Is a given input a valid TeX ⟨*number*⟩?

Udo Wermuth

Abstract

This article discusses the question of how one can determine if a given string of characters represents a valid number for TeX. A macro that looks and behaves like a Boolean conditional is implemented to answer the question.

1 Introduction

TeX operates with several data types and structures. We all know, for example, characters, numbers, dimensions, skips, token lists, boxes, files, and macros. Such structures are created and manipulated during the text processing and sometimes we need to get information about the currently stored contents. Thus, TeX provides a couple of conditional tests to gain insights; see pages 209–210 of *The TeXbook* [1]. Except for one, all of these tests return a Boolean result, i.e., *true* or *false*, and allow therefore two branches with different text and code. Five tests compare two items and two of them require a relation for the test as they don't test only for agreement of a single common characteristic.

All Boolean conditionals use the same scheme "`\if...` ⟨*true branch*⟩`\else` ⟨*false branch*⟩`\fi`" in which the `\else` and ⟨*false branch*⟩ can be omitted if this branch is empty. The structure itself is *expandable* ([1], p. 213) and the conditionals can be nested as TeX keeps track about the control words `\if...`, `\else`, and `\fi` even if they aren't executed; conditionals are *skippable* ([1], p. 211).

Plain TeX provides the command `\newif` so that users can create new Boolean conditionals ([1], p. 211). The user must set the conditional: Whatever the flag should mean its result must be computed before. The names of these flags must start with `\if` and thus these conditionals match the above scheme.

For example, TeX has no built-in test for the question if a given input string represents a valid number. So one must code a macro for this test and then a user-created Boolean conditional can be set to true or false. Unfortunately, the macro that must be coded turns out to be rather complex.

The TeX FAQ (accessible through TUG's Internet site or `https://texfaq.org/FAQ-isitanum`) contains information about this question. It focuses on short code snippets and so it limits itself to a discussion about "Is the input a not too large signed or unsigned decimal number?" without giving an answer. Similar limitations occur in the code shown on page 361f. of `https://ctan.org/tex-archive/info/apprendre-a-programmer-en-tex/output/apprendre-a-programmer-en-tex.pdf`.

Encodings. TeX knows many encodings for integers ([1], p. 269 and p. 118): decimal, octal, hexadecimal, and as an alphabetic constant. It accepts numbers in the range from $-2^{31}+1$ to $2^{31}-1$. For example, valid numbers are:

`+-"FF` (a *hexadecimal number*; value −255),
`+- -+"FF` (another hexadecimal number; 255),
`-'777` (an *octal number*; −511),
`` `a `` (an *alphabetic constant*; 97),
`+2147483647` ($=2^{31}-1$, the largest number)

whereas `2147483648` ($=2^{31}$) is invalid as it's out of TeX's range.

To check all cases that TeX allows as the encoding of a number and to do a range check is an unnecessary effort for most macro packages with user-supplied integer arguments. Only if the macro offers an interface for receiving integers from external sources does one need to implement TeX's syntax rules.

Contents. This article describes how to implement a TeX macro looking like a Boolean conditional that decides if a given input string forms a valid number. The macro is named `\ifisint`.

Section 2 lists a few expectations that the conditional should fulfill. Section 3 contains the code for `\ifisint`.

2 Expectations a.k.a. goals

Let's state as precisely as possible what we want to achieve with `\ifisint`.

(1) A Boolean conditional should be coded that has a structure similar to the other Boolean conditionals of TeX. It carries the name `\ifisint`. The argument that is tested for being a valid TeX number is delimited by `\Boolend`. Except for this control word the structure is familiar: `\ifisint` ⟨*argument*⟩ `\Boolend` ⟨*true branch*⟩`\else` ⟨*false branch*⟩`\fi`.

The conditional itself doesn't output anything; only the branches might output something.

(2) Any valid number for TeX in any allowed encoding either unbraced and then followed by any number of spaces or between braces and no spaces in front of `\Boolend` is recognized by `\ifisint` and the tokens in ⟨*true branch*⟩ are processed. Any other input makes TeX execute ⟨*false branch*⟩ if it is present.

