GMS, the ‘General Meta-Scenarios’: A proper extension to the l3expan package of the expl3 bundle and language, two years later

Grzegorz Murzynowski

Abstract

This paper presents the current state of the GM-Scenarios, an esoteric mini-language of one-char instructions covering and extending the functionalities of the l3expan package of the expl3/LaTeX3 bundle.

In an automata approach, the GMSs are described as a DPDA, deterministic pushdown automaton and a respective context-free language, and the arguments (not quite formal) are given to back this point of view.

A diagram of what I believe to be the automaton actually implemented, and a formal grammar that I believe to be a grammar of the GM-Scenarios language, are included.

In the final remarks, I accept the friendly critiques received about the GMSs at TUG@BachoTeX 2017, and reply in an “I’m fixing that” manner.

Contents

Motto:

Pani domu zaś, wydawszy przedtem dokładne wskazówki, sama powinna siedzieć przy stole wesoła i uśmiechnięta, i co najwyżej dawać służbie oczami znaki, gdy tego zajdzie potrzeba.

Concerning the lady of the house, she should sit at the table cheerfully and smiling and, having given the service exact instructions *before*, now give them signs with her eyes only if it is necessary.

Marja Ochorowicz-Monatowa,
“Uniwersalna książka kucharska”, 1910

Disclaimer 1. It’s not Computer Science.

I mean, I hope it to be so metaphorically, i.e., I hope the matter discussed in this paper is “crazy”. But it should be mentioned, and at the very beginning, that this paper is not a scientific article, and not on automata theory in particular. It’s just a presentation of a certain TeX program or set of macros, and the Reader should not be misled by the computer-scientific terms used.

The automaton and formal language presented in this paper might be at the very best considered an example or *exercise*, and the statements, especially those concerning the automaton’s and language’s classes, and computational complexity of the algorithm/program, considered hypotheses to be proven or disproved, or adjusted in their assumptions.

I’m not a computer scientist, i.e., I’m not educated in the theories of Computer Science, I’m just an “aspiring TeXnician” who hasn’t even read the entire *TeXbook* (you don’t use any quotation marks around the name of a Sacred Book, do you?), and just practices TeX in as Epicurean way as he [I] can. The only argument that might accrue to my benefit is that writing programs in TeX gets me my daily bread, and I’m still alive, and, moreover not sued for industrial sabotage or such.

Disclaimer 2. About (Non-)ASCII chars and the tailored font Ubu Stereo.

The expl3 language is kept strictly ASCII. Any characters outside of ASCII that occur in this paper, especially those from the “distant far-aways” of the Unicode, or even from the Private Use Area (**PUA** henceforth), are a sin of “mine, and mine only”.

Their rôle, and the rôle of expl3 in making me use them, is discussed in section 3.

All of them, as coming from different scripts, including the Chinese Traditional ‘記’ U+8A18 ‘write down, record, remember’, and ‘用’ U+7528 ‘use, apply, make use of’, and Math Fette Fraktur, and some even FontForge’d by myself, occur all together just in one (and only one) font in the world, named Ubu Stereo, based on Ubuntu Mono Regular, and enriched with glyphs only from fonts licensed freely.

1 Why again?

This paper follows my presentation at TUG@BachoTEX 2017 and summarizes the present state of the GMS mechanism that I conceived in the beginning of the year 2015 as a “just a bit more friendly interface to the l3expan macros”. While the general concept of the machine (automaton) remains the same, and so do most of its operators and constructs, there are new concepts I have added since, and, which might be more important, “now first it seems my thought begins to span it.” [1]

Which in turn gives the hope that I’ll present the subject in a more understandable manner this time.

Compared to the *TUGboat* 36:3 (2015) paper, the new things are:

- the name of the machinery, GM-Scenarios instead of GMOA;
- a meta-reflection discussed in sec. 3.3;
- shift in understanding from “DFA with Mysterious Something” to DPDA, Deterministic Push-Down Automaton;
- deprecating the pre-processors and pickers “homonymic” with the l3expan counterparts;
- a handful of new pre-ps. and pickers, ‘𝕯𝖉’, ‘ĿŀŁł⌦’, ‘Zz’;
- new aliases for those now deprecated: ‘Aa’, ‘Ää’, ‘Ee’, ‘Ēē’, ‘Ʋʋ’, ‘V̏v̏’,
- the “subs’n’refs” mechanism in FSM s;
- the “arguments from beyond” in FSM s;
- pure-ASCII and HTML-like alternate forms of operators.

What has to be mentioned, or: confessed, is that the GMS machinery is still in a state of development, and at the time of submitting this paper, some of the new things are not quite operational yet.

Much of this intense development comes from the fact I use the GMSs intensely, anywhere I can use my own TEX packages, [the source] of this paper being no exception.

[1] Walt Whitman / Ralph Vaughan Williams, “A Sea Symphony”: “O vast Rondure”.

1.1 The name

Why did I transition from GMOA, “General Manipulation Of Arguments”, to GMS, “General Meta-Scenarios”? The obvious part is, why the two initial (nomen-omen) letters remain the same.

False humility makes me say I should put my name on my work so that Humankind knows who to blame. But that aspect should not be overestimated, Gustav Mahler has the same initials, and were this paper and its dereference of no other use, may the mention of him and his cosmic *The Eighth* advocate me *in die illa tremenda* ☺ .

But, concerning the “G”, the mechanism is in fact quite general. My everyday work is programming in TEX, XƎLATEX to be precise, and I believe the fact I’m still employed by the same Company, and paid, is quite an argument for its usability and usefulness, at least in my hands. Quite general, dare I say, because the first thing it does is cover the functionality of l3expan.

The “GMOA” name focused attention on the ability of the machine to pre-process the arguments for a single expl3-function (usually, a macro). But the GMSs do more than that: they set and rearrange parts of the code before it’s actually run.

Further, because the mechanism operates on the “future” program code, it truly can be called “meta-”. And, because it tells how that “future” code should be executed, truly can it be called “scenario”. And, a “scenario” rather than “didascalia” or “markup”, as it is separated from the code it operates on.

Now, with just a tiny little modal collapse, i.e., the reasoning step “If sth. might be, then let it be”, we get — let the mechanism be called “Meta-Scenarios”, *quod erat demonstrandum.*

2 A brief history of logistic growth of resources or: What do we take for granted

Have you ever read the LATEX 2_ε sources? Thank Heavens, it’s richly commented, and the sub-structures, which in a more usual language would be called “subroutines” and “functions”, or “classes” and “methods”, are presented in pseudocode before they’re actually expressed in TEX.

An excerpt of it is presented in Fig. 1.

Why is it so obscure, why do even the primitives not bear shorter names, not for saving memory (negligible), but for the sake of readability?

LATEX 2_ε has been written in the times when memory was so precious and scarce, that William Henry Gates III, even though he didn’t utter exactly those famous words, actually was thinking, as “all”

```
\def\declare@robustcommand#1{%
  \ifx#1\@undefined\else\ifx#1\relax\else
     \@latex@info{Redefining \string#1}%
   \fi\fi
  \edef\reserved@a{\string#1}%
  \def\reserved@b{#1}%
  \edef\reserved@b{\expandafter\strip@prefix%
       \meaning\reserved@b}%
%<autoload> \aut@global
  \edef#1{%
   \ifx\reserved@a\reserved@b
     \noexpand\x@protect
     \noexpand#1%
   \fi
   \noexpand\protect
   \expandafter\noexpand\csname
     \expandafter\@gobble\string#1 \endcsname
  }%
  ...
}
```

Figure 1: A fragment of the LaTeX 2$_\varepsilon$ source, File d: ltdefns.dtx, 2004/09/18 v1.3g

of his contemporaries, that 640 kB of RAM would be enough for “anything”, and “at least for 10 years”.

In those times “this new TeX format, LaTeX”, had to do some serious “garbage collection” in order to run at all and finish the job.

That’s where `\@onlypreamble` comes from, and that’s why not only was the code written with as few new macros as possible, but also with reusing names, those reuses sometimes irrelevant to their contents and goal.

That’s where DocStrip comes from, whose primary task was to Strip the comments (Documentation) from the files, so as not to slow down their *reading in*.

Also, it was pure TeX not ε-TeX, whose expandable primitive `\strcmp` is neatly wrapped in expl3’s `\str_if_eq[_x]:nn[TF]`, and `\str_case:nn[TF]`, and also with no `\numexpr` or `\dimexpr` that allow expandable integer or dimen computations.

Let us think a moment, if we could write a TeX program or document that (with all the fonts, libraries, macro packages &c.!) works in no more than 500 kB of RAM. Yes, 500 *kilo* bytes, not megabytes.

So, in that time, and in those extreme conditions, that was the “optimum and beyond”. Let’s have this in mind and see what we take for granted in these days’ plenty, and the programmatic indulgence it’s causing.

Now, consider the following fragment of File r: ltfssdcl.dtx, dated: 2005/09/27, v3.0k, giving the definition of `\DeclareMathSymbol`:

```
\edef\reserved@a{\noexpand\in@{%
     \string\mathchar}{\meaning#1}}%
\reserved@a
\ifin@
...
```

It’s the shortest example I’ve found so far of what could be named “repetitive programmatic constructs”.

A macro is `\edef`ed just to put it in the input stream immediately after it’s defined. As can easily be guessed, it’s done to give well-prepared arguments to the macro `\@in`, that checks whether its `#1` is a sub-string, or rather, a sub-tokenlist of `#2`, and sets the Boolean switch `\ifin@` accordingly.

What is at hand at the point we wish to use `\@in` needs to be expanded in a certain way first.

Those “certain ways” of expanding first, and, in my version, not only expanding, we call **preprocessing** henceforth.

This particular schema above, with the macro `\@in` “frozen” with `\noexpand`, first argument consisting of `\string` and a c.s.(1), and the second of `\meaning` and a (supposed) c.s.(2), repeats on previous and subsequent pages quite a few times, just with different control sequences (1) and (2).

And that is just one schema, and the simplest/shortest, of many found just from the beginning till File r.

Then, maybe following The Sources’ example, most (LaTeX 2$_\varepsilon$-style) macro packages and document classes repeat the same manner of repeating those “repetitive programmatic patterns”. Not even with short aliases for `\expandafter` and `\noexpand` (cf. remark 3 on p. 236.)

“Repetitive programmatic patterns”. Do you see the irony? Isn’t the very essence of computer programming to make the machine do the repetitions, if possible?

Let’s repeat: this “if possible” is the answer to the question of why the LaTeX 2$_\varepsilon$ Authors repeat so many things: in those times, not-repeating them was *not* possible. But nowadays, it is. Let’s see what has l3expan got to offer.[2]

For the sake of disambiguation, let’s assume that ‘`#1`’ of the above code is the c.s. token ‘`\life`’. And don’t forget to set the catcode of ‘`@`’ to 11 ‘letter’, because with expl3, ‘`@`’ is ‘other’ by default.

[2] As a topic not fully relevant to the GMS, we skip the discussion on naming ‘`\@in`’ “the expl3 way”, with ‘`:nn`’ signature, and generating its ‘`:oo`’ variant.

Let us recall that the `expl3` catcode regime is that ASCII underscore '`_`' and colon '`:`' are made letters, blanks are ignored, including blank lines, and tilde '`~`' is made space$_{10}$ instead.[3]

In addition, there is Hungarian notation in the "function" names, i.e., the "function"'s signature added in its name after the colon, and other smart naming conventions.

