Quelques aspects de la programmation avec Expl3*

Yvon HENEL

31 octobre 2026

Résumé

Quelques aspects de la programmation avec Expl3 présentés au travers de code calculant sur les entiers : écriture des n premiers entiers premiers ; crible d'ÉRATOSTHÈNE ; suites de KAPREKAR.

Remerciements

Je remercie François PANTIGNY pour sa lecture attentive de la version 1.

*Version 1.4

Table des matières

1 Introduction **3**

I Écrire les premiers nombres premiers 3

2 Rappel **3**

3 Version avec Expl3 **4**
3.1 Une macro privée pour écrire les nombres 4
3.2 Commande vérifiant qu'un entier impair est premier 5
3.2.1 Compléments 7
3.3 Écrire la liste des nombres premiers 7
3.3.1 Deux macros utilitaires 7
3.3.2 La commande de document 8
3.4 Utilisation de la commande de document 9
3.5 Quelques commentaires subjectifs 9

II Le crible d'Eratosthène 9

4 Buts et algorithmes **10**

5 Création des vecteurs **10**
5.1 Une commande auxiliaire 11
5.2 Traitement des entiers pairs 12
5.3 Traitement des entiers impairs 12

6 Deux exemples d'utilisation **13**
6.1 Écriture du crible 13
6.2 Décomposition d'un nombre en facteurs premiers 14

III Les suites de Kaprekar 15

7 Suites de Kaprekar **15**

8 Le code **15**
8.1 Initialisation 15
8.2 De l'entier à la suite et retour 16
8.3 Commande interne 18

IV Annexes 20

Glossaire **21**

Commandes, fonctions et variables **21**

1 Introduction

À l'occasion du stage de Dunkerque 2019, j'ai eu envie, pour présenter Expl3, de revenir sur un code donné par Knuth dans le TeXbook et que Denis Roegel avait décortiqué dans l'article « Anatomie d'une macro » paru dans les *Cahiers Gutenberg*, nº 31 (1998), p. 19-28.

Cette macro, bien évidemment écrite en TeX, calcule les n nombres premiers pour n entier supérieur à 3.

J'ai décidé de réécrire une macro produisant le même résultat à l'aide de Expl3 pour utiliser le code comme base de mon exposé. Une fois lancé, j'ai écrit d'autres petits morceaux de code pour présenter quelques fonctionalités du langage que le premier exemple ne m'avait pas permis d'exposer. C'est tout cela que je vais présenter dans ce qui suit.

Première partie
Écrire les premiers nombres premiers

2 Rappel

Je commence par redonner le code de Donald E. Knuth mais je laisse le lecteur rechercher l'article précité de Denis Roegel pour y trouver toutes les explications nécessaires.

Knuth - *définitions*

```
\newif\ifprime      \newif\ifunknown
\newcount\n         \newcount\p
\newcount\d         \newcount\a

\def \primes#1{2,~3\n=#1 \advance \n by-2  \p=5
  \loop \ifnum \n>0 \printifprime \advance \p by2 \repeat}

\def \printp{, \ifnum \n=1 and~\fi
  \number \p \advance \n by -1 }

\def \printifprime{\testprimality \ifprime\printp\fi}

\def \testprimality{{\d=3 \global \primetrue
   \loop \trialdivision \ifunknown \advance \d by2 \repeat}}

\def \trialdivision{\a=\p \divide \a by\d
  \ifnum \a>\d \unknowntrue \else \unknownfalse \fi
  \multiply \a by\d \ifnum \a=\p \global \primefalse \unknownfalse \fi}

\endinput
```

L'appel `\primes{15}` permet d'obtenir « 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, and 47 ».

3 Version avec Expl3

Commençons par quelques remarques : comme le code est écrit dans un fichier `.tex` — et non dans un *module* — je dois commencer par enclencher la syntaxe propre à Expl3, ce que je fais dès la première ligne qui n'est pas un commentaire à usage interne — d'où le décalage de numéro — dans le premier fragment ci-dessous :

TdS-*définitions*

```
3  \ExplSyntaxOn
4
5  \cs_new:Nn \__yh_ecrire_nombre:n { \(\np{\int_to_arabic:n{ #1 }}\) }
```

fragment1

3.1 Une macro privée pour écrire les nombres

Sur la ligne 5, on lit la définition d'une macro *interne* c.-à-d. d'une macro qui n'a pas vocation à être appelée directement dans un document rédigé par l'utilisateur final. On y voit déjà quelques différences avec du code LaTeX 2ε que je vais préciser maintenant :

— les caractères _ et : sont promus au rang de lettres mais pas l'arobase @ cher aux programmeurs en LaTeX 2ε ;

— les espaces du code ne donneront pas d'espace dans le document. Voilà qui soulage grandement le programmeur puisque disparait la phase de recherche des *blancs inopportuns* — anglais *spurious blanks*.

Autre remarque, d'ordre typographique, la présentation que j'adopte dans ce document — pour des raisons de place — n'est pas la présentation recommandée par l'équipe du projet LaTeX3. On devrait plutôt, en effet, adopter la présentation qui suit :

```
\cs_new:Nn \__yh_ecrire_nombre:n
  {
    \(\np{\int_to_arabic:n{ #1 }}\)
  }
```

dont on m'accordera, j'espère, qu'elle occupe plus d'espace.

La commande que je définis s'appelle `\__yh_ecrire_nombre:n`. Ce nom comporte plusieurs parties :

1. les « `__` » initiaux signalent que cette commande est *privée* c.-à-d. que son usage est réservée au module — ici le simple fichier contenant le code — et n'est pas nécessairement documentée. Le programmeur ne s'engage à rien sur cette commande et un utilisateur du module ne doit pas s'attendre à ce que le comportement de cette commande reste stable au fil des versions ni même à ce qu'elle soit toujours disponible. Il ne s'agit ici que d'une **convention** car TeX est resté ce qu'il est et la gestion des espaces de noms n'est pas implémentée[1] ;
2. « `yh` » remplace ici ce qui est — dans un module — le nom du module ou, à tout le moins, une chaine de caractères réservée pour ce module ;
3. vient le nom explicite de la commande : « `ecrire_nombre` » dans lequel le souligné « _ » sert de séparateur de mots ;

1. L'équipe des développeurs de Expl3 nous dit, dans [3, p. 1] que celà pourrait se faire mais avec un surcoût prohibitif.

4. suivi du caractère « : » qui introduit la *signature* de la commande qui est, ici, « n ». Cela nous indique que la commande est une *fonction* — c'est le vocabulaire défini par le projet LaTeX3 — qui attend un seul argument fourni entre accolades.

La fonction `\cs_new:Nn` utilise la signature de la commande qui suit pour déterminer le nombre et le type des arguments que demande cette commande. Elle vérifie que la commande n'est pas déjà définie et produit une erreur si c'est le cas. De plus, elle définit la commande de manière globale — penser à `\gdef` avec les pouvoirs de `\newcommand`.

Elle possède plusieurs variantes dont, p. ex., `\cs_new:cn` qui attend comme premier argument le *nom* de la commande plutôt que la commande elle-même. J'aurais obtenu le même résultat avec :

```
\cs_new:cn { __yh_ecrire_nombre:n } { \(\np{\int_to_arabic:n{ #1 }}\) }
```

code que je livre compacté.

