\dante_tutorial:nn{expl3}{2022}

Nous publions ici, sous son titre original, un article de Marei Peischl originellement paru fin 2022 dans Die TeXnische Komödie [84]*. Il nous a semblé avoir un grand intérêt. Notons que cet article est également paru en anglais, traduit par son autrice, dans le premier numéro du* TUGboat *de l'année 2023* [85]*.*

De ce que nous en savons, le premier contact avec expl3 – la syntaxe de la couche de programmation de LaTeX3 – est souvent effrayante et trompeuse :

```
\ExplSyntaxOn
\clist_map_inline:nn \l_tmpa_clist {
 \__ptxcd_add_item:n {#1}
}
\ExplSyntaxOff
```

LaTeX3 n'est plus une nouveauté logicielle que la communauté attendrait depuis des décennies. LaTeX3 est parmi nous depuis belle lurette, il est utilisé par de nombreuses personnes sans pour autant être vraiment remarqué.

La couche de programmation de LaTeX3 constitue le socle de cet ensemble. Elle offre une interface normalisée que les développeurs de paquets comme les utilisateurs peuvent utiliser, directement ou non, que ce soit pour programmer des mécanismes complexes ou pour gérer du contenu de manière nettement plus flexible qu'avec des fichiers LaTeX classiques.

En définitive, les objectifs les plus importants de LaTeX3 sont :

- une interface unifiée pour les fonctions et variables ;
- une syntaxe modernisée ;
- une simplification du contrôle de l'expansion…

… ce qui permet de programmer très facilement avec LaTeX.

Programmer se révèle utile lorsque, en appliquant certains paramètres, on modifie la mise en page ou la structure générale d'un document. Un exemple classique nous vient de l'enseignement, où l'on compose un sujet d'examen en dissimulant ou non les solutions des exercices. Un autre exemple est l'utilisation de données externes – typiquement une liste [86], mise en forme de manière automatisée :

Exemple 34

code

```
\ExplSyntaxOn
\begin{enumerate}
```

84. http://archiv.dante.de/DTK/PDF/komoedie_2022_4.pdf
85. https://doi.org/10.47397/tb/44-1/tb136peischl-expl3
86. Note du traducteur – c'est volontairement que, dans la liste en question, nous n'avons pas traduit *eins, zwei, drei* : ils seront utilisés plus loin pour illustrer les capacités d'expansion de expl3. Or, de l'expansion à la récursivité, il n'y a qu'un pas, et nous utilisons l'éditeur de texte emacs, qui utilise le langage Lisp… nous nous souvenons des deux machines Lisp du MIT, le *Massachusetts Institute of Technology*. La première était appelée EINE, sigle récursif pour *Eine Is Not Emacs*, et la seconde ZWEI, pour *Zwei Was Eine Initially*. Nous manquons hélas d'informations sur DREI.

code (suite)

```
\clist_map_inline:nn { eins, zwei, drei } { \item #1 }
\end{enumerate}
\ExplSyntaxOff
```

résultat

1. eins
2. zwei
3. drei

La syntaxe expl3 paraît cryptique à des yeux habitués à d'autres langages de programmation. Les deux-points mêlés à des tirets bas et les conventions de nommage en vigueur créent une structure syntaxique singulière, qui trouve son origine dans le fait que LaTeX, et par conséquent expl3, n'est rien d'autre qu'un langage de macro-commandes. Les chaînes de caractères y sont remplacées par ce qu'elles désignent et non par des opérations réelles, comme c'est le cas dans les langages de scripts.

Pour comprendre la structure d'expl3, il est donc nécessaire de comprendre les bases de la macro-expansion et du concept de *catcode* (pour *category code*). C'est pourquoi les sections suivantes, qui décrivent la structure syntaxique de ce langage, explicitent ces concepts. Si ces derniers sont déjà familiers au lecteur, il lui suffit de sauter les sections correspondantes.

