第 1 章
Expanding TeX's \newif

第 1 节 Introduction

Like most of my colleagues, I use LaTeX to write papers, reports, notes, or what have you. In fact, I think all of the places that I regularly write supports some variable subset of LaTeX. Also like most of my colleagues, I'm not a TeXnician. I'm not proud to be ignorant in this regard, but there's only so many hours in a day, and the gains from properly learning a huge ecosystem like LaTeX seems minuscule compared to the initial buy-in cost.

Still, I was curious.

LaTeX and TeX, tomato tomato? Here's how I see it. If LaTeX is like C++20 — big, complex, confusing, full of cruft, but still very popular — then TeX is like C89 — small, simpler[1], confusing, a child of its time, and often neglected.

There's a certain pleasure in going far enough down the stack that the systems you are using becomes simple enough to reason about on a deep level. It's the feeling you might get sitting down one afternoon trying to write some assembly after a long week of debugging consistency errors in your sharded database across multiple kubernetes clusters[2]. No magic, no need to constantly search for other people who's had the same problems you're dealing with on StackOverflow. It's just you and the CPU, and likely the Intel Instruction Set Manual or something as big and scary. I wanted that, but with typesetting.

This was my romantic motivation to dig into TeX and try to see whether it really is rewarding to step back a few decades to avoid the complexity of newer and bigger typesetting systems. I bought the TeXbook, and read it from start to finish. Well, some paragraphs are marked with “dangerous bends”, signalling that the content covered or the background assumed for those paragraphs are more advanced. I read the single bends, but skipped the double bends, at least most of the time.

Somewhere in the book I found the definition of `\newif`, a macro that's used to define conditionals, which you can later query, and branch on. Booleans, in other words. I read it, and really didn't understand a single thing, and I figured that if I can manage to sit down and figure out what on earth this macro is doing and why, then I've had a good taste of what it's like digging down this low in the world of TeX.

This post is the result of that process.

（一） How Do I Write TeX?

This is not really as obvious as it might sound. After all, TeX produces a document, but when playing with macros we really want to see what forms expand to, which macros are defined, and so on. I have to say upfront that the method I used here probably wasn't the ideal, because I just started used `tex` (or sometimes `pdftex`, for the purposes of this post they seem to be exactly the same) , and started writing. The repl doesn't support `readline` bindings or arrow keys, or clicking to move the cursor, so if I wanted to add something in the middle of a line, I had to hold backspace all the way back to where I wanted to go and write out the rest of the expression. Sometimes I pasted back and forth from a text editor, which worked okay.

Here's exactly how I got started[3].

```
/h/martin$ tex
This is TeX, Version 3.141592653 (TeX Live 2021/Arch Linux) (preloaded format=tex)
**\relax % don't read input from a file

*\tracingall=1           % Give us lots of output
{vertical mode: \tracingstats}
{\tracingpages}
{\tracingoutput}
{\tracinglostchars}
{\tracingmacros}
{\tracingparagraphs}
{\tracingrestores}
{\showboxbreadth}
{\showboxdepth}
{the character =}
{horizontal mode: the character =}
{blank space  }

*\message{This will show somewhere}   % some sample message
{\message}
This will show somewhere          % here's the things you wrote above
{blank space  }

*\def\mymacro{from the macro}  % Make a new macro
{\def}
{blank space  }

*\message{\mymacro}      % \message will expand the macro
{\message}

\mymacro ->from the macro   % \mymacro is expanded to `from the macro`
from the macro          % ... and we get the fully expanded form out.
```

```
{blank space }

*
```

Input lines start with a $*$. It's very useful to set `\tracingall=1`, which makes TeX output a bunch of things some of which you care about. Note that I've changed up the formatting of the output throughout this post so that it's easier to see what's going on.

Another quick note: I didn't want to spend hours write an intro to TeX as well as whatever this is, so if you have never written a line of TeX or LaTeX, this might be difficult to follow. If you've written some LaTeX, and maybe defined your own simple macros, I think you'll be fine.

