\mymacro#1⟨toks sought⟩{...}          (*)
```

of a macro with match text [p. 203] will make \mymacro look for the first occurrence of the token list ⟨*toks sought*⟩ in the input after \mymacro[2] and make #1 be the token list (possibly empty) between the two.

This approach imposes a significant restriction on ⟨*toks sought*⟩ that is admittedly quite undesirable. Since it is a macro match text, ⟨*toks sought*⟩ *must contain no brace characters*, for if it did TEX would see a shorter macro definition in (*)!

2) If there is none before the next occurrence of \par an error will result, unless \long\def replaces \def.

Next observe that to prevent trouble in case ⟨*toks sought*⟩ is absent, we can apply such a macro to, for example:

⟨*toks to be searched*⟩`\premarker`⟨*toks sought*⟩`%`
`\postmarker\endmarker` (**)

Now, of course, the ⟨*toks sought*⟩ is always found and the search problem is converted into a question of *where* it is found: is it in the ⟨*toks to be searched*⟩ or between markers? For this, one can apply to (**) a macro with more complicated match text — as follows:

```
\def\searchmacro#1⟨toks sought⟩%
   #2#3\endmarker{...}
```

(We have still to decide on the macro substitution text `{...}`!) What happens when this is applied to (**)? Because of `\endmarker` the macro uses up the full text (**), which is all to the good — a leftover could cause havoc. The argument `#2` will be the token immediately following the first occurrence of ⟨*toks sought*⟩ in (**) and we conclude that `#2` is `\postmarker` precisely if ⟨*toks sought*⟩ failed to occur in ⟨*toks to be searched*⟩. Thus after setting out preliminary material

```
\newif\iffound
\def\postmarker{\uniquecs}
```

we specify the substitution text `{...}` to be:

```
{\def\this{#2}\ifx\this\postmarker
  \foundtrue\else\foundfalse\fi}
```

Putting all this together we have a search macro `\searchmacro` for a fixed toks ⟨*toks sought*⟩.

Several improvements are given in the 'production version' (3) below:

(a) allow ⟨*toks sought*⟩ to vary; this requires a somewhat confusing layer of indirection.
(b) allow both ⟨*toks sought*⟩ and ⟨*toks to be searched*⟩ to be specified in terms of a token register or macro as well as by direct typing; the solution is to specify ⟨*toks to be searched*⟩ by anything whose first expansion is ⟨*toks to be searched*⟩, and similarly for ⟨*toks sought*⟩.
(c) make direct typing of ⟨*toks sought*⟩ and ⟨*toks to be searched*⟩ convenient (our first attempt ignores initial spaces); the strings `0` (zero, not 'oh') and `@@` in the production version permit this.
(d) keep the macro and related apparatus out of the way of non-programmers by use of `@` with category 11 (letter).

The production version below was adapted from one in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX by Mike Spivak, which in turn was adapted from [p. 375]. I had to generalize somewhat to allow `#1` to be a token list rather than a character and to assure features (a)–(d). Also I spent a few extra control sequences on readability.[3)]

Roughly speaking, the `\IN@0#1@#2@` below sets the condition `\ifIN@` to true if the toks for `#1` is a sublist of the toks for `#2` and otherwise sets it to false. More precisely `#1` and `#2` should be things whose first expansions are the toks in question — so that arguments `#1` and `#2` can be of the form `\mymacro` or `\the\mytoks`.