Commutation syntaxique avec TeX, LaTeX et expl3

Quand TeX compile des données, il ne fait pas que lire une suite de caractères : il attribue une catégorie à chacun d'entre eux. Chaque catégorie détermine l'usage qui sera fait du caractère en question. L'attribution d'une catégorie à un caractère se fait via les fameux *catcodes* (abréviation de *category codes*, en anglais). À chaque caractère correspond un code de caractère, et à celui-ci est ensuite temporairement attribué un code de catégorie.

Au total, TeX définit seize catégories différentes. Toutefois, l'affectation d'une catégorie à un caractère peut varier à l'intérieur d'un même document. Le cas le plus courant est une variation en fonction d'une langue donnée, par exemple en utilisant babel [1]. C'est ainsi que la chaîne « "a » devient « ä » en allemand tandis que ces deux caractères seront traités séparément en français. Le comportement en allemand est dû à la catégorie 13, « active » ; les caractères actifs ne sont plus de simples caractères, mais des instructions – dans le cas du « "a », placer un tréma sur la lettre qui suit l'instruction. En français, aucun des deux caractères n'est placé dans la catégorie 13 : chacun d'entre eux est lu en tant que tel, d'où le résultat : « "a ».

La liste suivante présente toutes les catégories disponibles, avec des explications et des exemples qui vous seront probablement familiers.

0. Caractère d'échappement (\)
1. Début de groupe ({)
2. Fin de groupe (})
3. Passage en mode mathématique ($)

4. Alignement (&)
5. Fin de ligne (⟨*return*⟩)
6. Paramètre de commande (#)
7. Exposant mathématique (^)
8. Indice mathématique : (_)
9. Caractère ignoré (⟨*null*⟩)
10. Espace (␣)
11. Lettre (l'alphabet : A, B... a, b... – et, seulement avec XeTeX et LuaTeX, leurs versions diacritées)
12. « Autre » caractère (tout le reste : ., 1, :, @, etc.)
13. Caractère actif, à interpréter comme une séquence de contrôle (~)
14. Début de commentaire (%)
15. Caractère invalide : (⟨*delete*⟩)

Le caractère @ dans les macro-commandes de TeX et LaTeX

TeX utilise le caractère @ pour protéger ses macros privées : cela empêche les utilisateurs finaux d'y accéder. Ce n'est pas sans raison qu'elles sont ainsi protégées. Afin d'éviter des effets secondaires inattendus, d'éventuelles modifications doivent donc être effectuées avec prudence. Il est nécessaire de modifier la catégorie du caractère @ vers la classe 11 afin de pouvoir l'utiliser dans les noms de macros.

Pour ce faire, on utilise des commutateurs :

```
\makeatletter
\makeatother
```

Un exemple classique de l'utilisation du caractère @ est la création de variantes étoilées de certaines commandes. Pour cela, il faut définir en interne deux macros auxiliaires, qui elles-mêmes sont souvent protégées par le signe @.

```
\makeatletter
\newcommand*{\cmd}{\@ifstar\@cmdstar\@cmd}
\newcommand*{\@cmd}{sans *}
\newcommand*{\@cmdstar}{avec *}
\makeatother
```

La macro donne alors le résultat suivant :

Exemple 37

code

```
\cmd\\\cmd*
```

résultat

sans *
avec *

La syntaxe expl3

Comme la syntaxe de expl3 est conçue pour la programmation, LaTeX se comporte ici de manière fondamentalement différente de ce qui serait judicieux pour une sortie de texte.

- Les espaces et les retours à la ligne dans le code délimitent les chaînes de caractères ; autrement, ils sont ignorés.
- Les lignes vides ne marquent pas une fin de paragraphe.
- Les deux points (:) et le tiret bas (_) font partie intégrante des noms de macro.
- Il y a une distinction entre les fonctions et les variables.
- Il est recommandé de mettre des espaces autour des accolades si elles ne contiennent pas qu'un seul paramètre.

Ces modifications permettent de structurer un peu mieux le code, sans en modifier le sens. La syntaxe est introduite par un commutateur, analogue dans son utilisation au bien connu `\makeatletter` :

```
\ExplSyntaxOn
\ExplSyntaxOff                    (commutateur syntaxique)
```

Une autre caractéristique typique de LaTeX3 est l'utilisation de majuscules et de minuscules dans ses noms de macro. La lisibilité du code s'en trouve améliorée, y compris au sein de la syntaxe LaTeX classique.