第 2 节 The Goal

This is the definition we'll unravel, copied verbatim from The TeXbook.

```
\outer\def\newif#1{\count@=\escapechar \escapechar=-1
 \expandafter\expandafter\expandafter
  \def\@if#1{true}{\let#1=\iftrue}%
 \expandafter\expandafter\expandafter
  \def\@if#1{false}{\let#1=\iffalse}%
 \@if#1{false}\escapechar=\count@} % the condition starts out false
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}} % `if` is required
```

Don't despair if this is nonsense: the whole point of this post is to explain what's going on, and to get some better idea of how real and (somewhat) involved TeX macros work.

第 3 节 How TeX Reads Tokens

To start on the right foot, let's make sure that we properly understand how TeX reads tokens. A token is the input "unit" that TeX reads when it reads a document. For instance if you were to write `Let $n=\numb$ be a number.` then this will be transformed into a queue of tokens from which we will read one at a time. Exactly how the tokens are split up is not crucial to understanding, but in this example it looks something like this:

```
tokens = ['L', 'e', 't', ' ', $, 'n', '=', \numb, $, ...]
```

Notice three things. First, a letter is a token in of itself and we do not have one "word" be a token Second, `$` is not the character `'$'`, but the special begin/end math mode token. If we were to write `\$` we would get the character token `'$'`. Third, the whole macro `\numb` is one single token. When you hear "token", think "input unit".

So how does TeX read the tokens? One mental model is like this:

```
while tokens is not empty
```

```
t <- pop(tokens)
if shouldexpand(t)
   exp <- expand(t)
   tokens.push(ex)
else
   process(t)
```

Some tokens, like the `\newif` token we will figure out in this post, expand, and the expansion is another list of tokens, some of which might be regular character tokens, and some of which might be other tokens that also expand. Therefore when we expand a token we will push the result back onto the front of the queue.

Note that when we expand a macro that takes arguments, like `\def\paren#1{(#1)}` the expansion of `\paren` will pop more tokens from the queue, and then push the tokens of the expanded form back onto the queue.

What does it mean to "process" a token? For a character, this basically means to write that character at the current position on the page[4]. For a macro definition like `\def\bob{123}` it means to make the definition and storing it somewhere in memory so that if you ever encounter a `\bob` token you know that it expands to the three tokens `1`,2,3.

(一) *A Short Example*

Let `\def\A{a}\def\B{\A b} \def\C{\B\B}` and the input token queue be `[\C]` . To make sure we understand how this works, let's manually expand this whole thing. The left column is the token queue, and the left side of the queue is the front, which is the place at which we will be working. The right column explains what we're about to do.

Tokens | Current action: ——|:———— – `[\C]` | take `\C` out of the front of the queue `[]` | `\C` expands to `\B\B`, which we push back `[\B\B]` | take the first `\B` out `[\B]` | `\B` expands to `\A b``[\A b \B]` | `\A` is taken out, expanded to `a` and pushed back `[a b \B]` | `a` is taken out and processed, because it doesn't expand. `[b \B]` | `b` is taken out and processed. `[\B]` | you get the idea... `[\A b]` | `[a b]` | `[b]` | `[]` |

The end result of this execution is that we have sent the tokens `a`, `b`, `a`, `b` to the processing part of TeX.

第 4 节 A Primer on Catcodes

We need to know one more thing about tokens, or rather how the characters of your input are split into them. Each character have a *category code*, or catcode for short. Catcodes decide how to group and split characters into a token. There is a character code for letters (11), a code for space (10), and one for math shift (3) (there are also others). This way TeX knows that in the input `let $` consists of three characters, one space, and one "math shift". This is also how TeX figures out when the name of a macro ends and new tokens begin, as in `\hey3`: here we have one token with catcode 0 (the escape character \), three of catcode 11, and one of catcode

12（“others”，which include numbers). The name of a macro is only letters, so this way TeX knows that `\hey` is a macro and `3` is just the next token in the queue.

But catcodes can be changed. Why is this useful? Well, if we would like to make some macros that another user wouldn't accidentally redefine we have it include a character that, by default, isn't allowed to be in its name, like `@`. The catcode of `@` is 12, and so the input `\h@` will be read as two tokens `\h` and `'@'`. However, if we change the catcode of `@` to 11 it's as if `@` is just a regular letter, and `\h@` will be read as a single token `\h@`.

