Exploring \romannumeral and expansion

Joseph Wright

Abstract

The TeX \romannumeral primitive leads a double life. As well as its obvious use for making Roman numerals, it also offers a powerful method for controlling expansion. Here, I look at why this comes about, why we might want to use it, and give illustrative examples where only \romannumeral gives the results we want.

1 Expansion control

TeX is a macro expansion language, and so methods for manipulating exactly when tokens are expanded are a core part of its programmer toolbox. Experienced TeXers know that \expandafter will skip over *one* token and expand the next in the input stream, so for example

```
\def\foo{\baz}
\def\baz{a}
\expandafter\show\foo
```

gives

```
> \baz=macro:
->a.
\foo ->\baz
```

Another primitive that is used regularly to control expansion is \edef, which exhaustively expands everything in the input it is given. For example:

```
\def\foo{\baz}
\def\baz{a}
\edef\test{\foo}
\show\test
```

gives

```
> \test=macro:
->a.
```

The combination of \expandafter and \edef, along with the \noexpand primitive, can be used to carry out a wide range of reordering of TeX input. However, there are some provisos. The \expandafter primitive is itself expandable but only carries out *exactly one* expansion. Thus it cannot be used if we don't know exactly how many expansion might be needed.

On the other hand, \edef will completely expand input but is an assignment so cannot be used in an expansion context (for example, inside \csname, an assignment of a numerical register, *etc.*). Using \edef also gives us no (easy) way to stop an expansion part-way through some input. These cases need a different approach and take us away from primitives intended for expansion.

2 Enter \romannumeral

TeX's syntax for numbers (integers) allows an optional leading space (or spaces), optional leading sign (or signs), the integer itself and an optional trailing space. The number can be given literally (for example 1234), can be the result of expansion or can be an 'internal integer'. The latter is, for example, what using a count register or ` syntax results in: a 'complete' number where TeX will not look for any further digits.

How does this help with expansion? TeX needs a number (as it understands them) in several places, for example after \count (to select a register) and \number (to produce a literal number from various forms of input).

TeX also needs a number after \romannumeral: this primitive is of course meant for conversion of that number into a Roman numeral! However, in contrast to \number, which produces some output in all cases, \romannumeral yields *nothing at all* if the number it is given to work with is *negative.* Using \romannumeral for expansion is all about exploiting this fact in combination with TeX's definition of a number.

With the simple input

```
\romannumeral-1%
```

TeX will continue expanding after the 1 character to search for a continuation of the number or an optional space. It will expand tokens as it goes but will stop (without error) at the first non-expandable non-digit. Thus, an indirect redefinition (of \foo) like this:

```
\def\demo#1{%
  \begingroup
    \toks0=\expandafter
      {\romannumeral-1#1}%
    \showthe\toks0 %
  \endgroup
}
\def\foo{\baz}
\def\baz{\def\foo{}}
\demo{\foo}
```

will give as a result the "interior" definition of \foo (defined in \baz):

```
> \def \foo {}.
```

(where TeX has added spaces in the usual way to delimit tokens in the \show output). Contrast this with the result of trying to use \edef here, for example:

```
\def\demo#1{%
  \begingroup
    \edef\temp{#1}%
    \toks0=\expandafter
```

```
      {\meaning\temp}%
    \showthe\toks0 %
  \endgroup
}
\def\foo{\baz}
\def\baz{\def\foo{}}
\demo{\foo}
```

which results in:

```
! TeX capacity exceeded, sorry
  [input stack size=5000].
\baz ->\def \foo
                 {}
```

So, using `\romannumeral` for expansion looks useful but there's an issue: what if we supply additional digits? This probably won't be deliberate, but the input we want to expand might just start with some digits. The above will fail as such digits will be used as part of the number.

Happily, the syntax for an internal integer avoids this problem: TeX searches for an optional space but *not* for any more numerical input. The notation normally used to specify the internal integer is TeX's ` notation: `` `0 `` or `` `\q `` are commonly used but have no special meaning.

```
\def\demo#1{%
  \toks0=\expandafter
    {\romannumeral-`0#1}%
  \showthe\toks0 %
}
\def\foo{\baz}
\def\baz{123\def\foo{}456}
\demo{\foo}
```

gives the desired

```
> 123\def \foo {}456.
```

3 In use

This '`\romannumeral` trick' is commonly used where it's desirable to provide a macro that will give a result in a known number of expansions. In general, with a template like

```
\def\demo#1{%
  \romannumeral-`0%
    % ... expandable code here ...
```

we can be sure that `\demo` will expand in exactly two steps, provided of course that the code we've supplied doesn't stop the expansion (we'll deal with that below).

The one thing that *will* disappear from the input when using `\romannumeral` expansion is a leading space: remember that TeX is looking for an optional trailing space to the integer we've used to trigger expansion. Luckily, it's rare to be worried about retaining leading spaces in the contexts where we might want to use this approach.