Le 2e argument de `\cs_new:Nn` doit être fourni entre accolades, c'est la définition de la commande. Ici j'utilise `\np` de l'extension `numprint` en mode mathématique en ligne — à l'aide du bien connu couple `\(` et `\)` — afin d'obtenir « 3 257 » au lieu de « 3257 ».

La fonction `\int_to_arabic:n` appartient au module `int`, partie du noyau de LaTeX3, qui regroupe les commandes traitant les *entiers*. Elle prend un argument dont elle fournit la représentation en chiffres *arabes*.

3.2 Commande vérifiant qu'un entier impair est premier

La commande `\yh_impair_est_premier:nT` ne se déclare pas comme privée, dans un module elle devrait avoir une interface et une sortie stable et être documentée car son nom ne commence pas par `__` mais ça aurait pu être le cas.

2 étant le seul nombre pair premier, la question de la primarité ne se pose vraiment que pour les entiers impairs. Je pourrai donc n'essayer que des diviseurs impairs.

La signature de cette fonction, c.-à-d. `nT`, nous indique qu'elle attend deux arguments entre accolades, que le 1er sera un test qui renverra soit VRAI soit FAUX et que, *dans le seul cas* où le test est VRAI, le code contenu dans la 2e paire d'accolades sera considéré.

Regardons le code :

TdS-*définitions*

```
\cs_new:Nn \yh_impair_est_premier:nT {
  \bool_set_true:N \yh_continue_bool
  \bool_set_true:N \yh_est_premier_bool
  \int_set:Nn \l_tmpa_int {1}
  % FIN INITIALISATION
  \bool_do_while:Nn \yh_continue_bool { % BOUCLE `TANTQUE'
    % INCRÉMENTATION
    \int_add:Nn \l_tmpa_int { 2 }
    \bool_if:nTF {
    % TEST de divisibilité
      \int_compare_p:nNn { \int_mod:nn {#1} { \l_tmpa_int } } = { 0 }
    } { % SI VRAI
      \bool_set_false:N \yh_continue_bool
      \bool_gset_false:N \yh_est_premier_bool
    } { % SI FAUX
```

```
22        \int_compare:nNnT { \l_tmpa_int * \l_tmpa_int } > { #1 }
23        { \bool_set_false:N \yh_continue_bool }
24      } } % FIN DU SI ET FIN DE LA BOUCLE
25    \bool_if:nT {\yh_est_premier_bool} {#2} }
```
fragment2

Dans la partie d'initialisation — lignes 8 à 10 —, je crée deux *booléen*s — `\yh_continue_bool` et `\yh_est_premier_bool` — à l'aide de la fonction `\bool_set_true:N` qui, dans le même temps, leur donne la valeur VRAI.

Les fonctions dont le nom contient `set` définissent les commandes sans vérification et, à moins que le nom contienne `gset` — g comme « global » —, elles créent ces commandes *localement* c.-à-d. dans le groupe où a lieu la définition.

Ici, contrairement à ma devise[2], je cours le risque que ces deux booléens existent déjà, risque dont je pense qu'il est faible. On verra plus bas ce que j'aurais pu faire pour sécuriser la définition.

Ces deux booléens ne sont pas des fonctions mais des *variables* c.-à-d. des macros — au sens de TEX — dont le seul objet est de contenir une valeur et non pas de *faire* quelque chose. C'est pour cela que leurs noms n'ont pas de signature et ne se termine pas par : qui signalerait une fonction sans argument.

Dernière étape de l'initialisation, avec `\int_set:Nn` je donne à la variable entière **l**ocale temporaire `\l_tmpa_int` la valeur 1. Plusieurs modules du noyau de LATEX3 fournissent deux variables locales dont les noms comportent `tmpa` et `tmpb` respectivement et deux variables **g**lobales — avec les noms équivalents.

Suivant la convention générale qui veut que le nom d'une variable se termine par son type — c'est le cas des deux booléens que j'ai définis —, les variables temporaires fournies par un module du noyau ont une finale qui donne le type comme `\l_tmpa_int`. Cependant, contrairement à cette même convention de nommage, les variables temporaires n'ont pas, en tête de nom, le nom du module. On aura voulu éviter quelque chose comme `\int_l_tmpa_int` dont on peut penser que c'est légèrement redondant.

En ligne 12, je commence une boucle TANT QUE avec `\bool_do_while:Nn` qui attend un premier argument constitué d'un seul lexème — ici `\yh_continue_bool` qui est VRAI à l'entrée dans la boucle — et un deuxième argument entre accolades : le corps de la boucle.

Le corps de cette boucle commence avec l'accolade ouvrante du bout de la ligne 12 et s'achève avec la 2e accolade fermante de la ligne 24.

Je commence par ajouter 2 à la valeur courante de `\l_tmpa_int` à l'aide de la fonction `\int_add:Nn` ; `\l_tmpa_int` est ici le candidat diviseur.

`\bool_if:nTF` est une forme de SI… ALORS… SINON : on a bien les deux branches de l'alternative puisque la signature se termine en TF. Il s'agit de savoir si le candidat est bien un diviseur de l'entier représenté par #2. Pour ce faire, j'utilise la fonction `\int_compare_p:nNn` dont le 2e argument est un opérateur de comparaison c.-à-d. >, < ou =. Le 3e argument est simplement 0. Le premier, lui, est le reste de la division de #2 par `\l_tmpa_int` que l'on obtient à l'aide de la fonction `\int_mod:nn`.

Si le reste est nul, on exécute le code du 2e argument de `\bool_if:nTF`. On positionne les deux booléens à FAUX puisque #2, divisible par `\l_tmpa_int`, n'est pas premier et qu'il n'est donc pas nécessaire de reprendre la boucle.

Si le reste n'est pas nul, je compare l'entier #2 et le carré du candidat — on peut utiliser les opérateurs *classiques* d'opérations : +, -, *, / car, derrière ce code, c'est le mécanisme de la macro `\numexpr` de ε-TEX qui est à l'œuvre. Si le carré est plus grand, c'est que #2 est premier et qu'il est temps de s'arrêter d'où les valeurs données aux deux booléens. Le 3e argument du `\bool_if:nTF` s'arrête avec la 1re accolade fermante de la ligne 24.

2. À savoir : « ceinture et bretelles ».

À la sortie de la boucle, si #2 n'est pas premier il n'y a rien à faire, s'il est premier, on le retourne. Cela est accompli à l'aide de la fonction `\bool_if:nT` qui réalise un saut conditionnel SI… ALORS.

3.2.1 Compléments

Outre les deux fonctions `\bool_if:nTF` et `\bool_if:nT`, Expl3 fournit également la fonction `\bool_if:nF` qui réalise un saut conditionnel SI… SINON. Dans le manuel, on verra que la signature se termine par *TF* pour signaler la présence des trois signatures T, F et TF.

Par ailleurs, Expl3 fournit essentiellement quatre boucles :

— `\bool_do_while:Nn`, déjà rencontrée. C'est un TANT QUE dont le corps est exécuté puis le test effectué c.-à-d. qu'en toutes circonstances, le corps est exécuté au moins une fois ;

— `\bool_do_until:Nn`, qui inverse le sens du test. C'est un JUSQU'À dont le corps est exécuté au moins une fois ;

— `\bool_while_do:Nn`, TANT QUE qui commence par le test. Il se peut donc que le corps ne soit pas exécuté ;

— `\bool_until_do:Nn`, JUSQU'À commençant par le test.