This is how we change the catcode of `@` to 11 and then back to 12:

```
*\catcode`\@=11  % Category 11 consists of regular letters
*\catcode`\@=12  % Category 12 consists of "other characters"
```

第 5 节 Some Not So Bad Macros

We need to know about a few other macros that `\newif` uses internally. Most of these are pretty straight forward.

(一) \string

Takes an argument and replaces it by the non-expanded token list. `\string\foo` expands to the four tokens `\ f o o`, no matter what the macro `\foo` would expand to. A crucial detail which we will come back to is that the tokens `\string` produces will get catcode 12 (unless it's a space).

(二) \escapechar

The character which is used when a control sequence is outputted as text. Normally set to `\`. If this is set to for instance `@`, then `\string\foo` would expand to the four tokens `@ f o o` instead.

(三) \uccode

Short for uppercase code. This allows one to set the uppercase character code for another letter. Usually this would be `\uccodex=X \uccodeX=X` and so on, but this, like most things in TeX, can be changed, and changes, like most things in TeX, are local to the current group.

(四) \csname and \endcsname

Read and expand everything up until the matching `\endcsname`. The expansion result should be a list of character tokens, and this list will be made into a single control sequence token. If this is currently not defined it will be defined to `\relax`.

For instance `\csname hello\endcsname` will expand to the single token `\hello` and make the macro `\hello` expand to `\relax`. More interestingly, `\def\inner{hello}\csname\inner\endcsname` will do the same: Here the `inner` macro expands to the list of tokens `h e l l o`, and the `csname` pair of macros expand this macro, effectively replacing it with `\csname hello\endcsname`.

(五) \gdef

Normally definitions made with `\def` are local to your scope, just like in most programming languages. However, sometimes we want to define global macros, and `gdef` does exactly this. When a macro is defined with `\gdef` it is as if it was defined in the top level scope.

```
{
  \def\inner{hello}
  \inner  % expands to h e l l o
}
\inner  % this doesn't work, because \inner is no longer defined

{
  \gdef\inner{hello}
  \inner  % expands to h e l l o
}
\inner % also expands to h e l l o
```

(六) \outer

This is a safety measure that you put before a `\def` which ensures that this macro is not allowed to be an argument, in the parameter text, or in the replacement text of another macro.

第 6 节 The `\expandafter` Macro

Now that we've seen a few simple macros we turn to one that is slightly less simple. The `\expandafter` macro first reads the very next token in the queue without expanding it. Then, it'll read *and expand* the next token after that. Last, it will put the first token back in front, without expanding it. Here's a small example of how it runs:

```
*\def\first{first}
*\def\second{second}
*\expandafter\first\second
{\expandafter}

\second ->SECOND

\first ->FIRST
{the letter F}
*
```

Here the output shows that `\second` is expanded before `\first`, and that the first token that we process is `f`. Note that the second form is only *expanded* and not actually processed, so the following does **not** work:

```
*\expandafter\first\def\first{another first!}
```

The second term, the `\def` will be expanded, but it will not “run”, so when `\expandafter` later expands `\first` it will still have the same value as before, for instance not to be defined.

Due to how TeX expansion rules work, a macro doesn't have to have all of it's arguments in place when you use it; currying[5] is in a sense possible. We can use `\expandafter` to use this fact if the first token expands to a curried macro, and the first token in the *expansion* of the second token is the argument we want to give to the curried form.

Here's an example. Say we have a macro `\twoarray` that takes two things and wraps them in square brackets divided by a comma, as well as a macro `\tuple` that expands to two tokens `4` and `5`. If we want to have `\twoarray` wrap the two tokens from `\tuple`, it doesn't work out of the box:

```
*\def\twoarray#1#2{[ #1 , #2 ]}
*\def\tuple{4 5}
*\twoarray\tuple X  % X is just a placeholder for whatever's next; we don't want it.
[ 4 5 , X ]
% This does not work because `\twoarray` will read two tokens, `\tuple` and `X`

*\expandafter\twoarray\tuple X
[ 4 , 5 ] X
% This does work because `\tuple` is expanded before `\twoarray`, and so the token
% queue when we process `\twoarray` is `4 5 X`
```

