## TEX.StackExchange cherry picking, part 2: Templating

Enrico Gregorio

**Abstract**

We present some examples of macros built with expl3 in answer to users' problems presented on `tex.stackexchange.com`, to give a flavor of the language and explore its possibilities.

### 1 Introduction

This is the second installment[1] of my cherry picking from questions on TEX.SX that I answered using expl3. As every regular of TEX.SX knows, I like to use expl3 code for solving problems, because I firmly believe in its advantages over traditional TEX and LATEX programming.

This paper is mostly dedicated to "templating", an idea I'm getting fond of. There is actually a "templating layer" for the future LATEX3, but it's not yet in polished shape. The Oxford Dictionary of English tells us that the primary meaning of *template* is "a shaped piece of rigid material used as a pattern for processes such as cutting out, shaping, or drilling", more generally "something that serves as a model for others to copy" and this is how I think about *templating* here. This article is not about "document templates", the (often unfortunately bad) pieces of code that users are sometimes forced to fill in.

Basically, the templating I'm going to describe is achieved by defining a temporary function (macro, if you prefer) to do some job during a loop. This has the advantage of allowing us to use the standard placeholders such as `#1`, `#2` and so on instead of macros, as is done (for instance) with the popular `\foreach` macro of Ti*k*Z/PGF, with concomitant expansion problems. For instance, with

```
\foreach \i in {1,2,3} { -\i- }
```

the loop code doesn't "see" 1, 2 and 3, but `\i`, which *expands* to the current value. In expl3 we have

```
\clist_map_inline:nn {1,2,3} { -#1- }
```

and this is possible by the same templating technique I'll describe.

There are a couple of unrelated topics, just to show other applications of loops.

*Technical note.* The code shown here may differ slightly from the post on TEX.SX, but the functionality is the same. Sometimes I had afterthoughts, or decided to use new functions added to expl3 in the meantime. As in the first installment, the code I show should be thought of as surrounded by `\ExplSyntaxOn` and `\ExplSyntaxOff` (except for macro calls), and `\usepackage{xparse}` is mandatory.

[1] For the first installment, "TEX.StackExchange cherry picking: expl3", see *TUGboat* 39:1, pp. 51–59, `tug.org/TUGboat/tb39-1/tb121gregorio-expl3.pdf`. Some terminology introduced there is used here without explanation.

### 2 Euclid's algorithm

I've taught algebra classes for a long time and the Euclidean algorithm for the greatest common divisor has always been a favorite topic, because it's a very good way to introduce several ideas, particularly recursion. The algorithm consists in repeating the same operations until we get to 0. Thus any implementation must clearly show recursion at work. Here's the code for it.[2] (Well, a reduced version thereof for nonnegative numeric input only.)

```
\NewExpandableDocumentCommand{\euclid}{mm}
 {
  \egreg_euclid:nn { #1 } { #2 }
 }

\cs_new:Nn \egreg_euclid:nn
 {
  \int_compare:nTF { #2 = 0 }
    { #1 } % end
    {
     \egreg_euclid:nf
      { #2 }
      { \int_mod:nn { #1 } { #2 } }
    }
 }
\cs_generate_variant:Nn \egreg_euclid:nn { nf }
```

Some implementations do a swap if the first number is less than the second, but this is unnecessary, because the algorithm itself will perform it. The terminating condition is reached when we get a remainder of 0, in which case we output the first number. Otherwise the function calls itself with the first argument being the previous second one, and the second argument being the current remainder of the division.

Pure and simple: the only trick, for greater efficiency, is to call a variant of the main function in order to fully expand the mod operation.

It would be much more fun to implement the mod operation in fully expandable fashion, but somebody has already done it! Let's enjoy laziness!

My answer also features a possibly interesting `\fulleuclid` macro that *prints* the steps.

### 3 Mappings

Data can be available in "serial" form; expl3 has several data types of this kind: *clists* (comma separated lists), *sequences* and *property lists*. Further, token

[2] `https://tex.stackexchange.com/q/453877/`

lists can be seen as a list of tokens (or braced groups). For each of these data types expl3 provides mapping functions that process each item of the given data.