Ces quatres fonctions ont des variantes de signatures cn et nn. Avec cn, le 1er argument est le nom d'un booléen ; avec n, c'est une expression booléenne comme — exemple tiré de [1] —

```
\int_compare_p:n { 1 = 1 } &&
  (
    \int_compare_p:n { 2 = 3 } ||
    \int_compare_p:n { 4 <= 4 } ||
    \str_if_eq_p:nn { abc } { def }
  ) &&
! \int_compare_p:n { 2 = 4 }
```

qui permet de mesurer la complexité acceptée.

3.3 Écrire la liste des nombres premiers

Avant d'aborder la commande destinée à l'utilisateur — dont je dirai que c'est une *commande de document*, comme le suggère le nom de la macro qui permettra de la créer tout à l'heure. L'extension xparse [4] parle de *document-level command* —, je présente deux nouvelles macros utilitaires — qui elles aussi auraient pu être *privées* — à savoir `\yh_ecrire_separateur:n` et `\yh_ecrire_si_premier:n`.

3.3.1 Deux macros utilitaires

TdS-*définitions*

```
27 \cs_new:Nn \yh_ecrire_separateur:n { }
28
29 \cs_new:Nn \yh_ecrire_si_premier:n {
30   \yh_impair_est_premier:nT { #1 }
31   { \yh_ecrire_separateur:n { #1 }
32     \__yh_ecrire_nombre:n { #1 } } }
```

fragment3

On voit dans le code ci-dessus, en ligne 27, une pré-déclaration de `\yh_ecrire_separateur:n` qui permet juste de s'assurer que ce nom est bien disponible.

La macro suivante `\yh_ecrire_si_premier:n` fait exactement ce que son nom suggère.

Passons maintenant à la commande principale.

3.3.2 La commande de document

TdS-*définitions*

```
34  \NewDocumentCommand { \ListePremiers } { m } {
35    \group_begin:
36    % INITIALISATION
37    \int_new:N \l_compteur_int
38    \int_new:N \l_nombre_requis
39    \int_set:Nn \l_compteur_int { 2 }
40    \int_set:Nn \l_nombre_requis { #1 }
41    \cs_set:Nn \yh_ecrire_separateur:n {
42      \int_compare:nNnTF { ##1 } < { \l_nombre_requis - 1} {,~} {~et~} }
43    % LES DEUX PREMIERS NOMBRES PREMIERS NE SONT PAS CALCULÉS
44    \__yh_ecrire_nombre:n {2}
45    \yh_ecrire_separateur:n {1}
46    \__yh_ecrire_nombre:n {3}
47    % PRÉPARATION DE LA BOUCLE, \l_tmpb_int EST LE CANDIDAT À TESTER
48    \int_set:Nn \l_tmpb_int { 3 }
49    % BOUCLE `TANT QUE'
50    \int_while_do:nNnn { \l_compteur_int } < { \l_nombre_requis } {
51      \int_add:Nn \l_tmpb_int { 2 }
52      \yh_impair_est_premier:nT { \l_tmpb_int }
53      { \yh_ecrire_separateur:n { \l_compteur_int }
54        \__yh_ecrire_nombre:n{ \l_tmpb_int }
55        \int_incr:N \l_compteur_int } }
56    % ON FAIT LE MÉNAGE AVANT DE SORTIR
57    \cs_undefine:N  \l_compteur_int
58    \cs_undefine:N  \l_nombre_requis
59    \group_end: }
```

fragment4

Le code permettant de définir cette commande utilise l'extension `xparse` [4] et sa macro `\NewDocumentCommand`.

La syntaxe de déclaration d'argument change quelque peu ! Le `m`, 2e argument de la macro, indique que l'on définit une commande requérant un unique argument obligatoire fourni entre accolades — `m` pour *mandatory* c.-à-d. « obligatoire » . Le 1er argument est, comme on s'y attend, le nom de la commande créée.

Le corps de la définition commence en ligne 35 en ouvrant un groupe avec `\group_begin:` et finit en ligne 59 en fermant le groupe avec `\group_end:`.

Initialisation Dans le groupe, je crée en lignes 37 et 38 deux entiers locaux — d'où le `l` initial — mais qui seront créés *globalement*, car il n'y a pas moyen de faire autrement. Leur caractère local est donc conventionnel mais signale que l'on ne doit pas s'attendre à ce qu'ils aient une valeur pertinente à l'extérieur du groupe.

Pour respecter ce caractère local, je donne à ces entiers, une valeur avec `\int_set:Nn` et pas avec `\int_gset:Nn` qui, de fait, n'est qu'un alias de la première, à moins que ce soit l'inverse[3].

Afin que, en sortie de commande, ces deux entiers ne soient plus accessibles — *Vous êtes priés de laisser ces lieux dans l'état où vous auriez aimé les trouver en entrant!* —, je les annihile en lignes 57 et 58 à l'aide de `\cs_undefine:N`.

Il est temps à présent de définir, localement, avec `\cs_set:Nn`, la fonction `\yh_ecrire_separateur:n`, ce que je fais en ligne 41 et 42. Comme la définition est faite dans une définition, on doit, bien entendu, recourir au doublement des *dièses* d'où le `##1` dans le code de cette fonction.

Dans les 1er et 3e arguments de `\int_compare:nNnTF` on peut écrire *naturellement* des opérations sur les entiers. Je ne m'en prive pas.

Les tildes ~ que l'on voit apparaitre en ligne 42 permettent de coder un espace qui apparaitra dans le document final. Par contre, il faut utiliser autre chose pour coder un espace insécable — la solution est la macro `\nobreakspace`.

Je suis obligé de regarder si le séparateur est le dernier car, en français[4] il ne doit pas y avoir de virgule devant le *et*.

Comme l'annonce la ligne 43 je ne calcule pas les deux premiers nombres premiers — Donald Knuth ne le faisait pas — et je ne vérifie pas que l'utilisateur demande au moins trois nombres — même remarque —; pour faire comme dans les livres sérieux, je laisse cette amélioration comme exercice pour le lecteur.

Je prépare maintenant la boucle en initialisant, localement, le *candidat premier* à 3. On a déjà vu `\int_set:Nn`. Et je rentre dans la boucle en ligne 50 pour en sortir en ligne 55. La seule nouveauté de ce fragment de code est l'emploi de `\int_incr:N` qui permet d'incrémenter localement le compteur `\l_compteur_int`.

J'ai déjà expliqué ci-dessus l'utilité des lignes 57 à 59.

3.4 Utilisation de la commande de document

Il me reste à utiliser cette commande dans ce document : `\ListePremiers{15}` produit « 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 et 47 ».

Au passage, on notera avec satisfaction que `\primes` et `\ListePremiers` sont d'accord quant aux quinze premiers nombres premiers. C'est rassurant.

3.5 Quelques commentaires subjectifs

J'accorde sans peine qu'avec Expl3 le code est plus verbeux qu'avec TEX tout seul. Je ne peux, cependant, m'empêcher de penser qu'il est plus clair. Pour ma part, le gain énorme que je vois en passant cette fois de LATEX 2ε à Expl3 c'est le systématisme du nommage des macros, fonctions ou variables. L'équipe du projet LATEX3 en fait d'ailleurs, elle-même, un "*argument de vente*".