All this results in the first impression, at least mine, "what the [...] is this??", but then, when you get used to it, in growing and growing appreciation of the readability and "spaciousness" of the code. Also, you may see that errors are less likely to occur, and when they do, they are easier to backtrace.

Now, back to the example.

```
[\char_set_catcode_letter:N \@ ]

\::o \::o \:::
  \in@ {\string\mathchar}{\meaning\life}
```

The `expl3`-function `\::o` in the first step gets the respective ⟨balanced text⟩, i.e., the argument, next to "itself", i.e., to the 2nd-step macro. That next macro hits the ⟨text⟩ with `\expandafter` over the opening brace, and then the 3rd macro, that had been put next to `\expandafter`, puts the just-hit argument in the "storage of processed arguments", and passes control further.

As the `\::⍰` macros are essential for understanding further in this paper, let me explain them in more detail. The definitions are translated here to "Traditional TeX". In real `l3expan` they all use the `expl3` aliases and/or wrappers for the primitives.

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

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

So, the one-level expansion of the first `\::o` in the example above, results in:

```
#1 ->\::o %  "the tail of pre-ps. sequence" in general.
#2 ->\@in
#3 ->\string\mathchar

\expandafter \__exp_arg_next:nnn
\expandafter {\string\mathchar }
  {\::o } {\@in }
```

and after the `\expandafter`'s fire, we get

```
(\string)
\__exp_arg_next:nnn {\mathchar} {\::o } {\@in }
```

[\ to illustrate the fact that the backslash, and thus the whole control sequence, is "dead".]

Then `\__exp_arg_next:nnn` restores the order. Let's see that in slow motion.

```
\__exp_arg_next:nnn #1#2#3 -> {#2\::: {#3{#1}}}
#1 ->\mathchar, %  no space after the former c.s. is
        another sign it's "dead"
#2 ->\::o ,
#3 ->\@in
```

and thus we get:

```
\::o \::: {\@in {\mathchar}}
```

plus what was already there,

```
    {\meaning \life }
```

After performing two-level expansion of "this other `\::o`", i.e., replacing it with its definition, and firing the `\expandafter`'s, we get

```
(\meaning )
\__exp_arg_next:nnn {> \mathchar"2A.} %  \life is
        usually '> undefined', but once you give it some
        Deep Thought... ☺
    {} {\@in {\mathchar}}
```

and after expansion of `\__exp_arg_next:nnn`,

```
\::: {\@in {\mathchar}{> \mathchar"2A.}}
```

Now, the mysterious Triple Colon[4] Macro `\:::` that served as the delimiter of the tail of the preprocessors in/for the `\::⍰`'s ("Double-Colon-with-a-Letter" Macros). After all the `\::⍰`'s have expanded, it comes out as the "identity" macro, `\@firstofone` from LaTeX 2$_\varepsilon$:

```
> \:::=\long macro:
#1->#1.
```

and the braces covering the main macro and all the pre-processed arguments disappear, as if the End of Time came "and all things previously hidden are now revealed":

```
\@in {\mathchar}{> \mathchar"2A.}
```

This way, we saw "in slow motion", how the "repetitive programmatic patterns" found in the LaTeX 2$_\varepsilon$ sources and LaTeX 2$_\varepsilon$-style macro packages and document classes, might be vastly simplified and made more readable using the macros defined in the `l3expan` package.

And "here comes Mommie!"[5] — here come I, and say: look at those repeating backslashes and colons. What if we delegate repeating them to the machine, and we ourselves type just what's essential, i.e., the final letters?

[3] Not "ASCII tilde of catcode 10 'space' ", because it's not made "funny space" of the character code 0x7e via the `\lccode` trick, just honestly via `\catcode=`, ifx you know the difference.

[4] A Polish cartoon series "Kapitan Bomba" gives the term "triple colon" quite *another* meaning while describing the "Kurvinox" alien species' anatomy.

[5] Patrick Swayze in [spoiler alert] the opening scene of "To Wong Foo, thanks for everything, Julie Newmar".

3 The *inspiratio*: l3expan

3.1 The Pandora's box of new letters

Let me make another apparent digression, which is in fact an important explanation I owe the Readers and, maybe even more, the LaTeX3 Team.

Having the guts and nerve to abandon the Plain and LaTeX 2ε convention of making '@' catcode 11 'letter', and to make letters of '_' and ':' instead, is a huge inspiration and broadens my mind horizons, comparable with some kind of spiritual enlightenment, or with Dostoyevskian "Но есьли Бога нет, тогда всё довольно…" ['But, if there's no God, then anything is allowed…'].

I perceive this bold move as the main inspiration for my own attempts to "think out of the box". In my implementation it comes out more like Pandora's box, as has been kindly and amiably pointed out to me at this BachoTeX, as I'll discuss later.

In order to observe the naming conventions of expl3, especially the division of a c.s. into "scope" and module parts, and seeing the need for a more structured module part, I chose the letter 'ˈ' (Modifier Letter Vertical Line, U+02C8), with catcode 11 'letter' in XeTeX (as it is in Unicode), as the "unofficial" word separator.

```
\__gmeˈint_…:…
```

denotes a module-local LaTeX3-function of the module 'gme', and its submodule 'int'.

I use yet another letter, 'ᔥ', U+1525 Canadian Syllabics SH, as the character separating the final part of a c.s., intended to describe its particular role in a multi-step construct, or in a family of macros, e.g.:

```
\💎_makeˈgay:nnn
\💎_makeˈgayᔥtheˈYuletide:nn
\💎_makeˈgayᔥGigiˈtoday:nn
```

On the other hand, choosing a character rare enough so it could serve as an ideogram, like '⚸' U+26B8 Black Moon Lilith, or the one discussed next, let us structure the module part of names just with it and abbreviated description, like

```
\⚸int_…  \⚸tl_…  \⚸dim_…
```

which I use to name my additions and "completions"[6] to the expl3 respective modules.

As you see, it becomes "…тогда всё доволно" ('…then anything is allowed'), indeed. But — for the good cause of brevity, as brevity means better readability.

3.2 "Let's make it shorter and don't repeat…", or: how the GMSs began

Going still further in this direction, I make a letter also of '⋮', U+22EE, Vertical Ellipsis (there's also the Triple Colon character, U+2AF6, maybe it would be preferable), to get a symbol similar and referring to the Double and Triple Colons of l3expan, yet at the same time saying "I differ from them, be careful, I might be orthogonal!"

As already mentioned, one of my goals is to make code short and as free from repetitions as possible. This attitude resembles that of Webern towards music, toutes proportions gardées. And so with the TeX code making use of the GMS.[7]

Returning to the main narrative, let's replace Double-Colon-with-a-Letter s with Just-a-Letter s, i.e., let's trim leading '\::', and precede the whole thing with a two-(newly-declared)-letter word \⋮⋮. Since I'm "thinking in type not in sound", I've no idea how to pronounce this ideogram. What first, or rather *who* comes to my mind, is Aja.

```
        \::o \::o \:::
          \@in {\string\mathchar }{\meaning\life }
  \⋮⋮ I   o    o    :   …
```

And that's what the core and chronologically earliest part of the GMS does. Translates a sequence of letters into the sequence of l3expan \::⍰'s.

One thing that needs explaining is the letter 'I', which doesn't correspond to any of the l3expan "Double-Colon-with-a-Letter"s of the line above.

I found it a bit confusing in the l3expan convention that the first token, i.e., the assumed "function" for which the assumed arguments are pre-processed, is not reflected in the DCwL s, or rather is, but at the very end, in the Triple Colon. While thinking of what's going on here not as preparing arguments for a "function", but as a sequence of operators applied to a sequence of ⟨text⟩s (cf. sec. 3.3, p. 224), it becomes clear that the Identity operator is missing in the l3expan notation, and that's the 'I' in mine.

Then the desire for Symmetry wakes up and joins in, another monster conceived of LaTeX3 *inspiratio*,[8] namely, from looking at the "data types"

[6] The fact that some of my views, such as expecting an Esperanto-like symmetry from "the new LaTeX programmers' language", e.g., the Boolean constants, i.e., '\bool_const:Nn', just like there's any other '\⟨type⟩_const:Nn[…]', diverge a bit from what's actually there in expl3, neither does make it "incomplete", nor diminishes the utter respect and gratitude I have for its Authors. Hence the quotation marks.

[7] Typical reaction of a person listening to "normal" [Western] music at first hearing of a piece by Webern is: "What?? It's not music, it's some separate and random sounds!" The analogy with GMS holds.

[8] The Latin verb "inspiro" means 'to breathe into [something/someone]', and is used in the Vulgate and hence in the Christian narrative to describe G*d giving life to the first human after making his body out of earthly dust or clay,

and their declarators, setters, and naming conventions — a desire to make the pre-processing mechanism fully symmetric with respect to braced- and unbracedness of the results, as the `N`- and `n`-type arguments and l3expan pre-processors already do. Apparently. And, not quite, as Frank Mittelbach explained himself at BachoTEX 2015, correcting my (mis)understanding and thus unmasking the idea of "braced/unbraced" *result* to be "mine and mine only".[9]

While the original expl3's idea of `N`- and `n`-type arguments refers to un- and bracedness of the *arguments*, i.e., the *input*, my (mis)understanding, and the idea stemming from it in GMS, refers to un- and bracedness of the *results*, i.e., to whether the pre-processed ⟨balanced text⟩ should be "returned" in braces: when the letter is lowercase, or without them: when the letter is uppercase.

Therefore, "I've made my decision"[10] to deprecate the l3expan operators (letters), and maybe turn off handling of them in the future.

So, the convention of GMS is that the lowercase letters correspond to "returning" the pre-processed `{`⟨text⟩`}` in braces, while the uppercase "return" the ⟨text⟩ unbraced, which might seem strange at first glance, e.g., when the pre-processor hits a multi-letter control sequence with `\string`, but is useful, e.g., in defining a macro with parameters delimited with detokenized ('other'-ised) characters, e.g., ~~`pt`~~ or ~~`macro ->`~~ .

The idea of a set of operations closed in some aspects is usually a good idea, unless we are not scarce of memory or time, and these days we are not, oh, no, we aren't.

Also, while opposing or at best "orthogonal" to the concepts and conventions of expl3 at first glance, this `{`⟨lower⟩`}`/⟨upper⟩ convention fits with the more general paradigm of making the language fully regular, much as Esperanto is.

For those two reasons, i.e., not to mislead from the original expl3 concepts, and to observe the `{`⟨lower⟩`}`/⟨upper⟩ *result* convention, I replaced the ASCII lowercase '`o`' with the Latin lowercase a, '`a`', and thus the example use of `\⁝⁝` should be rewritten to:

```
      \::o \::o \:::
         \@in {\string\mathchar }{\meaning\life }
  \⁝⁝ I   a    a     : …
```

But "that's [not] all, folks", since the goal is to make the code short. And `\string` is used so often that it's worth its own pre-processor. And here it is: 's'/'S'. Also, `\meaning` might be very useful in some applications, and although I personally haven't yet had such a need, introducing a new pre-processor for it is a matter of five minutes of adding the respective macros, plus [indefinite time] to choose the letter/symbol.

I chose '𝖉' and '𝕯', having in mind that '`m`' stands in the ancient xparse, and also in gmcommand, for 'mandatory argument', and perhaps the most famous question about meaning is Freia's "Was deutet die Name?" in the finale of "Das Rheingold". Then, math Fraktur because ASCII '`D`' is already taken by the expl3 "Don't" pseudo-signature.

