low level TEX

macros

Contents

1 Preamble

This chapter overlaps with other chapters but brings together some extensions to the macro definition and expansion parts. As these mechanisms were stepwise extended, the other chapters describe intermediate steps in the development.

Now, in spite of the extensions discussed here the main ides is still that we have TEX act like before. We keep the charm of the macro language but these additions make for easier definitions, but (at least initially) none that could not be done before using more code.

2 Definitions

A macro definition normally looks like like this:[1]

```
\def\macro#1#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Such a macro can be used as:

```
\macro {1}{2}
\macro {1} {2}  middle space gobbled
\macro 1 {2}    middle space gobbled
\macro {1} 2    middle space gobbled
\macro 1 2      middle space gobbled
```

We show the result with some comments about how spaces are handled:

|1|2|

[1] The `\dontleavehmode` command make the examples stay on one line.

12 middle space gobbled
12 middle space gobbled
12 middle space gobbled
12 middle space gobbled

A definition with delimited parameters looks like this:

```
\def\macro[#1]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\hss}}
```

When we use this we get:

```
\macro [1]
\macro [ 1]    leading space kept
\macro [1 ]    trailing space kept
\macro [ 1 ]   both spaces kept
```

Again, watch the handling of spaces:

1
1 leading space kept
1 trailing space kept
1 both spaces kept

Just for the record we show a combination:

```
\def\macro[#1]#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this:

```
\macro [1]{2}
\macro [1] {2}
\macro [1] 2
```

we can again see the spaces go away:

12
12
12

A definition with two separately delimited parameters is given next:

```
\def\macro[#1#2]%
```

```
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

When used:

```
\macro [12]
\macro [ 12]     leading space gobbled
\macro [12 ]     trailing space kept
\macro [ 12 ]    leading space gobbled, trailing space kept
\macro [1 2]     middle space kept
\macro [ 1 2 ]   leading space gobbled, middle and trailing space kept
```

We get ourselves:

12	
12	leading space gobbled
12	trailing space kept
12	leading space gobbled, trailing space kept
1 2	middle space kept
1 2	leading space gobbled, middle and trailing space kept

These examples demonstrate that the engine does some magic with spaces before (and therefore also between multiple) parameters.

We will now go a bit beyond what traditional TEX engines do and enter the domain of LuaMetaTEX specific parameter specifiers. We start with one that deals with this hard coded space behavior:

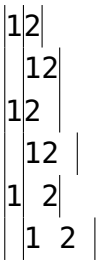
```
\def\macro[#^#^]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The #^ specifier will count the parameter, so here we expect again two arguments but the space is kept when parsing for them.

```
\macro [12]
\macro [ 12]
\macro [12 ]
\macro [ 12 ]
\macro [1 2]
\macro [ 1 2 ]
```

Now keep in mind that we could deal well with all kind of parameter handling in ConTEXt for decades, so this is not really something we missed, but it complements the to be discussed other ones and it makes sense to have that level of control. Also, availability

triggers usage. Nevertheless, some day the #^ specifier will come in handy.



We now come back to an earlier example:

```
\def\macro[#1]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

When we use this we see that the braces in the second call are removed:

```
\macro [1]
\macro [{1}]
```

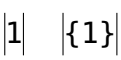
This can be prohibited by the #+ specifier, as in:

```
\def\macro[#+]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

As we see, the braces are kept:

```
\macro [1]
\macro [{1}]
```

Again, we could easily get around that (for sure intended) side effect but it just makes nicer code when we have a feature like this.



Sometimes you want to grab an argument but are not interested in the results. For this we have two specifiers: one that just ignores the argument, and another one that keeps counting but discards it, i.e. the related parameter is empty.

```
\def\macro[#1][#0][#3][#-][#4]%
  {\dontleavehmode\hbox spread 1em
     {\vl\type{#1}\vl\type{#2}\vl\type{#3}\vl\type{#4}\vl\hss}}
```

The second argument is empty and the fourth argument is simply ignored which is why we need #4 for the fifth entry.

```
\macro [1][2][3][4][5]
```

Here is proof that it works:

|1|3|5|

The reasoning behind dropping arguments is that for some cases we get around the nine argument limitation, but more important is that we don't construct token lists that are not used, which is more memory (and maybe even cpu cache) friendly.

