`multiexpand`
Trigger multiple expansions in one expansion step*

Bruno Le Floch†‡

Released 2017/11/29

Contents

1 Two user commands

- For $n > 0$, expanding `\MultiExpand{`n`}\macro` twice gives the n-th expansion of `\macro`.
- For $n > 0$, expanding `\MultiExpandAfter{`n`}\macroA\macroB` twice expands `\macroB` n times before expanding `\macroA`.

Note that neither functions work for $n = 0$.

These can typically be combined as

*This file describes version v1.5, last revised 2017/11/29.

†E-mail: blflatex@gmail.com

‡I have gathered ideas from various posts in the `{TeX}` community at `http://tex.stackexchange.com`. Thanks to their authors.

```
\MultiExpand{7}%
\MultiExpandAfter{4}\a\MultiExpandAfter{7}\b%
\MultiExpandAfter{3}\c\d
```

which would expand `\d` 3 times, then `\c` 5 times (2 of the 7 times were used to expand `\MultiExpandAfter{3}`), then `\b` twice $(4-2)$, and finally `\a` five times $(7-2)$. Note that all this happens in precisely *two* steps of expansion.

In some cases, one needs to achieve the same effect in *one* step only. For this, we use the first expansion of `\MultiExpand`, which is `\romannumeral \multiexpand`, or of `\MultiExpandAfter`, which is `\romannumeral \multiexpandafter`. In detail, expanding `\romannumeral \multiexpand{`n`}` once expands the following token n times, and similarly for `\romannumeral \multiexpandafter{`n`}`.

These are especially useful when we want to expand several times a very specific token which is buried behind many others. For instance, expanding the following code once

```
\expandafter\macroA\expandafter\macroB
\romannumeral\multiexpandafter{4}\macroC\macroD
```

will expand `\macroD` 4 times before the three other macros.

Note: as we mentionned, this breaks for $n = 0$. But in this case, consider using `\expandafter\empty`, or a variant thereof.

2 Implementation

1 ⟨∗package⟩

We work inside a group, to change the catcode of `@`. So we will only do `\gdef`s. Note that this code can be read several times with no issue; no need to bother to check whether it was already read or not.

```
2 \begingroup
3 \catcode `\@=11
```

2.1 Common to the ε-TEX and non-ε-TEX cases

For the "lazy", who do not want to use `\romannumeral`, we provide `\MultiExpand` and `\MultiExpandAfter`, simple shorthands. A drawback is that they require two steps of expansion rather than only one.

```
4 \gdef \MultiExpand {\romannumeral \multiexpand }
5 \gdef \MultiExpandAfter {\romannumeral \multiexpandafter }
```

2.2 Without ε-TEX's \numexpr

No need for the usual \begingroup\expandafter\endgroup to prevent \numexpr from being set to \relax, because we are already in a group.

```
\expandafter \ifx \csname numexpr\endcsname \relax
```

A helper.

```
\long \gdef \multiexpand@gobble #1{}
```

The user commands \multiexpand and \multiexpandafter, to be used after \romannumeral. They only differ a little bit.

```
\gdef \multiexpand {\multiexpand@aux \multiexpand@ }
\gdef \multiexpandafter {\multiexpand@aux \multiexpand@after }
```

The user commands receives a number, and to accept various forms of numbers we hit it with \number. If it is non-positive, stop the \romannumeral expansion with 0 and a space. Otherwise, reverse the number, to make it easy to subtract 1.

```
\long \gdef \multiexpand@aux #1#2%
  {\expandafter \multiexpand@test \number #2;#1}
\gdef \multiexpand@test #1;#2%
  {%
    \ifnum #1>0
      \multiexpand@reverse #1{?\multiexpand@reverse@end }?;;#2%
    \fi
    0 %
  }
```

The macro \multiexpand@reverse puts characters from the number one by one (as #1) after the semicolon, to reverse the number. After the last digit, #1 is {?\multiexpand@reverse@end}. The question mark is removed by \multiexpand@gobble, and the reverse@end macro cleans up. In particular, one should not forget to close the conditional using #5, which is the trailing \fi. At this stage, #4 is the function that distinguishes \multiexpand from \multiexpand@after, and #3 is the reversed number.

```
\gdef \multiexpand@reverse #1#2;%
  {\multiexpand@gobble #1\multiexpand@reverse #2;#1}
\gdef \multiexpand@reverse@end #1;?#2#3;#4#50
  {#5\multiexpand@iterate #41#3;}
```

The macro \multiexpand@iterate applies a ⟨*function*⟩ a certain number of times to what follows in the input stream. It expects to receive ⟨*function*⟩ ⟨*nines*⟩ 1⟨*reversed number*⟩;. The argument ⟨*nines*⟩, made entirely of the digit 9, is used to compute carries when subtracting 1, and is initally empty.