The first toy problem consists in making a short table of contents exploiting `\nameref`.[3] The user inputs something like

```
\procedurelist{
  PM-tyinglaces,
  PM-polishshoes,
  PM-ironshirt
}
```

where the input is a comma separated list of labels and the document contains something like

```
\section{Tying laces}\label{PM-tyinglaces}
```

for each label used. This should print a table with section numbers in the first column and section titles in the second column.

```
\NewDocumentCommand{\procedurelist}{m}
 {
  \begin{tabular}{cl}
  \toprule
  \multicolumn{1}{c}{\textbf{PM}} & \textbf{Name} \\
  \midrule
  \clist_map_function:nN
   { #1 }
   \__kjc_procedurelist_row:n
  \bottomrule
  \end{tabular}
 }
\cs_new:Nn \__kjc_procedurelist_row:n
 { \ref{#1} & \nameref{#1} \\ }
```

The `\clist_map_function:nN` command splits its first argument at commas, trimming spaces before and after each item and discarding empty items, and then passes each item to the function specified as the second argument. A variant is also available, `\clist_map_function:NN`, which expects as its first argument a *clist variable*.

In this case, the auxiliary function which the mapping is handed to is a template for a table row.

In contrast to other approaches, the operation is not hindered by being in a table, because the complete list of tokens

```
\__kjc_procedurelist_row:n {PM-tyinglaces}
\__kjc_procedurelist_row:n {PM-polishshoes}
\__kjc_procedurelist_row:n {PM-ironshirt}
```

is built before TEX starts expanding the first macro.

A similar approach with

```
\clist_map_inline:nn
 { #1 }
 { \ref{##1} & \nameref{##1} \\ }
```

would not work here, because the mapping would start in a table cell and end in another one, which is impossible.

However, the user here is far-seeing and is worried about their boss not liking the appearance of the table or perhaps wanting such tables printed in slightly different ways across the document.

We can make the auxiliary function variable! Here's the code.

```
\NewDocumentCommand{\procedurelist}{om}
 {
  \group_begin:
  \IfValueT{#1}
   {
    \cs_set:Nn \__kjc_procedurelist_row:n { #1 }
   }
  \begin{tabular}{cl}
  \toprule
  \multicolumn{1}{c}{\textbf{PM}} & \textbf{Name} \\
  \midrule
  \clist_map_function:nN
   { #2 }
   \__kjc_procedurelist_row:n
  \bottomrule
  \end{tabular}
  \group_end:
 }
\cs_new:Nn \__kjc_procedurelist_row:n
 { \ref{#1} & \nameref{#1} \\ }
```

We added an optional argument to the main macro. If this argument is present, then it will be used to redefine the auxiliary function. Grouping is used in order not to clobber the standard meaning of the auxiliary function. The same call as before would print the same table, but a call such as

```
\procedurelist[\ref{#1} & \textit{\nameref{#1}} \\]{
  PM-tyinglaces,
  PM-polishshoes,
  PM-ironshirt
}
```

would apply `\textit` to each section title. This is what I call *templating*.

## 4 Dummy variables

A user asked for a `\replaceproduct` macro so that

```
\replaceproduct{p_i^{\varepsilon_i}}{3}{n}
```

produces

$$p_1^{\varepsilon_1} p_3^{\varepsilon_3} p_3^{\varepsilon_3} \dots p_n^{\varepsilon_n}$$

The idea is that the "dummy variable" $i$ is replaced by the numbers 1, 2 and 3 (the value is given in the second argument) and by $n$ at the end (third argument).[4]

The idea of templating and using `#1` does not have the same appeal as before, because the call would need to be

```
\replaceproduct{p_#1^{\varepsilon_#1}}{3}{n}
```

which is more obscure. What if we make the dummy variable an optional argument, with default value `i`? Maybe we need to use `k` because the main template contains the imaginary unit or we want to be obscure at all costs (see the final example).

