GMOA, the ‘General Manipulation Of Arguments’: An extension to the l3expan package of the expl3 bundle and language

Grzegorz Murzynowski

Abstract

After an introduction on how and why we switched to expl3, we present general assumptions and conventions of this language. Then we take a more detailed look at the l3expan package and the marvellous improvements of code it facilitates.

Then we describe our generalization and extension to the machinery provided by l3expan. We call it “GMOA”, “General Manipulation of Arguments” and argue why it’s quite an earned description.

We point out that GMOA is based on a finite automaton yet parses an arbitrary properly braced ⟨specification⟩ which involves recognizing the Dyck language, and we explain how does this not contradict that fundamental theorem.

1 Switching to expl3 or: “Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb”

1.1 The Proverb

There is a saying about one’s perception of human-made things, especially technology: “Whatever you know before 20, is just a part of Nature. What you learn between 20 and 35, is a ‘novelty’ that you’re curious or even enthusiastic about. Whatever you heard of after 35, you want to neither learn nor even know, as it’s ‘Against Nature’ and an ‘Abomination in the Eyes of the LORD’. ”

Well, I understood I’d probably *have* to take expl3 seriously when Will Robertson rewrote his fontspec package in(to) it some six years ago. When I was 36. Yes, exactly as the Proverb says.

Overcoming the Proverb took me four years. A new project had to begin at my work to let me think it’s a good time to learn and deploy “this abominable nonsense”. ;-)

And voilà, since only two months later all the new TeX code of mine is shorter by some ten thousand `\expandafter`’s, so much more elegant (I hope) and noticeably less buggy.

As probably happens to all neophytes, the “Old Things” (here: LaTeX 2$_\varepsilon$) look to me ugly and obscure while code written in expl3, no matter how unreadable to any “normal” TeX user or even a TeXnician, seems to me to be a visible instance of utter readability and clarity[1].

The “functions” have their signatures as part of their names; the “variables” (data carriers) also tell their scope and type by themselves; there’s no fear of an open `\if...`, moreover, no fear of unreadably nested conditions and `\fi`’s in wrong places, as compound condition(als) might be written as Boolean expressions and computed in just one expansion of `\romannumeral -`0` (more later on this).

And so many more of such beauties (and oddities) that at least an entire volume the size of this *TUGboat* issue might be written about them.

So — who and why should or shouldn’t be afraid of expl3?

1.2 Who’s afraid of Virginia Woolf

As you probably have already noticed, I’m talking about *programming* in TeX. Not about directly *using it* for typesetting or writing texts.

The expl3 language is intended for TeX programmers (such as macro package writers), not for the end users.

Therefore, if you’re using LaTeX for writing/publishing things and when trouble strikes you do not hack some code yourself but look for an existing macro package to do the job, you have very little reason to worry and can safely stop reading here. :)

So far, we’ve discussed the “-love” of the section title. Now let’s get to the “Strange-”.

1.3 General assumptions and conventions

As the reference manual/documentation[2] is quite comprehensive and instructive, let us only summarize what we consider most important for understanding of further material.

First of all, a special `\catcode` regime. The blank characters are assigned catcode 9 “ignored” — which means there is no need for all those to indent and choose line breaks in code yet avoid spurious spaces in text and so on.

In expl3 you can use blanks as blanks, i.e., to make the code clearer for reading. “The blanks” include the line end (not by re`\catcode`ing it, but in another, beautifully twisted way) so also blank lines can be used just for organizing of code.

It took me some time to get used to this. E.g., under this catcode regime, with this code:

```
\ifnum 1=0
  1a
```

Editor’s note: Originally published in the BachoTeX 2015 proceedings, pp. 49–56 (GUST). Revised by the author.

[1] cf. O.Wilde, “The Importance of Being Earnest”.

[2] “The LaTeX3 Interfaces”, `http://mirrors.ctan.org/macros/latex/contrib/l3kernel/interface3.pdf`

```
\else
 0z
\fi
```

you get the condition satisfied and the conditional expanded to a, the oppose of what what a wannabe-TEXnician like myself is used to. (As the first end-of-line doesn't translate to a space (catcode 10) but is just ignored, and the number read at the right side of comparison is 01.

On the other hand, when you use expl3, you are not supposed to use TEX primitives (with their original names and syntax, at least).[3]

Other important re-catcodings are that _ and : are made letters (cat.11) and used in control sequence names as significant or generic word joiners and separators. Without going into too much detail here, let's give a couple of examples. First:

```
\l__gme_additional_letters_tl
```

means a local __ internal variable of type _tl, 'token list', of the "module" (macro package/document class) gme.

And second:

```
\prg_replicate:nn
```

means an expl3 "function" of the module "prg", taking two braced arguments (usually, like a TEX macro with two undelimited parameters; this will be discussed later).

The obligatory : separates the expl3 function name from its signature. The signature is a (finite, possibly empty) sequence of the letters:

c, D, f, n, N, o, p, T, F, w, x.

Each letter specifies one argument, except for w, which means "weird syntax" and D, which means "DON'T!", for TEX primitives and other commands that should not be used outside the expl3 kernel.

It should be emphasized that the "functions" of expl3 are not always macros, either in the sense of taking undelimited arguments or in implementation. One should rather think of them as a conceptual construct, like LATEX 2ε "commands".

[3] This "hide your TEX and forget you ever used it" (cf. Marquis' de Sade's "Speech to the women" at the beginning of "120 Days of Sodom") is probably the only thing I definitely dislike in expl3 (as I did in LATEX 2ε).

It's partially because of my intellectual inertia, I admit. But also, and more importantly, because of my strong belief that not many people (if anyone) could compete with Prof. Knuth in the τέχνε ('art/craft') of computer programming. Which implies that hardly anything written on top of TEX could be, I'm not saying: better than it; I'm saying: *comparable* with it. I believe that some of TEX's "beauties-oddities" will strike at some point anyway, and then looking down to some lowest-level things is inevitable.

It's also explicitly stated in the Reference Documentation that the uppercase letters ("argument types") N and V require a single token without braces, while all the others allow many tokens, in braces; for those latter, wrapping even a single token in braces is encouraged.

The p argument type is solely for macro parameter strings in the sense of *The TEXbook*'s chapter 20 and therefore can neither contain braces nor be wrapped with them. We'll see later how this works with the l3expan conventions and (not quite) with our extension to them.

Here is a variant of the previous function:

```
\prg_replicate:Vo
```

This requires the first argument to be an expl3 variable and renders its value; the second argument is hit with one \expandafter and only such preprocessed arguments are given the "original" or "primary" \prg_replicate:nn function.

As we are talking "functions" in expl3, they "return" a "result". We use these words in the sense of expl3 henceforth; bear in mind that, translated to our good old TEX jargon, they generally mean 'what at some point is left in TEX's "mouth" after execution of this stuff' or, most often, just 'expand to'.

Which brings us to part two, the pre-expansion of arguments with the l3expan package.

2 The l3expan package: Pre-expansion of macro arguments

Consider the not-so-unusual situation that before applying a macro, the soon-to-be arguments should be preprocessed ("tenderized"[4]). For instance, a control sequence \arg@i@int, a \count register or \numexpr, hit with \the; an {⟨arg.2⟩} subjected to \edef; and {⟨condition⟩{⟨arg.3T⟩}{⟨arg.3F⟩}} expanded either to {⟨arg.3T⟩} or {⟨arg.3F⟩} but not further, depending on ⟨condition⟩.