Spaces are always kind of special in TEX, so it will be no surprise that we have another specifier that relates to spaces.

```
\def\macro[#1]#*[#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

This permits usage like the following:

```
\macro [1][2]
\macro [1] [2]
```

|1|2| |1|2|

Without the optional 'grab spaces' specifier the second line would possibly throw an error. This because TEX then tries to match][so the] [in the input is simply added to the first argument and the next occurrence of][will be used. That one can be someplace further in your source and if not TEX complains about a premature end of file. But, with the #* option it works out okay (unless of course you don't have that second argument [2].

Now, you might wonder if there is a way to deal with that second delimited argument being optional and of course that can be programmed quite well in traditional macro code. In fact, ConTEXt does that a lot because it is set up as a parameter driven system with optional arguments. That subsystem has been optimized to the max over years and it works quite well and performance wise there is very little to gain. However, as soon as you enable tracing you end up in an avalanche of expansions and that is no fun.

This time the solution is not in some special specifier but in the way a macro gets defined.

```
\tolerant\def\macro[#1]#*[#2]%
```

```
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The magic `\tolerant` prefix with delimited arguments and just quits when there is no match. So, this is acceptable:

```
\macro [1][2]
\macro [1] [2]
\macro [1]
\macro
```

|1|2| |1|2| |1| |

We can check how many arguments have been processed with a dedicated conditional:

```
\tolerant\def\macro[#1]#*[#2]%
  {\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

We use this test:

```
\macro [1][2] \macro [1] [2] \macro [1] \macro
```

The result is: 2: |1|2| 2: |1|2| 1: |1|0: | which is what we expect because we flush inline and there is no change of mode. When the following definition is used in display mode, the leading `n=` can for instance start a new paragraph and when code in `\everypar` you can loose the right number when macros get expanded before the `n` gets injected.

```
\tolerant\def\macro[#1]#*[#2]%
  {n=\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

In addition to the `\ifarguments` test primitive there is also a related internal counter `\lastarguments` set that you can consult, so the `\ifarguments` is actually just a shortcut for `\ifcase \lastarguments`.

We now continue with the argument specifiers and the next two relate to this optional grabbing. Consider the next definition:

```
\tolerant\def\macro#1#*#2%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this test:

```
\macro {1} {2}
\macro {1}
\macro
```

We get:

|1|2| |1|\macro|

This is okay because the last \macro is a valid (single token) argument. But, we can make the braces mandate:

```
\tolerant\def\macro#=#*#=%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Here the #= forces a check for braces, so:

```
\macro {1} {2}
\macro {1}
\macro
```

gives this:

|1|2| |1| |

However, we do loose these braces and sometimes you don't want that. Of course when you pass the results downstream to another macro you can always add them, but it was cheap to add a related specifier:

```
\tolerant\def\macro#_#*#_%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Again, the magic \tolerant prefix works will quit scanning when there is no match. So:

```
\macro {1} {2}
\macro {1}
\macro
```

leads to:

|{1}|{2}| |{1}| |

When you're tolerant it can be that you still want to pick up some argument later on. This is why we have a continuation option.