La structure des noms

La structure des noms suffit à expl3 pour distinguer entre les fonctions, qui traitent des contenus/arguments, et les variables, qui ne font que stocker une valeur.

```
\Module_Description:Arguments                  (fonctions)
\Validité_Module_Description_Type-de-donnée (variables)
```

Les variables

Les variables stockent des valeurs. expl3 fournit pour cela différents types de données. En revanche, la structure du nom de chacun de ces types de données est identique. Techniquement, il s'agit d'une convention. Toutefois, le respect de ces règles permet aux utilisateurs de comprendre plus facilement le code.

```
\Validité_Module_Description_Type-de-donnée (variables)
```

Validité : constante (c), globale (g) ou locale (l).

Privée : les variables privées, qui ne doivent pas être utilisées par les utilisateurs finaux, présentent deux tirets bas entre l'indication de validité et le nom du module. Les variables normales n'en ont qu'un.

Module : nom du package [87], ceci pour éviter les doublons. L'enregistrement de ces noms se fait selon le processus décrit dans [2].

Description : que stocke la variable et pourquoi ?

87. NdT : ou de la classe.

Type de donnée : comment les différentes valeurs sont-elles enregistrées dans la variable ? Comment doit-elle être traitée ?

Un exemple de variable privée est la liste de *tokens*, c'est-à-dire une liste de caractères.

```
1  \l__siunitx_complex_sign_tl
```

Tous les types de données sont associés à un module. Les listes de *tokens* [88] proviennent du module `l3tl` [89] . D'autres types de données ont une autre désignation correspondant à cette abréviation.

Toutes les interfaces sont documentées en [3]. La liste suivante en donne un petit échantillon :

tl	*tokenlist*, chaîne de caractères qui ont encore différents *catcodes*
str	*string*, chaîne de caractères comprenant seulement des lettres, des espaces ou des caractères de catégorie 12 (autres)
bool	*boolean*, valeurs de vérité
box	boîte
coffin	« boîte avec poignées », une boîte dotée d'ancrages permettant un placement relatif à d'autres objets
dim	dimension
skip	dimension extensible
fp	*floating point*, nombres à virgule flottante
int	*integer*, nombre entier
prop	*property list*, liste de clés/valeurs
seq	*sequence*, séquence, suite
clist	liste de valeurs séparées par des virgules
stream	flux d'entrée/sortie pour la lecture/écriture de fichiers

Les fonctions

Le terme de fonction est peut-être ici trompeur par rapport aux langages de programmation. TEX, et donc expl3, est purement un langage de traitement

88. NdT : … ou lexème en français.

89. Note de Denis Bitouzé [90] : les *modules* dont il est question dans ce paragraphe sont ceux de expl3 et en constituent en quelque sorte des bibliothèques, chacune d'elles dédiée à un type de donnée. Ainsi, `l3tl` est-il celui dédié aux ***tl**okenlist*, `l3int` est-il celui dédié aux ***int**egers* (entiers). Ces *modules* sont à ne pas confondre avec ceux dont il est question ci-dessus et qui sont le nom du package ou de la classe que l'auteur est en train de créer ; ainsi le code du package siunitx, programmé en LATEX3, contient-il de nombreuses variables de la forme `\Validité__siunitx_Description_Type-de-donnée`, par exemple `\l__siunitx_complex_sign_tl`.

90. Le lectorat appréciera l'intervention de Denis Bitouzé, qui témoigne du sérieux de notre entreprise : la *Lettre* GUTenberg est certes irrégulomadaire et rédigée par des bénévoles, mais ce sont des bénévoles consciencieux, qui se relisent les uns les autres et n'hésitent pas à amender un texte, en l'espèce dans un but pédagogique. Voir la liste des impétrants page 113. Au sujet de l'intrication des notes, voir… la note 112 en page 100 !

de macros qui donne l'impression d'avoir des fonctions alors qu'il ne fait que substituer des chaînes de caractères. Néanmoins, dans expl3, on peut aussi considérer que les fonctions traitent leur contenu, si ce n'est qu'il n'y a pas de valeur de sortie au sens des langages de programmation traditionnels : la « valeur de sortie » des fonctions est envoyée dans le flux d'entrée — et donc écrite dans le document.