[3] https://tex.stackexchange.com/q/451423/

[4] https://tex.stackexchange.com/q/448389/

However, TEX doesn't allow placeholders for parameters other than #1, #2 and so on — but we have expl3 regular expression search and replace. Can we store the template in a token list variable and replace all appearances of the dummy variable with #1? Certainly we can: assuming that the template is the second argument in the macro we're going to define, we can do

```
\tl_set:Nn \l__egreg_rp_term_tl { #2 }
```

Now we can search and replace the dummy variable, which is given as (optional) argument #1, by

```
\regex_replace_all:nnN
 { #1 } % search
 { \cB\{ \cP\# 1 \cE\} } % replace
 \l__egreg_rp_term_tl % what to act on
```

so that each appearance of i (or whatever letter we specified in the optional argument) is changed into {#1}.[5] The prefix \cB to \{ is not standard in regexes[6]; in TEX we need to be careful with category codes, so \cB says that the next token must be given the role of a "begin group" characters. Similarly for \cE as "end group" and \cP for "parameter".

We now face the final problem: how to pass this to the auxiliary function that serves as internal template? We should do

```
\cs_set:Nn \__egreg_rp_term:n {⟨template⟩}
```

but the template is stored in a token list variable that we need to get the value of. That's a known problem and expl3 already has the solution: the argument type V, that denotes take the *value* of the variable and pass it, braced, as an argument to the main function. So we just need to define a suitable variant of \cs_set:Nn, namely

```
\cs_generate_variant:Nn \cs_set:Nn { NV }
```

We now have all the ingredients and we can bake our cake:

```
\NewDocumentCommand{\replaceproduct}{O{i}mmm}
 {% #1 = item to substitute
  % #2 = main terms
  % #3 = first terms
  % #4 = last term
  \group_begin:
  \egreg_rp:nnnn { #1 } { #2 } { #3 } { #4 }
  \group_end:
 }
\tl_new:N \l__egreg_rp_term_tl
\cs_generate_variant:Nn \cs_set:Nn { NV }
\cs_new:Nn \egreg_rp:nnnn
 {
  \tl_set:Nn \l__egreg_rp_term_tl { #2 }
  \regex_replace_all:nnN
   { #1 } % search
   { \cB\{\cP\#1\cE\} } % replace
   \l__egreg_rp_term_tl % what to act on
  \cs_set:NV \__egreg_rp_term:n \l__egreg_rp_term_tl
  \int_step_function:nN { #3 } \__egreg_rp_term:n
  \cdots
  \__egreg_rp_term:n { #4 }
 }
```

The only new bit is \int_step_function:nN, which is similar to the mapping functions, but uses the most natural series: the numbers from 1 up to the integer specified as end; each number, in its explicit decimal representation, is passed to the auxiliary function as in the mapping of the previous section. In the example call, the third argument is 3, so there will be three steps.

The function \int_step_function:?N comes in three flavors:

```
\int_step_function:nN
\int_step_function:nnN
\int_step_function:nnnN
```

In the two-argument version we only specify the end, in the three-argument version we specify the start and end; in the full version the three n-type arguments specify starting point, step and final point. Thus the calls

```
\int_step_function:nN {3} \__egreg_rp_term:n
\int_step_function:nnN {1} {3} \__egreg_rp_term:n
\int_step_function:nnnN {1} {1} {3} \__egreg_rp_term:n
```

are equivalent.

We can call the macro like

```
\[ \replaceproduct{p_i^{\varepsilon_i}}{3}{m} =
\replaceproduct[j]{i_j^{\eta_j}}{2}{n} \]
```

to get

$$p_1^{\varepsilon_1} p_2^{\varepsilon_1} p_3^{\varepsilon_1} \dots p_m^{\varepsilon_m} = i_1^{\eta_1} i_2^{\eta_2} \dots i_n^{\eta_n}$$

The same idea can be used for a "macro factory" where the task is to define one-parameter macros using the keyword PARAM in place of #1.[7] The problem here is that apparently plain TEX is used, but it's not really difficult: expl3 can also be used on top of it.