(一) *Chaining*

So what happens when we chain multiple `\expandafter`s together? Let's work it out with some notation: dashes under a line means `\expandafter` is skipping that line, and it's expanding the token above the hat `^`. Primed `a'` letters means expanded.

```
*\expandafter a  b  c  d ...
%             -  ^
% token list: a  b' c  d
```

With two `\expandafter`s this becomes

```
*\expandafter \expandafter a  b  c  d ...
%             ------------ ^
% token list:  \expandafter a' b  c  d
*\expandafter  a' b  c  d ...
%              -  ^
% token list:  a' b' c  d
```

It undid itself! The expansion order was `a` and then `b`. Let's try three expands in a row. Now we're getting somewhere, because when expanding the second token that `\expandafter` finds, we might end up reading *additional* tokens, *if* that token takes arguments. In this case this token is `\expandafter`, which does indeed take two arguments!

```
*\expandafter \expandafter \expandafter a  b  c  d ...
%        ------------   ^^^
%                 [eat 2 arguments]
*        \expandafter    a  b'     c  d ...
% This is just the first example again.
% token list:  a  b'' c  d ...
```

and we're again back to having the expansion order of `a` and `b` flipped. Despite this though, they are not identical, because `expandafter` does not expand a form until it only expands to itself, but only once. We can think of regular expansion as taking out the next token in the queue and if it is expandable we push back the expansion onto the queue.

Let's get concrete. As a warm up, here is the easy case where the two forms *are* identical, namely when expanding once is fully expanded. The list of `\A ->a` beneath each input line is the evaluation sequence such that the macro `\A` expands to the token `a`.

```
*\def\A{a}\def\B{b}\def\C{c}

*\A\B\C
\A ->a  \B ->b  \C ->c
*\expandafter\A\B\C
\B ->b  \A ->a  \C ->c
*\expandafter\expandafter\A\B\C
\A ->a  \B ->b  \C ->c
*\expandafter\expandafter\expandafter\A\B\C
\B ->b  \A ->a  \C ->c
```

Note that just like we said above, the first and third lines are the same, and the second and fourth are the same.

Next we make it slightly more interesting by expanding macros which body is another macro:

```
*\def\AA{\A}\def\BB{\B}\def\CC{\C}

*\AA\BB\CC
\AA ->\A  \A ->a    \BB ->\B   \B ->b  \CC ->\C  \C ->c
*\expandafter\AA\BB\CC
\BB ->\B  \AA ->\A  \A ->a     \B ->b  \CC ->\C  \C ->c
```

```
*\expandafter\expandafter\AA\BB\CC
\AA ->\A  \BB ->\B  \A ->a    \B ->b  \CC ->\C  \C ->c
*\expandafter\expandafter\expandafter\AA\BB\CC
\BB ->\B  \B ->b    \AA ->\A   \A ->a  \CC ->\C  \C ->c
```

The four lines have all distinct orders on which macros are expanded when, in contrast with the last example. With four `expandafter`s we are back to as if we had none.

What if we had `\AAA` and friends?

`\meaning\noexpand\foo`

第 7 节 Start Actually Expanding `\newif`

If you've made it this far, good job! I realize this is a fair amount of prerequisites before getting to the point of the post.

Here's the definition of `\newif` again, but formatted a little differently:

```
\outer\def\newif#1{
  \count@=\escapechar
  \escapechar=-1
  \expandafter\expandafter\expandafter \def\@if#1{true}{\let#1=\iftrue}%
  \expandafter\expandafter\expandafter \def\@if#1{false}{\let#1=\iffalse}%
  \@if#1{false} % the condition starts out false
  \escapechar=\count@
}
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
{
  \uccode`1=`i
  \uccode`2=`f
  \uppercase{\gdef\if@12{}}
} % `if` is required
```

Let's do this in parts, starting with the bottom group, then the middle `\def`, and then move on to the actual `\newif`. Note that only the first form is the actual body of `\newif` and that the bottom group and the `\def` in the middle is just part of the one-time setup. We'll start with the bottom group.

（一） The Bottom Group

```
{
  \uccode`1=`i
  \uccode`2=`f
  \uppercase{\gdef\if@12{}}
} % `if` is required
```

Recall from before that the `\uccode` macro sets the character code of the uppercase version of a character, so we can for instance change the uppercase of `g` to be `H` by writing `\uccodeg=H`. In our snippet we are setting the uppercase version of the numbers `1` and `2` to be `i` and `f`. Yes really. Also recall that the change is local to the current group, so this change will be undone after the third macro.