In LATEX 2ε we would write something like:

```
\newtoks \l@aux@args@toks
\newtoks \l@auxA@toks
\newtoks \l@auxB@toks

\l@auxA@toks = {{⟨arg.3T⟩}}
\l@auxB@toks={{⟨arg.3F⟩}}

\edef\aux@macro {%
 \if⟨test⟩ ⟨condition⟩
  \the\l@auxA@toks
 \else
  \the\l@auxB@toks
 \fi
```

[4] cf. http://myfirstdictionary.blogspot.com/2011/03/todays-word-is-shoo.html

```
}
```

```
\l@aux@args@toks
 \expandafter {\aux@macro } % "{⟨arg.3⍰⟩}"
```

```
\edef \aux@macro {⟨arg.2⟩}
\l@aux@args@toks
 \expandafter\expandafter\expandafter
 {\expandafter \aux@macro
 \the\l@aux@args@toks }
     % "{⟨arg.2-ed⟩}{⟨arg.3⍰⟩}"
```

```
\expandafter \def \expandafter
\aux@macro \expandafter{%
 \the \arg@i@int % it can be a numexpr,
      we don't know the num. of tokens
}
\l@aux@args@toks
 \expandafter\expandafter\expandafter
 {\expandafter \aux@macro
  \the\l@aux@args@toks }
     % "{⟨val.of arg.1⟩}{⟨arg.2-ed⟩}{⟨arg.3⍰⟩}"
```

```
% and, finally,
```

```
\expandafter \__mod_foo:nnn
 \the \l@aux@args@toks
```

As we can see, that's rather hard core. Even with the shorthands of our gmutils package(s), it wouldn't look much better.

Now, with l3expan, we can type

```
\::V \::x \::f \:::
\__mod_foo:nnn
\arg_i_int {⟨arg.2⟩}
{\⟨test⟩:nTF{⟨condition⟩} {⟨arg.3T⟩}{⟨arg.3F⟩}}
```

Four lines instead of ca. 30.

Or, if we expect such pre-expansions more often, we can introduce a *variant* of the expl3 function `\__mod_foo:nnn`:

```
\cs_generate_variant:Nn
 \__mod_foo:nnn {Vxf}
```

```
\__mod_foo:Vxf
\arg_i_int {⟨arg.2⟩}
 { \__⟨condition⟩:TF {⟨arg.3T⟩}{⟨arg.3F⟩} }
```

By the way, if the nature of pre-processing allows, all those `\::⍰` macros are expandable. How is that done? Except for `\:::`, all the `\::⍰`'s are `\long` 3-parameter macros with `#1` delimited with `\:::`, `#2` undelimited and `#3` depending on the nature of pre-processing, usually undelimited. For instance, `\::o` is defined like (TeX primitives are given here in their original names, l3expan uses them in expl3 aliases; please remember the blanks are cat.9 "ignored"):

```
\long\def
\::o #1 \::: #2#3
{ \expandafter \__exp_arg_next:nnn
  \expandafter {#3} {#1} {#2}
}
```

```
\long\def
\::f #1 \::: #2#3
{
 \expandafter \__exp_arg_next:nnn
 \expandafter { \romannumeral -`0 #3 }
 {#1} {#2}
}
```

```
\long\def
\__exp_arg_next:nnn #1#2#3
{ #2 \::: { #3 {#1} } }
```

Which means that `\::o` applies one `\expandafter` to its `#3` and then lets `\__exp_arg_next:nnn` do the rest.

Similarly, `\::f` applies `\romannumeral` in such a way that any leading expandable tokens of `#3` are expanded until the first unexpandable token is seen. Please note the ``-`0`` preceding `#3`. Thanks to it, even if `#3` expands to decimal digit(s), `\romannumeral` is satisfied with ``-`0`` as a complete ⟨*number*⟩ specification and, as this number is negative (the charcode of `0` is 48), expands to ⟨empty⟩ (empty sequence of tokens). Thus we get an "AFAP" ('As Far As Possible') expansion of `#3` in just one `\expandafter`.

(Very few things move me as deeply as this trick, if I may express my personal yet professional feelings here.)

Some of the pre-processors which render the value of a variable also use ``\romannumeral-`0``, depending on the variable type. The current implementation of the `_tl` type, for instance, as parameterless macros not, e.g., as `\toks` registers, allows for rendering their value with just an `\expandafter`, or even just the use of the variable name, in many contexts.

What happens next: the `\__exp_arg_next:nnn` macro appends the pre-processed `\::⍰`'s `#3` to the end of (that `\::⍰d`) `#2` (the result-so-far). All of that finally looks like:

```
⟨#1⟩ % remaining \::⍰'s
\::: {⟨#2⟩[{]⟨#3 pre-processed⟩[}]}
 ⟨further input⟩
```

The mysterious `\:::` delimiter control sequence is `\firstofone` or, in its expl3 naming, `\use:n`; so that when all the `\::⍰`'s do their job, it strips the outer braces off the final result.

For those pre-processors which pass their result stripped of braces, there's another version of the "pass-the-result" macro:

```
\long\def
```

```
\__exp_arg_next:Nnn #1#2#3
{ #2 \::: { #3 #1 } }
```

As you can clearly see, the difference in behaviour of these macros seems to be reflected in lower-/uppercase opposition of the letter(s) `n` vs. `N`.

However, both the Reference Documentation and further explanations from the LaTeX3 team discourage such an interpretation, underlining the expl3 goal to allow (force) the user "not to rely on implementation" and focus on the requirements (in most cases purely conventional): `N` "requiring" a single and un-braced token, most often a control sequence, and `n` "requiring" an arbitrary token(s) in braces, as in the ⟨*balanced text*⟩ of *The TeXbook*.

However, if we really stick to the conventions and don't rely on implementation[5] (how is *that* possible for a TeX programmer, let alone a TeXnician? ;-)), everything seems to work just fine.

Which leads us to some remarks concerning the pre-expansion of arguments and its generalization to GMOA. (With tidying-up the "argument types" with respect to it.)

3 GMOA, a 'General Manipulation Of Arguments' or: a DFA that seemingly recognizes the Dyck Language

Long story short, on top of the l3expan pre-expanders we add arbitrary rearrangements and/or grouping of arguments. In a "one-char" syntactic manner similar to the `tabular` specifications. And expandable (except for `x` and `X`). And with consistent naming conventions, coherent with l3expan to some degree.

And, last but not least, performed (at the stage of translation of the specifiers sequence into `\::⍰`-like macros) by a DFA, a (deterministic) finite automaton. Which includes a Dyck Language recognition. Without falsifying the theorem that there does not exist a DFA to do that.

[5] There is a temptation of using some functions as if they were (e)TeX primitives, like `\hbox:n` (as currently allowing the `\bgroup...\egroup` syntax) or `\tl_to_str:n` as the supposed expl3 alias for `\detokenize`. Just DON'T. Quoting Joseph Wright's email, "...for example, where the team needs to use the primitive behaviour of `\detokenize` [...], we use the 'raw' name `\etex_detokenize:D`".