With these "particulations", the example turns into a five-token GM-Scenario with one mandatory separator char between '`\⁝⁝`' and the pre-processors (remember it's not a space$_{10}$ in the expl3 or gme3u8 catcode regimes, it's an ignored char [space]$_9$, so it's not even officially read, yet it establishes the limit of a multi-letter control sequence, without which it would be '`\⁝⁝Is`' (a bit like, say, alcoholic addiction of a parent, which is not talked about, yet keeps the children from inviting friends home), and just three tokens of things processed. Let's put it all together:

```
  \edef\reserved@a{%
   \noexpand\in@{\string\mathchar}{%
       \meaning#1}}%
  \reserved@a                    % The LaTeX 2ε sources

  \::o \::o \:::
  \@in {\string\mathchar}{\meaning\life }% expl3

  \⁝⁝ Is𝖉 : \@in \mathchar \life        % GMS
```

Note that the 2nd and 3rd ⟨text⟩s are written without braces, since they are not necessary either for syntax correctness or for clarity, the latter provided by simplicity of this particular GMS.

By the way, all component macros and primitives of l3expan's "function" `\::o`, and thus also of GMS's `\⁝⁝ …a… :`, are expandable.

This is the case with all the l3expan and GMS pre-processors, if only their very nature allows it, i.e., if the pre-processing does not involve an assignment (other than this one and only assignment of `\relax` to a c.s. raised by '`\𝕿›… \𝕭 `').

As mentioned earlier, l3expan provides various types of arguments' pre-expansion.

The `\::f` preprocessor applies ``\romannumeral-`0``, which expands argument tokens until the first unexpandable token is seen. Because ``-`0`` is a complete ⟨number⟩ in the sense of *The TEXbook*, even if the argument expands to digit(s), `\romannumeral` is satisfied with the ``-`0`` and, as this number is negative (minus the character code of character '`0`', namely −48),

and also, conceiving the Child by the Most Venerable Virgin "from G*d's Spirit".

[9] cf. "Evita", "Eva's Final Broadcast".

[10] cf. "RuPaul Drag Race".

expands to ε (empty sequence of tokens). Thus we get an "AFAP" ('As Far As Possible') expansion. Not many things move me as deeply as this trick.

Some of the pre-processors rendering the value of a LaTeX3 variable also use `\romannumeral-`0`, depending on the LaTeX3 variable's type. The current implementation of the `_tl` type, for instance, as parameterless macros and not, e.g., as `\toks` registers, allows for rendering their values with just an '`\⸮`' (`\expandafter`), or even using that LaTeX3 variable "as is".

So far, the things described might be considered just another user interface for l3expan, maybe more user-friendly, if anything. Also, the operators that l3expan "lacks" are just superpositions of one that already exists, most often the [`o`]/`a`, with some of TeX's expandable primitives, like `Đð` for '`\::o∘\the`', `\::ð…( · )` == `\::o…(\the( · ))`.

So, what are the *real* enhancements I've made?

3.3 GMS as a nano-Copernican revolution (against l3expan (?))

Besides "stripping off one backslash and two colons" shown in the previous section, probably the most important enhancement made by me (if it may at all be attributed to me with such a strong inspiration[11]), and fundamental for any further development, is a change of the point of view, quite Copernican in this nano-scale:

Thinking of (l3expan and) the GM-Scenarios not as pre-processors of (individual) arguments for an expl3 function or macro, but —
as a sequence of operations applied, not necessarily 1:1, to a sequence of ⟨text⟩*s, where* ⟨text⟩ *is an undelimited or delimited argument of a resp. macro.*

"(?)" in the section's title, because it's not very likely that expl3's Authors do not realize the power of l3expan, rather they deliberately abstain from using it in its full glory (*if* they do in fact), and the "revolution" declared above is more of a shift of my own *understanding* of l3expan. Like Dr. Pierre Abelard says in the preface to his famous "Sic et non": "It's rather us lacking G*d's grace [ability] to read [and understand] than them [the Authors] lacking G*d's grace [ability] to write."

Let's go back to the excerpt from the LaTeX 2_ε sources and think of it this way: in the end, we wish to give TeX a two-parameter (undelimited) macro `\@in` with both of the arguments hit with some expandable operators.

What would be (conceptually) done, is:

1. hit the 1st argument with `\string`,
2. hit the 2nd argument with `\meaning`,
3. prepare `\@in` to go first,
4. put the tokens resulting from 1 next to `\@in` to go 2nd,
5. put the tokens resulting from 2 next to those of 4.

What if we wish to prepare the arguments as described above, but instead of knowing the "function" they are for already, give a placeholder for it and be able to pass any relevant macro "later", i.e., as an argument?

Or, if we wish to pass those arguments twice, for two different "functions"? For instance, having a c.s.(1) and an ⟨integer expression⟩(2), first declare (1) as an '`_int`' variable, and then initialize it with (2)?

One of the "basic needs" l3expan does not satisfy, is — changing the arguments' order. Another is replicating them, as LaTeX 2_ε's `\@dblarg` does, for instance. And yet another, grouping ⟨text⟩s together in one common pair of braces. The latter two actions are the reason why I mentioned it's not always a 1:1 correspondence.

Consider the following GMS:

```
\def \what's'the'Question #1 {
   \⁝⁝  ♮ 3̳ 1̳s 2̳ð : \mathchar \life #1  }
```

Knowing that the Musical Natural [Pitch] Sign '♮' declares a "natural permutation", I hope it's clear, what those "underlined digits" mean. How many of them might there be, i.e., how many ⟨text⟩s can an `\⁝⁝` handle at a time? Currently, up to 25, referring the first 9 with 1̳…9̳, and then with A̳…P̳, for the "uppercase" pre-processing, or {1}…{9}, {a}…{p} for "lowercase".

But anyway, the last operator counts for the un- or bracedness, so even though the '3̳', '1̳', and '2̳' are all "uppercase", only the ⟨text⟩ referred to as '3̳' is rendered without braces, while the other with, because of the lowercase '`s`' and '`ð`'.

And this, and other features, are worth a section of their own, so —

4 GMS: the automaton

The above example of changing the order of arguments is rather simple. And, seeing what is referred to as '1̳', '2̳', and '3̳', poses no problem.

But when there are more ⟨text⟩s to make a permutation of, it seems more reasonable to label them with some numbers. The following example uses

[11] It's infinitely easier to expand/develop something than to invent it in the first place. l3expan does things I've never thought of in the 11 years of my TeXnician's life. Or, if I ever 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 there are in the first one".

the characters ‘Opening Lenticular Bracket Ordinal Number Omega’, ‘⦅ω’, put at PUA+E9EA, and ‘Closing Lenticular Bracket Ordinal Number Omega’, ‘ω⦆’, at PUA+E9EB. (Both of them designed by me, together with the plain ‘Ordinal Number Omega’, ω, PUA+E9E9).[12]

‘⦅ω’ starts a part of a GMS being a permutation of ⟨text⟩s with possible repetitions and grouping, and additional pre-processing of them. Let’s call it: **Finite Sequence Manipulation**, henceforth **FSM**.[13]

‘ω⦆’ delimits the (explicitly labelled) sequence of ⟨text⟩ that’ll become the **elements** of this FSM, so that all may be absorbed by an internal macro delimited with this char.

(By the way, those two chars are declared in my Emacs as matching parentheses, so they are highlighted properly.)

```
[\gme'''on] %  set the gme3u8/expl3 catcodes
\pdef %  \protected\def
\make'exhyphen #1 {
  \⁞ I ⦅ω 1 ⟦ 2Ä=:2 ⟧216 3{42{5}} : %  a \GMS
  \lccode
  1 `
  2 \c_catcode_active_tl
  3 \lowercase
  4 {\protected\def }
  5 {\penalty\exhyphenpenalty \hskip%
      \c_zero_skip }
  6 #1 %  two tokens in definition, but replaced with a
      single char in runtime.
  ω⦆
  \ʢ_catcode:: `#1\ʢ_active::
}
[\gme'''off]
```

What we consider the automaton in this paper, is a set of macros that pass control to each other subsequently, just like states and transitions of a deterministic push-down automaton, and initialized by the macro `\⁞` (or another of the family, as will be discussed later), and finished by a colon.

In the example above, the GM-Scenario is the code commented as such, starting with ‘`\⁞`’ and finishing with ‘`:`’, if we take the syntactic approach, or all the code starting with `\⁞`, and finishing with ‘ω⦆’, if we look semantically. (It is not always clear where a GMS in the latter sense ends, since its instructions might be determined dynamically in the “runtime”, thanks to the “interᴿuptions”.)

The letter ‘`I`’ refers to the first ⟨balanced text⟩ following the colon, i.e., `\lccode`, and says ‘just take it and return as is, but without braces’. Then, the ‘⦅ω’ sign declares a **labelled FSM**, which means, that the automaton should now expect a permutation, possibly with repetitions, additional pre-processing, and grouping, of a certain number of ⟨balanced texts⟩, and that those ⟨texts⟩ have been already labelled, i.e., each of them preceded by proper alpha-digit (not necessarily starting from 1 and keeping the “increment by 1” rule), and that after all those texts there’s the delimiter ‘ω⦆’, so that absorbing all the permutation elements is performed in a one-level expansion of a macro with a ω⦆-delimited parameter.

Then, the fragments ‘`1`’, ‘`216`’, and ‘`3`’ are translated into macros “get the element with the label (·)”, and put those elements in given order. Only, before the element with the label ‘`2`’ is taken, the translation of the fragment ‘`⟦ 2Ä=:2 ⟧`’ hits that element with the “double `\expandafter`”, and replaces the original with the result of that 2-level expansion. We’ll discuss this in more detail in section 5.7.

Then, the part ‘`42{5}`’ translates into grouping the elements ♯4, ♯2, and ♯5 in one pair of braces together, and the element ♯5 within braces by itself. We discuss the grouping (bracing) mechanism, and its consequences in terms of the hierarchy class of the automaton, in section 5.6.

4.1 The automaton: diagram

Presenting the GM-Scenarios automaton, we use the following conventions:

- the symbol “`_•`” denotes ‘an arbitrary representative of the class •’, where “•” is the (usually one-character) name of the class, which may be homonymic with one of its members, or even the only one. One can think of it like an abbreviated Ruby Manual convention of typing an instance of a class, say, ‘`Array`’ as ‘`an_Array`’, just with the preposition stripped off; or, as sort-of conforming to the `expl3` convention of indicating the type of a variable (a data carrier) with underscore and type at the end of its name.
- “↓·” means ‘push (·) down the stack’, where (·) is 1 for an opening brace, or *i* for an opening ‘⟦’ bracket, which is used to mark the start of ⟨subs’n’refs⟩, and matching with ‘⟧’.

[12] The Ordinal Number ω not the Cardinal $\aleph_0$ because in the context of permutations, the ordering is fundamental, and the proper name for the Cardinal $\aleph_0$ in its Ordinal aspect is ω.

Then, ω not some arbitrary symbol, because “Sky is the limit”, theoretically number of ⟨text⟩s handled in an FSM is arbitrary (finite), and limited only by the capacities of the hardware and Time of our Universe, as the computing complexity of this part of GMS’s seems to be at least $O(n^2)$ Time, and at least $O(n)$ Space.