La puissance de `\NewDocumentCommand` — que l'on a à peine effleuré ci-dessus —, à elle seule, peut inciter à passer à Expl3 pour créer ses propres macros.

3. Je ne suis pas allé vérifier dans le source [2].
4. Encore que, si j'ai bien compris un article récent paru dans la presse britanique, même à Oxford on a fini par renoncer à la virgule devant le *and* précédant le dernier mot d'une énumération. *Tout fout l'camp, ma bonne dame!*

Deuxième partie

Le crible d'Eratosthène

Il s'agit maintenant de fournir quelques commandes de document permettant de créer un exemple du crible d'Eratosthène comme ceci :

1	**2**	**3**	$4^{\langle 2\rangle}$	**5**	$6^{\langle 2\rangle}$	**7**	$8^{\langle 2\rangle}$	$9^{\langle 3\rangle}$	$10^{\langle 2\rangle}$
11	$12^{\langle 2\rangle}$	**13**	$14^{\langle 2\rangle}$	$15^{\langle 3\rangle}$	$16^{\langle 2\rangle}$	**17**	$18^{\langle 2\rangle}$	**19**	$20^{\langle 2\rangle}$
$21^{\langle 3\rangle}$	$22^{\langle 2\rangle}$	**23**	$24^{\langle 2\rangle}$	$25^{\langle 5\rangle}$	$26^{\langle 2\rangle}$	$27^{\langle 3\rangle}$	$28^{\langle 2\rangle}$	**29**	$30^{\langle 2\rangle}$
31	$32^{\langle 2\rangle}$	$33^{\langle 3\rangle}$	$34^{\langle 2\rangle}$	$35^{\langle 5\rangle}$	$36^{\langle 2\rangle}$	**37**	$38^{\langle 2\rangle}$	$39^{\langle 3\rangle}$	$40^{\langle 2\rangle}$
41	$42^{\langle 2\rangle}$	**43**	$44^{\langle 2\rangle}$	$45^{\langle 3\rangle}$	$46^{\langle 2\rangle}$	**47**	$48^{\langle 2\rangle}$	$49^{\langle 7\rangle}$	$50^{\langle 2\rangle}$
$51^{\langle 3\rangle}$	$52^{\langle 2\rangle}$	**53**	$54^{\langle 2\rangle}$	$55^{\langle 5\rangle}$	$56^{\langle 2\rangle}$	$57^{\langle 3\rangle}$	$58^{\langle 2\rangle}$	**59**	$60^{\langle 2\rangle}$
61	$62^{\langle 2\rangle}$	$63^{\langle 3\rangle}$	$64^{\langle 2\rangle}$	$65^{\langle 5\rangle}$	$66^{\langle 2\rangle}$	**67**	$68^{\langle 2\rangle}$	$69^{\langle 3\rangle}$	$70^{\langle 2\rangle}$
71	$72^{\langle 2\rangle}$	**73**	$74^{\langle 2\rangle}$	$75^{\langle 3\rangle}$	$76^{\langle 2\rangle}$	$77^{\langle 7\rangle}$	$78^{\langle 2\rangle}$	**79**	$80^{\langle 2\rangle}$
$81^{\langle 3\rangle}$	$82^{\langle 2\rangle}$	**83**	$84^{\langle 2\rangle}$	$85^{\langle 5\rangle}$	$86^{\langle 2\rangle}$	$87^{\langle 3\rangle}$	$88^{\langle 2\rangle}$	**89**	$90^{\langle 2\rangle}$
$91^{\langle 7\rangle}$	$92^{\langle 2\rangle}$	$93^{\langle 3\rangle}$	$94^{\langle 2\rangle}$	$95^{\langle 5\rangle}$	$96^{\langle 2\rangle}$	**97**	$98^{\langle 2\rangle}$	$99^{\langle 3\rangle}$	$100^{\langle 2\rangle}$

Ce sera l'occasion d'évoquer un des « objets de haut niveau » fourni par Expl3. Non, je ne parle pas de programmation objet !

Une fois encore, je place le code dans un "bête" fichier `.tex` d'où les bien connus désormais `\ExplSyntaxOn` et `\ExplSyntaxOff` en début et fin de fichier.

4 Buts et algorithmes

L'objet dont il est question est appelé *intarray*. Il est global et, une fois créé, ne peut être détruit[5]. Dans certains langages on parle de *vecteurs* mais il s'agit ici d'un vecteur d'entiers (relatifs) et d'entiers seulement. Sa taille doit être donnée à la création et ne peut pas être changée.

Avec la commande `\eratosthene`, on crée deux *vecteurs d'entiers*. Le premier, `\__ERA_dv_ia`, contient en n^{e} place le plus petit diviseur premier de n. Le second, `\__ERA_qt_ia`, contient en n^{e} place le quotient de n et de ce diviseur premier dont je viens de parler.

La commande `\eratosthene` ne prend qu'un argument, un entier naturel, qui est la taille des deux vecteurs sus-définis.

Une fois les vecteurs créés, on pourra les utiliser pour écrire le crible d'Eratosthène comme vu précédemment ou, pour donner un autre exemple, obtenir la décomposition d'un entier comme produit de ses diviseurs premiers.

5 Création des vecteurs

commande eratosthene, initialisation

```
5  \NewDocumentCommand{ \eratosthene } { m } {
```

5. C'est du moins ce que je comprends à la lecture — que j'ai voulu rapide — du source. En tout cas Expl3 ne fournit pas d'outil pour ce faire.

```
6    \group_begin:
7    \int_new:N \l_ERA_diviseur_max_int
8    \int_set:Nn \l_ERA_diviseur_max_int { \fp_to_int:n { floor( sqrt ( #1 ))} }
9    \intarray_new:Nn \__ERA_qt_ia { #1 }
10   \intarray_new:Nn \__ERA_dv_ia { #1 }
```
fragment1

En ligne 6 j'ouvre un groupe avec `\group_begin:` qui sera fermé juste avant l'accolade fermante marquant la fin de la définition de `\eratosthene` en ligne 45.

Dans ce groupe je vais créer un *entier* avec `\int_new:N`. Il est créé globalement mais je le détruis en ligne 44 avec `\cs_undefine:N`. Il ne survivra donc pas à la macro.

Je crée également les deux vecteurs en lignes 9 et 10 à l'aide de `\intarray_new:Nn` qui prend en 2e argument la taille du vecteur qui est l'argument passé à `\eratosthene`. Initialement ces vecteurs ne contiennent que des zéros.

Les noms des vecteurs devraient se terminer par `_intarray` mais je me suis permis un raccourci en `_ia`. Leurs noms commencent avec le double souligné puisque ce sont des variables *privées*. `ERA` est le sigle de ce pseudo-module.

En ligne 8 je donne à la variable `\l_ERA_diviseur_max_int` la partie entière de la racine carrée de la taille des vecteurs — valeur donnée par `#1` — en *transtypant* le résultat de l'opération `floor( sqrt ( #1 ))` qui est un *flottant* — pour "nombre réel codé *en virgule flottante*" — en un entier.