In fact, we can exploit the fact that TeX is looking for a space: deliberately inserting one can be used to halt expansion at a known point, leaving potentially-expandable material untouched. That's handy if we want to stop once we have a 'result'. That naturally leads to the question of how we can arrange to produce a 'result' that consists of unexpandable tokens without ending up stopping expansion. It turns out to be easy enough, but is best shown by an example, as follows.

4 Two examples

To show some practical uses for the `\romannumeral` trick, I'm going to take a couple examples based on some code in the expl3 language (LaTeX3 team, 2015). To keep a focus on what we want to think about, these are somewhat simplified from the 'parent' versions, and in particular will work with TeX90: no ε-TeX primitives are used, at the cost of dropping a few features from the actual expl3 code.

The first example is a macro to pick arbitrary cases from a list of possible integer values. The `\ifcase` primitive is of course extremely fast but becomes highly inconvenient when the values involved are not close to 0 or are spread out. The approach we can take is as follows:

```
\catcode`\@=11 %
\long\def\@firstoftwo#1#2{#1}
\long\def\@secondoftwo#1#2{#2}
\long\def\intcase#1#2#3{%
  \romannumeral-`0%
    \intcase@loop{#1}#2{#1}{#3}\stop
}
\long\def\intcase@loop#1#2#3{%
  \ifnum#1=#2 %
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
    {\intcase@end{#3}}
    {\intcase@loop{#1}}%
}
\long\def\intcase@end#1#2\stop{\space#1}
```

Here, the idea is that we don't know how many times we will need to apply the `\ifnum` test, so if the case statement needs to be fully expanded to a result, using `\expandafter` won't be practical. On the other hand, by using `\romannumeral` we can be sure that exactly two expansions of `\intcase` will leave the result (and no other tokens). This behaviour is going to be useful in the second example.

TeX provides us with the primitives `\lowercase` and `\uppercase` to carry out case changing. These primitives are not expandable, which makes using them a bit tricky. It turns out to be possible to implement fully-expandable case changing even for complex cases. Here, I'll set up a (much) simplified version that only works with 'text' input and will ignore any spaces.

The `\LowerCase` user-level macro shown here sets the `\romannumeral` expansion then starts off a loop. Notice that we have an end macro too followed by an empty brace group: this is going to allow us to keep expansion going for as long as we want.

```
\def\LowerCase#1{%
  \romannumeral-`0%
    \lowercase@loop#1\lowercase@end{}%
}
```

The first internal macro simply looks for the end of the loop, and either picks the end-of-loop or case-change helper.

```
\def\lowercase@loop#1{%
  \ifx#1\lowercase@end
    \lowercase@end
  \else
    \lowercase@change#1%
  \fi
}
```

To perform the actual case change, we can use the `\intcase` macro we just saw. This can be forced to yield a result *before* storing the value, which is a benefit here in terms of performance (we end up with fewer tokens), but also comes into play if the result we are providing needs to be examined by some other code (which might also be forcing expansion!).

Notice that I've used another `\romannumeral` to avoid a long `\expandafter` chain in this forced expansion.

```
\def\lowercase@change#1\fi{%
  \fi
  \expandafter\lowercase@store
    \expandafter{%
      \romannumeral-`0%
      \intcase{`#1}
        { {`A}{a}{`B}{b}{`C}{c}{`D}{d}
          {`E}{e}{`F}{f}{`G}{g}{`H}{h}
          {`I}{i}{`J}{j}{`K}{k}{`L}{l}
          {`M}{m}{`N}{n}{`O}{o}{`P}{p}
          {`Q}{q}{`R}{r}{`S}{s}{`T}{t}
          {`U}{u}{`V}{v}{`W}{w}{`X}{x}
          {`Y}{y}{`Z}{z} }
        {#1}%
    }%
}
```

Storing the result of a case change is done by using the marker end-of-loop macro to add the processed token to the growing result: keeping the number of tokens down means TeX is doing less work. Finishing the loop on the other hand needs the outer `\romannumeral` to terminate, which is done by inserting a space then the final result.

```
\def\lowercase@store#1#2\lowercase@end#3{%
  \lowercase@loop#2\lowercase@end{#3#1}%
}
\def\lowercase@end#1\fi#2{\fi\space#2}
```

This code can then be expanded in exactly two steps to a result

```
\toks0=\expandafter\expandafter\expandafter
  {\LowerCase{abgT&HYRI$*Z}}%
\showthe\toks0 %
...
> abgt&hyri$*z.
```

or indeed we could once again use `\romannumeral`

```
\toks0=\expandafter
  {\romannumeral-`0%
    \LowerCase{abgT&HYRI$*Z}}%
\showthe\toks0 %
...
> abgt&hyri$*z.
```

5 Conclusions

Using `\romannumeral` can offer expansion control that is otherwise difficult or impossible in TeX. Particularly when creating expandable function-like macros, it is an invaluable tool in a TeX programmer's arsenal.

References

LaTeX3 team. "The expl3 programming language". http://ctan.org/pkg/l3kernel, 2015.

◇ Joseph Wright
Morning Star
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
joseph dot wright (at)
morningstar2.co.uk