[13] The religious allusion of this acronym is deliberate, may He be always *al dente*.

- "↑: 1¦0¦i" 'pop from the stack, and then there's 1¦0¦i resp. on top of the stack'. Note that the symbol i might be considered the initial symbol for the ⟨subs'n'refs⟩ sub-stack.
- "[⟨action⟩]" 'an action, usually assignment, performed "sideways" with a default value, in case of a transition that skips some intermediate state(s).
- "$\boxed{\text{-}\bullet}$" 'I'm not a usual label, in fact, I'm a sub-automaton'. Both of those sub-automata are depicted in the same figure.

5 GMS: the formal language, and program

There's a theorem about a correspondence between finite automata and formal languages, namely, coupling a formal language with an automaton that recognises it.

And since a formal language might be described with a formal grammar, there's also a natural correspondence between automata and formal grammars, namely, that $A \sim G$ iff $\exists L$ such that L is the language recognised by automaton A, and at the same time is defined by formal grammar G.

In this sense, the automaton depicted in fig. 2, and the grammar described in fig. 3, do correspond with one another.

Therefore, henceforth, I'll be describing the language, freely switching between its formal grammar, and the automaton recognising it.

And, since this is not Computer Science, just a presentation of a program, I'll be also explaining how the program works, which from the point of view of Automata Theory is all "side effects" at best.

5.1 The ⟨\⁝⁝ macro⟩ and ⟨specification⟩

The tokens of a ⟨specification⟩ are hit with `\string` one-by-one so they get catcode 12 "other", except those expanded within an "interᴿuption".

The $\{_1$ and $\}_2$ tokens may be used as they are, and if the ⟨\⁝⁝-macro⟩ works straightforwardly (only \⁝⁝ does, as for now), they don't even have to be balanced.[14] Also, their "rôle" might be played by ⦃ and ⦄, as they're recatcoded to letters in the `gme3u8` catcode regime, and might be translated in macros that work faster than the main GMS machinery.

Since the main iterating macro has one undelimited parameter, even in the usual catcode regime the blank chars are skipped and may serve just to improve readability. Of course, in the `expl3` regime, they are already ignored on the TEX reading level.

\⁝⁝ is a parameterless macro of the initial state, called `KN`, '[I] Know Nothing'. It first `\expandafter`s `\string` and then launches the macro. '⁊⟬' is a one-letter c.s. equivalent to `\csname`, Tironian Et U+E970 '⁊' another escape char, and '⁊⤼' is `\expandafter`:

```
\⁝⁝ -> ⁊⟬ ⁊⤼ \__⁝¹_s¨KN:N \string
```

Other ⟨\⁝⁝-macro⟩s do the same at some point, only first they absorb an entire ⟨specification⟩ as an argument delimited with '`:`'$_{11}$, and either check if such a specification has already been parsed and recorded (predefined), and use the pre-defined if so, or perform the predefinition instead of reorganising subsequent ⟨text⟩s, or, \⁝⁝_用記'…, are control sequences which the ⟨specification⟩ is part of.

We'll discuss the basic version, '\⁝⁝'.

The subsequent characters of ⟨specification⟩ are hit with `\string` and picked one-by-one, their "charclass" is determined, and proper transitions are performed accordingly, which TEXnically amounts to inserting 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 seen. Except, the immediate predecessor of such an `\endcsname` is `\expandafter`, and the token next to `\endcsname` is another `\csname`:

```
\csname name-1 \expandafter\endcsname
\csname name-2 …
… \endcsname
```

I think of this trick as an (architecture) arc or a 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 Ubu Stereo/ PUA signs based on Japanese quotation marks:

```
⁊⟬ name-1 ⁊⟭⟬ name-2 ⁊⟭⟬ … ⁊⟭
```

So, as a "side-effect" of parsing of the GMS of section 4, a translation is made:

```
⁊__⁝⁝_prepare'τ⦃ς⦄:w ⁊::I
⁊⁝⁝_prepare'FSM'⟅ω:w ⁊¨F♯1 ⁊¨I
⁊__⁝⁝_subs⁽ᵗ⁾refs〰start ⁊¨⦃0⦄
  ⁊¨F♯2 ⁊¨in'F: ⁊¨BÄ ⁊¨I ⁊¨subs⁽ᵗ⁾refs〰↓↓ ⁊¨⦃0⦄
  ⁊¨F¨2≔ ⁊¨subs⁽ᵗ⁾refs〰↓↓
⁊__⁝⁝_resume'FSM ⁊¨⦃0⦄
⁊¨F♯2 ⁊¨I ⁊¨F♯1 ⁊¨I ⁊¨F♯6 ⁊¨I ⁊¨F♯3 ⁊¨I
⁊¨Bε ⁊¨B♯4 ⁊¨B🖂 ⁊¨Bɔ ⁊¨B♯2 ⁊¨B🖂 ⁊¨Bɔ
⁊¨B♯5 ⁊¨B🖂 ⁊¨Bɔ ⁊¨¶i ⁊q__⁝⁝_FSM'craw〰start 6
⁊__⁝⁝_τ〰yield:w ⁊::: {}
```

where '⁊' symbolizes the '⁊⟭⟬' arch in the "⁝¹-run" (parsing-translation), and then, after turning them

[14] To be precise, each token of the GMS-charclass '{' has to be balanced with a token of GMS-charclass '}', but the catcodes are irrelevant now as all the tokens are hit with `\string` one-by-one. Also, it's possible, e.g., to generate missing }'s in an "interᴿuption" using `\Ucharcat`.

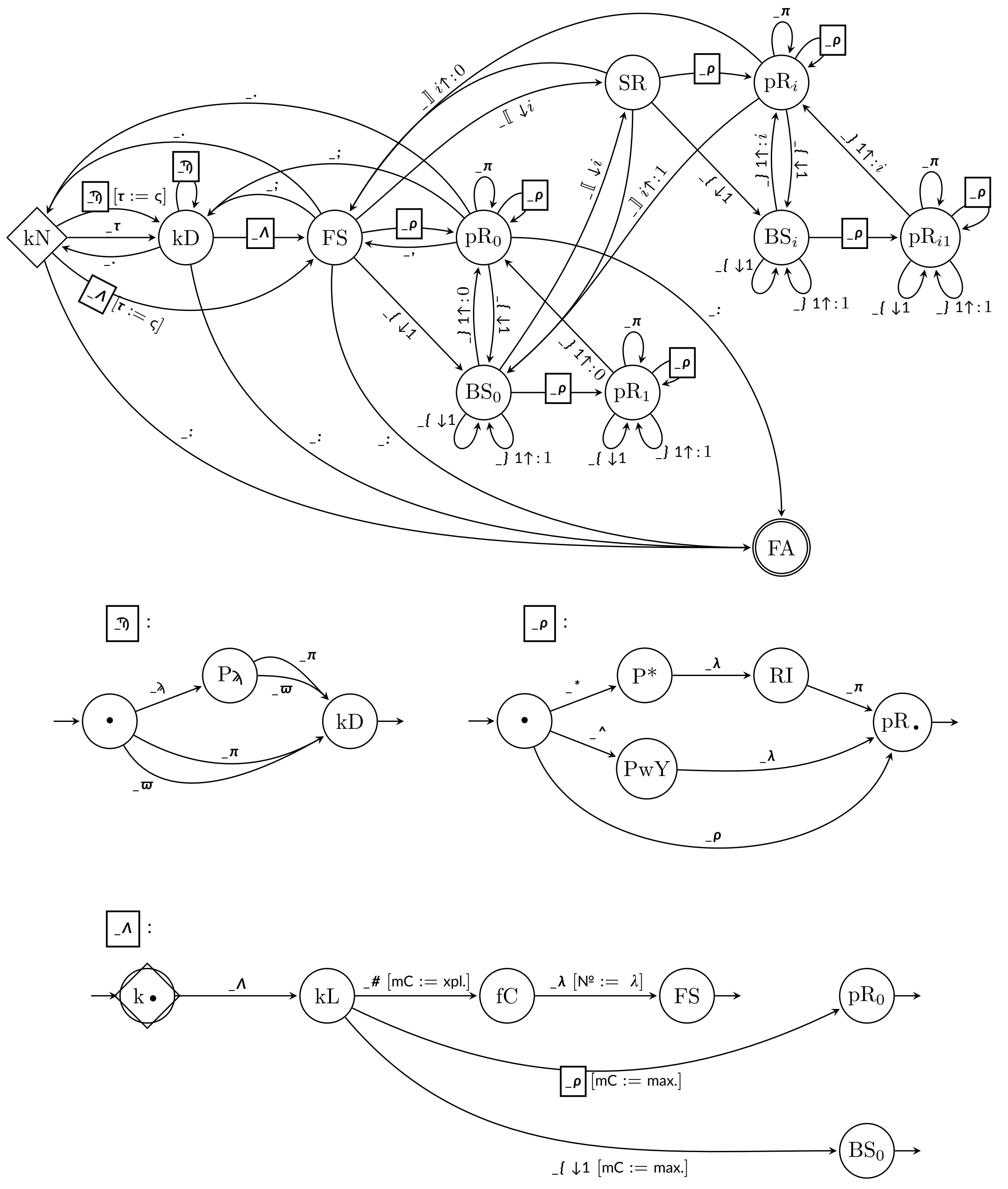

Figure 2: The GM-Scenarios deterministic push-down automaton.

GMS two years later. A complete madness. But — Turing-complete or not?

The meta-conventions and symbols defined elsewhere:

- ⟨opt. ⟨punct.x⟩⟩ – 0 or 1 punctuation character x, $x \in$ {'`.`', '`;`', '`.`'}
- + ᴿ – "interᴿuption" – ᴿ⟨an ᴿ-code⟩¦ˣ⟨args. for `\prg_replicate:nn`, with `#2` a sub-⟨specification⟩⟩ ¦ᴮ⟨expl3-⟨Boolean variable⟩⟩⟨T-arg.⟩⟨F-arg.⟩¦ᵇ⟨expl3-⟨Boolean expression⟩⟩⟨T-arg.⟩⟨F-arg⟩¦[an `&ASCII;`]
- + `&ASCII;` – right side of the rule should be doubled with the pure-ASCII, HTML-entity-like aliases of the symbols just-listed, in the form of `&`⟨pure-ASCII alias⟩`;`, and also with '`&U+`⟨hex⟩`;`' aliases, e.g., `ð` ≡ `&{the};` ≡ `&U+00F0;`, `Đ` ≡ `&the^;` ≡ `&U+00D0;`,
- ⟨an ᴿ-code⟩ – a(ny) TEX code that ᴿ-expands to a part of a ⟨specification⟩, where ᴿ denotes the `\romannumeral -`0` trick.