I'm not sure what a non-(LaTeX3 team) person should do in this case, since the Reference Documentation reads: "The `D` specifier means *do not use.* [...] Only the kernel team should use anything with a `D` specifier!" We're again at "Hide your TeX and forget you've ever used it"? Over my dead body.

The gme3u8.sty package (of mine) provides aliases for the (e)TeX primitives my way and anyone is invited to use them (at their own risk, of course). (The list is not complete, I just add what I needed so far.)

Roadmap for this section. After an excuse for using non-ASCII chars which need a special font, we give a formal definition of the GMOA specifications. Then we present an overview of the GMOA machinery, its *modus operandi*, and some of its macros alongside with their conceptual structure as a finite automaton.

In the subsequent (subusubsequent? ;-)) (subsub)sections we present groups of specifiers and respective automaton states / stages of parsing:

- the ⟨destination⟩ tokens,
- the ⟨prep-or-↓⟩s,
- the meta-operators,
- the digits / ⟨FSM⟩s,
- the braces / ⟨BDSM⟩s

We end this section with examples: one that is almost-comprehensive though not-necessarily-useful and a handful of "real-life" ones. All of them go beyond what's possible with l3expan only.

A disclaimer about (non-)ASCII chars and the custom font Ubu Stereo. Before we proceed, a remark. The expl3 language keeps strictly to ASCII. Any characters outside of ASCII that occur in the next part of this paper, especially those from the "distant far-aways" of the Unicode or even from the Private Use Area (PUA henceforth), are entirely my "fault and guilt".

I use some letters looking similar to some non-letter ASCII chars with the intention of making the names even more self-explanatory, while simultaneously respecting the expl3 naming conventions. Other characters are chosen for perfectly rational reasons (e.g., ϛ, the astrological symbol of the name of my dearest dog, which I use as my "trademark" in the expl3 module part of names). And, a some chars FontForged by myself to depict TeX primitives and other often-used things in one character each.

All are put together in one font named Ubu Stereo. The font is based on Ubuntu Mono with some characters copied from other libre fonts, especially DejaVu Sans and FreeSerif. I act with those fonts as I please (PL: "Wedle mojego widzimisię"), hence "Ubu" (cf. A. Jarry, "Ubu le Roi" &c.). "Stereo", because it's not all monospaced, as some wide characters are given double width for better distinction, such as —, and some combine to double width in pairs in a kind of typographical *rubato*, with one char half-width (declared as another escape char) and the other one-and-a-half-width, to provide one-char control sequences (I'm curious if they constitute reasonable mnemonics to anyone but me):

```
7↷ \expandafter
7⍂ \noexpand
```

`7⊠` `\unexpanded % e-TeX primitive`
`ꟻ▌ ... 7▙` `\csname ... \endcsname %` stator(s) of an electric motor; make the stuff between them spin.

With that description, we switch the mono font to Ubu Stereo from now on and get down to GMOA at last.

3.1 Description by examples

To give an idea of GMOA, let us rewrite an example from the previous section.

```
\⁝⁝ I Vxr :
\__mod_foo:nnn
\l__arg_i_int {⟨arg.2⟩}
{\<condition>:TF {⟨arg.3T⟩}{⟨arg.3F⟩} }
```

where `\⁝⁝` is the name of the main GMOA macro (subject to alias on user's request) and the subsequent letters are the specifiers of operations, "the operators" for short. So far, nothing more than in `l3expan`, as `I` stands for "**I**dentity" and just preserves `\__mod_foo:nnn` until the rest of the arguments are preprocessed, `V` and `x` act as (are translated to) `\::V` `\::x` and `r` translates to (an alias of) `\::f`, i.e., does the `` \romannumeral -`0 `` trick.

But let's have a look at some real-life use (courtesy of the PARCAT project, `parcat.eu`):

```
\⁝⁝  12̰{42̰5̰678̰} 13̰{43̰5̰679̰} :
\DeclareOption                          % 1
{oneside}{twoside}                   % 2, 3
\PassOptionsToClass                     % 4
{report}                                % 5
\protected\def                          % 6
\__ins'_page'oddity'count:              % 7
\c_one \c@page                       % 8, 9
```

This code declares a LaTeX 2$_\varepsilon$ document class options `oneside` and `twoside` that differ only in the meaning of `\__ins'_page'oddity'count:` and both pass their names to the basic document class `report`. I.e., `\⁝⁝...:` rearranges the code above into:

```
\DeclareOption {oneside}
{\PassOptionsToClass {oneside} {report}%
  \protected \def \__ins'_page'oddity'count:
      {\c_one }%
}
%
\DeclareOption {twoside}
{\PassOptionsToClass {twoside} {report}%
  \protected \def \__ins'_page'oddity'count:
      {\c@page }%
}
```

Some parts of the code have been replicated as many times as needed, their order changed, some groups of arguments were put into the same pairs of braces.

By writing all the *mutatis mutandis* text just once with the "mutandis" not repeated, the code is kept strictly parallel, i.e., change-robust, as any change need be made in only one place.

On the other hand, using this machinery has the obvious disadvantage that you have to learn the mini-language of specifiers. Hopefully, this is an acceptable expense. Another inconvenience is the counting of the arguments, which might be quite tricky, especially if there are mixed sequences of stand-alone operators and ⟨FSM⟩/⟨BDSM⟩ parts in a specification (discussed in the examples following the formal definition).

I dare think of `\⁝⁝` as superior[6] to both the `l3expan` low-level `\::▫` macros and the machinery of `\cs_generate_variant:Nn` in some aspects.

It provides much more general rearrangement and pre-processing options than either of the latter. It has much shorter syntax than the `\::▫` macros; moreover, it can also be used within the usual catcode regime (provided the very name is aliased properly), as the specifiers are parsed with the `\strcmp` comparisons, which are independent of the catcodes.

Except for braces, which are discussed in the section 3.2.6.

3.2 GMOA: Formal language definition

⟨GMOA⟩ ::= `\⁝⁝` ⟨specification⟩ `:`
⟨specification⟩ ::=
  ⟨destination⟩⟨FSoO⟩⟨optional `.`⟩
  ⟨specification⟩
⟨destination⟩ ::= ⟨empty⟩ | `ξ` | `ς` | `σ`
⟨optional `.`⟩ ::= ⟨empty⟩ | `.`
⟨FSoO⟩ "Finite Sequence of Operators" ::=
  ⟨empty⟩ | ⟨SAlos⟩⟨optional `;`⟩⟨FSoO⟩
  | ⟨FSM⟩⟨optional `;`⟩⟨FSoO⟩
⟨optional `;`⟩ ::= ⟨empty⟩ | `;`
⟨SAlos⟩ Stand-Alone's ::= ⟨preps'n'↓s⟩
  | ⟨SAlos⟩`(`⟨prep.seq.⟩`)`⟨SAlos⟩
  | ⟨SAlos⟩`` ` ``⟨prep⟩⟨SAlos⟩
  | ⟨SAlos⟩∗⟨prep⟩⟨SAlos⟩
  | ⟨SAlos⟩⁑⟨prep⟩⟨SAlos⟩
  | ⟨SAlos⟩⟨meta-ᴿ⟩⟨SAlos⟩
⟨preps'n'↓s⟩ ::= ⟨empty⟩
  | ⟨prep-or-↓⟩⟨preps'n'↓s⟩
⟨prep-or-↓⟩ ::= ⟨prep⟩ | ↓
⟨prep.seq.⟩ ::= ⟨empty⟩ | ⟨prep⟩⟨prep.seq.⟩

[6] It's infinitely easier to expand/develop something than to invent it in the first place. `l3expan` does things I've not even thought of in ten years of my TeXnician's life. Or if I did, it was: "Nah ... it's impossible; you just can't hit the 2nd undelimited argument with `\expandafter` since you don't know how many tokens are there in the first one".