As a concrete example, after `\multiexpand{302}` the successive calls to `\multiexpand@iterate` would go as follows.

```
\multiexpand@iterate \multiexpand@  1203;
\multiexpand@iterate \multiexpand@  1103;
\multiexpand@iterate \multiexpand@  1003;
\multiexpand@iterate \multiexpand@ 9 103;
\multiexpand@iterate \multiexpand@ 99 13;
\multiexpand@iterate \multiexpand@  1992;
\multiexpand@iterate \multiexpand@  1892;
\multiexpand@iterate \multiexpand@  1792;
```

Note in particular how carries are done in several steps. The details are left as an exercise to the reader. The most common case is when `#2` is empty and `#3` is a non-zero digit. Then `\number` is expanded, triggering `\ifcase` which shifts `#3` by one unit, and `#1` takes care of expanding the tokens are required by `\multiexpand` or `\multiexpandafter`. If `#3` is 0, then `\multiexpand@zero` is called, closing the conditional with `#1`, and iterating, this time with a non-empty ⟨*nines*⟩, which are the argument `#2` of a new call to `\multiexpand@iterate`. Those ⟨*nines*⟩ are put back into the number by `\multiexpand@iterate`, unless the next significant digit is also 0, in which case `\multiexpand@zero` is called again, until finding a non-zero digit; at each step, one more 9 is added to the ⟨*nines*⟩. If all digits are zero, we reach `;` this way, and end, after cleaning up.

```
\gdef \multiexpand@iterate #1#21#3%
  {%
    \ifx ;#3\multiexpand@end \fi
    \ifx 0#3\multiexpand@zero \fi
    \expandafter \multiexpand@iterate
    \expandafter #1%
    \number 1#2%
      \ifcase #3 \or 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or 8\fi
      #1%
  }
\gdef \multiexpand@zero #1#2\number 1#3\ifcase #4\fi #5%
  {#1\multiexpand@iterate #59#31}
\gdef \multiexpand@end #1#2\ifcase #3\fi #4{#10 }
```

Finally, the two different expansion commands.

```
\gdef \multiexpand@ #1;{#1\expandafter ;}
\gdef \multiexpand@after #1;{#1\expandafter ;\expandafter }
```

2.3 With ε-TEX

```
\else
```

With ε-TEX, everything is much easier, since the engine knows how to subtract 1.

The main looping macros expect their arguments as an integer followed by a semicolon. As long as the argument is at least 2, decrement it, and expand what follows. Once the argument is 1 (or less: the macros are not meant to handle that case), call `\multiexpand@end` to clean up and stop looping.

```
\gdef \multiexpand@ #1;%
  {%
    \ifnum #1<2 \multiexpand@end \fi
    \expandafter \multiexpand@
    \the \numexpr #1-1\expandafter ;%
  }
\gdef \multiexpand@after #1;%
  {%
    \ifnum #1<2 \multiexpand@end \fi
    \expandafter \multiexpand@after
    \the \numexpr #1-1\expandafter ;\expandafter
  }
```

The looping macros are used within an overarching `\romannumeral` expansion, which we end with a `0` and a space, as well as the appropriate `\expandafter`. Here, `#1` is `\fi` which needs to remain to close the conditional, `#2` is `\expandafter`, and there is a trailing `\expandafter` in the case of `\multiexpand@after`.

```
\gdef \multiexpand@end #1#2#3;{#10#2 }
```

Finally, user commands, used as `\romannumeral \multiexpand(after)`. Those evaluate their argument, and pass it to `\multiexpand@(after)`. The argument might contain `\par` tokens (who knows)

```
\long \gdef \multiexpand #1%
  {\expandafter \multiexpand@ \the \numexpr #1;}
\long \gdef \multiexpandafter #1%
  {\expandafter \multiexpand@after \the \numexpr #1;}
\fi
```

Close the group.

```
\endgroup
```

⟨/package⟩

Change History