So we've changed the uppercase of `1` and `2`, and next we're uppercasing a `gdef` which name is `i f@12`.

Let's make this slightly easier by only having one character we uppercase

```
*{\uccode`1=`M \uppercase{\gdef\bob1{bob}}}
*\bob
\bob M->BOB
```

Notice that the name of the macro is just `\bob`, not `\bob1` or `\bobM`.

(1) A note about more advanced parameter texts

TeX allows us to ensure that there are other tokens in the argument list of a macro expansion, or that the arguments are delimited by certain tokens. For instance consider the following:

```
*\def\commasep#1,#2{(#1, #2)}
*\message{\commasep 1 2 3 , 9 8 7}
(1 2 3 ,9) 8 7
```

We see that the first argument was not in fact just the first token, but all tokens up until we hit , which we had after the `#1` in the parameter text. The last argument however, was just the next token.

We can also do this:

```
*\def\mfirst m#1{(#1)}
*\message{\mfirst a a}
! Use of \mfirst doesn't match its definition.
<*> \mfirst a
         a
*\message{\mfirst m a}
(a)
```

Here we've said that we need an `m` before we get the next token as the first argument to the macro. If the next token is not an `m`, like in the first attempt, we error. It is basically a very simple version of pattern matching.

(2) Back to Bob

In our definition of `\bob` we have ensured that the parameter text should end with an uppercase `1`, which was `M`. There is a problem though:

```
*\bob M
! Use of \bob doesn't match its definition.
<*> \bob M

?
```

The reason this doesn't work is that while the uppercase of `1` is temporarily set to `M` and the macro really does expect to be called as `\bob M`, the `M` we send in now has the wrong character code: it's a letter and not a number. We can temporarily change this in a group, and it will work.

```
*{\catcode`M=12 \bob M}
{begin-group character {}
{entering simple group (level 1)}
{\catcode}
{changing \catcode77=11}
{into \catcode77=12}

\bob M->BOB
{the letter B}
{end-group character }}
{restoring \catcode77=11}
{leaving simple group (level 1)}
{blank space  }

*
```

Now we are ready to understand the current snippet

```
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}} % `if` is required
```

This will define a macro `\if@` that ensures that the first two tokens after it is `i` and `f` with category code `12`. Also note that it will expand to nothing, but it will eat the matched tokens in the parameter list. In other words:

```
*\def\eat h{H} \message{\eat hello}
Hello
```

The `h` is eaten and replaced with the body of the macro, `H`, and the rest of the tokens `ello` are just characters so nothing is done to them, and the result is `Hello`.

To summarize, we've now globally defined a macro `if@` which ensures that when applied the next two tokens in the token list will be two tokens with catcode 12 that is `i` and `f`, and these tokens will be taken out of the token list.

(二) *The Middle* `\def`

Moving on to this part:

```
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
```

Let's peel the onion. We've got a `csname/endcsname` pair, so the output of the function will be a control sequence name, which will, unless already defined, be defined to expand to `\relax`. The name will be the result of `\expandafter\if@\string#1#2`; the arguments passed to `\@if` (the `def` we're looking at) will thus be sent to `\if@`, but the first argument will be eaten by `\string` first. We just learned that the only thing that `\if@` does is to ensure that the first two tokens given are `i f` of catcode 12. And it just so happen that the tokens that we get from expanding `\string` are exactly of catcode 12!

Let's try to expand `\@if{ifeven}{true}`:

```
\@if{ifeven}{true}
\csname \expandafter\if@\string{i f e v e n}{t r u e}\endcsname
\csname \if@ i f e v e n {t r u e}\endcsname
\csname e v e n {t r u e}\endcsname
\csname e v e n t r u e\endcsname  % csname doesn't care about grouping
eventrue
```

The result is a single control sequence token with the name `eventrue`. That's it! As long as the `\string` expansion of the first argument starts with `i f` we will get a control sequence token that is the concatenation of the two arguments.

