Hints & Tricks

Interesting loops and iterations — second helping

Paweł Jackowski

Abstract

Where on earth does a programmer have to implement a loop construct himself? In TEX! TEX as a programming language is akin only to itself. Its interesting feature, rarely to be found among programming languages, is the lack of a built-in loop construct. However, thanks to TEX dealing perfectly well with recursive definitions and its ability to check conditions there are no obstacles to defining DIY loops. It has been done by Donald Knuth in plain TEX, extended by Alois Kabelschacht, Kees van der Laan, Marcin Woliński and many others and used by every practicing TEXie. This article sums up what every TEXie should know about loops. We will not shy away from dirty tricks which users need not know about.

Taking on `\loop`

Let's yet again review the traditional `plain`-ish loop definition ([1], p. 352):

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body \let\next\iterate
 \else \let\next\relax\fi \next}
\let\repeat=\fi
```

The definition is pretty readable and understandable thanks to the supporting macros suggestively named `\next` and `\body`. However, an unnecessary assignment is performed at every iteration. This assignment gives a meaning to the `\next` instruction as well as the `\body` instruction which, in principle, should not be used anywhere else. As the instructions are hidden from the user a name conflict can easily arise.

There are several well-known enhancements of this traditional definition which use `\expandafter` instead of the scratch `\next` macro. For example, from [2]:

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body\expandafter\iterate\fi}
```

or even simpler, from [6]:

```
\def\loop#1\repeat{%
 \def\iterate{%
  #1\expandafter\iterate\fi}%
 \iterate}
```

In the first case we are getting rid of the superfluous definition of `\next` and in the second also of the definition of `\body`. In yet another construction (for an extended description see [3]) the whole contents of the loop is executed outside of a conditional block `\if...\fi`:

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body\else\etareti\fi\iterate}
\def\etareti\fi\iterate{\fi}
```

A summary of these and other solutions may be found in [4].

A loop in a loop

The above constructs, though correct and elegant, do not allow loop nesting. In each of them the first operation remembers the content of the loop in an instruction. Embedding it would cause a conflict for the inner and outer loops.

Is there a way out? Yes. At the cost of slightly slowing the loop one may use a macro parameter instead of a definition. For example, instead of repeating the `\body` at every `\iterate`, we can set the repeated code fragment as an argument of the `\iterate` instruction. For convenience the `\long` prefix is used, to enable the use of `\par` within the loop. We also define the `\gobbleone` macro, which is called just before processing leaves the loop and gobbles the superfluous argument just after the `\fi` ending the conditional.

```
\long\def\loop#1\repeat{%
 \iterate\gobbleone{#1}}
\long\def\iterate\gobbleone#1{%
 #1\expandafter\iterate\fi
 \gobbleone{#1}}
\long\def\gobbleone#1{}
```

The `\gobbleone` definition plays a second role — it delimits the `\iterate` macro (i.e., is a *macro delimiter*). When the `\iterate` instruction is being

---

This is a translation of the article "Ciekawe pętle i iteracje na drugą nóżę", which first appeared in *Biuletyn GUST* nr 22 (2005), 3–6. Reprinted by permission. Translation by Jerzy Ludwichowski.

executed, the immediately following `\gobbleone` is swallowed as an unused fragment of the parameter. At the end of the loop `\iterate` is skipped, but `\gobbleone` swallows the loop content argument.

This loop might be used like the traditional form, the difference being that it can be nested, as shown in the following example:

```
\count100=9
\loop{\count101=65 % ASCII ‘A’
 \advance\count100 by-1
 \ifnum\count100>0
 \leavevmode\loop
  \char\count101 \the\count100
  \advance\count101 by1
  \ifnum\count101<73 \space
\repeat\par
}\repeat
```

The code produces something akin to a chess field. The row elements are typeset by the inner loop and the rows are produced by the outer loop.

```
A8 B8 C8 D8 E8 F8 G8 H8
A7 B7 C7 D7 E7 F7 G7 H7
A6 B6 C6 D6 E6 F6 G6 H6
A5 B5 C5 D5 E5 F5 G5 H5
A4 B4 C4 D4 E4 F4 G4 H4
A3 B3 C3 D3 E3 F3 G3 H3
A2 B2 C2 D2 E2 F2 G2 H2
A1 B1 C1 D1 E1 F1 G1 H1
```

One should note the use of grouping in the outer loop block:

```
\loop{...\loop...\repeat...}\repeat
```

This group affects only the scope of the argument reading. The content of the outer loop is not executed within the group. Thanks to this, the outer loop can use the assignments done in the inner loop. Grouping is necessary — without it TEX would cease reading the outer loop just after seeing the first `\repeat`.

Let it resolve

The capabilities of TEX do not end in incrementing and checking the counter value. Moreover, TEX iterations are not restricted to `\loop...\repeat` constructions. Often there is a need to execute some procedure for each token of a group, in a context where assignments cannot be used (e.g., when creating definitions with `\edef`, `\xdef`, inside `\write`-s, `\special`-s and `\mark`-s). Here it is worth citing the beautiful-in-its-simplicity macro `\fifo`, described in more detail in [3]:

```
\def\fifo#1{\ifx\ofif#1\ofif\fi
 \process#1\fifo}
\def\ofif#1\fifo{\fi}
```

In the example below `\fifo` is used to create a crib sheet of codes of some diacritics:

```
\def\process#1{(#1 -> \number‘#1)}
\immediate\message
 {\fifo áéó\ofif}
```

Counting of iterations is replaced here by executing the `\process` instruction for consecutive arguments. At the start of each iteration `\ifx` checks if the just-found argument is the `\ofif` token. The latter both delimits the token list and is a macro ending the condition executed after the last iteration.

Number games

No one needs convincing that expandable macros (without assignments) are more convenient. But how can assignments be avoided in loops operating on numbers? The most typical use of loops is repeating code some defined number of times. The previously shown `\loop...\repeat` constructs achieve this by iteratively incrementing or decrementing a counter, but this requires assignments.

The task is not hopeless, however. As the preceding example shows, the `\number` instruction expands “on the fly” any TEX representation of a number into its decimal form. In the basic version of TEX every arithmetic operation requires an assignment. To the rescue comes ε-TEX, which offers several convenient operations that allow dodging inconvenient assignments. The `\numexpr` instruction will serve as an example. It executes, in an expandable way, the basic operations on numbers (addition, subtraction, multiplication and division).

Let us use `\numexpr` to build a `\replicate` macro which repeats an arbitrary piece of code a given number of times. The first parameter is the number of repetitions, the second is the content of the loop.

```
\long\def\replicate#1#2{%
 \ifnum\numexpr#1>0
  #2\replicate{#1-1}{#2}\fi}
```

The loop starts with the check for the counter being positive, i.e., if the repetition should be executed. If so, then the contents of the loop, given as the second parameter, is executed and then a recursive call is being made to the `\replicate` procedure with the counter subtracted by 1 and the second parameter unchanged.

This construct suffers from two serious drawbacks. First, each repetition is executed within accumulating \ifnum...\fi blocks, which threatens catastrophe if a large number of iterations is required. Second, the length of the first parameter of the macro is increased by two at each turn of the loop, hence during the check of the counter value TEX must each time evaluate an ever longer expression of the form \numexpr100-1-1-1....

Therefore let us try to modify the \replicate macro so as to execute each repetition outside of the \ifnum...\fi condition and give the parameter representing the counter a more elegant form.

```
\long\def\replicate#1#2{%
 \ifnum\numexpr#1>0
  #2\expandafter\replicate\expandafter
  {\number\numexpr#1-1\expandafter}%
 \else
  \expandafter\gobbleone
 \fi{#2}}
```

Again we start by checking if the loop counter is positive, i.e., if the repetition should be executed. If so, the content of the loop (the second parameter) is processed, after which \expandafter in connection with \number\numexpr decrements the counter by one and enters the \replicate procedure with the new value of the counter. The second parameter to the \replicate procedure is passed on unchanged and immediately follows the \fi instruction ending the conditional. When the counter reaches 0 (or if we mischievously start the loop with the parameter being not greater than 0), \expandafter kills the remaining \fi after which the already described \gobbleone procedure swallows the superfluous parameter.

We use here the previously mentioned beneficial feature of the \number instruction which causes the macros following it to be expanded completely, i.e., until the decimal representation of the number is produced. During the expansion of \numexpr, \expandafter is executed which as if in passing (during the number expansion!) causes the loop condition block to disappear. TEX then "notices" that the expression cannot be expanded further and returns to the \replicate instruction. The latter is executed with the numerical argument in decimal representation and the second argument being the immutable loop content. This happens outside of the conditional block.

Here is an example of \replicate in a context in which the traditional TEX loop with assignment would fail. The \replicate macro is expandable and can be nested.

```
\immediate\message
 {\replicate{100+1}
  {I will be using eTeX%
    \replicate{3}{!} }}
```

Let's move on to a more complicated example. We will try to define a \fixed macro which puts the digit 0 in front of all other digits in such a way as to complement the number to a set length. For example,

```
\fixed{4}{12}
```

should expand to 0012. We begin by defining a helper macro to "measure" the length of a sequence.

```
\long\def\abacus#1{\addabacus#10}
\long\def\addabacus#1#2#3{%
 \ifx#3#1#2\else
  \expandafter\addabacus
  \expandafter#1\expandafter
  {\number\numexpr#2+1\expandafter}%
 \fi}
```

The \abacus macro (from Latin: a calculating tool) counts tokens appearing between a pair of two other tokens.

```
\count100=\abacus|Llanfairpwllgwyngyll%
gogerychwyrndrobwllllantysiliogogogoch|
\edef\numofletters{%
 \abacus\relax Antidisestablish%
  mentarianism\relax}
```

At each turn of the loop the macro tests if the upcoming token is the delimiting token of the measured sequence. If not, the macro in the already described manner increments the counter by 1 and moves on to the next iteration. If yes, it simply returns its counter which is the number of tokens between the freely chosen delimiters.

Now, we can use \replicate and \abacus to define a macro to pad the sequence with a chosen character to a given length.