5.1 Une commande auxiliaire

Je crée maintenant, à l'intérieur de la définition de `\eratosthene` la commande `\ERA_marquer_de_a_par:nnn` que, de fait, je n'utiliserai qu'une fois, en ligne 33, et dont, par conséquent, j'aurais pu faire l'économie, au risque d'embrouiller considérablement le code suivant.

commande eratosthene, auxiliaire
```
12   \cs_set:Nn \ERA_marquer_de_a_par:nnn {
13     \int_set:Nn \l_tmpa_int { ##1 / ##3}
14
15     \int_step_inline:nnnn { ##1 } { ##3 } { ##2 } {
16       \int_compare:nNnT { \intarray_item:Nn  \__ERA_dv_ia { ####1 } } = { 0 }
17       { \intarray_gset:Nnn \__ERA_dv_ia  { ####1 } { ##3 }
18         \intarray_gset:Nnn \__ERA_qt_ia  { ####1 } { \l_tmpa_int } }
19       \int_incr:N \l_tmpa_int }
20   }
```
fragment2

Je crée cette commande localement grâce à `\cs_set:Nn`, elle ne survivra donc pas au groupe. Je commence, dans la définition de `\ERA_marquer_de_a_par:nnn`, par donner comme valeur à la variable *locale* `\l_tmpa_int` le quotient entier des 1er et 3e arguments de la commande.

Puis, lignes 15 à 19, j'utilise `\int_step_inline:nnnn` pour accomplir le travail. C'est une boucle POUR. L'index — implicite et anonyme — parcourt les entiers en partant du premier argument de cette dernière macro jusqu'au troisième argument avec un pas donné par le deuxième. Le 4e argument est le code *en ligne* dans lequel `#1` est l'indice de la boucle. Enfin, `#1` serait l'indice de la boucle si on utilisait directement la macro `\int_step_inline:nnnn` au niveau du document. Ici comme je l'utilise dans une définition incluse elle-même dans une définition, je dois doubler les dièses deux fois d'où le `####1` que l'on voit dans le code.

En ligne 19, j'incrémente \l_tmpa_int. De la ligne 16 à la ligne 18, j'ai un test SI... ALORS avec \int_compare:nNnT : les actions des lignes 17 et 18 ne seront accomplies que s'il est vrai que la ####1-ième composante du vecteur __ERA_dv_ia est nulle c.-à-d. que ce nombre n'a pas encore été traité. Dans ce cas, je place la valeur du pas ##3 — c'est le plus petit diviseur premier du nombre considéré — à la bonne place dans le vecteur des diviseurs et je place le quotient à la place idoine dans le vecteur des quotients. L'incrémentation faite en-dessous me dispense de calculer ce quotient à l'aide d'une division entière.

Comme le nom l'indique la macro \intarray_gset:Nnn réalise une affectation **g**lobale dans le vecteur donné en 1er argument. C'est bien ce que je veux. Le 2e argument donne l'indice, le 3e la valeur à y placer.

5.2 Traitement des entiers pairs

Je retourne maintenant au niveau de la définition de la commande \eratosthene et je commence par traiter les nombres pairs.

commande eratosthene, nombres pairs

```
23   \int_set:Nn \l_tmpa_int { 1 }
24
25   \int_step_inline:nnnn { 2 } { 2 } { #1 }
26   { \intarray_gset:Nnn \__ERA_dv_ia  { ##1 } { 2 }
27     \intarray_gset:Nnn \__ERA_qt_ia  { ##1 } { \l_tmpa_int }
28     \int_incr:N \l_tmpa_int }
```

fragment3

À ce moment de l'action, les deux vecteurs ne contiennent que des zéros, cela nous dispense du test tel qu'il apparaît dans la macro auxiliaire vue ci-dessus. De plus, le premier quotient (2/2) est clairement 1 d'où la ligne 23.

On retrouve ensuite \int_step_inline:nnnn pour traiter les nombres pairs en partant de 2, avec un pas de 2, et en finissant à la taille des vecteurs #1.

5.3 Traitement des entiers impairs

Le traitement des entiers impairs se fera en deux étapes, la deuxième étant d'une simplicité déconcertante.

commande eratosthene, impairs

```
31   \int_set:Nn \l_tmpb_int { 3 }
32
33   \int_while_do:nNnn { \l_tmpb_int } < { \l_ERA_diviseur_max_int }
34   { \ERA_marquer_de_a_par:nnn { \l_tmpb_int } { #1 } { \l_tmpb_int }
35     \int_do_until:nNnn {\intarray_item:Nn  \__ERA_dv_ia { \l_tmpb_int }} = { 0 }
36     { \int_add:Nn \l_tmpb_int { 2 } } }
```

fragment4

Nous avons déjà vu tout ce que j'utilise ici. On notera, lignes 35 et 36, la boucle permettant, en avançant de 2 en 2, de trouver le prochain nombre impair non traité avec \int_do_until:nNnn analogue, pour réaliser une boucle sur les entiers, de \bool_do_until:Nn présentée page 7.

La dernière boucle traite les nombres impairs qui n'ont pas encore été vus. On pourrait améliorer la chose en commençant avec le premier impair non-vu, une fois encore je laisse ça en exercice!

commande eratosthene

```
  \int_step_inline:nnnn { 3 } { 2 } { #1 }
  { \int_compare:nNnT { \intarray_item:Nn  \__ERA_dv_ia { ##1 } } = { 0 }
    { \intarray_gset:Nnn \__ERA_dv_ia  { ##1 } { ##1 }
      \intarray_gset:Nnn \__ERA_qt_ia  { ##1 } { 1 } } }
```

fragment5

6 Deux exemples d'utilisation

Les deux vecteurs ayant été créés, on peut les utiliser. Je donne deux exemples : l'écriture du « crible », l'écriture d'un nombre comme produit de ses facteurs premiers — version améliorable.

6.1 Écriture du crible

Je présente maintenant la commande permettant de créer le tableau présenté en page 10.

écriture du crible, commande de document

```
\NewDocumentCommand { \EcrireCribleEratosthene } { m }{
  \par \noindent \int_step_inline:nn { #1 } { \ERA_presenter_nieme:n {##1} } }
```

fragment1

L'unique argument de `\EcrireCribleEratosthene` doit être un entier positif plus petit que la taille des deux vecteurs créés auparavant sinon on aura une erreur d'accès aux vecteurs dû à un indice hors bornes.

Bien entendu, cette commande se contente de refiler le boulot à la commande interne `\ERA_presenter_nieme:n` par l'intermédiaire de la boucle `\int_step_inline:nn`.

Soulevons le capot.

écriture du crible, commande interne

```
\cs_new:Nn \ERA_presenter_nieme:n {
  \group_begin:
  \int_set:Nn \l_tmpa_int { \intarray_item:Nn \__ERA_qt_ia { #1 } }
  \int_set:Nn \l_tmpb_int { \int_mod:nn { #1 } { 10 } }

  \framebox[4em] {\strut \footnotesize
    \int_compare:nNnTF { \l_tmpa_int } = { 1 }
      {\textbf}           % premier
      {\textcolor{gray}}  % composé
    {\int_to_arabic:n { #1 }}
    \int_compare:nNnT { \l_tmpa_int } > { 1 }
    { \int_set:Nn \l_tmpa_int { \intarray_item:Nn \__ERA_dv_ia { #1 } }
      ~\({}^{\langle \int_to_arabic:n { \l_tmpa_int } \rangle}\) } }
  \kern-\fboxrule
  % saut de paragraphe tous les 10
  \int_compare:nNnT { \l_tmpb_int } = { 0 }
  { \par \nointerlineskip \kern-\fboxrule \noindent }
  \group_end: }
```

fragment2

Pour la création du tableau, je copie servilement le code que fourni Christian TELLECHEA dans [5]. J'utilise quand même la macro \framebox fournie par LaTeX 2_ε car personne n'est obligé de réinventer la roue chaque matin !