The grammar:

⟨GMS⟩ ::= ⟨\⁝⁝-macro⟩⟨specification⟩`:`
⟨\⁝⁝-macro⟩ ::= `\⁝⁝␣` | `\⁝⁝_ˀ記ᴱᵗ用記␣` | `\⁝⁝_記記␣` | `\⁝⁝_用記␣` | `\⁝⁝_用記'` + `&ASCII;` + ᴿ
⟨specification⟩ ::= ⟨1st dest.⟩⟨FSoO⟩⟨subseq. specification⟩
⟨1st dest.⟩ ::= ε | ⟨τ⟩
⟨τ⟩ "το τέλος", 'destiny/destination' ::= `ξ` | `ς` | `σ` + `&ASCII;` + ᴿ
⟨subseq. specification⟩ ::= ε | ⟨dest.reset⟩⟨FSoO⟩⟨subseq. specification⟩
⟨dest.reset⟩ ::= `.` | ⟨opt. `.`⟩⟨τ⟩
⟨FSoO⟩ "Finite Sequence of Operators" ::= ε | ⟨SAlos⟩⟨opt. `;`⟩⟨FSoO⟩ | ⟨FSM⟩⟨opt. `;`⟩⟨FSoO⟩
⟨SAlos⟩ "Stand-Alone's" ::= ⟨($\pi^*\varpi^*$)*⟩ | ⟨SAlos⟩⟨ƛ⟩⟨π⟩⟨SAlos⟩
⟨($\pi^*\varpi^*$)*⟩ ::= ⟨π^*⟩⟨($\pi^*\varpi^*$)*⟩ | ⟨ϖ^*⟩⟨($\pi^*\varpi^*$)*⟩
⟨π^*⟩ "the pre-p's" ::= ε | ⟨π⟩⟨π^*⟩
⟨π⟩ "a **p**re-**p**rocessor" ::= `c` | `f` | `n` | `N` | `o` | `T` | `F` | `v` | `V` | `x` % l3expan's, deprecated
  | `A` | `a` | `Ä` | `ä` | `Ć` | `ć` | `Đ` | `ð` | `𝕯` | `𝖉` | `E` | `e` | `Ē` | `ē` | `I` | `i` | `K` | `k` | `Ṅ̡` | `Ṉ́` | `R` | `r` | `S` | `s` | `V̕` | `v̕` | `V̏` | `v̏` | `Z` | `z`
  | `Ṅ` | `ṅ` | `Ḋ` | `ḋ` | `Ḟ` | `f̈` | `Ġ` | `ġ` + `&ASCII;` + ᴿ
⟨ϖ^*⟩ "the special pickers" ::= ε | ⟨ϖ⟩⟨ϖ^*⟩
⟨ϖ?⟩ "optional picker" ::= ε | ⟨ϖ⟩ | i | I + ᴿ
⟨ϖ⟩ "a special **p**icker" ::= `p` % l3expan's, deprecated
  | `H` | `h` | `Ħ` | `ħ` | `Hꜝ` | `Ɨ` | `ɨ` | `Ŀ⟨$\delta$⟩` | `ŀ⟨$\delta$⟩` | `Ł⟨$\delta$⟩` | `ł⟨$\delta$⟩` | `⌦⟨$\delta$⟩` | `Q` | `q` | `Q̧` | `ꝗ` + `&ASCII;` + ᴿ
⟨ƛ⟩ "a star prefix" ::= ⁎ (Low Asterisk U+204E) | ⁑ (Double Asterisk U+2051) + `&ASCII;` + ᴿ
⟨FSM⟩ "Finite Sequence Manipulation" ::= ⟨labellity⟩⟨opt.cardinality⟩⟨FSM w. mCard.⟩
⟨labellity⟩ ::= ε | ⟨Λ⟩
⟨Λ⟩ ::= ♮ (musical Natural sign) | ω | 𝕎 + ᴿ
⟨opt.cardinality⟩ ::= ε | `|`⟨λ⟩`|`
⟨FSM w. mCard.⟩ "FSM with [known] **me**ta**Card**inality" ::= ⟨FSM chunk⟩⟨opt. `,`⟩ ⟨FSM w. mCard.⟩
⟨FSM chunk⟩ ::= ⟨($\varrho\pi^*$)*⟩ | ⟨BDSM⟩ | ⟨subs'n'refs setting⟩
⟨($\varrho\pi^*$)*⟩ "**r**ender-pointers w. **p**re-**p**rocessors" ::= ε | ⟨ϱ⟩⟨π^*⟩⟨($\varrho\pi^*$)*⟩ | ⟨$\varrho\backslash r$⟩⟨π⟩⟨π^*⟩⟨($\varrho\pi^*$)*⟩
⟨ϱ⟩ "a **r**ender-**p**ointer" ::= `1̳` | `2̳` | `3̳` | `4̳` | `5̳` | `6̳` | `7̳` | `8̳` | `9̳` | `A̳` | `B̳` | `C̳` | `D̳` | `E̳` | `F̳` | `G̳` | `H̳` | `I̳` | `J̳` | `K̳` | `L̳` | `M̳` | `N̳` | `O̳` | `P̳`
  | `{1}` | `{2}` | `{3}` | `{4}` | `{5}` | `{6}` | `{7}` | `{8}` | `{9}` | `{a}` | `{b}` | `{c}` | `{d}` | `{e}` | `{f}` | `{g}` | `{h}` | `{i}` | `{j}` | `{k}` | `{l}` | `{m}` | `{n}` | `{o}` | `{p}` + ᴿ
  | `^`⟨λ⟩ | `_`⟨λ⟩
⟨$\varrho\backslash r$⟩ "a no-render pointer", or "bare pointer" ::= `*`⟨λ⟩ (ASCII asterisk and an FSM label) + ᴿ
⟨λ⟩ "an FSM **l**abel" ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` + ᴿ
⟨BDSM⟩ "Braces'n'Digits Sequence Manipulation" ::= `{`⟨FSM chunk⟩`}`⟨π^*⟩
⟨subs'n'refs setting⟩ "substitutions and references setting" ::= ⟦ ⟨subs'n'refs⟩ ⟧ + ᴿ
⟨subs'n'refs⟩ ::= ⟨s.un.prefix⟩⟨λ⟩ | ⟨FSM chunk⟩⟨s.bin.infix⟩⟨λ⟩ | ⟨s.amb.prefix⟩⟨ϖ?⟩⟨λ⟩ | ⟨FSM chunk⟩
  | ⟨subs'n'refs⟩⟨subs'n'refs⟩
⟨s.un.prefix⟩ "subs'n'refs prefix operator" ::= ※ + ᴿ
⟨s.bin.infix⟩ "subs'n'refs binary infix operator" ::= `≕` | `≕⁜` + ᴿ
⟨s.amb.prefix⟩ "subs'n'refs 'ambiguary' prefix operator" ::= ⁁ + ᴿ

Figure 3: A formal grammar of GMS.

all in control sequences "backwards" from right to left, the escape chars of the resulting macros.

Not wishing to repeat everything that has already been said in the *TUGboat* 36:3 (2015) paper, let's just restate that:

- '\¨F⋕2' is a macro of the ⟨FSM⟩ part, performing "Get the element ⋕2 from the "shelf" and put it on the "slab".
- '\¨I' is "Put the result of previous pre-processing to the FSM's result storage, without the outer braces.
- '\¨Bε' begins the ⟨BDSM⟩ part, and is an "empty ⟨text⟩" argument for further binary operation of reverse concatenation, denoted with —
- '\¨Bɔ', that takes the two most recently processed ⟨partial result⟩s, and (conceptually) glues them into one, in reverse order.
- '\¨B🖂' is a BDSM unary of "No wrap"/"Strip off the braces" — "Pass it further in an open envelope", while
- '\¨B✉' is (also unary) "Pass it further in a sealed envelope", i.e., "Wrap it in braces".
- '\¨ꟼi' finishes the ⟨BDSM⟩ part and puts its result into the enclosing ⟨FSM⟩'s result container. The Epigraphic Reversed P, U+A7FC, 'ꟼ', indicates "Reverse Polish Notation", as this is what's going on in the ⟨BDSM⟩'s.

And the result of running this, is:

```
\ʢ_lccode:: `* %  a \noexpand-ed active char acquired
          by two-level expansion of \c_active_tl.
              `#1
\ʢ_lowercase:: {\ʢ_pdef:: *}{
  \ʢ_penalty:: \ʢ_exhyphenpenalty::
  \ʢ_hskip:: \c_zero_skip }
```

The transitions are labeled not with particular characters but with equivalence classes of: $\langle\pi\rangle$s, $\langle\tau\rangle$ (destination tokens) &c.

It's probably not a significant savings of memory or other costs of computation, but a great simplifying of the code. And making it more change- and development-robust as e.g., adding a new argument type, which is denoted with a char of equivalence class $\langle\pi\rangle$, does not require any changes in the automaton.

5.2 The destination, $\langle\tau\rangle$

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

ε If no explicit destination token is given,[15] the usual "just once" is assumed, as l3expan's \::⍰/__exp_arg_next:nnn do. This is equivalent to the use of ς.

ς Greek letter small sigma final form, the open variant, for "συναγωγή πολύ" /synagoge poly/, '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".) Useful if a GMS (is expandable and) is to prepare a single argument or {⟨balanced text⟩}.

ξ for Greek "ξανα", '[use] again': the result is put back as input for further parts of the specification, quite like the ruminants do.

5.3 The pre-ps. and pickers, $\langle(\pi^*\varpi^*)^*\rangle$

Most of these letters directly correspond to an expl3 "argument type" and the respective \::⍰ macro, or extrapolate their ideas, even maybe towards a kind of a completeness or full(er) symmetry.

With respect to their actions, they could be divided in four groups:

𝟙 the identity operators, Ii (TeXnically, they *pick* an undelimited argument, so they can be also described as "pickers");

⚗ the pre-processors (TeXnically also pickers, as above);

✔ the special pickers, picking a delimited argument, but not applying anything to it;

✘ the discarders (or destroyers), picking an un- or delimited argument and discarding it.

We have

$\langle\pi\rangle = 𝟙 \cup ⚗$,

$\langle\varpi\rangle = ✔ \cup ✘$,

and the main reason for distinguishing between (the 𝟙 s and) the ⚗ s (the $\langle\pi\rangle$ s), and the ✔ s and ✘ s (the $\langle\varpi\rangle$ s) is that the latter are not allowed in ⟨FSM⟩s, and in fact don't make much sense there.

So, let us see what they do. The ones homonymic with l3expan "argument types" are enclosed in [square brackets]. The group assignment of the above four is indicated, and the symbols '↻̸' and '?↻' denote non-expandability and uncertain expandability, resp. The symbol '?✘' means ✘ group

[15] I.e., the first char met is none of ς σ ξ.

assignment uncertain. No symbol concerning expandability at an operator means it's expandable.[16]

[o] A a ⚗ One-level expansion with `\expandafter`. (As currently implemented in l3expan.)

Ä ä ⚗ Two-level expansion with `\expandafter`. Used by me for pre-processing of macros that should be expanded to their content and that content hit once more, e.g.:

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

[c] Ć ć ⚗ The uppercase is just an alias for `c`, i.e., applying ꟻ𝐈·ꟷ𝐋 before passing the argument on *without* braces. The lowercase `ć` does the same only passes the result on in braces. What could be it useful for? First, for all the TEX primitives that require a {⟨balanced text⟩}. Then, for the constructs like

```
name 1 ꟷ𝐈 name 2 …
```

Đ ð ⚗ (Latin letter Eth/eth) Hits 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 `Đ` to `VI`. `v` is equivalent ∗`cð` in stand-alone contexts or `cð` as ⟨π s⟩ 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.

𝕯 𝖉 ⚗ Hits the argument with `\meaning`. ("Was 𝖉eutet die Name?")

[x] E e ⚗↺ Submit the argument to `\edef`. The lowercase `e` translates to the `\::x` macro of 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.

Ē ē ⚗↺ Submit the argument to, approximately, `\protected@edef` in the sense of LATEX 2ε. (We can think of the horizontal bar over the `\edef` "e"'s as a protective shield.)