La convention de nommage des fonctions est la suivante :

```
\Module_Description:Spécifications-des-arguments
```

Le `Module` et la `Description` nom sont identiques aux mêmes champs que nous avons déjà décrits pour les variables. La différence entre une fonction locale et une fonction globale est toutefois inscrite dans la description. Par convention, on insère le mot `set` devant une affectation (locale) et on y ajoute un `g` pour une affectation globale : `gset`. Les fonctions permettant de définir des variables entières en sont des exemples :

```
\int_set:Nn
\int_gset:Nn
```

Avec la syntaxe expl3, les arguments attendus sont indiqués après les deux points. Ainsi, les deux exemples ci-dessus attendent chacun deux arguments : un de type `N` et un de type `n`. On peut ainsi voir directement le nombre et le type des arguments attendus par une fonction. Cela sera important plus tard pour le contrôle du développement des macros, mais pour le moment, nous nous limitons aux types `N`/`n`. La lettre `n` (majuscule ou minuscule) signifie *No manipulation* [92]. L'argument est donc transmis à la fonction sans autre traitement.

L'usage de la majuscule ou de la minuscule indique si la fonction attend ici un *token* unique ou un argument groupé. Les majuscules correspondent à des *tokens*. Les minuscules correspondent à des groupes. La spécification d'argument `:Nn` déclare que la fonction attend d'abord un *token* et ensuite un groupe. Ainsi, la macro `\int_set:Nn` pourrait donc être utilisée comme suit :

```
\int_set:Nn \l_tmpa_int { 5 }
```

L'exemple ci-dessus donne à une variable entière (premier argument, dans ce cas `\l_tmpa_int`) la valeur 5 (donnée dans un groupe).

La fonction est donc très similaire à la classique `\setcounter` LaTeX, mais l'argument de `\int_set:Nn` peut également être calculé. La fonction `\int_use:N` donne la valeur de la variable donnée en argument, et dans l'exemple qui suit, l'imprime.

Cf. exemple 45 page suivante.

92. NdT : Pas de manipulation.

Exemple 45 (cf. page 74)

code

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { 5 + 2 * 3 }
\int_use:N \l_tmpa_int
\ExplSyntaxOff
```

résultat

11

Un exemple de type de données : l3dim

Cette section est consacrée au type de données identifiées par dim dans la liste précédente, et servant pour les longueurs. Les fonctions les plus courantes y sont abordées. Il existe des fonctions similaires pour d'autres types de données[89]. On trouvera un passage en revue complet dans [3].

Création et initialisation — type d'argument N/n

```
\dim_new:N
\dim_const:Nn
```

L'instruction \dim_new:N crée une nouvelle variable. Celle-ci existe alors globalement, mais elle peut aussi n'exister que localement. La convention de nommage est ici importante. Si nous souhaitons créer une variable locale et une variable globale pour le tutoriel, nous pouvons le faire ainsi :

```
\dim_new:N \l_dante_test_dim
\dim_new:N \g_dante_test_dim
```

Il est donc possible qu'il y ait deux variables avec la même description, dont l'une est définie localement et l'autre globalement.

Quand on crée une constante, on renseigne directement sa valeur lors de la déclaration :

```
\dim_const:Nn \c_dante_test_dim { 5cm }
```

Pour les variables, qui sont donc modifiables, l'affectation se fait séparément :