(3) Of course, not all characters might appear in ⟨*argument*⟩. For example, TeX's comment character, the percent sign, is never part of a number; TeX reports an error if `\Boolend` is commented out. On

doi.org/10.47397/tb/45-1/tb139wermuth-isint

the other hand, the input “`2^3`” should throw no error message although the math shifts are missing.

The input “`{2^3}`” can be passed to *any* macro as an argument without error. But without braces “`2^3`” as a single argument throws an error if the argument is not delimited. Entered as ⟨*argument*⟩ to `\ifisint` there should not be an error message.

Not all of TeX’s special characters can occur. For example, a single ‘`{`’ starts a group and without an ending ‘`}`’ TeX will report an error.

If a user wants to test any input string without error messages, TeX’s special characters need other category codes. Plain TeX provides the macro `\dospecials` that helps in this task; see page 380 of [1]. Furthermore, TeX’s special *double-hat notation* ([1], p. 45; “hat” p. 369) doesn’t work if ‘`^`’ does not have category 7. The valid encoding of -1 as “`^^m^^31`” is then rejected. So leave the hat character as special if that is possible.

(4) The ⟨*argument*⟩ should receive written-out input. It doesn’t make sense to test data stored in, say, a `\count` register to find out if it represents a valid TeX number. But a simple macro that stores a number in its replacement text should be accepted; undefined macros shall be reported.

This makes `\ifisint` different from, for example, `\ifodd`, as this conditional accepts, for example, count registers for its test. With `\ifisint`, code this: `\expandafter \ifisint \the\count`⟨*n*⟩␣ `\Boolend`. Of course, if “⟨*n*⟩” isn’t allowed after `\count`, say, because $n > 255$, an error is raised.

(5) The conditional must be skippable, i.e., the following input with nested `\if`s

```
\iffalse\ifisint 117\Boolend\message{A}%
        \else\message{B}\fi
\else\message{C}\fi
```

generates no error message and outputs “C” on the terminal.

(6) It is *not* expected that the new conditional `\ifisint` is expandable.

3 The code for \ifisint

What is a valid integer? This is specified in detail on pages 268–269 of [1]. There are four types of integers in ⟨*normal integer*⟩: a) the ⟨*integer constant*⟩, b) the ⟨*octal constant*⟩ that starts with a right quote, c) the ⟨*hexadecimal constant*⟩ that starts with the ditto mark, and d) the alphabetic constant built from a left quote and a ⟨*character token*⟩. We are not interested in the syntactic quantity ⟨*internal integer*⟩ as it stands for valid integers stored in control words of TeX; see page 271 of [1]. Any type can be followed by an optional space. Moreover, integers can have signs of category 12: ‘+’ and ‘−’. One can use a chain of signs and separate them by spaces: Page 268 defines ⟨*plus and minus*⟩ and on page 269 it’s stated that in ⟨*optional signs*⟩, the signs might be followed by ⟨*optional spaces*⟩.

And what does TeX do if it expects a number but finds none? For an answer we have to look into [2], part 26, “Basic scanning subroutines”. Sections 440–446 contain the code for the *scan_int* procedure that reads a number. Here we also find the three error messages that can occur. Section 442 presents the first error message “Improper alphabetic constant”, section 445 contains “Number too big”, and section 446 includes the code for the third message “Missing number, treated as zero”.

The last message tells us what TeX does if, for example, a letter is read but a digit was expected: It recovers by inserting the number 0; nothing is removed from the input. In the first error it happens too, as explained in the help text. TeX uses its largest known integer 2147483647 when it finds a number whose absolute value is too big; again, information from the help text. In this case all digits of the large number are read and digested by TeX.

Analysis. Thus, in essence there are six cases that our new macro must distinguish.

1. TeX reads a valid number; no more input.
2. TeX reads a valid number; more input available.
3. TeX doesn’t find a number, uses 0 instead; no more input.
4. TeX doesn’t find a number, uses 0 instead; more input available.
5. TeX reads a number that’s outside of its range, uses 2147483647 instead; no more input.
6. TeX reads a number that’s outside of its range, uses 2147483647 instead; more input available.

Only case 1 is a valid TeX number. When we are able to determine if more input is available then cases 2, 4, and 6 are detectable.

Only the following input strings fulfill case 3: the empty input, ‘`'`’, ‘`"`’, and ‘`` ` ``’. All can be preceded by any number of signs.