```
\tolerant\def\foo       [#1]#*[#2]#:#3{!#1!#2!#3!}
\tolerant\def\oof[#1]#*[#2]#:(#3)#:#4{!#1!#2!#3!#4!}
\tolerant\def\ofo       [#1]#:(#2)#:#3{!#1!#2!#3!}
```

Hopefully the next example demonstrates how it works:

```
\foo{3} \foo[1]{3} \foo[1][2]{3}
\oof{4} \oof[1]{4} \oof[1][2]{4}
\oof[1][2](3){4} \oof[1](3){4} \oof(3){4}
\ofo{3} \ofo[1]{3}
\ofo[1](2){3} \ofo(2){3}
```

As you can see we can have multiple continuations using the `#:` directive:

```
!!!3! !1!!3! !1!2!3!
!!!!4! !1!!!4! !1!2!!4!
!1!2!3!4! !1!!3!4! !!!3!4!
!!!3! !1!!3!
!1!2!3! !!2!3!
```

The last specifier doesn't work well with the `\ifarguments` state because we no longer know what arguments were skipped. This is why we have another test for arguments. A zero value means that the next token is not a parameter reference, a value of one means that a parameter has been set and a value of two signals an empty parameter. So, it reports the state of the given parameter as a kind if `\ifcase`.

```
\def\foo#1#2{ [\ifparameter#1\or(ONE)\fi\ifparameter#2\or(TWO)\fi] }
```

Of course the test has to be followed by a valid parameter specifier:

```
\foo{1}{2} \foo{1}{} \foo{}{2} \foo{}{}
```

The previous code gives this:

[(ONE)(TWO)] [(ONE)] [(TWO)] []

A combination check `\ifparameters`, again a case, matches the first parameter that has a value set.

We could add plenty of specifiers but we need to keep in ind that we're not talking of an expression scanner. We need to keep performance in mind, so nesting and backtracking are no option. We also have a limited set of useable single characters, but here's one that uses a symbol that we had left:

```
\def\startfoo[#/]#/\stopfoo{ [#1](#2) }
```

The slash directive removes leading and trailing so called spacers as well as tokens that represent a paragraph end:

```
\startfoo [x ] x \stopfoo
\startfoo [ x ] x \stopfoo
\startfoo [ x] x \stopfoo
\startfoo [ x] \par x \par \par \stopfoo
```

So we get this:

[x](x) [x](x) [x](x) [x](x)

The next directive, the quitter `#;`, is demonstrated with an example. When no match has occurred, scanning picks up after this signal, otherwise we just quit.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par

\tolerant\def\foo[#1]#;#={/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;#2{/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1/// /2/// /3///
//1// //2// //3//
///1/ ///2/ ///3/

I have to admit that I don't really need it but it made some macros that I was redefining

behave better, so there is some self-interest here. Anyway, I considered some other features, like picking up a detokenized argument but I don't expect that to be of much use. In the meantime we ran out of reasonable characters, but some day #? and #! might show up, or maybe I find a use for #< and #>.

+	keep the braces
-	discard and don't count the argument
*	ignore spaces
/	remove leading an trailing spaces and pars
=	braces are mandate
_	braces are mandate and kept
^	keep leading spaces
:	pick up scanning here
;	quit scanning

3 Runaway arguments

There is a particular troublesome case left: a runaway argument. The solution is not pretty but it's the only way: we need to tell the parser that it can quit.

```
\tolerant\def\foo[#1=#2]%
  {\ifarguments 0\or 1\or 2\or 3\or 4\fi:\vl\type{#1}\vl\type{#2}\vl}
```

The outcome demonstrates that one still has to do some additional checking for sane results and there are alternative way to (ab)use this mechanism. It all boils down to a clever combination of delimiters and `\ignorearguments`.

```
\dontleavehmode \foo[a=1]
\dontleavehmode \foo[b=]
\dontleavehmode \foo[=]
\dontleavehmode \foo[x]\ignorearguments
```

All calls are accepted:

```
2:|a|1|
2:|b|
2:|
1:x]|
```

Just in case you wonder about performance: don't expect miracles here. On the one hand there is some extra overhead in the engine (when defining macros as well as when

collecting arguments during a macro call) and maybe using these new features can sort of compensate that. As mentioned: the gain is mostly in cleaner macro code and less clutter in tracing. And I just want the ConTEXt code to look nice: that way users can look in the source to see what happens and not drown in all these show-off tricks, special characters like underscores, at signs, question marks and exclamation marks.

For the record: I normally run tests to see if there are performance side effects and as long as processing the test suite that has thousands of files of all kind doesn't take more time it's okay. Actually, there is a little gain in ConTEXt but that is to be expected, but I bet users won't notice it, because it's easily offset by some inefficient styling. Of course another gain of loosing some indirectness is that error messages point to the macro that the user called for and not to some follow up.

4 Introspection

A macro has a meaning. You can serialize that meaning as follows:

```
\tolerant\protected\def\foo#1[#2]#*[#3]%
  {(1=#1) (2=#3) (3=#3)}

\meaning\foo
```

The meaning of `\foo` comes out as:

```
tolerant protected macro:#1[#2]#*[#3]->(1=#1) (2=#3) (3=#3)
```

When you load the module `system-tokens` you can also say:

```
\luatokentable\foo
```

This produces a table of tokens specifications:

tolerant protected macro:#1[#2]#*[#3]->(1=#1) (2=#3) (3=#3)

tolerant protected control sequence: foo					
488768	19	49	match		argument 1
491814	12	91	other char	[	U+0005B
488942	19	50	match		argument 2
379676	12	93	other char	]	U+0005D
491847	19	42	match		argument *
491815	12	91	other char	[	U+0005B
488845	19	51	match		argument 3

498558	12	93	other char	]	U+0005D
148002	20	0	end match		
487013	12	40	other char	(	U+00028
487025	12	49	other char	1	U+00031
487031	12	61	other char	=	U+0003D
487778	21	1	parameter reference		
487850	12	41	other char	)	U+00029
498226	10	32	spacer		
491908	12	40	other char	(	U+00028
30475	12	50	other char	2	U+00032
487079	12	61	other char	=	U+0003D
30459	21	3	parameter reference		
491001	12	41	other char	)	U+00029
379705	10	32	spacer		
492012	12	40	other char	(	U+00028
488928	12	51	other char	3	U+00033
491754	12	61	other char	=	U+0003D
488831	21	3	parameter reference		
491052	12	41	other char	)	U+00029

A token list is a linked list of tokens. The magic numbers in the first column are the token memory pointers. and because macros (and token lists) get recycled at some point the available tokens get scattered, which is reflected in the order of these numbers. Normally macros defined in the macro package are more sequential because they stay around from the start. The second and third row show the so called command code and the specifier. The command code groups primitives in categories, the specifier is an indicator of what specific action will follow, a register number a reference, etc. Users don't need to know these details. This macro is a special version of the online variant:

```
\showluatokens\foo
```

That one is always available and shows a similar list on the console. Again, users normally don't want to know such details.

5 nesting

You can nest macros, as in:

```
\def\foo#1#2{\def\oof##1{<#1>##1<#2>}}
```

At first sight the duplication of # looks strange but this is what happens. When TEX scans the definition of `\foo` it sees two arguments. Their specification ends up in the preamble that defines the matching. When the body is scanned, the `#1` and `#2` are turned into a parameter reference. In order to make nested macros with arguments possible a # followed by another # becomes just one #. Keep in mind that the definition of `\oof` is delayed till the macro `\foo` gets expanded. That definition is just stored and the only thing that get's replaced are the two references to a macro parameter

control sequence: foo						
487872	19	49	match		argument 1	
488927	19	50	match		argument 2	
491816	20	0	end match			
487774	115	1	def			def
491507	133	0	tolerant call			oof
491831	6	35	parameter			
30427	12	49	other char	1	U+00031	
491810	1	123	left brace			
487665	12	60	other char	<	U+0003C	
491796	21	1	parameter reference			
252074	12	62	other char	>	U+0003E	
487663	6	35	parameter			
508550	12	49	other char	1	U+00031	
491209	12	60	other char	<	U+0003C	
488867	21	2	parameter reference			
491807	12	62	other char	>	U+0003E	
491755	2	125	right brace			

Now, when we look at these details, it might become clear why for instance we have 'variable' names like `#4` and not `#whatever` (with or without hash). Macros are essentially token lists and token lists can be seen as a sequence of numbers. This is not that different from other programming environments. When you run into buzzwords like 'bytecode' and 'virtual machines' there is actually nothing special about it: some high level programming (using whatever concept, and in the case of TEX it's macros) eventually ends up as a sequence of instructions, say bytecodes. Then you need some machinery to run over that and act upon those numbers. It's something you arrive at naturally when you play with interpreting languages.[2]

[2] I actually did when I wrote an interpreter for some computer assisted learning system, think of a kind of interpreted Pascal, but later realized that it was a a bytecode plus virtual machine thing. I'd just applied what I learned when playing with eight bit processors that took bytes, and interpreted opcodes and such.

So, internally a #4 is just one token, a operator-operand combination where the operator is “grab a parameter” and the operand tells “where to store” it. Using names is of course an option but then one has to do more parsing and turn the name into a number[3], add additional checking in the macro body, figure out some way to retain the name for the purpose of reporting (which then uses more token memory or strings). It is simply not worth the trouble, let alone the fact that we loose performance, and when TEX showed up those things really mattered.

It is also important to realize that a # becomes either a preamble token (grab an argument) or a reference token (inject the passed tokens into a new input level). Therefore the duplication of hash tokens ## that you see in macro nested bodies also makes sense: it makes it possible for the parser to distinguish between levels. Take:

```
\def\foo#1{\def\oof##1{#1##1#1}}
```

Of course one can think of this:

```
\def\foo#fence{\def\oof#text{#fence#text#fence}}
```

But such names really have to be unique then! Actually ConTEXt does have an input method that supports such names, but discussing it here is a bit out of scope. Now, imagine that in the above case we use this:

```
\def\foo[#1][#2]{\def\oof##1{#1##1#2}}
```

If you’re a bit familiar with the fact that TEX has a model of category codes you can imagine that a predictable “hash followed by a number” is way more robust than enforcing the user to ensure that catcodes of ‘names’ are in the right category (read: is a bracket part of the name or not). So, say that we go completely arbitrary names, we then suddenly needs some escaping, like:

```
\def\foo[#{left}][#{right}]{\def\oof#{text}{#{left}#{text}#{right}}}
```

And, if you ever looked into macro packages, you will notice that they differ in the way they assign category codes. Asking users to take that into account when defining macros makes not that much sense.

So, before one complains about TEX being obscure (the hash thing), think twice. Your demand for simplicity for your coding demand will make coding more cumbersome for

There’s nothing spectacular about all this and I only realized decades later that the buzzwords describes old natural concepts.

[3] This is kind of what MetaPost does with parameters to macros. The side effect is that in reporting you get `text0`, `expr2` and such reported which doesn’t make things more clear.

the complex cases that macro packages have to deal with. It's comparable using TEX for input or using (say) mark down. For simple documents the later is fine, but when things become complex, you end up with similar complexity (or even worse because you lost the enforced detailed structure). So, just accept the unavoidable: any language has its peculiar properties (and for sure I do know why I dislike some languages for it). The TEX system is not the only one where dollars, percent signs, ampersands and hashes have special meaning.

6 Prefixes

Traditional TEX has three prefixes that can be used with macros: `\global`, `\outer` and `\long`. The last two are no-op's in LuaMetaTEX and if you want to know what they do (did) you can look it up in the TEXbook. The ε-TEX extension gave us `\protected`.

In LuaMetaTEX we have `\global`, `\protected`, `\tolerant` and overload related prefixes like `\frozen`. A protected macro is one that doesn't expand in an expandable context, so for instance inside an `\edef`. You can force expansion by using the `\expand` primitive in front which is also something LuaMetaTEX.

Frozen macros cannot be redefined without some effort. This feature can to some extent be used to prevent a user from overloading, but it also makes it harder for the macro package itself to redefine on the fly. You can remove the lock with `\unletfrozen` and add a lock with `\letfrozen` so in the end users still have all the freedoms that TEX normally provides.

