1 Making sense of error messages

Interpreting TEX's error messages can sometimes be like going to your physician with a complaint that you're feeling fatigued and being handed, in response, a breakdown of your blood chemistry. The explanation of your distress is probably there, but it's not easy to figure out what it is. A few simple rules will go a long way in helping you to understand TEX's error messages and get the most benefit from them.

Your first goal should be to understand what you did that caused TEX to complain. Your second goal (if you're working interactively) should be to catch as many errors as you can in a single run.

Let's look at an example. Suppose that your input contains the line:

```
We skip \quid a little bit.
```

You meant to type '`\quad`', but you typed '`\quid`' instead. Here's what you'll get from TEX in response:

```
! Undefined control sequence.
l.291 We skip \quid
                  a little bit.
?
```

This message will appear both at your terminal and in your log file. The first line, which always starts with an exclamation point (`!`), tells you what the problem is. The last two lines before the '`?`' prompt (which in this case are also the next two lines) tell you how far TEX has gotten when it found the error. It found the error on line 291 of the current input file, and the break between the two message lines indicates TEX's precise position within line 291, namely, just after `\quid`. The current input file is the one just after the most recent unclosed left parenthesis in the terminal output of your run (see p. '`infiles`').

This particular error, an undefined control sequence, is one of the most common ones you can get. If you respond to the prompt with another '`?`', TEX will display the following message:

```
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
1 or ... or 9 to ignore the next 1 to 9 tokens of input,
H for help, X to quit.
```

Here's what these alternatives mean:

- If you type ⟨return⟩, TEX will continue processing your document. In this case it will just ignore the `\quid`.
- If you type '`S`' (or '`s`'—uppercase and lowercase are equivalent here), TEX will process your document without stopping *except* if it encounters a missing file. Error messages will still appear at your terminal and in the log file.
- If you type '`R`' or '`r`', you'll get the same effect as '`S`' except that TEX won't even stop for missing files.
- If you type '`Q`' or '`q`', TEX will continue processing your document but will neither stop for errors nor display them at your terminal. The errors will still show up in the log file.
- If you type '`X`' or '`x`', TEX will clean up as best it can, discard the page it's working on, and quit. You can still print or view the pages that TEX has already processed.
- If you type '`E`' or '`e`', TEX will clean up and terminate as it would for '`X`' or '`x`' and then enter your text editor, positioning you at the erroneous line. (Not all systems support this option.)
- If you type '`H`' or '`h`', you'll get a further explanation of the error displayed at your terminal and possibly some advice about what to do about it. This explanation will also appear in your log file. For the undefined control sequence above, you'll get:

  ```
  The control sequence at the end of the top line
  of your error message was never \def'ed. If you have
  misspelled it (e.g., `\hobx'), type `I' and the correct
  spelling (e.g., `I\hbox'). Otherwise just continue,
  and I'll forget about whatever was undefined.
  ```

- If you type '`?`', you'll get this same message again.

The other two alternatives, typing '`I`' or a small integer, provide ways of getting TEX back on the track so that your error won't cause further errors later in your document:

- If you type '`I`' or '`i`' followed by some text, then TEX will insert that text as though it had occurred just after the point of the error,

at the innermost level where TEX is working. In the case of the example above, that means at TEX's position in your original input, namely, just after '`\quid`'. Later you'll see an example that shows the difference between inserting something at the innermost level and inserting it into your original input. In the example above of the undefined control sequence, if you type:

```
I\quad
```

TEX will carry out the `\quad` command and produce a quad space where you intended to have one.

- If you type a positive integer less than 100 (not less than 10 as the message misleadingly suggests), TEX will delete that number of tokens from the innermost level where it is working. (If you type an integer greater than or equal to 100, TEX will delete 10 tokens!)

Here's an example of another common error:

```
Skip across \hskip 3cn by 3 centimeters.
```

The error message for this is:

```
! Illegal unit of measure (pt inserted).
<to be read again>
                   c
<to be read again>
                   n
l.340 Skip across \hskip 3cn
                              by 3 centimeters.
```

In this case TEX has observed that '`3`' is followed by something that isn't a proper unit of measure, and so it's assumed the unit of measure to be points. TEX will read the tokens of '`cn`' again and insert them into your input, which is not what you want. In this case you can get a better result by first typing '`2`' to bypass the '`cn`'. You'll get the message:

```
<recently read> n

l.340 Skip across \hskip 3cn
                            by 3 centimeters.
```

Now you can type '`I\hskip 3cm`' to get the skip you wanted (in addition to the `3pt` skip that you've already gotten).[1]

If you type something that's only valid in math mode, TEX will switch over to math mode for you whether or not that's what you really wanted. For example:

```
So \spadesuit s are trumps.
```

[1] By typing '`I\unskip\hskip 3cm`' you can get rid of the `3pt` skip.