6.2 Décomposition d'un nombre en facteurs premiers

Regardons à présent la commande permettant d'obtenir :
$78 = 2 \times 3 \times 13$ $75 = 3 \times 5 \times 5$ $64 = 2 \times 2 \times 2 \times 2 \times 2 \times 2$
avec le code
\EcrireDiviseurs{78} \qquad \EcrireDiviseurs*{75} \qquad \EcrireDiviseurs*{64}.

Voici le code :

facteurs premiers

```
73  \NewDocumentCommand { \EcrireDiviseurs } { s m } {
74    \group_begin:
75    \int_new:N \l_ERA_nv_qt_int         % quotient
76    \int_set:Nn \l_ERA_nv_qt_int { #2 }
77    \int_new:N \l_ERA_vx_dv_int         % ancient diviseur
78    \int_set:Nn \l_ERA_vx_dv_int { 0 }
79    \int_new:N \l_ERA_nv_dv_int         % nouveau diviseur
80    \int_set:Nn \l_ERA_nv_dv_int { \intarray_item:Nn \__ERA_dv_ia { #2 } }
81
82    \IfBooleanF{#1}{\par}
83    \( \int_to_arabic:n { #2 } =
84    \int_while_do:nNnn { \l_ERA_nv_qt_int } > { 1 }
85    { \int_compare:nNnT { \l_ERA_vx_dv_int } > { 0 } { \times }
86      \int_to_arabic:n { \l_ERA_nv_dv_int }
87      \int_set:Nn \l_ERA_vx_dv_int { \l_ERA_nv_dv_int }
88      \int_set:Nn \l_ERA_nv_qt_int {
89        \intarray_item:Nn \__ERA_qt_ia { \l_ERA_nv_qt_int } }
90      \int_set:Nn \l_ERA_nv_dv_int {
91        \intarray_item:Nn \__ERA_dv_ia { \l_ERA_nv_qt_int } } }
92    \)
93    \cs_undefine:N \l_ERA_nv_qt_int
94    \cs_undefine:N \l_ERA_vx_dv_int
95    \cs_undefine:N \l_ERA_nv_dv_int
96    \group_end: }
```

fragment3

Comme on l'aura remarqué, la commande \EcrireDiviseurs admet une variante étoilée. On l'obtient facilement avec \NewDocumentCommand en donnant comme descriptif de 1er argument le s que l'on voit en ligne 73.

Avec \IfBooleanF de la ligne 82, on ne place un saut de paragraphe — avec \par — que s'il n'y a **pas** d'étoile.

Pour réagir à la présence ou l'absence d'un lexème — une étoile * avec le descripteur d'argument s, n'importe quel lexème avec t — on dispose également de \IfBooleanT et de \IfBooleanTF.

On retrouve les mêmes techniques de création d'entiers dans un groupe et leur destruction en sortie du corps de la définition de la commande. J'évite les entiers de brouillon fournis par Expl3 car je préfère, pour ne pas me perdre, des noms plus explicites. De toute façon, j'ai besoin d'au moins trois entiers locaux. On devrait pouvoir écrire la décomposition de 75 sous la forme $75 = 3 \times 5^2$ à l'aide d'un entier supplémentaire. Là encore, l'exercice etc.

Troisième partie

Les suites de Kaprekar

7 Suites de Kaprekar

Une suite de Kaprekar est définie par la donnée d'un nombre entier, p. ex. 125, d'une base — je prendrai systématiquement et uniquement 10 comme base de l'écriture des nombres entiers — et d'un nombre N de chiffres que j'appellerai la *taille* du nombre. On obtient un terme u_{n+1} à partir du terme précédent u_n en appliquant l'algorithme suivant :

1. on écrit u_n avec N chiffres dans la base choisie quitte à compléter avec des zéros ;
2. on range les N chiffres ainsi obtenus dans l'ordre décroissant pour obtenir v_n ;
3. on range les N chiffres dans l'ordre croissant pour obtenir w_n ;
4. enfin, $u_{n+1} = v_n - w_n$.

Une telle suite est évidemment finie. La commande que nous allons détailler s'arrête sur le dernier nombre déjà obtenu.

En voici deux exemples.

On obtient « 125 ; 5 085 ; 7 992 ; 7 173 ; 6 354 ; 3 087 ; 8 352 et 6 174 » avec `\Kaprekar{125}` qui, par défaut, travaille avec $N = 4$, et « 125 ; 520 875 ; 849 942 ; 753 543 ; 420 876 ; 851 742 ; 750 843 ; 840 852 ; 860 832 ; 862 632 et 642 654 » pour lequel on a pris $N = 6$.

L'objet qui va nous servir à créer ces suites est la *sequence*, en français “suite” — dont le sigle est `seq` — que fournit l'extension `l3seq`.

8 Le code

Je rappelle que tout le code est placé entre `\ExplSyntaxOn` et `\ExplSyntaxOff` pour les raisons déjà exposées.

La commande de document est `\Kaprekar` dont voici le code.

Kaprekar, commande de document

```
\NewDocumentCommand{\Kaprekar}{ O{4} m }{
  \KAP_ecrire_tous_les_suivants:nn { #1 } { #2 } }
```

fragment1

On y retrouve l'habituelle `\NewDocumentCommand` mais un nouveau descripteur d'argument, à savoir `O{4}` : le premier argument est optionnel et sa valeur par défaut est 4. Si on veut en fournir une autre, on le placera entre crochet comme d'habitude.

La commande n'est, de fait et à part le truc de l'argument optionnel, qu'un alias de `\KAP_ecrire_tous_les_suivants:nn` que je vais maintenant détailler. Mais commençons par la phase d'initialisation.

8.1 Initialisation

Kaprekar, initialisation

```
\seq_new:N \kaprun_seq
\seq_new:N \kaprekar_seq
\int_new:N \l_tmpc_int
```

fragment2

Avec `\seq_new:N` je crée deux *sequences* : `\kaprun_seq` et `\kaprekar_seq`. La première variable a vocation à recevoir la suite des chiffres du nombre traité. La seconde contiendra la suite de KAPREKAR.

Je crée ensuite un entier *local* en suivant le modèle de l'extension `int`.

8.2 De l'entier à la suite et retour

J'ai besoin de commandes internes — qui pourraient être privées — pour *découper* un entier en la suite de ses chiffres et pour créer un entier à partir d'une suite de chiffres. Le fait de travailler en base fixée — et en base 10 qui plus est — facilite largement le travail. Là encore, si le cœur vous en dit, exercice !