The input \foo{qix}{:: PARAM ::} should be equivalent to

```
\def\barqix#1{:: #1 ::}
```

and this can be accomplished easily.

```
\input expl3-generic

\ExplSyntaxOn
\tl_new:N \l__egreg_param_tl

\cs_new_protected:Npn \foo #1 #2
 {
  \tl_set:Nn \l__egreg_param_tl { #2 }
  \regex_replace_all:nnN
   { PARAM }
   { \cP\#1 }
   \l__egreg_param_tl
  \cs_set:NV \__egreg_param:n \l__egreg_param_tl
```

[5] TEXnical remark: things are set up so that # tokens are not unnecessarily doubled when stored in a token list variable.

[6] Informal abbreviation for "regular expressions".

[7] https://tex.stackexchange.com/q/355568/

```
  \cs_new_eq:cN {bar#1} \__egreg_param:n
 }
\cs_generate_variant:Nn \cs_set:Nn { NV }
\ExplSyntaxOff

\foo{qix}{:: PARAM ::}

\barqix{hi world}

\bye
```

The function `\__egreg_param:n` is temporary, but necessary: a variant such as `\cs_new:cpV` cannot be defined because of the parameter text which can consist of an arbitrary number of tokens to jump over. Here I exploit the fact that `\cs_set:Nn` computes its parameter text from the signature.

## 5 Double loops

We want to be able to generate and print a triangular diagram such as

$$\begin{array}{llll} 1\times 1=1 & & & \\ 2\times 1=2 & 2\times 2=4 & & \\ 3\times 1=3 & 3\times 2=6 & 3\times 3=9 & \\ 4\times 1=4 & 4\times 2=8 & 4\times 3=12 & 4\times 4=16 \end{array}$$

with as little effort as possible.[8]

This calls for using `array`, but it also needs a double loop, which makes it inconvenient to use `\int_step_function:nN` as before, because only one argument is passed to the auxiliary function. The posted answer by Jean-François Burnol (jfbu) is as usual very nice, but not easily extendable — we may want only the operations, instead of also showing the result; or to do addition instead of multiplication; etc.

My idea is to define a macro `\lowertriangular` which takes as arguments the number of rows and what to do with the indices in each cell; for instance, the diagram above would be generated by

```
\lowertriangular{4}{#1\times #2 = \inteval{#1*#2}}
```

The macro `\inteval` is provided by xfp, which is part of the xparse family accompanying expl3 and needs to be loaded. A lower triangular matrix can be generated by

```
\left[\lowertriangular{4}{a_{#1#2}}\right]
```

After all, I teach linear algebra courses, so triangular matrices are my bread and butter.

The placeholders `#1` and `#2` stand, respectively, for the row and column index.

We need nested loops, the outer one stepping the row index, the inner one stepping the column index, but only up to the row index. However, since we have to make an `array`, we need to build the body beforehand and then feed it to the matrix building environment. Small complication: the indices provided by the outer loop are called `##1`, those relative to the inner loop are called `####1` (it's quite predictable, but has to be mentioned).

As I said, we have to use the "inline" form for the loop, that is, `\int_step_inline:nn` (which has siblings like those described before for the similar `\int_step_function:nN`). When we are at the "cell level", we will use the auxiliary function defined with the template given as second argument to the user level macro.

```
\NewDocumentCommand{\lowertriangular}{mm}
 {
  \group_begin:
  \egreg_lt_main:nn { #1 } { #2 }
  \group_end:
 }

\tl_new:N \l__egreg_lt_body_tl

\cs_new_protected:Nn \egreg_lt_main:nn
 {
  % an auxiliary function for massaging the entries
  \cs_set:Nn \__egreg_lt_inner:nn { #2 }
  % clear the table body
  \tl_clear:N \l__egreg_lt_body_tl
  % outer loop, #1 rows
  \int_step_inline:nn { #1 }
   {
    % inner loop, ##1 columns
    \int_step_inline:nn { ##1 }
     {
      % add the entry for row ##1 (outer loop),
      % column ####1 (inner loop)
      \tl_put_right:Nn \l__egreg_lt_body_tl
       { \__egreg_lt_inner:nn { ##1 } { ####1 } }
      % if ##1 = ####1 end the row,
      % otherwise end the cell
      \tl_put_right:Nx \l__egreg_lt_body_tl
       {
        \int_compare:nTF { ##1 = ####1 }
         { \exp_not:N \\ } % end row
         { & }             % end cell
       }
     }
   }
  % output the table
  \begin{array}{ @{} *{#1}{c} @{} }
  \l__egreg_lt_body_tl
  \end{array}
 }
```