Case 5 remains. To distinguish it from case 1 we must be able to check the infinite number of inputs that represent the largest number of TeX. The number is infinite as we can always add another plus or minus sign and more leading zeros in the input.

What we need is a *canonical form* into which we transform the input. With unsigned numbers only a few forms for case 3 remain. An unsigned number with exactly one leading zero leaves for case 5 only three forms: the largest number with a leading zero in decimal, octal, and hexadecimal notation.

Thus, a couple of comparisons solve the main task if we find answers to the following problems.

A. Find a way to detect if the number is followed by more input.
B. Find a way to construct the canonical form.

Problem A is solved with an assignment of the input string to a count register *inside* an hbox. This box is empty if only a number is input.

```
\setbox0=\hbox{\count255=<input>}%
\ifdim\wd0>0pt % <input> is not a number
```

Problem B is solved with two macros: The first removes the signs and the second leading zeros but also assures that one is present. They use the technique called *tail recursion* ([1], p. 219) to do their job. For example, the following code removes signs.

```
\def\II@rmsign #1{\ifx#1+\else\ifx#1-\else
 \II@endrm#1\fi\fi\II@rmsign}
\def\II@endrm #1\fi\fi#2{\fi\fi#1}
```

If the argument to `\II@rmsign` is '+' or '−', one of the two `\ifx` becomes true, the sign is gobbled as nothing is done in the branch, but at the end the macro is called again. The macro stops if the argument isn't a sign; a trick to shuffle the argument and the `\fi`s is needed, though. To make sure that the macro stops we add a *sentinel*, the letter 'W', to the input. This moves the detection of a bad alphabetic constant from case 3 to the box-width check.

But this solution has a shortcoming. It removes not only signs but also signs in curly braces, while such symbols aren't allowed in a valid number. Well, the validity of the number is determined by other means. We don't care that valid and invalid numbers are mapped to the same canonical form at this stage.

To summarize: We do the following steps in a TeX macro.

Step 1: 1) Remove signs; add sentinel. 2) Test that case 3 is excluded; otherwise return false.
Step 2: Create canonical form.
Step 3: 1) Assign the input to a `\count` register inside an hbox. 2) Test that the box width is the width of the sentinel; 3) otherwise return false (cases 2, 4, 6).
Step 4: 1) Return true if the number isn't TeX's maximum. 2) Otherwise test if the canonical form is TeX's maximum. If yes, return true (case 1). 3) Otherwise return false (case 5).

Note the procedure works with errors that are generated intentionally. As TeX limits the number of errors in a single paragraph to 100 ([2], §76) the macro shouldn't be applied, for example, in a loop.

One task is still open: How to suppress TeX's error messages? We cannot do that but we can switch to `\batchmode` so that the messages aren't displayed on the terminal. The terminal gets a blank line when we switch between `\batchmode` and another mode.

There is a little problem as modes are globally set and in the original TeX we don't know to which mode we must return. The macro uses a configurable parameter; the default is `\errorstopmode`.