Kaprekar, de l'entier à la suite

```
9  \cs_new:Nn \KAP_int_to_seq:NN {
10   \int_set:Nn \l_tmpa_int { #1 }
11   \int_do_until:nNnn { \l_tmpa_int } = { 0 }
12   { \int_set:Nn \l_tmpb_int { \int_mod:nn { \l_tmpa_int } { 10 } }
13     \seq_put_left:NV #2 \l_tmpb_int
14     \int_set:Nn \l_tmpa_int
15     { \int_div_truncate:nn { \l_tmpa_int - \l_tmpb_int } { 10 } } } }
```

fragment3

Le premier argument de `\KAP_int_to_seq:NN` est le nombre à découper, le second la suite qui contiendra ses chiffres. Dans les deux cas, la commande attend une variable.

Nous avons déjà rencontré la majorité des commandes utilisées dans la définition. Avec `\seq_put_left:NV` je place en première place — à gauche *left* — la valeur contenue dans la variable `\l_tmpb_int`. Le `V` dans la signature assure que la commande `\l_tmpb_int` sera traitée de telle sorte que l'on récupère bien son contenu.

Dans la boucle, on trouve `\int_div_truncate:nn` qui permet de récupérer le quotient entier de #1 par #2.

Kaprekar, de la suite à l'entier

```
17 \cs_new:Nn \KAP_seq_to_int:NN {
18   \seq_pop_right:NNT #1 \l_tmpa_tl
19   { \int_set:Nn #2 { \l_tmpa_tl }
20     \bool_do_until:nn { \seq_if_empty_p:N #1 }
21     { \seq_pop_right:NNT #1 \l_tmpa_tl
22       \int_set:Nn #2 { 10 * #2 + \l_tmpa_tl } } } }
```

fragment4

Cette commande transforme une suite de chiffres en un entier. On trouve, en ligne 18, la commande `\seq_pop_right:NNT` qui prend la valeur la plus à droite dans la suite passée en 1er argument pour la placer dans la variable donnée en 2e argument. Si cette opération est possible — c.-à-d. si la pile n'était pas vide — le 3e argument est développé.

La suite, après cette opération, est amputée de sa dernière valeur. J'utilise ici la suite comme une pile — c'est le fameux LIFO pour *Last In First Out* . Expl3 fournit aussi des commandes qui permettent d'utiliser la suite comme une queue — FIFO pour *First In First Out*.

Dans la boucle, j'utilise `\seq_if_empty_p:N` qui prend la valeur VRAI si la *sequence* est vide. Tant qu'elle n'est pas vide, je multiplie par 10 la valeur conservée dans la variable donnée en #2 et j'y ajoute la dernière valeur extraite de la pile.

De fait, le chiffre le plus à droite est celui de *poids* le plus grand.

Kaprekar, remplissage

```
24  \cs_new:Nn \KAP_remplir_seq:Nn {
25    \int_while_do:nNnn { \seq_count:N #1 } < { #2 }
26    { \seq_put_left:Nn #1 { 0 } } }
```

fragment5

Cette commande assure que la suite contient le bon nombre de chiffres en ajoutant des zéros si nécessaire. La seule nouveauté est `\seq_count:N` qui donne le nombre de valeurs contenues dans la suite.

Kaprekar, tri

```
28  \cs_new:Nn \KAP_ranger_seq:N {
29    \seq_sort:Nn #1
30    { \int_compare:nNnTF { ##1 } > { ##2 }
31      { \sort_return_swapped: }
32      { \sort_return_same:    } } }
```

fragment6

`\seq_sort:Nn` trie la *sequence* donnée — par son « nom » — en 1er argument à l'aide du code donnée en 2e argument. Ici, j'échange ##1 et ##2 si le premier est strictement supérieur au second, autrement dit, je trie les éléments de la suite du plus petit au plus grand. Les commandes `\sort_return_swapped:` — retourner en échangeant — et `\sort_return_same:` — retourner le même — assurent le tri.

Kaprekar, calcul du terme suivant

```
34  \cs_new:Nn \KAP_suivant:nNN {
35    \KAP_int_to_seq:NN #2 \kaprun_seq
36    \KAP_remplir_seq:Nn \kaprun_seq { #1 }
37    \KAP_ranger_seq:N \kaprun_seq
38
39    \seq_set_eq:NN \kaprdx_seq \kaprun_seq
40    \seq_reverse:N \kaprdx_seq
41
42    \KAP_seq_to_int:NN \kaprdx_seq \l_tmpa_int
43    \KAP_seq_to_int:NN \kaprun_seq \l_tmpb_int
44    \int_set:Nn #3 { \l_tmpb_int - \l_tmpa_int } }
```

fragment7

Le terme courant est donné par une variable comme 2e argument, le 1er est la taille, le 3e doit être la variable qui recevra le terme suivant.

`\kaprun_seq` prend en ligne 35 les chiffres du terme courant puis on remplit puis on range dans l'ordre croissant.

Avec `\seq_set_eq:NN`, en ligne 39, on crée, localement et sans contrôle, `\kaprdx_seq` comme copie de `\kaprun_seq` puis, en ligne suivante, on la range dans l'ordre décroissant.

Les trois dernières lignes traduisent l'algorithme de Kaprekar tel qu'exposé en 15.

Il ne reste plus qu'à utiliser tout cela.

8.3 Commande interne

Kaprekar, commande interne

```
46  \cs_new:Nn \KAP_ecrire_tous_les_suivants:nn {
47    \group_begin:
48    \bool_set_true:N \l_tmpa_bool
49    \seq_clear:N \kaprekar_seq
50    \int_set:Nn \l_tmpc_int { #2 }
51    \seq_put_right:NV \kaprekar_seq \l_tmpc_int
52    \KAP_suivant:nNN {#1} \l_tmpc_int \l_tmpc_int
53    \seq_if_in:NVT \kaprekar_seq \l_tmpc_int {\bool_set_false:N \l_tmpa_bool}
54    \bool_while_do:nn { \l_tmpa_bool }
55    { \seq_put_right:NV \kaprekar_seq \l_tmpc_int
56      \KAP_suivant:nNN {#1} \l_tmpc_int \l_tmpc_int
57      \seq_if_in:NVT \kaprekar_seq \l_tmpc_int { \bool_set_false:N \l_tmpa_bool } }
58    % \seq_show:N \kaprekar_seq
59    % \seq_use:Nnnn \kaprekar_seq {~et~} {,~} {~et~}\par{}
60    \int_set:Nn \l_tmpa_int { \seq_count:N \kaprekar_seq }
61    \seq_map_inline:Nn \kaprekar_seq
62    { \( \np{##1} \)
63      \int_decr:N \l_tmpa_int
64      \int_case:nnF { \l_tmpa_int }
65      {
66        { 1 } {~et~}
67        { 0 } { }
68      }
69      {~;~}
70    }
71    \group_end: }
```

fragment8

La commande `\KAP_ecrire_tous_les_suivants:nn` prend la valeur de N — la *taille* — en premier argument et le premier terme de la suite comme 2e argument.

Une fois de plus, tout se passe dans un groupe, ouvert en ligne 47 et fermé en ligne 71.

J'utilise la variable booléenne locale `\l_tmpa_bool` fournie par Expl3. Je lui donne la valeur VRAI en ligne 48. Quand elle prendra la valeur FAUX il sera temps de sortir de la boucle — lignes 54 à 57 — qui remplit la *sequence* `\kaprekar_seq`.