⟨prep⟩ ::= `c`|`ć`|`Ć`|`ð`|`Đ`|`f`|`h`|`H`|`i`|`I`|`k`|`K`|`n`|`N`
  |`o`|`O`|`ⴱ`|`ϴ`|`p`|`P`|`q`|`Q`|`r`|`R`|`s`|`S`|`T`|`F`|`v`|`V`|`x`
  |`X`

⟨meta-ᴿ⟩ ::= ᴿ⟨arbitrary-meta-ᴿ⟩
  |×⟨number specification⟩⟨meta-replicated⟩

⟨arbitrary-meta-ᴿ⟩ ::= a(ny) TEX code that ᴿ-expands to [consistent part of] a ⟨specification⟩

⟨number specification⟩ ::= the `\prg_replicate:nn`'s first argument

⟨meta-replicated⟩ ::= the `\prg_replicate:nn`'s second argument

⟨FSM⟩ "Finite Sequence Manipulation" ::= ⟨opt.cardinality⟩⟨FSM w.card.par.⟩

⟨opt.cardinality⟩ ::= ⟨empty⟩ | |⟨digit⟩|

⟨FSM w.card.par.⟩ "FSM with cardinality paradigm known" ::=
  ⟨FSM chunk⟩⟨optional `,`⟩
  ⟨FSM w.card.par.⟩

⟨FSM chunk⟩ ::= ⟨digits with prep.seqs.⟩
  |⟨BDSM⟩

⟨digits with prep.seqs.⟩ ::= ⟨empty⟩
  |⟨digit⟩⟨prep.seq.⟩⟨digits with prep.seq.⟩

⟨digit⟩ ::= `1`|`2`|`3`|`4`|`5`|`6`|`7`|`8`|`9`
  |*A*|*B*|*C*|*D*|*E*|*F*|1|2|3|4|5|6|7|8|9|*a*|*b*
  |*c*|*d*|*e*|*f*

⟨optional `,`⟩ ::= ⟨empty⟩ | `,`

⟨BDSM⟩ "Braced Dyck-language Sequence Manipulation" ::=
  `{`⟨FSM chunk⟩`}`⟨prep.seq.⟩

The chars of a specification are hit with `\string`, one by one, so each gets catcode 12 (other) unless it has catcode 5, 9, 10, 14, or 15.

The only cat.1 and cat.2 characters allowed are `{` and `}` and they have to be balanced.

Blank chars, either in `expl3` catcode regime (i.e., cat.9 "ignored") or in the usual (cat.10) are skipped and may be used at will to improve readability.

Let us now explain the semantics, i.e., "see what this mouse trap really does".[7]

3.2.1 Overview of the DFA

`\⁝` is a macro with one parameter delimited with `:`$_{11}$ (catcode "letter"). Taking an entire ⟨specification⟩ at once serves only for checking if there is a predefined macro to handle the given case.

If no such shortcut is predefined, the code

```
⸢ \__⁝¹_str῀s¨KN:w ⟨specification⟩ :{}
```

is executed, where `⸢` is just `\csname` and the next control sequence is a GMOA first-stage (`⁝¹`) `\string`-preceded (`str`) macro for the state (`῀s`) named `KN`, 'Know Nothing'. Which is, as one can guess, the initial state.

From there on the characters of ⟨specification⟩ are hit with `\string`, picked one-by-one, tested, and transitions performed accordingly, which TEXnically amounts to putting further and further "telescopic" `\csname`'s, i.e., the sequences of tokens that could and should be transformed into a control sequence at the very moment the matching `\endcsname` is met.

Only, the immediate predecessor of each such `\endcsname` is ... `\expandafter`. And the token next to `\endcsname` is ... another `\csname`:

```
\csname cs-name-1 \expandafter\endcsname%
       \csname cs-name-2 ...
```

I think of this trick as of an (architectural) arch or bridge; and I think of `\csname...\endcsname` as the stator(s) of an electric motor, which make(s) the stuff between them spin. Hence the PUA/Ubu signs based on Japanese quotation marks:

```
⸢ cs-name-1 ⸥⸢ cs-name-2 ⸥
```

Now, a GMOA

```
\⁝ ξIo ςIir :
```

is at some point translated into:

```
⸢ ::I∗ ⸥⸢ ::o∗ ⸥⸢ ::I ⸥⸢ ::i ⸥⸢ ::r ⸥⸢ ::: ⸥
```

and turns into the `\::⍰` macros from right to left.

At the moment of submitting this paper for printing, GMOAs DFA consists of 12 states with 64 transitions (and 11 ⟨meta-ᴿ⟩ interruptions).

The transitions are labeled not with particular characters but with equivalence classes of: ⟨prep⟩s, ⟨∗meta-operator⟩s, ⟨destination⟩ tokens, &c.

It's probably not a significant savings of memory or other computational costs, but does result in a great simplifying of the code. It also makes the code more change- and development-robust; e.g., adding a new argument type, which is denoted with a char of equivalence class ⟨prep⟩, does not require any changes in the automaton.

3.2.2 The destination

Parsing of a ⟨specification⟩ starts from determining of the "destination", i.e., the way the result of the next ⟨FSoO⟩ is yielded:

⟨empty⟩ If no explicit destination token is given,[8] the usual "just once" is assumed, like `l3expan`'s `\::⍰`/`\__exp_arg_next:nnn`. This is equivalent to the use of `ς`.

`ς` Greek letter small sigma final form, the open variant, for "συναγωγή πολύ" /synagoge poly/,

[7] cf. William Tenn, "Errand Boy", in "The Seven Sexes", 1968.

[8] i.e., the first char met is none of `ς`,`σ`,`ξ`.

'gather (as) many' (with intended associations with the correlation between social diversity and open-mindedness of people). Therefore let us call this "just once and multi".

σ stands for "συναγωγή μόνο" /synagoge mono/, 'gather [as] one', Greek letter small sigma middle form, the closed variant, to be associated with enclosing of all the picked and pre-processed arguments in one common resulting pair of braces. ("Just once and as one".)

ξ for Greek "ξανα", '[use] again': the result is put back as input for further parts of specification, after this ⟨destination⟩ is done (quite like ruminants do). ("We'll meet again", as in the final credits of "Dr. Strangelove or ...".)

[We skip the description of the "Grand Gathering" post-meta ⟨SAlos⟩ Σ and Ξ as having originated in the formative stage of the GMOA language and most probably doomed for deletion. Basically, they act like ς and ξ, only applying to "everything done so far" and have notably less elegant syntax.]