```
\dim_set:Nn \l_dante_test_dim { 3cm }
\dim_gset:Nn \g_dante_test_dim { 1cm + 5mm }
```

Comme pour les nombres, il est également possible de réaliser des calculs au moment de l'affectation. Il est important de noter que la précision est limitée à celle de TEX pour les longueurs. La plus petite unité est ici 1 sp = 0,000 02 pt. Ce qui est définitivement assez petit pour avoir une précision plus que suffisante pour les productions imprimées.

Outre l'affectation, il existe également des commandes pour l'addition et la soustraction de longueurs. La structure des noms reste identique. Tous les détails peuvent être consultés dans [3].

Requêtes et boucles — type d'argument T/T/TF

Lors du traitement des variables, il est également possible d'en comparer les valeurs et, le cas échéant, de créer des boucles à partir de celles-ci. Pour les longueurs, la commande la plus simple est la suivante :

```
\dim_compare:nTF
```

Cette macro est donc utilisée pour comparer des longueurs. Les opérateurs de comparaison sont :

égalité	= ou ==
supériorité ou égalité	>=
supériorité	>
infériorité ou égalité	<=
infériorité	<
inégalité	!=

Il existe quelques variantes de la fonction \dim_compare qui n'autorisent qu'une partie de ces opérateurs, ce qui permet de réduire les temps de traitement. Nous présentons ici la variante qui s'utilise le plus facilement. Comparons les longueurs que nous venons de créer :

Exemple 51

code

```
\ExplSyntaxOn
\dim_compare:nTF
{ \g_dante_test_dim >= \l_dante_test_dim }
{ est~supérieur~ou~égal }
{ est~inférieur }
\ExplSyntaxOff
```

résultat

est inférieur

Comme décrit plus haut, chacune des lettres du type d'argument TF correspond à un argument, qui peut être dans ce cas aussi bien un groupe qu'un *token* [93]. T et F signifient respectivement *true* et *false*. Mais le langage expl3 présente ici une particularité : si une seule réponse est attendue, il suffit d'omettre les types d'argument inutilisés — ici le T !

Exemple 52

code

```
\ExplSyntaxOn
\dim_compare:nF
{ \g_dante_test_dim > \l_dante_test_dim }
{ est~non~supérieur }
\ExplSyntaxOff
```

93. NdT :. Contrairement au couple N/n, T et F sont toujours écrits en majuscule.

résultat (suite)

est non supérieur

Sorties pour le débogage

Lors d'un travail complexe de programmation, il est utile de vérifier de temps à autre que le traitement des valeurs se déroule correctement. Pour ce faire, expl3 offre des commandes qui permettent d'afficher dans le terminal l'état actuel d'une variable ou de l'écrire dans un fichier journal.

```
\dim_show:N
\dim_show:n
\dim_log:N
\dim_log:n
```

Les macros présentant le type d'argument n donnent une expression de longueur et non une variable de longueur. Cela permet donc d'y effectuer des calculs ou d'y utiliser une macro, contenant par exemple une indication en centimètres, qui sera évaluée.

Spécification des types d'arguments de fonction

Outre les types d'arguments exposés plus haut, il en existe quelques autres, dont le comportement diffère de celui des arguments LaTeX classiques.

Type	Nom et description
N/n	***n**o manipulation*
TF/T/F	***t**rue/**f**alse*
c	***c**sname*
V/v	***v**alue*
o	*expand **o**nce*
e	***e**xhaustive expansion* (et si besoin la macro elle-même est développable)
x	*e **x**haustive expansion*
f	***f**ull expansion*
p	***p**aramètre*
w	***w**eird*

N et TF ont été exposés ci-avant. Nous en venons au type d'argument c. Les types o, f, x, e et V viendront ensuite. Les types p et w sont rarement utilisés par les néophytes ; nous n'en parlerons donc pas. Ils n'ont été mentionnés que par souci d'exhaustivité. On trouvera plus d'informations à leur sujet dans [4], par exemple.

csname/endcsname — type d'argument c

L'un des principes fondamentaux de LaTeX est la possibilité de créer des noms de macro de manière dynamique.