```
\newif\ifIN@
\def\IN@{\e\INN@\e}
\def\INN@0#1@#2@%
 {\def\NI@##1#1##2##3\ENDNI@
   {\ifx\m@rker##2\IN@false
   \else\IN@true\fi}%
  \e\NI@#2@@#1\m@rker\ENDNI@}
\def\m@rker{\m@@rker}                    (3)
```

There are some reasonable technical restrictions on this macro. It is to be defined and used inside macro packages where `@` has been given catcode 11 (= letter). Neither token list produced by `#1` and `#2` should contain a `\par`[4)], nor a character token `@` with catcode 11 — something easily avoided as they are either under the programmer's control or come from the user's world where `@` has catcode 12 (= other) or 13 (= active). Further, neither should contain a token (like `\m@rker`), whose expansion begins with `\m@@rker`.

There is also one annoying restriction explained above. *The toks for* `#1`*, i.e.,* ⟨*toks to find*⟩*, must contain no braces.*[5)] However, braces (balanced of course) are permitted in ⟨*toks to be searched*⟩.

The above production version is admittedly very technical; fortunately no understanding of how all the the details work together is essential for what follows. Incidentally, the splitting macro below is more transparent and could serve as a stepping stone.

---

3) For hints on recovering these examine [p. 375].
4) To allow `\par` one uses `\long\def` in place of `\def`.
5) One way to work around this restriction without resorting to the slow token-by-token approach of [p. 376 (middle)] might be to use the `\meaning` primitive to first convert braces to category 12 characters, cf. Ron Whitney's note [Wh]. This also gets around the blanket restriction to 'balanced' token list. However, it may require you to use `\write` and `\read` to reconstitute control sequence tokens from category 12 characters.

Splitting at a sublist

Suppose we know (from the test above, for example) that the toks (with no braces) in \xtoks is a sublist of the toks in \ztoks. Then we typically want to put into \atoks the segment of \ztoks up to the first occurrence of \xtoks and put into \btoks the segment following that occurrence of \xtoks. This is to be accomplished by the syntax

```
\SPLIT@0\xtoks @\ztoks @
\atoks=\Initialtoks@
\btoks=\Terminaltoks@
```

where \SPLIT@0#1@#2@ carries on the basic conventions and design features for \IN@0#1@#2@ set out above. The macro definitions required are

```
\newtoks\Initialtoks@
\newtoks\Terminaltoks@
\def\SPLIT@{\e\SPLITT@\e}
\def\SPLITT@0#1@#2@%
 {\def\TTILPS@##1#1##2@%
   {\Initialtoks@{##1}%
   \Terminaltoks@{##2}}%
  \e\TTILPS@#2@}                    (4)
```

We have now established basic processing functions for TEX's token lists that are generalizations of well known text processing functions, and that execute at a useful speed. They can be used to edit pieces of text before printing them, and more importantly to build new macros that provide users with syntax with flexible options. This second 'parsing' theme will be pursued in the next section.

I also recommend use of token list processing deep within macro packages; for hints on this sort of application I suggest reading about Knuth's list macros [Appendix D, p. 378–379] and Appelt's stack macros [App, Chap. 5]. Incidentally, TEX offers some *ready-made* text processing control sequences such as \uppercase and \lowercase.

Section 2. From Text Processing to Keyword Parsing

One of the most powerful, convenient, and wide spread syntaxes one encounters on classical computers is the 'keyword option' system. W. Appelt [App] has advertised this system in TEX programming, and provided a practical sort of recipe to implement it, after a first simple example by Knuth [p. 376 (top)]. Here we will provide recipes offering improvements such as more general syntax, potentially greater speed or capacity, or more compact formulas. The most general recipe is the the second below, called (A); it will be simple application of our token list processing of section 1. But a more subtle process (B) will often give better results in case the keywords are macros.

An ad hoc parsing process

The keyword option system will be illustrated first by a \special command from Tom Rokicki's dvips postscript printer driver for TEX. His syntax summary is:

```
\Special{psfile="filename"[ key=value]*}   (1)
```

Here the possible keys are the words: hoffset, voffset, hsize, vsize, hscale, vscale, angle, and each of these keys calls for a suitable quantity in place of value. I have perversely written \Special for \special here so that (1) and (2) can soon be assigned another meaning.

A specific example is