```
                \def\foo{foo} 1: \meaning\foo
          \frozen\def\foo{foo} 2: \meaning\foo
     \unletfrozen    \foo      3: \meaning\foo
\protected\frozen\def\foo{foo} 4: \meaning\foo
     \unletfrozen    \foo      5: \meaning\foo
```

1: macro:foo
2: macro:foo
3: macro:foo
4: protected macro:foo
5: protected macro:foo

This actually only works when you have set `\overloadmode` to a value that permits redefining a frozen macro, so for the purpose of this example we set it to zero.

A `\tolerant` macro is one that will quit scanning arguments when a delimiter cannot be matched. We saw examples of that in a previous section.

These prefixes can be chained (in arbitrary order):

```
\frozen\tolerant\protected\global\def\foo[#1]#*[#2]{...}
```

There is actually an additional prefix, `\immediate` but that one is there as signal for a macro that is defined in and handled by Lua. This prefix can then perform the same function as the one in traditional TEX, where it is used for backend related tasks like `\write`.

Now, the question is of course, to what extent will ConTEXt use these new features. One important argument in favor of using `\tolerant` is that it gives (hopefully) better error messages. It also needs less code due to lack of indirectness. Using `\frozen` adds some safeguards although in some places where ConTEXt itself overloads commands, we need to defrost. Adapting the code is a tedious process and it can introduce errors due to mistypings, although these can easily be fixed. So, it will be used but it will take a while to adapt the code base.

One problem with frozen macros is that they don't play nice with for instance `\futurelet`. Also, there are places in ConTEXt where we actually do redefine some core macro that we also want to protect from redefinition by a user. One can of course `\unletfrozen` such a command first but as a bonus we have a prefix `\overloaded` that can be used as prefix. So, one can easily redefine a frozen macro but it takes a little effort. After all, this feature is mainly meant to protect a user for side effects of definitions, and not as final blocker.[4]

A frozen macro can still be overloaded, so what if we want to prevent that? For this we have the `\permanent` prefix. Internally we also create primitives but we don't have a prefix for that. But we do have one for a very special case which we demonstrate with an example:

```
\def\FOO % trickery needed to pick up an optional argument
  {\noalign{\vskip10pt}}

\noaligned\protected\tolerant\def\OOF[#1]%
  {\noalign{\vskip\iftok{#1}\emptytoks10pt\else#1\fi}}

\starttabulate[|l|l|]
    \NC test \NC test \NC \NR
    \NC test \NC test \NC \NR
```

[4] As usual adding features like this takes some experimenting and we're now at the third variant of the implementation, so we're getting there. The fact that we can apply such features in large macro package like ConTEXt helps figuring out the needs and best approaches.

```
    \FOO
    \NC test \NC test \NC \NR
    \OOF[30pt]
    \NC test \NC test \NC \NR
    \OOF
    \NC test \NC test \NC \NR
\stoptabulate
```

When TEX scans input (from a file or token list) and starts an alignment, it will pick up rows. When a row is finished it will look ahead for a `\noalign` and it expands the next token. However, when that token is protected, the scanner will not see a `\noalign` in that macro so it will likely start complaining when that next macro does get expanded and produces a `\noalign` when a cell is built. The `\noaligned` prefix flags a macro as being one that will do some `\noalign` as part of its expansion. This trick permits clean macros that pick up arguments. Of course it can be done with traditional means but this whole exercise is about making the code look nice.

The table comes out as:

test test
test test

test test

test test

test test

One can check the flags with `\ifflags` which takes a control sequence and a number, where valid numbers are:

1	frozen	2	permanent	4	immutable	8	primitive
16	mutable	32	noaligned	64	instance		

The level of checking is controlled with the `\overloadmode` but I'm still not sure about how many levels we need there. A zero value disables checking, the values 1 and 3 give warnings and the values 2 and 4 trigger an error.