```
\csname nom\endcsname
```

Si l'on s'en tient à l'aspect visuel, l'environnement \csname/\endcsname ne fait rien d'autre que d'insérer une contre-oblique (\) avant le contenu dudit environnement :

Exemple 55

code

```
\LaTeX{}
=
\csname LaTeX\endcsname
```

résultat

LaTeX = LaTeX

Les références internes en sont un exemple classique d'utilisation. Ici, la macro \label crée une macro interne en lui donnant comme nom r@ suivi de son argument. La seconde ligne de l'exemple permet d'afficher la valeur de cette macro interne dans le terminal [94].

Exemple 56

code

```
\label{frame:csname}
\expandafter\meaning\csname r@frame:csname\endcsname
```

résultat

```
macro:->{}{78}
```

Une référence interne est ainsi créée sous la forme d'une macro dont la définition est constituée de deux groupes contenant le numéro de l'élément (ici vide, car non numéroté) et le numéro de la page où il se trouve. La commande \ref utilisera le premier groupe dans la valeur de la macro tandis que \pageref utilisera le second [95]. On trouvera plus d'informations à ce sujet, ainsi que de nombreux exemples, dans les actes de la conférence prononcée par Amy Hendrickson lors du TUG 2012 [5].

expl3 reprend ce concept pour le type d'argument c – c comme csname.

Exemple 57

```
\dim_set:Nn \l_tmpa_dim { 1cm }
\dim_set:cn { l_tmpb_dim } { 2cm }
\dim_set_eq:cc { l_tmpa_dim } { l_tmpb_dim }
```

Contrôle d'expansion

L'expansion désigne le remplacement d'une macro par sa signification. En s'en tenant aux commandes classiques de LaTeX, le résultat de l'expansion d'une macro peut être affiché en utilisant les commandes suivantes (la première affiche dans le document, la seconde dans le terminal) :

94. NdT : nous développons ici le texte original pour plus de clarté.
95. Voir note 94 ci-dessus.

```
\meaning\commande
\show\commande
```

On a ainsi :

Exemple 59

code

```
\newcommand*{\variable}{def}
\meaning\variable\\
\newcommand*{\fonction}[1]{fonction avec argument (#1)
   }
\meaning\fonction
```

résultat

```
macro:->def
macro:#1->fonction avec argument (#1)
```

La macro `\variable` est donc simplement remplacée par la chaîne de caractères `def`.

La macro `\fonction` accepte un argument et le place entre les parenthèses au lieu du `#1`.

Pour mettre en œuvre des structures plus complexes, d'autres macros sont souvent utilisées à l'intérieur des définitions de macros. Une expansion en plusieurs étapes est alors nécessaire.

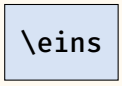
Expansion en plusieurs étapes

Nous utilisons les définitions suivantes[86] à titre d'exemple pour illustrer la manière dont LaTeX traite les commandes.

Exemple 60