My implementation. Note, Step 1 includes the expansion in an `\edef`; see the discussion in section 2, no. 4. And `\Boolend` is given the significance of `\iffalse` to make the macro skippable; see section 2, no. 5. Moreover, `\hbox{\II@font W}` has width 10.2778 pt if `\II@font` represents `cmr10`.

```
\catcode`\@=11 %  use the private prefix ``II@''
\newif\ifII@itis      % main result of the macro
%% helper macros
\def\II@rmsign #1{\ifx#1+\else\ifx#1-\else
 \II@endrm#1\fi\fi\II@rmsign}
\def\II@endrm #1\fi\fi#2{\fi\fi#1}
\def\II@zeros #1{\ifx#1''\else\ifx#1""\else
 \II@cont#1\fi\fi\II@zeros}
\def\II@cont #1\fi\fi#2{\fi\fi\II@hdlzero#1}
\def\II@hdlzero #1{\ifx#10 \else
 \II@xchgfi #1\fi\II@hdlzero}
\def\II@xchgfi #1\fi#2{\fi\ifx#1`\else0\fi#1}
%% constants with the sentinel `W'
\def\II@cfd{02147483647W}%  canonical forms with
\def\II@cfh{"07FFFFFFFW}% W of TeX's max integer
\def\II@cfo{'017777777777W}%    in dec, hex, oct
\def\II@W{W}\def\II@hexW{"W}% all unsigned input
\def\II@octW{'W}% with W for which TeX inserts 0
%% assignments
\let\Boolend=\iffalse \font\II@font=cmr10
\let\IIcurrentmode=\errorstopmode    % CONFIGURE
%% main macro
\def\ifisint #1\Boolend{\II@itisfalse % see S1.2
 \edef\II@digs{\II@rmsign#1W}% S1.1 with 2 \edef
 \edef\II@digs{\expandafter\II@rmsign\II@digs}%
 \ifx\II@digs\II@W\else\ifx\II@digs\II@octW
 \else\ifx\II@digs\II@hexW\else  % S1.2 finished
  \edef\II@cf{\expandafter\II@zeros\II@digs}% S2
  \wlog{=== start ignore}\batchmode\begingroup
   \setbox0=\hbox{\count255=\II@cf
    \xdef\II@val{\the\count255}}%
   \setbox0=\hbox{\II@font\count255=#1W}%   S3.1
   \xdef\II@wd{\the\wd0}%
  \endgroup\IIcurrentmode\wlog{=== stop ignore}%
  \ifdim\II@wd=10.2778pt % \wd of hbox `W'; S3.2
   \II@itistrue \ifnum\II@val=2147483647 %  S4.1
    \ifx\II@cf\II@cfd
    \else\ifx\II@cf\II@cfh
    \else\ifx\II@cf\II@cfo                 % S4.2
    \else \II@itisfalse                    % S4.3
   \fi\fi\fi\fi
  \else \II@itisfalse                      % S3.3
 \fi\fi\fi\fi \ifII@itis}
\catcode`\@=12
```

A few remarks. The second `\edef` for `\II@digs` can be deleted if macro expansion as in section 2, no. 4, is not needed. Currently it's possible to code:

```
\def\mynum{-1234 }\ifisint\mynum\Boolend ...
```

This expansion is performed outside of `\batchmode` so that errors are shown to the user. I did this to avoid misinterpretations if the user enters a faulty sequence and thinks the contents of the macro was tested, i.e., if the user enters something erroneous like this: `\ifisint\maynum\Boolend ...`

The control word `\Boolend` is used in the macro `\ifisint` as delimiter, i.e., `\ifisint` has a *delimited parameter* ([1], p. 203f.). But in the case of delayed execution of `\ifisint`, for example, if the primitive `\expandafter` precedes it, the user must be careful not to execute `\Boolend`.

As mentioned above, `\Boolend` receives via a `\let`-assignment the meaning of `\iffalse`. Thus if TeX executes `\Boolend` it also executes `\ifisint`'s ⟨*false branch*⟩. Therefore, the use of a `\count` register in section 2, no. (4), requires a space between the number of the `\count` register and the delimiter `\Boolend` to avoid the erroneous execution of `\Boolend` that destroys the macro `\ifisint`.

A second `\let`-assignment gives the control sequence `\IIcurrentmode` the meaning of TeX's primitive `\errorstopmode`. A user can change this by another `\let`-assignment so that `\ifisint` returns to the mode that is currently active. (With ε-TeX one can query the current mode and return to it after `\ifisint` has done its work.)

Execution time. TeX needs more time to execute `\ifisint` than it needs to perform `\ifodd`, i.e., the only built-in conditional with a single number. Measurements on my system with my own TeX port in Pascal show that `\ifisint` is ≈ 8.5 times slower than `\ifodd` when the "real" times of 100,000 calls of "`\ifodd 255\fi`" and of 100,000 calls of "`\ifisint 255\Boolend\fi`", with the additional assignment "`\def\wlog #1{}`", are compared.

References

[1] Donald E. Knuth, *The TeXbook*, Volume A of *Computers & Typesetting*, Boston, Massachusetts: Addison-Wesley, 1984.

[2] Donald E. Knuth, *TeX: The Program*, Volume B of *Computers & Typesetting*, Boston, Massachusetts: Addison-Wesley, 1986.

◇ Udo Wermuth
Dietzenbach, Germany
`u dot wermuth (at) icloud dot com`