An almost straightforward modification of the code allows for producing upper as well as lower triangular matrices. It's sufficient to add a test to make the inner loop go on all the way, instead of stopping at the diagonal.

```
\NewDocumentCommand{\lowertriangular}{mm}
 {
  \group_begin:
  \egreg_tm_main:nnn { #1 } { #2 } { >= }
  \group_end:
 }
\NewDocumentCommand{\uppertriangular}{mm}
 {
```

[8] https://tex.stackexchange.com/q/435349/

```
  \group_begin:
  \egreg_tm_main:nnn { #1 } { #2 } { <= }
  \group_end:
 }

\tl_new:N \l__egreg_tm_body_tl

\cs_new_protected:Nn \egreg_tm_main:nnn
 {% #1 = size, #2 = template, #3 = < or >
  % an auxiliary function for massaging the entries
  \cs_set:Nn \__egreg_tm_inner:nn { #2 }
  % clear the table body
  \tl_clear:N \l__egreg_tm_body_tl
  % outer loop, #1 rows
  \int_step_inline:nn { #1 }
   {
    % inner loop, #1 columns
    \int_step_inline:nn { #1 }
     {
      % add the entry for row ##1 (outer loop),
      % column ####1 (inner loop) only if
      % ##1 #3 ####1 is satisfied
      \int_compare:nT { ##1 #3 ####1 }
       {
        \tl_put_right:Nn \l__egreg_tm_body_tl
         { \__egreg_tm_inner:nn { ##1 } { ####1 } }
       }
      % if ####1 = #1 end the row,
      % otherwise end the cell
      \tl_put_right:Nx \l__egreg_tm_body_tl
       {
        \int_compare:nTF { ####1 = #1 }
         { \exp_not:N \\ } % end row
         { & }             % end cell
       }
     }
   }
  % output the table
  \begin{array}{ @{} *{#1}{c} @{} }
  \l__egreg_tm_body_tl
  \end{array}
 }
```

Now `\uppertriangular{5}{a_{#1#2}}` prints the body of an upper triangular matrix and my linear algebra course can go on. Particularly because I can also define

```
\NewDocumentCommand{\diagonal}{mm}
 {
  \group_begin:
  \egreg_tm_main:nnn { #1 } { #2 } { = }
  \group_end:
 }
```

and get the diagonal matrices I need. I leave as an exercise the further extension of defining an optional template for filling the otherwise empty cells.

The integer comparison `\int_compare:n(TF)` accepts quite complex tests in its first argument, but here we're interested in what operators are allowed; they are '`= < > != <= >=`' and their meaning should be obvious. In a perfect world they would be '`= < > ≠ ≤ ≥`', but let's be patient.[9]

[9] This could easily be added for Unicode engines such as XeTeX or LuaTeX; it would be more complicated to support legacy 8-bit engines and those symbols in every encoding that has them. Code portability is much more important.

## 6 A templating puzzle

First let me present the code:[10]

```
\NewDocumentCommand{\automagic}{mm}
 {
  \begin{figure}
  \clist_map_inline:nn { #1 }
   {
    \cs_set:Nn \__oleinik_automagic_temp:n
     {
      \caption { #2 }
     }
    \begin{subfigure}[t]{0.33\textwidth}
    \includegraphics[
      width=\textwidth,
    ]{example-image-##1}
    \__oleinik_automagic_temp:n { #1 }
    \end{subfigure}
   }
  \end{figure}
 }
```

If one uses

```
\automagic{a,b,c}{Figure #1 from the set: ``##1''}
```

the result would show the three subcaptions

This is figure a from the set: "a,b,c"
This is figure b from the set: "a,b,c"
This is figure c from the set: "a,b,c"

The trick is that `\clist_map_inline:nn` does its own templating. The interested reader may enjoying solving the puzzle.