```
\Special{psfile=myfile
  angle=90 hscale=50 vscale=50}        (2)
```

which prints the PostScript graphics file myfile rotated 90 degrees at scale 50 percent. The central point to note is that the user can specify *any number* (or zero) of keys in *any order* he pleases.

This command is interpreted by dvips (a printer driver) after a preliminary expansion by TEX.[1] But *let us imagine that we want TEX to interpret a control sequence with this sort of syntax.* For example, one might want a TEX macro \Special with *identical* syntax, that provides, in addition to what Rokicki's \special gives, a TEX box into which the printed graphics nicely fits. Of course, such a \Special will normally also appeal to \special after composing a suitable box.

How can TEX understand or 'parse' (2)?

By making \Special a one-argument macro, TEX can efficiently isolate the guts of (2), namely psfile=myfile ... vscale=50, and store it as a token list T in a token list register, say \Ttoks. This is the first (easy) step of parsing.

Now T is a concatenation (see section 1):

$$T = aa^*bb^*cc^*...zz^* \qquad (3)$$

where a, b, ... are 'keys' taken in any order from a known family $\mathcal{K}$ and a^*, b^*, ... are user supplied token lists; we call a^* the *argument* or *field* of the key a, and b^* the argument of b, etc...

The main step of parsing is to store a^*, b^*, ... in token list registers (or macros) associated to the keys a, b, ... When this parsing of T has been accomplished, TEX has a firm grip on

1) Recall that \special is one of the cohorts of \edef mentioned at the end of section 0.

the information encoded in T and typesetting can proceed.

Our purpose in this section is to propose ways to parse T in a few cases of practical importance.

First consider the specific example (1). We are just as happy (or happier) with the full expansion a^{**} of a^*, discussed at the end of section 0, since one can readily believe that in the specific context of (1) the expansion is unlikely to cause the sort of trouble mentioned there.[2)]

The argument a^* might well have TEX conditions and arithmetic (including =), while the full expansion a^{**} should be a dry number or dimension. In particular, it will not contain =, which we can then use as a tell-tale sign for a key.

We expand the whole of T (using `\edef`; see section 0) and note that this gives $aa^{**}bb^{**}\ldots$, i.e., the keys are intact. Since = is a tell-tale sign for the next key b, we can readily determine b. More precisely, in Rokicki's syntax, the keyword is delimited on the left by a space and on the right by =.[3)] We can thus split at b — or for greater speed use just the idea of formula $(*)$ in section 2 — to get a^{**} and $bb^{**}cc^{**}\ldots$. We store away a^{**} for key a, then iterate the process to get a grip on b^{**}, c^{**}, ... similarly.

In summary, in case the next key is always readily accessible, keyword parsing is a straightforward process. The time required seems then to be the least time for all the processes we will consider. Qualitatively speaking, the time per key is constant and independent of the number of keys.

The syntax discussed by Appelt [App, Chap. 5 (end)] is of this simple sort; his next keyword lies between the next semicolon and the next equal sign. (Appelt formulas nevertheless run through all keywords to find the next key, something to be avoided if there are many keys.)

The accessibility of the next key in Rokicki's case was probably a well-planned accident — related to Rokicki's driver wanting to parse this syntax in a hurry. In the absence of the tell-tale = above, or if we had wanted unexpanded arguments, the following general method would have worked.

Parsing process (A) — Substitution and self-analysis

This is a simple and general process that depends heavily on our token list processing in section 1. It is practical if there are just a few keys.

For each key k in $\mathcal{K}$, search[4)] for k in the toks T of form (3) and, if k is present, replace it (using splitting and concatenation of section 1) by the two tokens `\zzz\macro`k. Thus (after doctoring the extremities), we readily give altered T (in `\Ttoks`) the form:

$$\texttt{\textbackslash macro}a\ a^*\texttt{\textbackslash zzz\textbackslash macro}b\ b^*\texttt{\textbackslash zzz}\ldots \quad \texttt{\textbackslash macro}z\ z^*\texttt{\textbackslash zzz} \qquad (4)$$

This completes the substitution step.

Now for each k in $\mathcal{K}$, we are at liberty to define `\macro`k`#1\zzz` as a one-argument macro which places token list `#1` in a token list register (or macro) associated to k. Then writing `\the\Ttoks` as a command, we execute (4), and the result is to complete the parsing. The idea of this second step that we have subtitled 'self-analysis' has been used by Knuth in dealing with the TEX data structure called 'list' [Appendix D, p. 378–379].

Note that if there are N keys in $\mathcal{K}$, the parsing process always has N nontrivial steps and each applies to the whole token list. Thus the time of execution can be estimated as roughly proportional to nN where n is the number of keys actually present in the token list T. Consequently for N sufficiently large the process will be intolerably slow. How large? My tests suggest you should be worried for $N > 5$.

The $\mathcal{AMS}$-TEX bibliography reference macro `\ref` ... `\endref` is one that has well over a dozen keys; for it, one needs a better parsing technique. It has the peculiarity that the keys are all macros. Consider the example