3.2.3 The ⟨prep-or-↓⟩'s

If one understands (the beauty of) the l3expan preprocessors, then the ⟨prep⟩s are the easy part.[9] All those letters either directly correspond to some expl3 "argument type" and the respective `\::⊔` macro (and are internally translated to it), or extrapolate their idea, perhaps even towards a kind of a completeness or full(er) symmetry.

They can be divided into two groups: "argument pre-processors" and "special pickers", the latter being `H`,`h`,`p`,`Q`,`q`. They pick (scan) the arguments delimited in special ways as described below, and don't do anything else to them. All the others, the "argument pre-processors", take undelimited arguments (again in the sense of *The TEXbook*, chapter 20) and submit them to various kinds of expansion or "rendering".

What needs to be emphasized is the expl3 concept of "argument types", although mostly consistent with my idea of upper- and lowercase operators, uses the opposition of upper- and lowercase with a different purpose. The original intent, expressed both in the Reference Documentation and in the LATEX3 team members' responses to my emails, is that `N` and `V` require a single and unbraced token, usually a control sequence, while all the others allow many tokens, in that case in braces (except the `p` type for obvious reasons).

[9] cf. "Devil's Advocate", Al Pacino as Satan talking salary with Tom Cruise.

(And, not to forget that the `T` and `F` argument types refer to the argument originally braced yet returned without braces.)

This is, as far as I understand it, part of the general goal of creating an abstract programming language, not necessarily using TEX in the future.

My ideas are more moderate. I'd like just to provide some easier ways to memorize and shortcut some programmatic constructs *in TEX*, for a person who is acquainted with TEX and with at least some of its beauties-oddities. Therefore the GMOA mini-language of one-char specifiers consistently (to the maximum extent permitted by expl3) "thinks" of the uppercase specifiers as referring to the resulting arguments unbraced and their lowercase counterparts as referring to the arguments braced, up to introducing "lowercase digits", as we'll see later.

So, let us see what the pre-processors do. The ones homonymic with l3expan "argument types" are put in [square brackets].

`[c]`,`ć`,`Ć` The uppercase `Ć` is an alias for `c`, i.e., applying `\csname...\endcsname` before passing the argument on *without* braces. The lowercase `ć` does the same, only passing the result on in braces. What is this useful for? First, for the TEX primitives that require a `{`⟨balanced text⟩`}`. Second, for theoretically possible constructs like

```
cs name 1\expandafter\endcsname\csname cs
        name 2 ...
```

`ð`,`Ð` (Icelandic, etc., letter eth/Eth) Hit the argument with `\the`. In the current implementation of expl3, it's almost equivalent to `V` for some expl3 data types, namely: `_int`, `_dim` and `_skip`.

`ð` is equivalent to `V`, and `Ð` – `VI`. `v` is equivalent to `*cð` in stand-alone contexts or `cð` as ⟨prep.seq.⟩ of an ⟨FSM⟩ or ⟨BDSM⟩.

However, due to the "Don't rely on implementation" rule one should *always* use the `v` or `V` specifier to render the value of an expl3 data carrier. (Which is a bit more expensive than the `\the`.)

`h`,`H`,`[p]` Pick the `#{`-delimited argument and return it without braces (`H`, `p`) or wrapped in braces (`h`).

`i`,`I` **I**dentity operation, braced or unbraced with respect to the lettercase.

`k`,`K` `\detokenize` the argument.

`[n]`,`[N]` **N**o pre-processing. Equivalent to `i/I` in the current implementation of l3expan (described separately as that implementation should not be relied upon).

`[o]`,`[O]` **O**ne level expansion with `\expandafter`. (As currently implemented in l3expan.)

`ꙩ,Ꙩ` Cyrillic letter lowercase/uppercase binocular o. Two-level expansion with `\expandafter`. Used by the author for elegant (in his opinion) pre-processing of macros that should be expanded to their content and that content hit again, e.g.:

```
\def\number_of_page:{\the\c@page}
```

`q,Q` Pick the argument delimited with `\q_stop`.

`[f],r,R` Apply `\romannumeral -`0` to the argument. This causes the leading token(s) of that argument to be fully expanded until an unexpandable token is seen. So, it's called `f` for the "full" and *not* called so by myself for the "until first unexpandable". Here comes a point where I prefer to refer to (some) knowledge of TeX, namely, of the `\romannumeral` primitive.

`s,S` Hit the argument with `\string`. It's worth emphasizing that the uppercase variant returns the result without braces, which for a control sequence means at least two bare tokens (except for `\␣`).

`[T],[F]` Strictly parallel to the l3expan/expl3 homonyms (functionally equivalent to our `I` but intended to indicate the conditional branches).

`[v],[V]` Strictly parallel to the l3expan/expl3 homonyms (rendering the value of a data carrier (an expl3 variable or constant), given as a control sequence for `V` or as the tokens of its name for `v`). Related to `ð` and `Đ`; see above.

`[x],X` Hit the argument with `\edef`. The lowercase variant translates to the `\::x` macro in l3expan/expl3 which, in its current implementation, returns the result in braces. The uppercase variant returns the result without braces and is not present in expl3.

The `↓` operator discards the respective argument. Therefore there is no need to use it in the ⟨FSM⟩s (including ⟨BDSM⟩s), as there you just skip a digit for that purpose.

When used as SAlos, the ⟨prep-or-↓⟩s refer to and are applied to subsequent arguments from the input.

When following a ⟨digit⟩, the ⟨prep⟩s refer to and are applied to the ⟨digit⟩th argument from the input, counting as explained later.

When following a BDSM, the ⟨prep⟩s refer to that BDSM as if it was a single argument taken from the input.

Before we deal with the ⟨FSM⟩s and ⟨BDSM⟩s, a word on the meta-operators as they apply to the just-presented ⟨prep⟩s, and other specifiers that constitute categories (GMOA "char-classes") by themselves.

3.2.4 The meta-operators

The `, ∗, ⁑ and `(...)` meta-operators, i.e., the operators modifying actions of ⟨prep⟩s, are intended to allow applying multiple pre-processings to the same argument without calling the ⟨FSM⟩ machinery (which is much more expensive, as we'll see). As they are redundant with respect to the general power of GMOA, we leave the details for an interested reader in the documentation.

The `ᴿ` meta-operator (or rather: interruptor) suspends parsing of ⟨specification⟩, hits whatever is next to it with `\romannumeral -`0` and then hopefully resumes the parsing. That allows you to branch the very specification of a given GMOA, not only its arguments. Including nesting of GMOAs. Does it increase its expressive power? Probably not, but it lets you write code in a more meta- way. Also in a way much more obscure, yet shorter.

`×` is a shorthand for `ᴿ\prg_replicate:nn`; thus it requires two pairs of braces to follow, the first containing a ⟨number specification⟩ and the other the things you wish to replicate. This way, instead of

```
\⁝⁝ ... ↓↓↓↓↓↓↓↓ ...:
```

you can type

```
\⁝⁝ ... ×8↓ ...:
```

As you may have noticed, at this point I do rely on the current implementation of `\prg_replicate:nn`, both on its expandability and on the fact that so far it's a macro with two undelimited arguments (i.e., the presence of braces is not obligatory in fact).