Here's TEX's error message:

```
! Missing $ inserted.
<inserted text>
                $
<to be read again>
                   \spadesuit
l.330 So \spadesuit
                  s are trumps.
```

Since the `\spadesuit` symbol is only allowed in math mode, TEX has inserted a '`$`' in front of it. After TEX inserts a token, it's positioned in *front* of that token, in this case the '`$`', ready to read it. Typing '`2`' will cause TEX to skip both the '`$`' and the '`\spadesuit`' tokens, leaving it ready to process the '`s`' in '`s are trumps.`'. (If you just let TEX continue, it will typeset '`s are trumps`' in math mode.)

Here's an example where TEX's error diagnostic is downright wrong:

```
\hbox{One \vskip 1in two.}
```

The error message is:

```
! Missing } inserted.
<inserted text>
                }
<to be read again>
                   \vskip
l.29 \hbox{One \vskip
                     1in two.}
```

The problem is that you can't use `\vskip` when TEX is in restricted horizontal mode, i.e, constructing an hbox. But instead of rejecting the `\vskip`, TEX has inserted a right brace in front of it in an attempt to close out the hbox. If you accept TEX's correction, TEX will complain again when it gets to the correct right brace later on. It will also complain about anything before that right brace that isn't allowed in vertical mode. These additional complaints will be particularly confusing because the errors they indicate are bogus, a result of the propagated effects of the inappropriate insertion of the right brace. Your best bet is to type '`5`', skipping past all the tokens in '`}\vskip 1in`'.

Here's a similar example in which the error message is longer than any we've seen so far:

```
\leftline{Skip \smallskip a little further.} But no more.
```

The mistake here is that `\smallskip` only works in a vertical mode. The error message is something like:

```
! Missing } inserted.
<inserted text>
```

```
                }
<to be read again>
                   \vskip
\smallskip ->\vskip
                    \smallskipamount
<argument> Skip \smallskip
                              a little further.
\leftline #1->\line {#1
                         \hss }
l.93 ...Skip \smallskip a little further.}
                                                But no more.
```

The error messages here give you a tour through the macros that are used in plain TEX's implementation of `\leftline`—macros that you probably don't care about. The first line tells you that TEX intends to cure the problem by inserting a right brace. TEX hasn't actually read the right brace yet, so you can delete it if you choose to. Each component of the message after the first line (the one with the '!') occupies a pair of lines. Here's what the successive pairs of lines mean:

1) The first pair indicates that TEX has inserted, but not yet read, a right brace.
2) The next pair indicates that after reading the right brace, TEX will again read a '`\vskip`' command (gotten from the macro definition of `\smallskip`).
3) The third pair indicates that TEX was expanding the `\smallskip` macro when it found the error. The pair also displays the definition of `\smallskip` and indicates how far TEX has gotten in expanding and executing that definition. Specifically, it's just attempted unsuccessfully to execute the `\vskip` command. In general, a diagnostic line that starts with a control sequence followed by '`->`' indicates that TEX has been expanding and executing a macro by that name.
4) The fourth pair indicates that TEX was processing a macro argument when it found the `\smallskip` and also indicates TEX's position in that argument, i.e., it's just processed the `\smallskip` (unsuccessfully). By looking ahead to the next pair of lines we can see that the argument was passed to `\leftline`.
5) The fifth pair indicates that TEX was expanding the `\leftline` macro when it found the error. (In this example the error occurred while TEX was in the middle of interpreting several macro definitions at different levels of expansion.) Its position after `#1` indicates that the last thing it saw was the first (and in this case the only) argument to `\leftline`.
6) The last pair indicates where TEX is positioned in your input file. Note that this position is well beyond the position where it's inserting

the right brace and reading '`\vskip`' again. That's because TEX has already read the entire argument to `\leftline` from your input file, even though it's only processed part of that argument. The dots at the beginning of the pair indicate a preceding part of the input line that isn't shown. This preceding part, in fact, includes the `\leftline` control sequence that made the `\vskip` illegal.

In a long message like this, you'll generally find only the first line and the last pair of lines to be useful; but it sometimes helps to know what the other lines are about. Any text that you insert or delete will be inserted or deleted at the innermost level. In this example the insertion or deletion would occur just before the inserted right brace. Note in particular that in this case TEX puts any text you might insert *not* into your input text but into a macro definition several levels down. (The original macro definition is of course not modified.)

You can use the `\errorcontextlines` command (p. '`\errorcontextlines`') to limit the number of pairs of error context lines that TEX produces. If you're not interested in all the information that TEX is giving you, you can set `\errorcontextlines` to 0. That will give you just the first and last pairs of lines.

Finally, we'll mention two other indicators that can appear at the start of a pair of error message lines:

- `<output>` indicates that TEX was in the middle of its output routine when this error occurred.
- `<write>` indicates that TEX was in the middle of executing a `\write` command when this error occurred. TEX will detect such an error when it is actually doing the `\write` (during a `\shipout`), rather than when it first encounters the `\write`.