```
\ref
 \key W \by A. Weil
 \paper Sur quelques
  r\'esultats de Siegel
 \journal Summa Brasil Math.
 \vol 1 \yr 1946 \pages 21--39
\endref
```

Note that only `\ref` has a balancing terminator `\endref`; it lets us scoop up the whole token list

2) The reason is that TEX always does this sort of expansion on the argument of `\special` before stuffing the result into the `.dvi` file for further processing by the printer driver. Clearly, the user will have himself to blame if he attempts for `\Special` what fails for `\special`!

3) To be more user-friendly, it would be advisable to allow space between (for example) `hsize` and =. Although this may double the time to locate the next keyword, one still does not have to run through all the keywords.

4) If one key is a subset of another, e.g., "SCALE" and "VSCALE", deal with the larger one first.

from \key to 21--39 as a macro argument. Once again, we have a parsing problem as described for (3).

There are six keys here: \key, \by, \journal, \vol, \yr, \pages. But parsing process (A) above would require searching with well over twice as many keys. The feature that each key is a control sequence lets us use a new process (B) which will be given in full detail.

Parsing process (B) — Sequestered self-analysis

This process usually applies when each key is a control sequence; it requires a few extra conditions which will become clear when the process has been described.

Since process (A) does indeed apply here, and what follows is comparatively difficult, I had better explain very clearly what (B) attempts to gain! Suppose that the set $\mathcal{K}$ of keys is big, say N of them (perhaps 25), and getting bigger year by year. We ask for a process that on a given example using n keys (perhaps $n = 5$) does not run substantially slower each year as N increases. We would like to get by with a few times n steps — to be more precise, not more than $a + bn$, where a and b are constants independent of N. In contrast, the similar estimate for (A) would be $a' + b'nN$. Thus in the usual succinct mathematical terminology, process (A) requires $O(nN)$ steps while (B) requires $O(n)$. The latter seems, qualitatively speaking, a 'nec plus ultra' of good behavior, because it means that the cost of parsing per field actually present is constant.[5)]

The 'box register' alternative to token list parsing that is actually used by $\mathcal{AMS}$-TEX for the \ref...\endref macro system enjoys the sort of linearity that we are promising for (B). On the other hand, the keyword parsing provided by W. Appelt [App, Chapter 5 (end)] simply does not apply.

The idea for (B) is to make a preliminary pass over the material between \ref and \endref to determine, for each key $\backslash k'$ that is present, the key $\backslash k$ that follows, and then define a macro $\backslash k'$#1$\backslash k$ (with argument #1 and delimiter $\backslash k$), which will, on a second pass, serve, much as in (A), to sweep up the field of $\backslash k'$ and store this toks in a corresponding macro expansion. For this first pass, subtitled 'sequestered self-analysis', one assigns special temporary definitions to each key to carry out this plan. A major difficulty is that I cannot prevent extraneous typesetting activity during the first pass; the best remedy known is to 'sequester' this extraneous material in an \hbox and annihilate it. Unfortunately, this \hbox involves a grouping that entraps definitions — unless one uses some global definitions. Perhaps surprisingly, this secondary difficulty is overcome without losing the expected behavior of the parsing process with respect to TEX grouping, namely that it change nothing outside braces enclosing the whole process.

Now we get down to programming process (B). The functioning prototype is given in a sidebar, but will probably have to be understood as it was built — by stages. The programmer has to define for each key $\backslash k$ in $\mathcal{K}$ (say \paper, to be specific) an artificial expansion that combines $\backslash k$ (say \paper) with key $\backslash k'$ (say \author) stored as author in a register called (for good reason!) \LastKeytoks@. The definition of \paper goes roughly as follows.