Let's now deal with the ⟨digit⟩s, that is, the general permutations.

3.2.5 The general permutations, or the ⟨FSM⟩s without regrouping

The ⟨digit⟩s refer to subsequent arguments from the input. However, the counting starts after the earlier ⟨FSoO⟩s are done and the preceding ⟨SAlos⟩ from the current ⟨FSoO⟩.

Which means that we skip the arguments picked and processed within earlier (possibly implicit) `ς` and `σ` destinations and the `↓`'s independent of destination, and include the ones destined `ξ`, and all standalone ⟨prep⟩s from the current destiny. Consider

```
\⁝⁝ ς Iii  ξ o↓.  1343 :
\__gme_foo:nnn {⟨1st arg.⟩}{⟨2nd arg.⟩}
{\__gme_arg'2B:N \__gme_⤼'ed:}%  and then d.1
{⟨to be ↓-discarded⟩}
{⟨d.2-arg (discarded)⟩}{⟨d.3-arg⟩}{⟨d.4-arg⟩}
```

The text of the 3rd line gets hit by `\expandafter`. The argument from the next line is discarded by

↓ (no matter that it's within a ξ /xana/ destination). As the `.` is seen, the ξ ⟨FSoO⟩ is finished, i.e., the `\expandafter`ed argument is put back for further processing. Which means it becomes the "digit 1" of the last ⟨FSoO⟩ (with implicit destination ς just like the first).

Three subsequent braces are picked according to their specifications and since the digit 2 does not occur in the ⟨FSoO⟩, the {⟨d.2-arg...⟩} brace is discarded.

The result looks like:

```
\__gme_foo:nnn {⟨1st arg.⟩}{⟨2nd arg.⟩}
{⟨one-lev.exp.of⟩\__gme_arg'2B:N \__gme_?'ed:}
{d.3-arg}{d.4-arg}{d.3-arg}
```

The processing of a "general permutation" can be described as two stages: (stage one) preparation of the "slab"[10] and then (stage two) picking numbered arguments from it.

Stage one consists of picking of proper number of arguments (only undelimited so far, but one can put some ξ ⟨FSoO⟩ before that contains some "special pickers", right?), (re)wrapping them in braces and separating with indices which will be the argument delimiters for the "digit"-picking macros.

In this role we have just "bare" digits and Latin capital letters (hex digits so far), which is safe since the (GMOA) arguments' contents is "invisible" to TEX's macro argument scanner, thanks to the braces.

For instance, a slab (or, closer to TEX digestive tract metaphors, a "craw") for the 4-permutation in the example above looks as follows:

```
⟨the FSM translated into \::\?-like macros⟩
\::: {} % container for the result
\::_\__⁝⁝_FSM˜args %  start-delimiter of the "slab"
1{⟨d.1-arg⟩} 2{⟨d.2-arg⟩} 3{⟨d.3-arg⟩} 4{⟨d.4-
        arg⟩}
\q__⁝⁝_FSM'craw˜stop %  end-delimiter of the "slab"
        or "craw"
{⟨tail of the (whole) ⟨specification⟩⟩\:::} %
        delimited just like in l3expan
{⟨the result of earlier part of ⟨specification⟩⟩}%
```

The slab is functionally a one-dimensional array (a vector). The accessor macros read all the stuff until their proper index and the argument after that index and add that argument to the result container, and return all the stuff including the "original" copy of their argument back to the slab.

It looks very expensive; it would probably be more efficient to define index-named macros whose contents would be the ⟨FSM⟩'s arguments. Then access to an element would cost only one `\csname...\endcsname` plus one one-level expansion of it. (Plus one initial `\long\edef{\unexpanded{`⟨arg.⟩`}}`[11] each.)

But when implemented this way, it stays expandable. Why is that so important? I'm not sure. But it's certainly fun to be able to process quite complex rearrangements with just one `\romannumeral-`0`!

But, one may ask, how does GMOA know how many arguments should be taken for an ⟨FSM⟩?

As ⟨opt.cardinality⟩ is ⟨empty⟩ here, the largest value of ⟨digit⟩ is assumed. Again, in an expandable way, via an initial assumption of 0 and comparing current ⟨digit⟩ with the largest-so-far, which is passed as an argument to the next step of parsing-expansion.

If the ⟨opt.cardinality⟩ value is given explicitly, the comparisons are performed anyway and an error raised at a digit exceeding the cardinality declared.

Each ⟨digit⟩ of an ⟨FSM⟩may be followed by ⟨prep.seq.⟩, in which case a respective sequence of operations is applied to the resp. copy of the resp. argument, as stated above.

To allow arguments beyond the 9th, the `A`...`F` symbols are used with those "ribbon-accents" to distinguish the hexadecimal digits `A`–`F` from their Latin-letter counterparts (in this role, Unicode points from the PUA, U+E990–5 as provided in the Ubu Stereo font, and also Elisp functions for inserting them in my GNU Emacs), whereas the latter might become ⟨prep⟩s in the future (the letter `D` is already in use by expl3, although not handled by GMOA). (Yet internally represented as hexadecimal digits `A`...`F`.)

The "lowercase" hex digits `1`...`9`,`a`...`f` serve as shorthands for `1i`...`Fi`, i.e., while `1`...`F` render the respective arguments without braces, `1`...`f` pass their arguments (re)braced (which is why in the Ubu Stereo font they have those tiny under- and over-braces).

So, it seems we are handling a dynamic-length data structure within purely expandable sub-TEX. Are we really? Yes, to some degree. Namely, to the largest number (of arguments) for which "slab"-preparing and "slab"-referring macros are previously defined.

3.2.6 The ⟨BDSM⟩'s mystery explained or: how a DFA can recognize a Dyck language ;-)

A ⟨BDSM⟩, "Braced Dyck-language Sequence Manipulation", is an enrichment of an ⟨FSM⟩ with (balanced) braces (of cat.1 and cat.2).

[10] "Let's go to the lab 'n' see what's on the slab", "The Rocky Horror Picture Show".

[11] The need for `\long` is clear. `\edef{\unexpanded` serves to avoid "the hash clash".

As stated above, any closing brace can be followed by ⟨prep.seq.⟩ as if it was a single ⟨digit⟩, in which case the sequence of operators is applied to that entire brace as one argument.

The GMOA machinery can parse arbitrary (re)-grouping specifications, limited of course by TeX's capacity and other resources.

The translation is performed by (a part of) a DFA, a deterministic finite automaton.That is, by a construct whose well-known limitation is its inability to recognize a language of properly paired and arbitrarily nested parentheses. Or, as "naturally" comes to a TeXnician's mind, curly braces; a.k.a. the Dyck language.[12]

Nonexistence of such a DFA is proven *ordine geometrico*. A proof formalizes the intuition that a (given) machine with only finite set of states, say n, cannot properly "count" the braces nested deeper than n levels.

As stated above, each token of a GMOA ⟨specification⟩ is hit with `\string` so it's actually the opening brace $\{_{12}$ ("other") taken into further processing from every ⟨BDSM⟩ chunk.

But — the mystery is unveiled — although the very first and opening brace of a given ⟨BDSM⟩ is hit with `\string` and thus turned to cat.12 "other", and so are all the remaining ones, it's *not* done character by character.