(三) *The First* `\def`

Phew, back at the top. Here it is, once more:

```
\outer\def\newif#1{
  \count@=\escapechar
  \escapechar=-1
  \expandafter\expandafter\expandafter \def\@if#1{true}{\let#1=\iftrue}%
  \expandafter\expandafter\expandafter \def\@if#1{false}{\let#1=\iffalse}%
  \@if#1{false} % the condition starts out false
  \escapechar=\count@
}
```

We're almost there; it's just a matter of piecing together some of the parts that we've already unravelled. First we can note that we are temporarily setting `\escapechar` to be `-1` and then restoring it at the end. There are two questions we can answer here: (1) why do we set it, and (2) why can't we group it instead?

1. We want the argument to `\newif` to be a control sequence, like `\newif\ifred`, and we also need to check that the given control sequence starts with `if`, which we do in `\if@` through the `\string` macro. If naively applied, `\string\ifred` would expand to `\ i f r e d`, but we need it to be `i f r e d`. By setting `\escapechar=-1` we make `\string` output nothing for `\`, and we are good.
2. Had we used grouping the `\defs` we have inside would be local to the group and effectively destroyed by the time we are done expanding `\newif`. If we were to use `\gdef` then all defined macros with `\newif` would have to be global. This way we can have the user define `\newifs` that are local to their groups.

That only leaves three lines in the macro body, and two of them are of the same form. From earlier we remember that three `\expandafter` would expand the second token in the token list twice. Let's assume `#1 = \ifred`. With the total form

```
\expandafter\expandafter\expandafter \def \@if \ifred {true} {\let \ifred = \iftrue}
```

we would first expand `\@if`, which will eat two tokens, `#1` and `{true}` and be replaced with the body of the macro, as seen above. Then we need a second expansion to expand the `csname` pair, and this will expand to the control sequence token `redtrue`. This would be put back in the token queue,

```
\expandafter \def \csname \expandafter\if@\string\ifred{true}\endcsname{\let \ifred
= \iftrue}
\def \redtrue{\let \ifred = \iftrue}
```

and at the end we have a familiar form. The same happens with the `false` variant. The next line is then ran:

```
\@if\ifred{false} % expand:
\csname \expandafter\if@\string\ifred{true}\endcsname  % eval the csname pair
\redfalse  % we just defined this macro
\let\ifred=\iffalse  % run this
```

At last, we restore `\escapechar` to whatever it was initially.

第 8 节 In Conclusion

Taking it all together, running `\newif\ifred` expands to this:

```
% In the preamble we have the forms
```

```
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}} % `if` is required

% The user writes
\newif\ifred
% .. which expands to
\count@=\escapechar
\escapechar=-1
\expandafter\expandafter\expandafter \def\@if\ifred{true}{\let\ifred=\iftrue}
\expandafter\expandafter\expandafter \def\@if\ifred{false}{\let\ifred=\iffalse}
\@if\ifred{false}
\escapechar=\count@
% ... which is basically the same as
\def\redtrue{\let\ifred=\iftrue}
\def\redfalse{\let\ifred=\iffalse}
\redfalse
```

and that's it! So hey, we had to peel a few onions[6], but in the end we managed to unravel the mystery and really understand what's going on in `\newif`; it turns out it's quite a lot, though the main functionality seems that we don't have to write these three lines every time we want to define a new conditional, but that only one suffices.

If you want to know more "real" definition and edge cases, check out this site; I went back and forth on that and in the TeXbook when writing this post, and having a searchable index of basically the entire language is, well, indispensable. Of course, if you don't know much about TeX from before I can only assume that the reference will be hard to dig into.

Notes, comments, questions, and tomatoes can be sent to my public inbox.

Hope you learned something, and thanks for reading.

第 9 节 Footnotes

1. I couldn't call C89 or TeX simple in good faith.
2. I don't know what I'm talking about here; can you tell?
3. btw I use arch
4. This isn't really how it works, but for the purposes of this post we might as well pretend it is.
5. This example is more close to destructuring, but I didn't want to get in the weeds of constructing an example that looked more like currying. Here's a sketch: you can have a macro in the body of another macro `\func #1 x y` such that `#1` expands to another macro. If we `\expandafter` the `#1` here we might get something like `\func u v w x y` and so we've effectively constructed a function `f(g) = h(g(), x, y)`.
6. Something something crying when peeling an onion.