```
\def\fixedprefix#1#2#3{%
 \expandafter\replicate\expandafter
 {\number
   \numexpr#1-\abacus\relax#2\relax}
 {#3}#2}
```

If we now write

```
\edef\test{\fixedprefix{4}{ab}{*}}
```

the \test instruction will be assigned the value of **ab. It remains to construct a specialized version of the \fixedprefix macro which will format numbers in such a way that they will have the specified number of digits by prepending with zeros if needed. Because the \fixed macro should operate on numbers, the first operation to be performed is

to expand the argument to a sequence of digits only. We know this trick already.

```
\def\fixed#1#2{%
 \expandafter\fixedzero\expandafter
  {\number\numexpr#1\expandafter}%
 \expandafter{\number\numexpr#2}}
\def\fixedzero#1#2{%
 \fixedprefix{#1}{#2}{0}}
```

We also know that TEX expands numbers tirelessly until the end. We also know that it has no problems with long sequences of tokens swallowed as arguments. The `\rnum` (*read number*) macro presented below exploits both TEXniques of iteration to read numbers in different notations, from binary to hexadecimal.

```
\def\rnum#1#2{\dornum{#1}{0}#2\relax}
\def\dornum#1#2#3{\ifx#3\relax#2\else
  \expandafter\dornum\expandafter
  {\number
    \numexpr#1\expandafter}\expandafter
  {\number
    \numexpr#1*#2+"#3\expandafter}%
 \fi}
```

We thus taught TEX to understand what, e.g., 1 000 000 000 000 means in binary notation:

```
\count100=\rnum{2}{1000000000000}
```

The reader may have noticed the character '`"`' which was used in the second-to-last line of the `\dornum` macro. As is known, for TEX this means: "read the digits as hexadecimal". Without it, TEX would not properly understand the digits `A` through `F`.

For dessert we propose the `\xnum` macro, which does the opposite of `\rnum`. It rewrites decimal numbers into other notations, from binary to hexadecimal and, of course does this in a fully expandable way. If the reader made it to this point, he should have no problems in understanding the following code. However, those who do not use ε-TEX deserve two explanations.

1. If during calculating `\numexpr` ε-TEX encounters the `\relax` token, it immediately stops reading the expression and `\relax` disappears without a trace.
2. If `\numexpr` contains non-integer division, the result will be rounded, unlike in TEX, where it will be truncated to the integer part.

More about ε-TEX constructions is in [5].

```
\def\hexdigit#1{%
 \expandafter\hexdigits
 \number\numexpr#1\relax\relax}
\def\hexdigits#1\relax
{\ifcase#1
  0\or1\or2\or3\or4\or5\or
  6\or7\or8\or9\or A\or
  B\or C\or D\or E\or F\fi}

\def\xnum#1#2{%
 \expandafter\doxnum\expandafter
 {\number
  \numexpr#1\expandafter}\expandafter
 {\number\numexpr#2}}
\def\doxnum#1#2{%
 \ifcase
  \ifnum#2<\numexpr#2/#1*#1\relax
   0 \else1 \fi
 \expandafter\doxnumdown\or
 \expandafter\doxnumup\fi
 {#1}{#2}}
\def\doxnumdown#1#2{%
 \ifnum#1>#2 \else
  \expandafter\doxnum\expandafter
  {\number#1\expandafter}\expandafter
  {\number\numexpr#2/#1-1\expandafter}\fi
 \hexdigit{#2-(#2/#1-1)*#1}}
\def\doxnumup#1#2{%
 \ifnum#1>#2 \else
  \expandafter\doxnum\expandafter
  {\number#1\expandafter}\expandafter
  {\number\numexpr#2/#1\expandafter}\fi
 \hexdigit{#2-#2/#1*#1}}
% test
\count100=\rnum{2}{1000000000}
\immediate\message
 {\xnum{16}{\count100}}
```

Bibliography

[1] Donald E. Knuth: *The TEXbook* (1990), Addison-Wesley.

[2] Alois Kabelschacht: `\expandafter` vs. `\let` and `\def` in conditionals and a generalization of plain's `\loop`. *TUGboat*, Volume 8 (1987), No. 2, 184–185.

[3] Kees van der Laan: FIFO and LIFO sing the BLUes. *Biuletyn* GUST, nr 4 (1992), 20–26.

[4] Marcin Woliński: O pewnych konstrukcjach warunkowych i iteracyjnych [On some conditional and iterative constructs]. *Biuletyn* GUST, nr 7 (1996), 5–9.

[5] Peter Breitenlohner: The ε-TEX Manual, Version 2, February 1998, 9.

[6] Victor Eijkhout: The bag of tricks. *TUGboat*, Volume 21 (2000), No. 1, 91.

◇ Paweł Jackowski
GUST
P dot Jackowski (at) gust dot org dot pl