## 7 ISBN and ISSN

Every book has a number, called ISBN (International Standard Book Number) and each serial journal has an ISSN (International Standard Serial Number).

Originally, ISBN consisted of ten digits (with the final one being possibly `X`); later the code was extended to thirteen digits, but in a way that allowed old numbers to fit in the scheme by adding '978' at the beginning and recomputing the final digit, which is a checksum. For instance, *The TeXbook* originally had ISBN 0201134489, while more recent editions have 9780201134483. After the leading 978 there is a 0, which means the book has been published in an English-speaking country. The rest denotes the publisher and the issue number internal to the publisher. Books published in Brazil will start with 97865 or 97885; books published in Italy with either 97888 or 97912. The 979 prefix is a more recent extension for coping with a greater number of books.

On the contrary, the eight digit ISSN doesn't convey information about the place of publication; it's basically a seven digit number with a final checksum (which can be `X`). Why this strange possibility?

[10] `https://tex.stackexchange.com/q/410913/`

Because the checksum is computed modulo 11, so the remainder can be from 0 to 10 and `X` represents 10. This is also the case for old style ISBN, whereas the new codes compute the checksum modulo 10.

The algorithm for verifying correctness of an old ISBN is simple: the first digit is multiplied by 10, the second by 9 and so on up to the last digit (or `X`) which is multiplied by 1. All numbers are added and the result should be a multiple of 11. This method is guaranteed to catch errors due to transpositions of adjacent digits, but is not otherwise foolproof. For ISSN it is the same, but starting with multiplication by 8.

For a new style ISBN, the first digit is multiplied by 1, the second by 3, the third by 1 and so on, alternating 1 and 3. The sum of all numbers so obtained should be a multiple of 10 (no `X` needed).

We would like to have a macro for checking the validity of an ISBN or ISSN.[11] The package ean13isbn can be used for printing the bar code corresponding to a valid ISBN.

I provided a solution with TEX arithmetic a while ago. Now it's time to do it in expl3. The numbers may be presented with various hyphens, for separating the relevant information, but this is neither recommended nor required. Thus the macros first remove all hyphens and act on the string of numerals that result.

Since the methods for computing checksums are very similar, we can dispense with much code duplication. I define two user level macros, `\checkISBN` and `\checkISSN`. Both first remove the hyphens and then check the lengths. If this test passes, control is handed to a function that has as arguments the length and the modulo (11 for ISSN and old style ISBN, 10 for new style ISBN). The multipliers are kept in constant sequences defined beforehand.

This function will set a temporary sequence equal to the one corresponding to the length, then computes the checksum and the remainder of the division with the prescribed modulo. If the remainder is 0, the code is deemed valid.