```
\newcommand\eins{eins}
\newcommand\zwei{\eins,zwei}
\newcommand\drei{\eins,\zwei,drei}
```

Imaginons que chaque macro soit une boîte dont le contenu varie en fonction de sa définition :



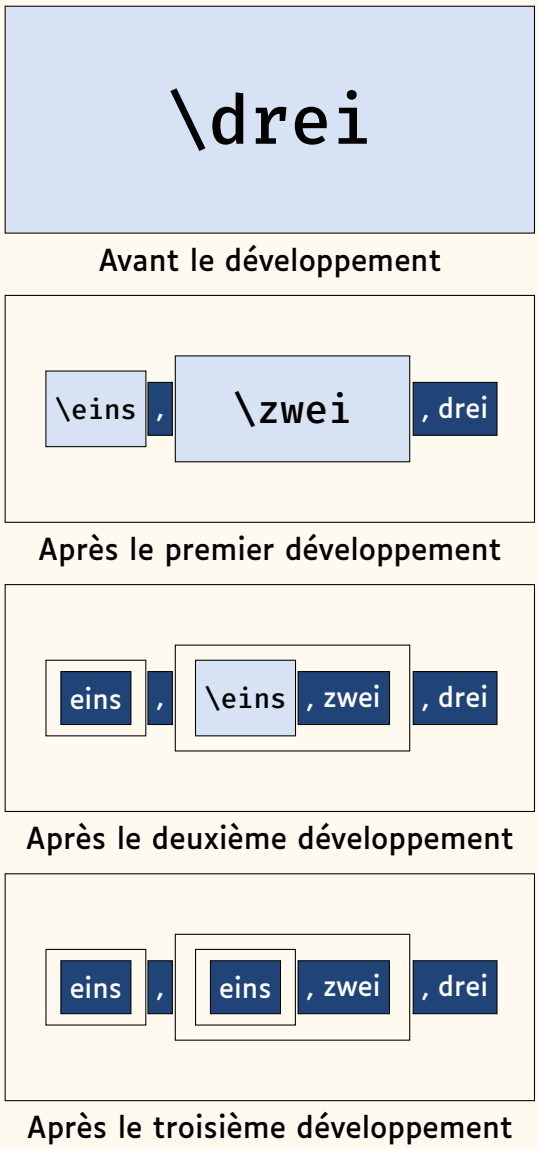
Sans expansion, TeX ne voit qu'un *token*. Si l'on veut poursuivre le traitement, il faut que la boîte soit déballée :

Comme cette macro ne contient que la chaîne de caractères `eins`[86], ce processus ne peut pas être répété. La macro est déjà entièrement développée.

Si la macro `\eins` n'est développable qu'une fois, `\drei` peut être développée plusieurs fois. Les différentes étapes de développement sont illustrées

en figure 21.

FIGURE 21 – Illustration des étapes de développement dans des boîtes.



expl3 permet, via les arguments, de contrôler explicitement — et très précisément — jusqu'à quel point les boîtes sont développées avant d'être traitées par la fonction.

On peut dès lors différencier les types d'arguments selon ce critère en utilisant directement une syntaxe expl3.

L'instruction `\tl_to_str:n` insère directement dans le document la chaîne de caractères qui représente son argument. Pour contrôler les niveaux de développement, on peut la faire précéder de `\exp_args:No` ou d'autres variantes de cette commande. Le premier argument de `\exp_args` est la macro dont on va traiter l'argument : il ne sera donc pas développé. Le deuxième argument de `\exp_args` est l'argument de la macro qui constitue son premier argument; on en contrôle le développement grâce à la seconde lettre qui suit les deux points, comme le montrent les exemples suivants [96] :

96. Voir note 95 page 78.

Exemple 61

code

```
\ExplSyntaxOn
% sans développement
\tl_to_str:n { \drei }              \\
% développement simple
\exp_args:No \tl_to_str:n { \drei } \\
% développement total
\exp_args:Nf \tl_to_str:n { \drei } \\
% développement ininterrompu (e)
\exp_args:Ne \tl_to_str:n { \drei }
\ExplSyntaxOff
```

résultat

```
\drei
\eins ,\zwei ,drei
eins,\zwei ,drei
eins,eins,zwei,drei
```

Comme on le voit, il est possible de contrôler précisément l'ordre dans lequel le développement des arguments des fonctions sera effectué. Il est ainsi possible de regarder à l'intérieur des boîtes avant qu'elles ne soient traitées. Ainsi, on peut par exemple vérifier dans quel format une date apparaîtra et, en fonction de cela, faire un choix définitif :

Exemple 62

```
\cs_new:Nn \__dantetut_parse_date:n {
  % La chaîne de caractères est divisée
  % suivant les traits d'union :
  \seq_set_split:Nnn \l_tmpa_seq { - } {#1}
  \int_compare:nTF {\seq_count:N \l_tmpa_seq > 1}
  {
    % Il y avait effectivement des traits d'union :
    % on suppose que la date était au format ISO.
    \seq_item:Nn \l_tmpa_seq { 3 } .
    \seq_item:Nn \l_tmpa_seq { 2 } .
    \seq_item:Nn \l_tmpa_seq { 1 }
  }{
    % D'autres possibilités existent.
    % Dans ce cas, on suppose que la date
    % est déjà mise en forme correctement.
    #1
  }
}
```

La macro vérifie que l'argument contient un trait d'union. Si c'est le cas, la date est interprétée comme une date ISO (AAAA-MM-JJ). Sinon, on suppose que la date a le format JJ.MM.AAAA. Il est bien sûr possible de le vérifier ensuite.

Pour l'exemple, imaginons maintenant que la date qui a été renseignée au

début du document via la commande \date passe dans notre macro. En interne, la valeur est stockée dans la macro \@date. L'argument passé à __dantetut_parse_date est alors le *token* \@date, et non une chaîne de caractères : pour pouvoir examiner le contenu de la date, il faut donc d'abord développer la macro [97].

Une possibilité serait d'utiliser la commande \exp_args. Mais expl3 fournit un mécanisme permettant de créer des variantes d'une commande de base :