```
\def\paper
 {\global\def\authorAgent@
  {\def\author####1\paper
    {\def\authorBag@{####1}\paper}%
     \global\let\authorAgent@=\relax}%
   \LastKeytoks@={paper}%
   \aftergroup\authorAgent@
   \def\paper{\errmessage
    {  *** A key has been used
    twice. Once is max. ***}}%
 }                                    (5)
```

The programmer unfortunately is not in a position to write something so explicit — for example he does not know the actual name of the key that will precede \paper. Standard indirect methods involving \csname...\endcsname apply nevertheless. This macro depends on $\backslash k$ in a very simple way; so the TEXpert can get away with writing just one (nasty) macro \SetKeyDef@ (see sidebar) so designed that executing \SetKeyDef@{k} for $\backslash k$ in $\mathcal{K}$ sets things up once and for all.

To facilitate the parsing we use an extra terminal key \t@il, as well as an initial key \he@d.[6)]

One then executes:

```
\LastKeytoks@={he@d}
\setbox0=\hbox{\the\Ttoks\t@il}
\setbox0=\hbox{}
\let\t@il=\relax
\he@d\the\Ttoks\t@il                  (6)
```

5) In contrast, I do not know how to use TEX to reverse the order of a list of n tokens in $O(n)$ steps!

6) The (category 11) @ in he@d and t@il keep these out of the user's way. Likewise, \Ttoks, \e, \n, \cs, \ecs should be protected elsewhere.

The first appearance of \the\Ttoks is a dirty trick! What we really wanted to do is execute in order just the keys $abc\ldots$ found in T. But, as these are buried in T and not directly accessible, we execute all of T instead. This (unfortunately) causes spurious typesetting activity, but we catch the detritus in \box0 and annihilate it! [7)]

The global definitions are essential to pass information out of the first \hbox{...} group in (6), using \aftergroup. This is accomplished as follows. The tokens \aftergroup\authorAgent@ occur inside the grouping of \hbox{...} and cause the globally defined macro \authorAgent@ to be executed after the closing brace. This in turn prepares a second definition of \author for a second execution of \the\Ttoks in the last line.

This second execution will be similar to the last step in (A): the macro \author (new definition) will cause the field of the key \author to become the expansion toks of \authorBag@ — *outside* the \hbox{...} group. This process is carefully designed to cause no net global changes in case it occurs within a larger group; in particular \authorAgent@ is globally \relax before and after. The resulting 'escape from braces' without net global change seems in itself a worthwhile trick.

Both executions of \the\Ttoks cost $O(n)$ steps. While the first involves futile typesetting, the second is fast and purely syntactical. Let us have a closer look at the second: after a couple of expansion steps, what TEX sees is $aa^*bb^*cc^*\ldots$; then the first three tokens $a\,a^*b$ act together to put the field a^* in the expansion of the macro[8)] aBag@ leaving behind $bb^*cc^*\ldots$. Then bb^*c act together, and so on until all fields have been 'bagged' and only \t@il is left, which evaporates as we have set it equal to \relax.

The use of \aftergroup restricts the number n of keys used in any one parsing example to be less than the size of TEX's save stack space. This may mean $n < 100$ in current implementations of TEX; but soon your limitations should be much more liberal; already, OzTEX's configuration file of 1990 lets one push n up to 2000, and the total number N of keys to about 5000. Perhaps squeamishness about the use of \aftergroup [p. 374] can be relegated to the past. In any case, \aftergroup could be replaced by some global definitions without prejudicing the linear performance we have achieved.

This process (B) does have some drawbacks beyond the fake typesetting. (You expect a dirty trick to have some!)

(i) It assumes that T is fit to be put in an \hbox, and that, on execution of T, the key macros a, b, ... will be executed in that order.

This is a very mild restriction that refers to the first pass; it should in practice hold if each key field is fit to be composed on its own. If (i) is not satisfied, one can hope it will be if one suitably alters the TEX environment for the first pass.

(ii) The speed of parsing is a bit disappointing to me; I get about 50–100 key fields parsed per second with a 1987 microcomputer (16 mhz and 32 bit bus). In implementing (B), one has a great deal of latitude in programming style; perhaps I have made some bad choices; if so I hope some reader will offer better coding. This slowness may not be a serious fault if you have a sufficiently fast computer, or if this parsing is not going to be proportionally a major activity of TEX, or if the other TEX material is already slow to process — for example commutative diagrams, tables, or verbatim material.

A good feature of (B) that I did not expect is the brevity of the coding.

In summary, the one rather general parsing process (A) is firmly based on our token list processing, and is delightfully simple and safe, but, used with a large number of possible key options, it becomes slow. That has led us to consider process (B), whose time cost per key field actually parsed is essentially constant[9)] and independent of the total number of possible keys. In practice it seems that (B) is faster than (A) for $N > 5$.