An important feature we exploit is that in the first and third arguments to `\int_compare:nNnTF` any *integer denotation* is allowed, with full expansion; so we can use our friend `\int_step_function:nN` to extract the multiplier and the digit, insert `*` between them (for multiplication) and add a trailing `+`. The final digit is treated specially, because it may be `X`; in this case 10 is used.

```
\NewDocumentCommand{\checkISBN}{m}
 {
  \__egreg_check_normalize:Nn
   \l__egreg_check_str
   { #1 }
  % ISBN can have length 10 or 13
  \int_case:nnF { \str_count:N \l__egreg_check_str }
   {
    {10}{\__egreg_check:nn { 10 } { 11 }}
    {13}{\__egreg_check:nn { 13 } { 10 }}
   }
   {Invalid~(bad~length)}
 }
\NewDocumentCommand{\checkISSN}{m}
 {
  \__egreg_check_normalize:Nn
   \l__egreg_check_str
   { #1 }
  % ISSN must have length 8
  \int_compare:nNnTF
   { \str_count:N \l__egreg_check_str } = { 8 }
   { \__egreg_check:nn { 8 } { 11 } }
   {Invalid~(bad~length)}
 }

\str_new:N \l__egreg_check_str

\seq_const_from_clist:cn {c_egreg_check_8_seq}
 { 8,7,6,5,4,3,2,1 }
\seq_const_from_clist:cn {c_egreg_check_10_seq}
 { 10,9,8,7,6,5,4,3,2,1 }
\seq_const_from_clist:cn {c_egreg_check_13_seq}
 { 1,3,1,3,1,3,1,3,1,3,1,3,1 }

% remove hyphens
\cs_new_protected:Nn \__egreg_check_normalize:Nn
 {
  \str_set:Nn #1 { #2 }
  \str_replace_all:Nnn #1 { - } { }
 }

% the main macro
\cs_new_protected:Nn \__egreg_check:nn
 {% #1 = length, #2 = modulo
  % use the appropriate constant sequence
  \seq_set_eq:Nc
   \l__egreg_check_seq
   { c_egreg_check_#1_seq }
  % compute the checksum and check it
  \int_compare:nNnTF
   {
    \int_mod:nn
     { \__egreg_check_aux_i:n { #1 } }
     { #2 }
   }
   = { 0 }
  {Valid}
  {Invalid~(bad~checksum)}
 }
\cs_new:Nn \__egreg_check_aux_i:n
 {% do a loop from 1 to 7, 9 or 12
  \int_step_function:nN
   { #1-1 }
   \__egreg_check_aux_ii:n
  % and add the last digit
  \str_if_eq:eeTF
   { \str_item:Nn \l__egreg_check_str { #1 } }
   { X }
   { 10 }
   { \str_item:Nn \l__egreg_check_str { #1 } }
 }
```

[11] https://tex.stackexchange.com/q/39719/

```
% the auxiliary function extracts the items from
% the sequence (multiplier) and the string (digit)
\cs_new:Nn \__egreg_check_aux_ii:n
 {
  \seq_item:Nn \l__egreg_check_seq { #1 }
  *
  \str_item:Nn \l__egreg_check_str { #1 }
  +
 }
```

Check with the following test:

```
\checkISBN{12345}          % invalid
\checkISBN{111111111X}     % invalid
\checkISSN{1234-56789}     % invalid
\checkISSN{1234-567X}      % invalid
\checkISBN{0201134489}     % TeXbook
\checkISBN{978-0201134483} % TeXbook
\checkISSN{0896-3207}      % TUGboat
```

With the same idea one could devise a fully expandable macro that takes as input a string of digits, applies a sequence of weights and computes a check digit based on a modulo operation.

## 8 Catcode tables

Every TEX user is fond of category codes, particularly when they put sticks in the wheels.[12] How to print on the terminal and log file the current status of category codes?[13]

We need a macro that calls a loop; with legacy TEX engines, the table is limited to the range 0–255, but with Unicode engines we can go much further. Another issue is that characters in the range 0–31 and 127–255 may fail to print in the log file, so I'll adopt for them the usual `^^`⟨*char*⟩ or `^^`⟨*char*⟩⟨*char*⟩ convention. For instance, character 0 is represented by `^^@`, character 127 by `^^?`, but character 128 by `^^80`.

The macro can be called like `\catcodetable`, `\catcodetable[255]` or `\catcodetable[0-255]` all meaning the same thing: no optional argument implies `0-255`; a single number specifies the end point, starting from 0; two numbers separated by a hyphen specify start and end points.

I use an `\int_step_function:nnnN` loop, the auxiliary function prints the code point (in decimal), then a representation of the character, then its category code in verbose mode. The interesting bit here, besides the complex tests for `\int_compare:nTF`, is `\char_generate:nn`. This function takes as arguments two numeric expressions; the first one denotes the code point, the second one the category code to assign. Of course only some of these catcodes are meaningful: 9, 14 and 15 aren't; also 13 cannot (yet) be used with XeTEX. I use here 12, for safety.

[12] In Italian we say *mettere i bastoni fra le ruote* when somebody tries to impede our endeavor.