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

Ħ ħ ✘ Pick a `#{`-delimited argument and discard it.

Ƕ ✔ Pick everything until the digit (⟨FSM⟩-label ⟨λ⟩) `1` and "return" without braces, leaving the ⟨λ⟩ `1` at input. (Used to jump right to a labelled FSM.) (Latin Capital Letter HV, shape modified in my Ubu Stereo font.)

I i 𝟙 Identity operation, braced or unbraced with respect to the lettercase.

Ɨ ɨ ✘ Pick and discard an undelimited argument.

K k ⚗`\detokenize` the argument.

Ŀ⟐ ŀ⟐ ✔ ?↺ Pick an argument delimited with the delimiter ⟐; if ⟐ is not yet declared, i.e., there's no internal macro with a parameter delimited with it to do the job, define it dynamically (and not expandably, in this case).

Ł⟐ ł⟐ ⌦⟐ ✘ ?↺ Pick and discard an argument delimited with ⟐, possibly declaring ⟐ dynamically, as with 'Ŀ' and 'ŀ' above.

[N] [n] 𝟙 No pre-processing. Equivalent to `i`/`I` in the current implementation of l3expan; listed separately here in observance of the admonition in "The LATEX3 Interfaces" by The LATEX3 Project [Team?][17] (henceforth "L3Interfaces"): "the implementation should not be relied upon".

Ṇ̃ Ṅ ⚗ Hit the argument, which should be a single character token, with (expandable) in- or decrement by 1 respectively (expandable).

Q q ✔ Pick an argument delimited with `\q_stop`.

Ꝗ ꝗ ✘ (Latin Capital/Small Letter Q with Stroke through Descender, U+A756/U+A757) Pick and discard an argument delimited with `\q_stop`.

[f] R r ⚗ Apply ``\romannumeral -`0`` to the argument. This fully expands the leading token(s) of the argument until an unexpandable token is seen. So, it's called `f` for "full" and *not* called so by myself for "until first unexpandable". I chose the letters '`R`'/'`r`' to refer clearly to the primitive `\romannumeral`, as this expansion is *not* "full" in principle; and if might be described without it, that would be "`\expandafter` *quantum satis*".

S s ⚗ Hit the argument with `\string`. It's worth underlining that the uppercase version "returns" the result without braces, which for a control sequence means at least two "bare" tokens.

[V] Ѵ ѵ ⚗ (expl3: Latin Capital Letter V; me: Cyrillic letter Izhitsa, U+0474/U+0475.) Render the value of a data carrier (an expl3 "variable" or "constant"), given as a control sequence. Related to `ð` and `Đ`, see above.

[v] Ѷ ѷ ⚗ (expl3: Latin letter lowercase v; me: Cyrillic letter Izhitsa with Kendima, U+0476/U+0477.) Render the value of a data carrier given as a name (first submit the argument to ꟻ𝐈 · ꟷ𝐋).

Z z ?✘ ?↺ Insert the resp. ⟨text⟩ in the stream of this GMS translation (`::`-run macros).

[16] Like, toutes proportions gardées, in Orthodox Jewish districts or state(s), "If something isn't *terefah* by its very nature, in which case there's a warning, then it's *kashrut*."

[17] as of May 18, 2016.

Since ‘Z’ is considered a symbol of things last or ultimate, and with this operator/s you can do literally anything, “from ‘A’ to ‘Z’ ”.[18]

Dangerous, experimental, liability excluded to the maximum extent permitted by Law.

Don’t use it unless you read and understood its current implementation.

Ṅ ṅ Ḋ ḋ Ġ ġ ⚗ Submit the argument to `\the\numexpr`, `\the\dimexpr`, or `\the\glueexpr` respectively, i.e., in one-level expansion evaluate the argument as `\int_eval:n`, `\dim_eval:n`, and `\skip_eval:n` do in the everyday expl3. (Added in July, 2017.)

Ḟ ḟ ⚗ Applies `\fp_eval:n` to the argument, that is, “floating point” evaluation in the sense of expl3. (Added in August, 2017.)

When used as ⟨SAlos⟩, the $\langle(\pi^*\varpi^*)^*\rangle$ (pre-ps. and pickers) refer to / are applied to subsequent arguments from the input.

When following a $\langle\varrho\rangle$, the $\langle\pi\rangle$ s refer to / are applied to the resp. $\langle\lambda\rangle$th argument from the input, counting as explained later.

When following a close brace in a ⟨BDSM⟩, including the outermost, the $\langle\pi\rangle$ s refer to that ⟨BDSM⟩ as if it were a single argument taken from the input, and a $\langle\varpi\rangle$ raises an error.

5.4 The meta-operators, ⟨ƛ⟩

The ⁎ and ⁑ meta-operators are allowed only in the ⟨SAlos⟩, and modify actions of $(\pi^*\varpi^*)^*$. By returning a pre-processed ⟨text⟩ to input, they allow multiple operations on the same ⟨text⟩ without launching the (more expensive) ⟨FSM⟩/gBDSM machine. They don’t increase the expressive power of the language, as they might be expressed as follows:

- ‘⁎$\langle\pi\rangle$’ ≡ ‘ξ$\langle\pi\rangle$.’
- ‘⁑$\langle\pi\rangle$’ ≡ ‘ξ ♮1̳1̳. $\langle\pi\rangle$.’

The ᴿ meta-operator (or rather: interruptor) suspends parsing of ⟨specification⟩, hits whatever is next with `\romannumeral -`0`, i.e., “the `f`-type expansion” in the L3Interfaces, and then hopefully resumes parsing. That allows you to branch the very specification of a given GMS, not only its arguments. Including nesting of GMS’s. Does it increase expressive power? Yes. It brings virtually the whole “mouth” of TEX into the GM-Scenarios, and that’s Turing-complete (cf. an expandable implementation of lambda calculus at `ctan.org/pkg/lambda-lists` and a more general discussion at `tex.stackexchange.com/questions/35039/why-isnt-everything-expandable`.)

[18] Also, Zelenka’s *Missæ ultimæ* might be recalled as a mnemo.

And that (the “interᴿuptions”) let you write code in a more “meta” way. And more obscure, yet shorter.

× is a shorthand for ᴿ`\prg_replicate:nn`, which means it requires two pairs of braces to come next, the first containing a ⟨number specification⟩ and the second 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`, namely, on its expandability.[19])

Let’s now deal with the “render-pointers” $\langle\varrho\rangle$s, that is, the general permutations.

5.5 The general permutations, or the ⟨FSM⟩ without grouping

The processing of a “general permutation” can be described as two stages: (Stage One) preparation of the “**shelf**” or “substrates’ storage”, or “**craw**”, and then (Stage Two) picking labelled **elements** from the “shelf” and putting on the “**slab**”[20] (a permutation consists of or is applied to elements (of some set), not arguments, isn’t it?).

As labels, the “bare” digits and Latin capital letters are used: ‘`1`’..‘`9`’, ‘`A`’..‘`P`’. This is safe since the (GMS’s) arguments’ contents are “invisible” to TEX’s macro argument scanner, thanks to the braces. Two important arguments for this choice are:

- these chars are easy to type in (for the “explicit labels” version), at least with Western input devices;
- no one changes their catcodes (not even me).

The “shelf” is functionally a one-dimensional array (a vector). For each label $\langle\lambda\rangle$, an accessor or “**F**etch!” macro ‘¨F⋕$\langle\lambda\rangle$’ exists that resp. render-pointer translates to. It “gets a copy” of the ⟨text⟩ put next to $\langle\lambda\rangle$ on the “shelf” onto the “slab”, i.e., absorbs that ⟨text⟩, and puts one copy of it on the “slab”, and another copy back on the “shelf”.

Then the $\langle\pi\rangle$ s are applied (if any), and the result is appended to what’s already in the “result container”.

It seems expensive, $O({l_s}^2)$, l_s being the length of ⟨specification⟩, and it would probably be more effective to define index-named macros whose contents would be the ⟨FSM⟩’s elements. Then access

[19] But why does the L3Interfaces indicate expandability of its “functions” if one should not rely on implementation? (That makes me feel confused ☺ .)

[20] “Let’s go to the lab ’n’see what’s on the slab”, “The Rocky Horror Picture Show”.

to any of them would cost just one ꟻ⁅ … ⁊⁆ plus one one-level expansion of it. But practically, for up to 25 ⟨text⟩s for an ⟨FSM⟩, it works just fine.

But when implemented the way it is now, it stays expandable. Why is that so important? I'm not quite sure. First of all, it's more fun. But also, many TeX and ε-TeX primitives expand macros in search of {⟨balanced text⟩} to absorb. Then, if a GMS is expandable, it is possible to write, e.g.,

```
\toks\<number>= \⁝⁝ ⟅ω {…} : 1 … ω⟆
```

and get the scenario to return the {⟨text⟩} for the `\toks` assignment.

In addition, the prefixes `\global`, `\outer`, and `\protected` expand expandable tokens, so that's possible to define, say, `\ʢ_def:Nn` that computes and sets the proper parameters string out of its `#1`'s signature, and at the same time accepts prefixing with `\global` or `\protected`, and acts accordingly.[21]

So, it seems we are handling a dynamic-length data structure within purely expandable sub-TeX. Are we really? Not quite. It is dynamic in length *and* expandable only up to the largest number for which "shelf"-preparing and -referring macros were previously defined. For now, this number is 25, as I haven't needed more so far, especially since more than 10 already makes a GMS a tool of "Security by Obscurity" rather than of shortening the code or making it more bug-robust.

5.6 Parsing the braces, or: ⟨BDSM⟩

In the first go, as presented in the previous paper, I perceived the GMS as a DFA (Deterministic Finite Automaton) "with Mysterious Something" that allowed it to handle the braces properly.

That "Mysterious Something" had been implemented as the native TeX's argument scanner with additional tricks to roll back the effect of '`\string{`', in a sense, and instead put a special token 'ꟼ' next to the outermost closing brace, translated to the '`\¨ꟼ`' macro mentioned above.

Currently my understanding is that those tricks are not necessary, and we can process the ⟨specification⟩ char-by-char until a colon '`:`', which allows to use not only the braces $\{_1$ and $\}_2$ as the group opening and group closing symbols, but also other characters I'd give this charclass(es), e.g., '⦃' and '⦄'.

The only thing we need to do is add another argument to the relevant states and transitions, one which bears the nesting level.

[21] `\ʢ_def:Nn` is actually defined this way in gme3u8, while l3expan's `\cs_new:Nn` contains a preliminary/auxiliary `\def` any prefix is "earthed" at, or even raises an error.

And what is such an argument, i.e., an integer (Natural) number, that's initially 0, and is increased by 1 with each opening brace, and decreased by 1 with each closing? It's nothing (in a sense) other than a pushdown stack (i.e., a stack with top-only access), with the initial symbol 0 and just one symbol pushed down or popped, 1. Then, we can think of the number representing current nesting depth of braces as the stack storing this many 1's.

In this sense, the GMS automaton is a DPDA, Deterministic Pushdown Automaton.

The (informal) argument that it's a proper DPDA (not a Turing Machine), is that it does only what fig. 2 depicts. In particular, it rejects (raises an error at) the ⟨BDSM⟩ braces and ⟨subs'n'refs⟩ double-stroke brackets interlacing

```
\⁝⁝ … ♮ {⟦}⟧ ; … :
```

while all three '♮ {}⟦⟧ ;', '♮ {⟦⟧} ;', and '♮ ⟦{}⟧ ;' are accepted.