```
% TestbedB.tex

%% Sidebar: Testbed for parsing method (B)
%% L. Siebenmann 1991

\chardef\CatAt\the\catcode`\@\catcode`\@=11
\newtoks\Ttoks@ \newtoks\LastKeytoks@
\let\e=\expandafter \let\n=\noexpand
```

7) One naturally wonders whether there is a much neater trick. One can marginally speed up this trick using the 'dummy' font device of the last dirty trick 9 in [Appendix D, p. 401], but if you are not careful you will instead lose time through overhead.

8) I have not used a token list register here to store the key field; this permits the number of keys to exceed the total number (256) of token list registers!

9) This assumes fields of constant size; if not, the dependence of time cost per field on the size of the field is more or less linear, with a substantial positive constant term.

```
\let\cs=\csname \let\ecs\endcsname
%% Texpert protect \e,\n,\cs,\ecs with @

\def\SetKeyDef@#1%
 {\e\def\cs#1\ecs{\MasterMacro@{#1}}}

\def\MasterMacro@#1%
 {\e\xdef\cs\the\LastKeytoks@ Agent@\ecs
   {\def\e\n\cs\the\LastKeytoks@\ecs
           ####1\e\n\cs#1\ecs
      {\def\e\n\cs\the\LastKeytoks@
                 Bag@\ecs
           {####1}\e\n\cs#1\ecs}%
    }%
    \e\aftergroup
        \cs\the\LastKeytoks@ Agent@\ecs
    \LastKeytoks@={#1}%
 }

\def\@K.#1.{\SetKeyDef@{#1}}

\@K.by.\@K.key.\@K.paper.\@K.jour.\@K.yr.
\@K.pages.\@K.issue.\@K.no.\@K.vol.
\@K.publ.\@K.eds.\@K.bysame.\@K.paperinfo.
\@K.book.\@K.publaddr.\@K.lang.
\@K.bookinfo.\@K.finalinfo.\@K.t@il.

\def\ref#1\endref{\Ttoks@={#1}%
 \LastKeytoks@={he@d}%
 \setbox0=\hbox{\the\Ttoks@\t@il}%
 \let\t@il\relax
 \e\he@d\the\Ttoks@\t@il\Typeset@}

\let\Typeset@\relax % stop after parsing

%% begin test
\def\R{% for time test
 \ref
  \key W \by A. Weil\paper Sur quelques
                    r\'esultats de Siegel
  \jour Summa Brasiliensis Math.
  \yr 1946 \pages 21-39
 \endref}