[13] `https://tex.stackexchange.com/q/60951/`

```
\NewDocumentCommand{\catcodetable}
 {
  >{\SplitArgument{1}{-}}O{0-255}
 }
 {
  \catcodetablerange#1
 }
\NewDocumentCommand{\catcodetablerange}{mm}
 {
  \IfNoValueTF{#2}
   {
    \egreg_cctab:nn { 0 } { #1 }
   }
   {
    \egreg_cctab:nn { #1 } { #2 }
   }
 }

\str_const:Nn \c_egreg_cctab_prefix_str { ^ ^ }

\cs_new_protected:Nn \egreg_cctab:nn
 {
  \int_step_function:nnnN
   { #1 } % start
   { 1 }  % step
   { #2 } % end
   \egreg_cctab_char:n
 }

\cs_new_protected:Nn \egreg_cctab_char:n
 {
  \iow_term:x
   {
    Code~\int_to_arabic:n { #1 }:~(
    \int_compare:nTF { 0 <= #1 < 32 }
     {
      \c_egreg_cctab_prefix_str
      \char_generate:nn { #1+64 } { 12 }
     }
     {
      \int_compare:nTF { #1 = 127 }
       {
        \c_egreg_cctab_prefix_str
        \char_generate:nn { #1-64 } { 12 }
       }
       {
        \int_compare:nTF { 128 <= #1 < 256 }
         {
          \c_egreg_cctab_prefix_str
          \int_to_hex:n { #1 }
         }
         {
          \char_generate:nn { #1 } { 12 }
         }
       }
     }
     )~\__egreg_cctab_catcode:n { #1 }
   }
 }
\cs_new:Nn \__egreg_cctab_catcode:n
 {
  \int_case:nn { \char_value_catcode:n { #1 } }
   {
    {0}{escape}
    {1}{begin~group}
    {2}{end~group}
    {3}{math~shift}
    {4}{alignment}
```

```
    {5}{end~of~line}
    {6}{parameter}
    {7}{superscript}
    {8}{subscript}
    {9}{ignored}
    {10}{space}
    {11}{letter}
    {12}{other~character}
    {13}{active~character}
    {14}{comment}
    {15}{ignored}
   }
 }
```

A selected part of the output from `\catcodetable` with 8-bit LaTeX:

```
Code 0: (^^@) ignored
Code 1: (^^A) active character
Code 2: (^^B) active character
[...]
Code 31: (^^_) active character
Code 32: ( ) space
Code 33: (!) other character
Code 34: (") other character
Code 35: (#) parameter
Code 36: ($) math shift
Code 37: (%) comment
Code 38: (&) alignment
Code 39: (') other character
[...]
Code 63: (?) other character
Code 64: (@) other character
Code 65: (A) letter
Code 66: (B) letter
[...]
Code 90: (Z) letter
Code 91: ([) other character
Code 92: (\) escape
Code 93: (]) other character
Code 94: (^) superscript
Code 95: (_) subscript
Code 96: (`) other character
Code 97: (a) letter
Code 98: (b) letter
Code 122: (z) letter
Code 123: ({) begin group
Code 124: (|) other character
Code 125: (}) end group
Code 126: (~) active character
Code 127: (^^?) ignored
Code 128: (^^80) active character
Code 129: (^^81) active character
[...]
```

Running `\catcodetable["10FFFF]` with XeLaTeX also works, and produces a 37 MiB[14] log file ending with

```
Code 1114109: (<U+10FFFD>) other character
Code 1114110: (<U+10FFFE>) other character
Code 1114111: (<U+10FFFF>) other character
 )
Here is how much of TeX's memory you used:
 9287 strings out of 492956
 183011 string characters out of 6133502
 204291 words of memory out of 5000000
 [...]
```

The `<U+10FFFF>` is an artifact of `less` on my system.

A small curiosity about the code for the string constant that prints the two carets when needed:

```
\str_const:Nn \c_egreg_cctab_prefix_str { ^ ^ }
```

There *must* be a space between the carets, otherwise the standard TeX convention would prevail and the string would end up containing character 32+64 = 96, that is, `. The (ignored) space in between the carets separates them and so we get our desired two carets in the output string.

Happy LaTeX3ing!

◇ Enrico Gregorio
Dipartimento di Informatica
Università di Verona
and
LaTeX Team
enrico.gregorio@univr.it

[14] "mebibyte"; 1 MiB = $2^{20}$ bytes.