Then, the depth of the BDSM braces is limited only by the capacity of TeX (and at the ⁝¹-run it's about memory not the maximum group level, since there are no "groups" (that belong to the semantics), just syntax, i.e., the symbols of the GMS language's alphabet). This means it's stronger than a Deterministic Finite Automaton, since it's capable of recognizing the Dyck language.

The "stack integer" is implemented as a single character interpreted as a number via '`' (TeX backquote), and expandably in- or decremented thanks to `\numexpr` and `\Ucharcat`.

I consider it a data type and call it 'Ń', hence the 'Ń̟' and 'Ń̠' pre-processors using the expandable in- and decrement mentioned above.

With setting the number 0 to be represented by the character '`0`', the maximum character number available in XeTeX is 1114063. So, far far more than the maximum grouping level handled by TeX, so it can parse (accept and translate into the rearrangement macros) far deeper nesting than TeX can execute.

If we meet an opening brace when 0 is at the top of the stack, we move to the BDSM sub-automaton, and do basically the same things as we do in the FSM states, but pushing or popping 1's down or from the stack.

Once 0 is seen again, i.e., once all the 1's are popped from the stack, we know that the outermost opening brace (of this BDSM) has been matched, so we possibly apply some ⟨π⟩s to it, and then yield, and move back to the state of general FSM.

The case of "subs'n'refs" is conceptually analogous, although the current implementation is more tricky than honestly "automaton-ic".

5.7 The ⟨subs'n'refs⟩

Whenever the automaton, being already in some of the ⟨FSM⟩ states, meets an opening '⟦' bracket, it conceptually pushes yet another symbol down the stack, and moves us into the "subs'n'refs" sub-automaton(s).

This new symbol may be considered the initial symbol of the ⟨subs'n'refs⟩ sub-stack. Then, again, whenever an opening brace is met, 1 is pushed down, and popped at meeting a closing brace. Seeing that "yet another symbol" on top of the stack after a closing brace was popped, says that that brace is outermost with respect to this ⟨subs'n'refs⟩.

As you see, the matter discussed gets a bit complex, and hence I chose i as that "yet another symbol". ☺

The ⟨subs'n'refs⟩ cannot be nested, and that's why only one i is allowed on the stack, and why there's no '⟦'- or '⟧'-labelled edge between two states of the ⟨subs'n'refs⟩ sub-automaton.

The ⟨subs'n'refs⟩ sub-automaton might also be considered two distinct sub-automata, one reached from the general ⟨FSM⟩ states and returning to them, and the other from the ⟨BDSM⟩, and going back to that ⟨BDSM⟩.

The difference lies on the top of the stack (literally) before pushing down the symbol i, and after popping it: if the ⟨subs'n'refs⟩ sub-automaton is reached from a general ⟨FSM⟩, 0 is on top of the stack, and if the ⟨subs'n'refs⟩ has been reached from within a ⟨BDSM⟩ state, then 1 is on top of the stack.

The diagram is complex already; so as not to make it messy, those two sub-automata have been merged, in a sense. Instead of drawing the two independently, only the "foreign" edges are drawn as distinct, while the states and the "domestic" edges are present once, and the two sub-automata are topologically homeomorphic.[22]

5.7.1 The replacements, '=:'

Consider the part '⟦ 2̳Ä=:2 ⟧' of the example in section 4. As has already been mentioned, this part of ⟨specification⟩, and more precisely, of its ⟨FSM⟩, translates into macros that replace the original element ⋕2 with its "double 7⤵" expansion, prepared to be returned without braces.

[22] The macros for the two are separate, though, because within ⟨BDSM⟩ the local "shelf" and "slab" are prepared slightly differently, and so the parameter delimiters differ.

So far, there is no "symbol i" used in the implementation. The transition to the ⟨subs'n'refs⟩ part of the automaton is implemented with "memorizing" current nesting level, which would be the current length of the stack in the terms of this paper, as a(nother) numchar, and confronting it with the {}-nesting level at ⟧, i.e., at the end of ⟨subs'n'refs⟩.

This is done in a way similar to the "usual" processing of the ⟨FSM⟩ elements, i.e., by taking a copy of the element into the "operation table" or "slab", applying the ⟨π⟩s, and, here comes the difference, putting the result not in the "result container", but back in the "craw" or "shelf", effectively replacing it at the given label.[23]

The idea of this sub-automaton stemmed exactly from the craving for "as few repetitions as possible", namely, in situations where I'd apply the same sequence of pre-processors ⟨π^*⟩ to the same element of an ⟨FSM⟩ more than once.

In the given example, it's a bit of an overkill, as the element ⋕2 is used only in two copies. But, even twice might be too many, if we think of the points to remember to change something, say, a single '7⤵' to "double-7⤵".

In this example, the render-pointer ⟨ϱ⟩ at the left side of '=:' corresponds with the label ⟨λ⟩.

But it's not a *sine qua non*. The mechanism is general enough (at no additional cost) to process any correct ⟨FSM⟩ put on the left (including ⟨BDSM⟩), and make the replacement of it at the label typed on the right side of the '=:' "assignment" symbol.

And, I used quotation marks for the word "assignment", because the replacement operator '=:' (binary infix) is expandable.

5.7.2 "The arguments from beyond", 'ʌ'

Also expandable is the "ambiguary" prefix operation ʌ⟨ϖ?⟩⟨λ⟩, that gets the next argument from the "input", i.e., from beyond all the "slabs", "shells", "fridges" and "containers" of the ⟨FSM⟩ and of the entire ⟨specification⟩, and puts it as the element ⋕⟨λ⟩, thus replacing whatever was there before.

It also works if there was nothing at that label earlier, i.e., if ⟨λ⟩ was until now immediately followed by another label, or by a delimiter of the "shelf". (Garbage warning as above.)

Again, it's done with macros with a parameter delimited by the respective label(s).

[23] With the assumption that such replacements are "not too many", and in order to allow "empty labels", the old version of an element remains "at the back", and is discarded only at the very end. (Garbage warning.)

Since this feature was implemented only last week, only an undelimited argument picker is handled at the moment. But other pickers are *in pectoris*, as described in the Grammar, fig. 3.

Also, in this (early) version of this functionality, presence of the label ⟨λ⟩ already in the slab is assumed and required.

Again, this feature emerged from a need (or will) to be able to write a GMS consisting mostly of an FSM, and make an anonymous function of it,[24] and yet declare the FSM with labels, as it's more readable this way.

The example which follows is based on a quasi-iterator used in some *really real* TeX program. First of all (conceptually), there is a list of control sequences that should be at some point defined (or not, that's why it's not done on the "ground level" of code). Then, if they are defined (and only if), they should be initialized, as they're defined as variables (of various types). Then, if they joined in the action, it has to be known how to set them (s), and also, how to reset (rs). That makes a 4-tuple of things for each of those control sequences, with the (s) function used also in (rs), only with the special value (rsv), and that (rsv) is specified as the 4th element of each tuple.

Each of those control sequences requires specific "methods" of its own, and initialization is performed once if at all, so instead of defining macros, I used a GMS to allow the contents of the braces (*), i.e., that anonymous 1-argument function, to be put by the loop, and given control sequence (*b) as the argument.

```
...
{⟨initializations of:⟩}  \g__💎_aux'1̳_str
...
{ %  (*)
  \⁝⁝  ⟦ω ⟦⋏5⟧ 1̳5̳ 2̳5̳3̳: } %  declare & init. box to
        empty \hbox
{  \⁝⁝  ⟦ω ⟦⋏5⟧ 2̳5̳4̳: }  %  reset the box to void
{ %  (**)
    1  \box_new:N
    2  {\ʢ_global::\ʢ_setbox::}
    3  {=\hbox{}}
    4  \c_ʢ_void_box
    5
  ω⟧
}
\c__💎_aux'1̳_box %  (*b)
...
```

[24] Although at the time of writing this feature I was not aware it was to be an anonymous function. Not only am I not a computer scientist, but also not a graduate of a formal course of computer programming 😸.

Thus, when the code (*) and (**) are put (without braces), and followed by the c.s. (*b), the first thing done, written down as '⟦⋏5⟧', is absorbing (*b) to the #5.

And then the initialization is performed, i.e., the result of the above GMS is:

```
\box_new:N \c__💎_aux'1̳_box
\ʢ_global:: \ʢ_setbox:: \c__💎_aux'1̳_box =
  \hbox {}
```

The optional picker ⟨ϖ?⟩, if present, makes the machine pick not the next ⟨text⟩ undelimited, but delimited as specified with the ⟨ϖ?⟩ (for symmetry, specifying 'i' or 'I' is also allowed).

Note, by the way, that any GMS might be considered an anonymous function (unless a "predef" of it is made), and also an explicit sequence of l3expan '\::⍰' 's, but not the 'inline' (1st) arguments of the expl3 \⟨type⟩_map_inline:n[n|N] iterators, as the latter are internally assigned a (one-parameter) macro in the usual way, only hidden.

5.7.3 Snapshots and references, '※'

Described last, as unexpandable by their nature, are the ※⟨λ⟩ operations, 'snapshot the element #⟨λ⟩ and make a reference to it'.

The idea is very simple: allow referencing the permutation elements within the reorganized code, so as not to be forced to divide everything into the "before the element/argument part" and "after ~~ part".

So, putting '⟦ ›… ※7 ›… ⟧' within an ⟨FSM⟩, makes a "snapshot" of the element #7 as it is at the point of ⁝⁝-run of the (translation of) this operator, available as the contents of an expandable macro, or, speaking in expl3, a _tl variable, that may be rendered via '₇※7' to get that contents wrapped in \unexpanded, or via '₇⁜7', for not protected.

Nesting one GMS within another is allowed, and to avoid messing up the snapshots and references in such a case, a record of its level (depth) is kept, and updated expandably as long as purely expandable ⟨subs'n'refs⟩ are used, which is checked by the automaton in the ⁝⁝-run.

For now, only '※' and '=⁝⊧' ⟨subs'n'refs⟩ operators "destroy expandability", the latter being a superposition of expandable '=:' and '※', as follows.

The binary infix operator '=⁝⊧', used as

'⟦›…⟨FSM⟩=⁝⊧⟨λ⟩›…⟧',

first replaces the element ⟨λ⟩ with the result of the ⟨FSM⟩ from the left side, and then also makes a snapshot of it, referrable as described above, via ₇※⟨λ⟩ for "\unexpanded'ed", or ₇※⟨λ⟩, for "bare".

To be honest, I've used this mechanism only a few times so far, as it denies expandability by its very nature. Each of those few uses is large, of more than 10 elements, therefore I'll show just fragments of the simplest one.

```
\⁝⁝ ⟅ω ⟦3̳=⁜9 % (curr. contents of) #3 is put on #9,
        and made ※9
  2̳Ć=:7⟧
  1̳ 7̳{5} % the c.s. built above is now defined
  ... ⁚
...
2 { c__💎_ #1 transition'from⟨ #2 ⟩ᔑvia⟨ 7⁜9
     ⟩ᔑresult_clist }
3 { #3 }
9
...
ω⟆
```

Note how the "snap'n'ref" is used: the primary goal of this feature is to allow putting placeholders in the subject code, and have them replaced with the respective element. Somewhere in the middle of the text, and possibly, also nested.