When the automaton meets an opening brace which has already been "petrified" (if we are fans of Platform 9¾) or "denatured" (if we'd like to $\boxed{\texttt{Br}}$eak $\boxed{\texttt{Ba}}$d), it uses this trick (↷ is `\expandafter`, remember?):

```
↷{\ifnum 0=↷`↷}\fi
```

to put an unbalanced $\{_1$ back and then use TeX's argument scanner to pick the entire ⟨BDSM⟩ chunk.

So, it's not the GMOA DFA which "recognizes a von Dyck language" but TeX itself. We have not subverted Computer Science!

But that's just the beginning.

Next, an outermost {⟨FSM chunk⟩} like this is `\detokenize`d and a special delimiter appended to it.

Then, this part of ⟨specification⟩ is parsed again, char-by-char, with the opening brace starting a new branch of (binary) concatenations, terminated with a (unary) operation of bracing when the closing brace is met, with the leafs being ⟨digit⟩s, and rewritten to Reverse Polish Notation (RPN) (with a variant of the Shunting Yard Algorithm, of course).

Once the special BDSM's delimiter is met, the automaton goes to the state "Yield what you've Reverse-Polished" which results in emptying another kind of a "craw" into the main result container.

3.3 An almost-comprehensive usage example (yet not necessarily useful)

Assume (redefining core commands just for brevity):

```
\escapechar \c_minus_one
\def\w{wia} \def\v{\u} \def\t{tro} \def\u{\t}
      \quark_new:N\q_pia
```

and set

```
↷ \showtokens ↷ { \_ᴿ\_ᴿgo:  % debug
      show-commands
\⁝⁝ % <<<<< % My Precioussssssssssss
  ξ h↓↓ %  ξ as in "ξανα", '[use] again'; h as in "hash",
      parameter delimited with #{
  ⁑I  % "Eat the cookie and have the cookie:" take
      an argument from input, yield one copy to the
      result and put another back at input.
  .   %  the limit of ξ's scope.
      %  a new ⟨destination⟩ begins; as the next char
      is not a ⟨destination token⟩, an implicit ς,
      "συναγωγή πολύ", 'gather (as) many' is assumed.
  21   %  take two (undel.) arguments from input and
      put them in reversed order (outer braces off).
  ⁑I
  ξ    %  an explicit ⟨destination⟩ token terminates the
      scope of the previous one.
  io
  (ooo)%  (parentheses) apply all three o's to the same
      argument.
  hii
  `oo. % apply o to an arg. and leave for further
      preprocessors, i.e., for the second o; equivalent
      to (oo).
  2345671 {283}848{58687}1
  28Å871 B̊86871  79C̊79C̊79C̊1.
  11111111. %  equivalent to ⁑I⁑I⁑I⁑I⁑I⁑I⁑II.
  σ 1 4369785 2.
  : %  end of specifiers
%  now the text(s) to be passed through GMOA:
ta␣ metoda␣ nazywa␣ się {␣-↓1}{␣—↓2}
{^^J} % 1
  \w \v   areo {te}{ra}
% 2  3    4    5   6
{ ↷ \use_none:nn \_ᴿ\_ᴿgo:\cs_to_str:N \q_pia}
      % 7
  - ! {tra} {ae} {␣ }
% 8 9 Å     B̊     C̊
{␣}
  ⟦⟧.AaBrsu
% 123456789
^^J^^J
}%  end of text for \showtokens.
```

[12] More precisely, the language recognized apparently "by GMOA's DFA" is a Dyck language's closure with respect to insertions of /(⟨digit⟩⟨prep.seq.⟩)*/'s and concatenations with /(⟨SAlos⟩)*/'s, where "/…/" denotes a regexp.

And the result is (typed out in the terminal):

```
\::_ ->
{\showtokens}
>
ta metoda nazywa się
wiatroareoterapia
{{wia}-{tro}}-{areo}-{{te}-{ra}-{pia}}
wia-tra-pia
ae-ra-pia
pia! pia! pia!
      {⟦A.Bursa⟧}

.
l.43 }
     % end of text for \showtokens
?
```

And the "`\::⍰`-like macros" are (`\escapechar=0␣`):

```
__⁝⁝_prepare'τ⟨ξ⟩:w % the letter τ for Greek "τo
        τέλος", 'the End, the Final Destiny' :3
::h ::↓ ::↓ ::I⁑ __⁝⁝_τ∽yield:w
__⁝⁝_prepare'τ⟨ς⟩:w ⁝⁝_prepare'FSM:w ¨F⋕2 ¨I
        ¨F⋕1 ¨I
q__⁝⁝_FSM'craw∽start 2 % the digit of cardinality
::I⁑ __⁝⁝_τ∽yield:w
__⁝⁝_prepare'τ⟨ξ⟩:w ::i ::o ::o∗ ::o∗ ::o ::h
        ::i ::i ::o∗ ::o __⁝⁝_τ∽yield:w
__⁝⁝_prepare'τ⟨ς⟩:w ⁝⁝_prepare'FSM:w
¨F⋕2 ¨I ¨F⋕3 ¨I ¨F⋕4 ¨I ¨F⋕5 ¨I ¨F⋕6 ¨I ¨F⋕7
        ¨I ¨F⋕1 ¨I ¨Bɛ ¨B⋕2 ¨B¨wrap¨i ¨Bɜ ¨B⋕8
        ¨B¨wrap¨I ¨Bɜ ¨B⋕3 ¨B¨wrap¨i ¨Bɜ ::i ¨F⋕8
        ¨I ¨F⋕4 ¨i ¨F⋕8 ¨I ¨Bɛ ¨B⋕5 ¨B¨wrap¨i ¨Bɜ
        ¨B⋕8 ¨B¨wrap¨I ¨Bɜ ¨B⋕6 ¨B¨wrap¨i ¨Bɜ
        ¨B⋕8 ¨B¨wrap¨I ¨Bɜ ¨B⋕7 ¨B¨wrap¨i ¨Bɜ ::i
        ¨F⋕1 ¨I ¨F⋕2 ¨I ¨F⋕8 ¨I ¨F⋕A ¨I ¨F⋕8 ¨I
        ¨F⋕7 ¨I ¨F⋕1 ¨I ¨F⋕B ¨I ¨F⋕8 ¨I ¨F⋕6 ¨I
        ¨F⋕8 ¨I ¨F⋕7 ¨I ¨F⋕1 ¨I ¨F⋕7 ¨I ¨F⋕9 ¨I
        ¨F⋕C ¨I ¨F⋕7 ¨I ¨F⋕9 ¨I ¨F⋕C ¨I ¨F⋕7 ¨I
        ¨F⋕9 ¨I ¨F⋕C ¨I ¨F⋕1 ¨I
q__⁝⁝_FSM'craw∽start C % the digit of cardinality
__⁝⁝_τ∽yield:w
__⁝⁝_prepare'τ⟨ς⟩:w ⁝⁝_prepare'FSM:w ¨F⋕1 ¨I
        ¨F⋕1 ¨I ¨F⋕1 ¨I ¨F⋕1 ¨I ¨F⋕1 ¨I ¨F⋕1 ¨I
        ¨F⋕1 ¨I ¨F⋕1 ¨I
q__⁝⁝_FSM'craw∽start 1 __⁝⁝_τ∽yield:w
__⁝⁝_prepare'τ⟨σ⟩:w ⁝⁝_prepare'FSM:w ¨F⋕1 ¨I
        ¨F⋕4 ¨I ¨F⋕3 ¨I ¨F⋕6 ¨I ¨F⋕9 ¨I ¨F⋕7 ¨I
        ¨F⋕8 ¨I ¨F⋕5 ¨I ¨F⋕2 ¨I
q__⁝⁝_FSM'craw∽start 9 __⁝⁝_τ∽yield:w :::
{}% the main result container
ta␣ metoda␣ nazywa␣ się ... % the input
```

3.4 Real-life uses of GMOA

The GMOA mechanism can be used anywhere that l3expan can be, and with no need of "generating variants", yet still with shortcuts for the typical cases, as the ⟨specification⟩ is first tested for existence of a predefined macro like l3expan `\exp_args:...`.