\def\RR{\R\R\R\R\R\R\R\R\R\R}
\def\RRR{\RR\RR\RR\RR\RR\RR\RR\RR\RR\RR}

\show\key % all set?
\RRR % do 100 iterations

%\show\keyBag@    \show\byBag@ % checks?
%\show\paperBag@ \show\jourBag@
%\show\yrBag@     \show\pagesBag@
\catcode`\@=\CatAt % restores catcode
\end
```

Should token list parsing have fixed the bug in the $\mathcal{AMS}$-TEX reference macros?

As we will see presently, the answer is no, but it seems worth examining the pros and cons since many of them would have to be examined in any large scale application of parsing based on token lists.

The bug, located in `amsppt.sty` (versions $\leq$ 2.0), prevented hyphenation after explicit hyphens, or after mathbins and mathrels, for line-breaking in references. As Michael Downes so nicely explained [Do], this bug (and also the residual problems with Knuth's fix) have occurred because, if a fragment of a reference is put into an hbox or even a vbox, certain stages of line-breaking may be done prematurely and hence inappropriately in that box.

The plan for using token registers is very simple. Place the various parts of a single bibliography reference into as many token lists using parsing method (B), edit these token lists as necessary using the text processing of section 1, concatenate them in the desired order[1)] to make a single token list for the reference in question, ultimately releasing the whole reference 'en block' into TEX's intestines for typesetting.

The bug will not occur since one replaces the troublesome boxes by token registers. Indeed those aspects of typesetting related to line breaking simply do not take place in token registers; they are delayed until the full reference is ready for typesetting.

Ron Whitney and Mike Downes tell me that the idea of using token lists in place of boxes was well known but considered to be an impasse (no way!).

The token list approach has some intrinsic advantages over the box-oriented approach. We have already mentioned the possibility of doing some text editing before printing (say to replace AMS by Amer. Math. Soc.). In this vein, there is the possibility, not well afforded by the box approach, of having any key's data influence the action taken for any other, for example, when the reference is

1) Such ordering should have a cost proportional to $n \log n$ when n out of N keys are present. But in $\mathcal{AMS}$-TEX a cost aN with a very small is tolerated instead for simplicity.

a book the style of several entries could reasonably change. One can also output the references to a file in a convenient 'data structure' format, to facilitate further processing. This might, for example, facilitate preparation of a citation index (or other bibliographic data base) for journals using $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX. As this idea applies to reprocessing archived articles, the toks parsing approach may ultimately be complementary to the box-oriented approach even in the specific context we are considering. Another advantage already mentioned is the virtual inexhaustability of token registers: although there exactly 250 token registers or box registers in the strict sense, macros, of which there are thousands available, can be employed as auxiliary token registers; they indeed were in the testbed for (B).

Nevertheless, we will have to wait for some future occasion to see a large-scale test of the above token-parsing ideas. There are a host of reasons that, taken together, are quite cogent. Repair of `amsppt.sty` (where the faulty macros reside) has already been successfully made by Mike Downes (for `amsppt.sty` version 2.1 of July 1991) using Knuth's `\vbox` approach plus extra work to suppress undesirable side-effects. A very practical consideration is that Knuth's approach is comparatively close to Spivak's, so that much less rewriting of this hefty complex of macros was required.

Furthermore, the box approach is exceedingly fast — to the point that bibliographies are composed faster than most mathematics. This turns out to be more than twice as fast as the next fastest contender, the parsing approach (B), which is in turn more than twice as fast as (A). See [p. 385].

Finally, there is a general weakness (mentioned on [p. 381–382, p. 385]) afflicting all macros having arguments which are blindly scooped up as chunks of input, namely: *category changes within the arguments will be ignored because category is fixed at input.* We did propose to scoop up T above using `\ref#1\endref`! One impact is that, with our present approach, `\verb` (of LATEX) and `\lit` (of $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX) would become inoperative within a reference.[2)] This may prove annoying, but one can live with it by 'hand setting' or by importing literal material in a box register. Another impact is that language changes would have to be made so as not to involve category changes, which fortunately is possible. It might be be desirable to set up some warning using `\message{...}` to be triggered by uses of `\catcode` within `\ref...\endref` and give indication of alternatives. These category problems are annoying but they are not debilitating.

In summary, token list parsing in version (B) compared to the box register alternative seems a promising alternative because of multiple hitherto unused possibilities we have mentioned; it equals box registers in dealing 'linearly' with increasing loads, but is always slower by a small integer factor; and finally it may, alas, be penalized for blocking category change. I rate that an honorable second place on a tough course.

The role of token registers in Plain TEX, LATEX, or $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX was quite marginal. But recently, their role has become quite significant, for example in M. Spivak's $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX (released recently into the public domain). Now that TEX is no longer evolving, I expect TEX programming will still advance a long way by increasingly calling upon currently underused resources.

2) One can *in principle* pick up the token list T one token at a time watching for `\verb` and `\lit`, in order to avoid this weakness, but that sort of procedure is much too slow. For just this reason we have ignored parsing procedures based on sequential token-by-token examination, using `\futurelet`.

Bibliography

[**App**] W. Appelt, *TEX für Fortgeschrittene*, Programmiertechniken und Makropakete, Addison-Wesley, Bonn, 1988.

[**Do**] M. Downes, Linebreaking in `\unhboxed` text, *TUGboat* 11, no. 4 (Nov. 1990) 605–612.

[**Kn**] D. Knuth, *The TEXbook*, copyright 1984, Amer. Math. Soc., Volume 1 of *Computers and Typesetting*, Addison-Wesley.

[**Wh**] R. Whitney, Sanitizing control sequences under `\write`, *TUGboat* 11, no. 4 (Nov. 1990) 620–622.

◇ L. Siebenmann
Matématique, Bât 425
Univ. de Paris-Sud
91405 Orsay, France
`lcs@matups.matups.fr`