So, there are the placeholders in the middle of ⟨text⟩ of an element. And, they are valid only within their respective ⟨FSM⟩, thanks to the record of GMS nesting mentioned above, and checking it.

And outside their own FSM, those placeholders/references issue an error. So, to make any use of them, one has to apply some kind of full expansion to the elements that contain them.

And here it is: the element #2 is a long csname built with 7⁜9, and the 'Ć' operator, i.e., ƒ⁅·7⁆, performs such expansion. The resulting c.s. is put instead of the original {⟨text⟩}.

Note BTW, that the c.s. raised from #2 is not put instead of it(self), but at #7. That's because '#3' being the contents of the dereference #9, is "alive", and "then" expands to something other than "now".

6 Rough budgeting, a.k.a. cost estimation

If we take the "interᴿuptions" into account, the estimation is simple: anything is possible, including arbitrary elongation of the resulting "interᴿuption"-less ⟨specification⟩. That elongation might come from, e.g., '×⟨N⟩{∗i}, where N is a decimal or hex. representation of a positive integer.

Then the resulting length of ⟨specification⟩ becomes $O(B^{l_N})$, where B is the base of the representation of N used, and l_N its length in this representation.

So, in the full-featured version, potentially "exponentially explosive", but no more so than any loop accepting numerical limits in power-position notation.

What about GM-Scenarios with "interᴿuptions" put aside, i.e., the proper DPDA of it?

What basic operations should we consider here? If we think of each operator as a (constant) sequence of macros, as the definition of the operators does not change in the runtime, and getting the next ⟨text⟩ from input as just one step (unit cost), then the time cost of a GMS that doesn't involve ⟨FSM⟩ is, putting l_s as the length of ⟨specification⟩, $O(l_s)$.

Then, including ⟨FSM⟩ in our consideration, we see that one character (plus a constant number of its "context"), may result in apparently arbitrarily large numbers of ⟨text⟩s to take from input.

```
\⁝⁝ … ♮|7|; … ⁚
```

But it cannot happen, as the alphabet (not Unicode, not the charset handled by the TeX engine used, but the theoretical alphabet of the language considered) is finite,[25] so there is an upper bound for the numbers expressible with the ⟨λ⟩'s, so, as we're in the realm of Naturals, there exists the maximum of those numbers. Let's denote it by M. Then, including ⟨FSM⟩'s in this "budgeting", we get an estimation

$$O(l_s) + O(M \cdot l_s) = O(l_s),$$

still within linear time.

But, is the assumption of the unit cost of getting new ⟨text⟩ reasonable, no matter how far we have to jump over the tail of ⟨specification⟩, and over the partial result, i.e., the storage of the ⟨text⟩s already processed?

It seems not. As the ⟨specification⟩ is executed, the partial result "pessimistically" grows at the same rate as the ⟨specification⟩ shortens, and we have to jump over both of them in order to get the next ⟨text⟩ for pre-processing. So, if we consider "jump over one ⟨text⟩" a unit cost, the estimation becomes $O(l_s)^2$, it seems. Still, polynomial time, not so bad (it seems).

The space cost appears even nicer, as no ⟨text⟩ at the input can be copied more than l_s times, and there can be one ⟨FSM⟩"shelf" at a time, so no more than M additional ⟨text⟩s at a time. That allows the following estimation of the space cost SC:

$$SC \leq 2l_s + M,$$

with given alphabet and fixed set of ⟨ϱ⟩'s and ⟨λ⟩'s, M is constant, and so we get $O(2l_s + M)$, and that's just $O(l_s)$. Just great, it seems.

[25] …and the language is too weak to express a description like "The largest number expressible with less than 70 characters"!☺

However, consider

```
\⁝⁝ ξ♮{1̳1̳1̳}. : a                   % (s1)
\⁝⁝ ξ♮{1̳1̳1̳}. ξ♮{1̳1̳1̳}. : a          % (s2)
\⁝⁝ ξ♮{1̳1̳1̳}. ξ♮{1̳1̳1̳}. ξ♮{1̳1̳1̳}. : a % (s3)
...
```

It looks we've found an "exponential explosion", as each next 'ξ♮{1̳1̳1̳}.' replicates the result of the previous one three times, and n times in general, n being the number of '1̳''s within braces.

Even though the length of ⟨specification⟩ grows by $n+K$, $K = 4$ being fixed, so it's not exactly $3^{l_s/n}$ but more like $3^{l_s/(n+K)}$ — still exponential. Both in time and space, as it's the partial result that grows so fast.

But this is a weird and theoretical example.[26] In practice, let me repeat, it works just fine, as users intuitively avoid the 'ξ' "destination", and don't do such silly things as mere replication of one ⟨text⟩; the previous estimation, of $O({l_s}^2)$ time and $O(l_s)$ space, seems to hold in all reasonable cases.

7 Friendly critiques at TUG@BachoTEX 2017

The respective section in the GMOA paper was called "Real-life uses of GMOA". I'm not sure whether such a machinery, which with its full features is rather an esoteric language than a friendly tool for reasonable users, might be much used in "real life". I'm afraid that, by mere using of it, the respective part of "life" would be made "un-real". At least, in the sense of total obscurity for anyone else but me.

That's the most important thing my colleagues, or better say: friends, pointed out after the presentation of GMS at TUG@BachoTEX 2017.

In more detail, it's because:

1. using "distant" and PUA Unicodes does not help at all, since most users are still in pure ASCII, at least concerning the control layer, like control sequences and special characters;
2. it's too complex and obscure, and for most people it's simply easier and clearer to write the same code twice, or more times, than to try to decipher from the one-character instructions how the pieces should be repeated, and how modified;
3. "`\expandafter` does strange and complex things, therefore it should have a long and strange name, and not single-character!", and similar argument about other primitives;
4. it doesn't seem useful, "… and I understood, it doesn't have to be: because you don't develop it to be useful, you develop it as your artistic expression."

@ rem. 1, I totally agree. Also, not even yet shown, non-English control sequences, such as the eschatological-appearing '`\__⁝⁝_αποκαταστaθεί'FSM`' ('apokatastathei FSM', reminding one of the Neo-Platonic or Gnostic visions of Apokatastasis at the End of Time), changed to '`\__⁝⁝_SᴱᵗR∽resume'FSM`' herein, along with all other Greek or Latin ones.

And, to ease typing of ⟨specification⟩s to those few who may not be completely familiar with things like Opening Lenticular Bracket Ordinal Number Omega, PUA+E9EA '', I provide a pure ASCII and HTML-like "input method": in the most general version, one may type '`&U+`⟨hex⟩`;`', and then `\Ucharcat ·␣12␣` will be applied to ⟨hex⟩, i.e., the resp. char$_{12}$ rendered, as if it were there in the first place.

Then, there are some "ASCII approximations" of the symbols, like '`&w[;`' and '`&w];`' for '' and '', or '`&VY;`' for the Capital Izhitsa with Kendima, 'Ѷ', or '`&{the};`' for 'ð'.

The '`&…;`' "entities" are (more or less) "interᴿ-uptions", and can be used anywhere. They are translated internally to the respective original symbols, so using a native Unicode engine remains obligatory.

For the pointer-renderers of ⟨FSM⟩ elements, i.e., the ⟨ϱ⟩ symbols of the formal grammar, the HTML-like forms: '`&_1;`'…'`&_9;`', '`&_A;`'…'`&_P;`' for the "lowercase" '{1}'…'{p}', and '`&^1;`'…'`&^9;`', '`&^A;`'…'`&^P;`' for "uppercase" '1̳'…'9̳', 'A̳'…'P̳'.

To avoid possible confusion of '`&_1;`' and '1̳', think of ASCII Underscore as the sign of "generic sub-ness", as in standard TEX for subscripts, so may it be also for lowercase, and of the graphical element '‗' "double underline" as the proofreading sign "make this uppercase".

But in case of the ⟨ϱ⟩ "render-pointers", the HTML-like notation is not necessary, i.e., the '`&`' and '`;`' might be omitted, and '`_9`' is also fine, as shown both in the automaton graph in fig. 2, and in the formal grammar in fig. 3.

@ rem. 2, I admit: yes, GMS are complex, and maybe even mad, and might easily become obscure. But, and this is one of its goals, they allow reducing the number of repetitions of at least some parts of code to 1, and that in turn makes things fixable at just one point. For instance, having defined two macros that differ only with the printed text, and put the same skips before it, if I wish later to change the amounts, then, having used `\⁝⁝` properly, I change "both" of them only once, namely, at the label '`A`':

[26] Remember Murphy's Law?

```
\GMS  &w[; ^1^2 ^7 {^A^3^9}
          ^1^4 ^7^8 {^A^8 _5 ^9}  :
1  \def
2  \macro1   3  {indigo}
4  \macro2   5  {indigenous}
7  {#1#2#3}  8  {#4}   9  {#2}
A  {\hskip 17pt\relax } %  later changeable
&w];
```

As with probably all things in this world, it's a matter of balance. Here, between only the primitives at one extreme, and just one active character expanding to the entire document, at the other. Oh, not even one: an empty file, that expands to the entire document thanks to `\everyeof`.

For me, that balance seems to be in the l3expan iterators and in not too long, but on the other hand, nontrivial, GM-Scenarios.

@ rem. 4, I also admit that the main reason I'm doing this is fun, or, to put it in a less hedonistic way, intense intellectual satisfaction.

But, again, that doesn't exclude usefulness *per se*, and striving to make GMS meet not only my requirements, but also those of other people, might be as much fun, and as much art.

@ rem. 3, let me just say:

′

as in: $(f(x) \cdot g(x))' = f'(x)g(x) + f(x)g'(x)$. 😸

8 Final remarks

8.1 "Thank Heavens, it's not the Premium Class"

At the end, let's recap the question posed in the title. We already know GMS's are a complete madness. But — are they Turing-complete?

The answer has already been given, and this answer is: No.

Ignoring the "interᴿuptions", the[27] automaton is deterministic pushdown, and the GMS language appears to be context-free.

So, it's not a Premium Class machine, i.e., Turing, and that's a relief in a sense, as it shows I did not "rewrite TeX in TeX" [yet].

On the other hand, the GM-Scenarios allow for making parts of code noticeably shorter, clearer, and less repetitive, and this way more readable and bug-robust. Provided that they (GMS s) are kept at bay on their own, i.e., not too long, and not too complex.

[27] Actually, *an* automaton, since there exist many automata equivalent to the one just presented, in the sense of recognizing exactly the same language.

8.2 The end, or ἔσχατον

When I think of all those symbols, the automaton, its states and transitions, adding "the arguments from beyond", the correspondence between it and the formal language of GMS, the most "finale-al" finale I know of, the eschatological and apokatasthatic "Chorus Mysticus" in the cosmic Mahler *Eighth Symphony* comes in handy. ☺

Alles Vergängliche	All things under Transition
Ist nur ein Gleichnis;	are just a Symbol;
Das Unzulängliche,	What l3expan couldn't express,
Hier wird's Ereignis;	here is performed;
Das Unbeschreibliche,	What could not be described,
Hier ist es getan;	here is just done;
Das Ewigweibliche	les Femmes Puissantes,
Zieht uns hinan.	protect and bring us beyond.

— Goethe, "Faustus"[28] *&*
Mahler, the Eighth Symphony

◇ Grzegorz Murzynowski
PARCAT.eu
g.murzynowski (at) parcat dot eu
natror.croolik.sryc (at) gmail dot com

[28] English translation mine, adapted and adjusted for the needs of this paper.