Thanks to the `p`/`H` and `h` ⟨prep⟩s, we are able to pick any number of unbraced tokens delimited with an opening brace. That allows for, among other things, creating "semi-transparent" conditionals that disappear in one branch or discarding such `h` sequences in the other. Or just pick an entire left side of an assignment with one `h`/`H` (again, courtesy of the PARCAT project):

```
\⁝⁝ Hr :
\ʢ_pdef:Npn
\PagesTotal
{ % sth. expandable to sth. hopefully useful

  \cs_if_exist:NTF
  \c__ins'_pages'total_int % if a count register /
        _int variable is defined—
  { \int_use:N \c__ins'_pages'total_int }% ...
        then brrrring it on!—
  { \_ᴿstop: \textbf{??} }% ... otherwise expand to
        "We don't know".

}
```

The *mutatis mutandis* multiple definitions in a single piece of code were discussed in section 3.1.

Another interesting use has been noticed only while preparing this paper for print:

The well-known trick,

```
\begingroup
\obeylines %
\firstofone{\endgroup %
  \def
  {⟨sth.useful⟩}%
}
```

(the line end following the last `}` is back of the usual cat.5, i.e., there's *no* line end at all ;-) yet the cat.13 "active" line end has been redefined to ⟨sth.useful⟩) could not be easily applied to things other than catcodes. Such as locally recording a locally changed font size and bringing that local record beyond the scope of the changed font size.

Well, now it can be:

```
\smaller % set the font one-step smaller than current
        \font ((gm)relsize)
\⁝⁝ Io :
{ \group_end:
  \ʢ_def:: \__ʢ'font'size∽smaller: }
\f@size % the inner LaTeX 2ε macro bearing current
        font size is expanded and wrapped in braces
        before closing current group, i.e., before the font
        size is reverted to its previous value.
```

The code above opens a group, then changes the font size (locally), including redefinition of the LaTeX 2_ε

macro `\f@size` that bears the size value in `pt`; then the GMOA preprocessing expands that (locally) redefined macro to its contents, i.e., to the literal value of current font size and (re)wraps it in braces; then puts `\group_end:...` before the expanded and braced literal font size value, thus making it robust to the closing of group and making it the body of the macro `\__ʢ'font'size˷smaller:`.

It's true that that could be written in "pure l3expan", although with different bracing and more `\::⍰`'s:

```
\group_begin:
\smaller
\::N \::N \::o \:::
\group_end:
\ʢ_def:: \__ʢ'font'size˷smaller:
{ \f@size }
```

But what if we also need another macro that bears both the value of the smaller `\f@size` and of the normal `\baselineskip`? (So that the `\acro` command changes only the size of letters and not the line spacing.)

```
\group_begin:
\⁝: ξ {12}o. %  render current baseline skip, wrap it
        in brace and return back to the input —

      21 %  ... and revert the order of braces, stripping
        off the originally-second; these ⟨digits⟩ constitute
        a separate ⟨FSM⟩ with the ⟨destination⟩ implicit
        ς and refer to the rendered and braced value
        of \baselineskip and the brace with the inner
        GMOA.

   :
\the\baselineskip
{
 \smaller
 \⁝: ξ Io.  I  124, 13{45} : %  the last ⟨digit⟩ 5
        refers to the value of (normal) \baselineskip
        rendered by the outer GMOA.

 { \group_end:
   \ʢ_edef::
   \__ʢ'font'size˷smaller:
   \__ʢ'font'sizes˷smaller:
 }
 \f@size
}
```

3.5 GMOA as a part of the **gme3u8** package and *in statu viæ*

We realize that GMOA, along with the whole gme3u8 macro package, with all its "far" and even PUA Unicode usage that require a tailored font and special input methods (such as Elisp functions for GNU Emacs), are not useful for anyone except the author of this article.

However, we intend to make it usable for the others as soon as we get some signals of interest. Indeed, we'll be grateful for any remarks concerning GMOA, especially suggestions for development, and programming in expl3 in general.

◇ Grzegorz Murzynowski
PARCAT.eu
g.murzynowski@parcat.eu
natror@sent.at

Production notes

Karl Berry

As a reader might imagine, editing this article posed unusual challenges. Grzegorz provided his `Ubu Stereo` font (discussed in the text) to make it possible.

To process the article, we used XƎLaTeX, using filename lookups for the fonts: `FreeSerif.otf`, `DejaVuSans.ttf`, `Carlito-Regular.ttf`, `UbuStereo-Regular.ttf`. Avoiding system font lookups allows the article to be processed on different systems without having to change their font configurations, highly desirable for *TUGboat*.

For the actual editing, however, it was necessary to make it work on GNU/Linux, so I edited my configuration file `~/.fonts.conf` to contain the line:

```
<dir>/some/directory</dir>
```

in the `<fontconfig>` block, where `/some/directory` is the directory where I saved `UbuStereo-Regular.ttf`. To my knowledge, all GNU/Linux systems use Fontconfig (`fontconfig.org`) to find application fonts.

To make Fontconfig know about the new directory:

```
fc-cache -fv # | sort >/tmp/fc
```

The commented-out part redirects lots of possibly-but-not-necessarily interesting output from the terminal.

This command shows the names of all the (scalable) monospaced fonts available:

```
fc-list :spacing=mono:scalable=true family |sort
```

Sadly, `Ubu Stereo` does not show up here, as it is technically not monospaced (per the article). Eliminating the `:spacing=mono` selector (i.e., listing all scalable fonts), it does appear.

To use the font in a standard terminal:

```
xterm -fa 'Ubu Stereo' -fs 19
```

The font size (`-fs`) is what worked best on my monitor.

Then I ran GNU Emacs (`gnu.org/s/emacs`) within the xterm: `emacs-nw`. Running Emacs directly under X had complications I didn't need to track down. I used the latest Emacs (24.5), compiled from the original source, as Unicode support is one of the most active development areas in Emacs.

I didn't need Grzegorz's Elisp code (referred to in the article), since I could use the existing unitext.

A final lament: I find that `xterm`, `emacs`, and other programs just drop characters from UTF-8-encoded source when input and font do not match perfectly. What happened to "be liberal in what you accept"? Beware ...