```
\cs_generate_variant:Nn \__dantetut_parse_date:n {x}
```

Maintenant qu'existe la commande __dantetut_parse_date:x [98], il est possible de rédiger le code suivant :

Exemple 64

code

```
\ExplSyntaxOn
\__dantetut_parse_date:n { 23.06.2022 }
=
\__dantetut_parse_date:n { 2022-06-23 } =
\__dantetut_parse_date:x {
  \use:c { @date } % ou \makeatletter\@date
   \makeatother
}
\ExplSyntaxOff
```

résultat

23.06.2022=23.06.2022=23.06.2022

Résumé conclusif et perspectives

Nous avons vu que, malgré – ou grâce à – une syntaxe qui peut paraître un peu étrange au premier abord, LaTeX3 et expl3 étendent et simplifient la création de macros, en les rendant plus facilement adaptables. En complément du type de données de base (et unique) token, des types structurés de données sont introduits et une distinction est faite entre le traitement et le stockage des données. En outre, expl3 rend le contrôle du développement des macros beaucoup plus transparent qu'auparavant.

Il est ainsi possible d'automatiser le traitement de différents contenus et d'écrire des macros dont le comportement varie en fonction des arguments fournis et de leur structure. En outre, la structure uniforme des interfaces facilite énormément la lecture et la compréhension de codes écrits par d'autres [99].

De très nombreuses macros, qui ont été redéfinies à plusieurs reprises dans de nombreux packages, sont ainsi obsolètes, car il existe désormais des fonctions prêtes à l'emploi dans le noyau lui-même. Des familles entières de

97. Voir note 94 page 78 (promis, cette fois-ci pas de double renvoi !).
98. NdT : ... qui développe complètement son argument, selon le sens de x vu plus haut.
99. NdT : une fois familiarisé avec les conventions de la syntaxe !

packages sont déjà remplacées par des modules standards LaTeX3, comme par exemple les nombreux packages permettant d'implémenter le parsage des options du type clé/valeur. À long terme, cela réduira les conflits entre les packages, tout comme d'autres extensions du noyau, et rendra le système LaTeX encore un peu plus stable.

Cet article ne donne cependant qu'un très petit aperçu des possibilités. Celles-ci étaient certes déjà illimitées avec LaTeX classique, mais les fonctions avancées nécessitaient souvent des *patchs* et des *hacks* de bas niveau. Le package expl3 m'a permis beaucoup plus simplement de programmer directement en LaTeX.

J'espère vraiment que cet article permettra d'éclairer un peu le chemin qui nous mène vers expl3.

`\prg_do_nothing:` ou `\relax`

Marei Peischl

traduit par Patrick Bideault

Références

[1] Johannes L. Braams et Javier Bezos. *Babel, Localization and internationalization*. English. 2022. URL : http://mirrors.ctan.org/macros/latex/required/babel/base/babel.pdf.

[2] Joseph Wright. *Registering `expl3` module*. English. 2012. URL : https://www.texdev.net/2012/11/04/registering-expl3-module/.

[3] The LaTeX Project. *The LaTeX3 Interfaces*. English. 2022. URL : http://mirrors.ctan.org/macros/latex/contrib/l3kernel/interface3.pdf.

[4] *The `expl3` package and LaTeX3 programming*. English. 2022. URL : http://mirrors.ctan.org/macros/latex/contrib/l3kernel/expl3.pdf.

[5] Amy Hendrickson. *TUG 2012 Conference Proceedings*. English. 2012. URL : https://www.tug.org/tug2012/booklet/hendrickson/AmyTugProc.pdf.