Avec `\seq_clear:N`, en ligne 49, je vide la suite `\kaprekar_seq`.

En ligne 51 je place le 1er terme de la suite dans la *sequence* `\kaprekar_seq`. Cette fois je remplis par la droite avec `\seq_put_right:NV` — j'utilise la suite comme une *queue*.

C'est parce que j'ai besoin que cette valeur soit placée dans la variable `\l_tmpc_int` que j'utilise `\seq_put_right:NV`. S'il suffisait de placer la valeur directement dans la suite, on pourrait utiliser `\seq_put_right:Nn` avec le code

```
\seq_put_right:Nn \kaprekar_seq {#2}
```

Je place dans `\l_tmpc_int` le terme suivant calculé avec `\KAP_suivant:nNN` en ligne 52 et si ce terme est déjà présent dans la suite — cas ou $u_1 = u_0$ — je donne à `\l_tmpa_bool` la valeur FAUX. C'est `\seq_if_in:NVT` qui permet de vérifier que la valeur #2 est présente dans la suite #1 et de développer, si besoin le code placé dans #3.

À la fin de la ligne 57 la suite a été créée.

J'ai laissé, commentée, la ligne 58 que j'ai utilisée à titre de diagnostic : la commande `\seq_show:N` permet d'écrire le contenu de la *sequence* dans le fichier `.log`.

À la ligne suivante, la commande `\seq_use:Nnnn` permet d'écrire le contenu de la *sequence*, donnée en 1er argument. Les trois arguments suivants donnent, dans l'ordre, le séparateur du premier et deuxième élément quand il n'y a que deux éléments ; ce qui précède chaque élément, sauf le dernier, quand il y plus de deux éléments ; enfin, ce qui sépare l'avant-dernier du dernier élément. Comme je veux que les nombres soient écrits à l'aide de `\np` je ne peux pas me contenter de cette ligne d'où ce qui suit.

Après avoir placé dans un compteur le nombre d'éléments de la suite, j'utilise `\seq_map_inline:Nn` pour appliquer à chaque élément de la suite passée en 1er argument le code contenu dans le 2e argument — ligne 62 à 70.

Dans ce code, j'écris l'élément en en donnant la valeur à `\np` en mode mathématique en ligne puis je décrémente le compteur à l'aide de `\int_decr:N`.

Avec `\int_case:nnF`, je sélectionne le séparateur à placer : si le compteur vaut 1 j'écris « `~et~` », s'il vaut 0, je n'écris rien — ligne 67 — et, dans tous les autres cas — ligne 69 —, j'écris « `~;~` ».

Quatrième partie

Annexes

Références

[1] The LaTeX 3 Project Team. *The LaTeX3 Interfaces*. Anglais. manual. 25 juill. 2019. 299 p.

[2] The LaTeX 3 Project Team. *The LaTeX3 Source*. Anglais. manual. 25 juill. 2019. 1302 p.

[3] The LaTeX 3 Project Team. *The expl3 package and LaTeX3 programming*. Anglais. manual. 25 juill. 2019. 16 p.

[4] The LaTeX 3 Project Team. *The `xparse` package, Document command parser*. Anglais. 28 mai 2019. 15 p.

[5] Christian Tellechea. *Apprendre à programmer en TeX*. 21 sept. 2014. ISBN : 978-2-9548602-0-6.

Index

boucle, 6, 7, 11
    jusqu'à, 7
    pour, 11
    tant que, 6, 7

jusqu'à, 7

pour, 11
présentation du code, 4
saut, 6, 7, 12
si, 6, 7, 12
    si alors, 7, 12
    si alors sinon, 6
    si sinon, 7

tant que, 6, 7

Glossaire

booléen *boolean* 6, 7

entier c.-à-d. entier relatif, *integer* 5, 8–11, 14, 16

flottant *floating point number*, nombre *réel* représenté en *virgule flottante* 11

fonction *function* 5–7, 9

module *module* 4–6

sequence *sequence*, en français « suite » 15–19

signature *signature* 5, 6

variable *variable* 6, 9, 11, 16–18

Commandes, fonctions et variables

`\bool_do_until:Nn` e3 7, 12

`\bool_do_while:Nn` e3 6, 7

`\bool_if:nF` e3 7

`\bool_if:nT` e3 7

`\bool_if:nTF` e3 6, 7

`\bool_set_true:N` e3 6

`\bool_until_do:Nn` e3 7

`\bool_while_do:Nn` e3 7

`\cs_new:cn` e3 5

`\cs_new:Nn` e3 5

`\cs_set:Nn` e3 9, 11

`\cs_undefine:N` e3 9, 11

`\EcrireCribleEratosthene` 13

`\EcrireDiviseurs` PERSO 14

`\__ERA_dv_ia` PERSO 10, 12

`\ERA_marquer_de_a_par:nnn` PERSO 11

\ERA_presenter_nieme:n PERSO 13
__ERA_qt_ia PERSO 10
\eratosthene PERSO 10–12
\ExplSyntaxOff e3 10, 15
\ExplSyntaxOn e3 10, 15
\framebox 14
\gdef TK 5
\group_begin: e3 8, 11
\group_end: e3 8
\IfBooleanF xparse 14
\IfBooleanT xparse 14
\IfBooleanTF xparse 14
\int_add:Nn e3 6
\intarray_gset:Nnn el3 12
\intarray_new:Nn el3 11
\int_case:nnF e3 19
\int_compare:nNnT e3 12
\int_compare:nNnTF e3 9
\int_compare_p:nNn e3 6
\int_decr:N e3 19
\int_div_truncate:nn e3 16
\int_do_until:nNnn 12
\int_gset:Nn e3 9
\int_incr:N e3 9
\int_mod:nn e3 6
\int_new:N e3 11
\int_set:Nn e3 6, 9
\int_step_inline:nn el3 13
\int_step_inline:nnnn el3 11, 12
\int_to_arabic:n e3 5
\KAP_ecrire_tous_les_suivants:nn PERSO 15, 18
\KAP_int_to_seq:NN PERSO 16
\kaprdx_seq PERSO 17
\Kaprekar PERSO 15
\kaprekar_seq PERSO 16, 18
\kaprun_seq PERSO 16, 17
\KAP_suivant:nNN PERSO 18
\l_compteur_int PERSO 9
\l_ERA_diviseur_max_int PERSO 11

\l_tmpa_bool e3 18
\l_tmpa_int e3 6, 11, 12
\l_tmpb_int e3 16
\l_tmpc_int PERSO 18
\newcommand L2e 5
\NewDocumentCommand xparse 8, 9, 14, 15
\nobreakspace e3 9
\np numprint 5, 19
\numexpr eTeX 6
\par 14
\seq_clear:N e3 18
\seq_count:N e3 17
\seq_if_empty_p:N e3 16
\seq_if_in:NVT e3 18
\seq_map_inline:Nn e3 19
\seq_new:N e3 16
\seq_pop_right:NNT e3 16
\seq_put_left:NV e3 16
\seq_put_right:Nn e3 18
\seq_put_right:NV e3 18
\seq_set_eq:NN e3 17
\seq_show:N e3 19
\seq_sort:Nn e3 17
\seq_use:Nnnn e3 19
\sort_return_same: e3 17
\sort_return_swapped: e3 17
\yh_continue_bool PERSO 6
__yh_ecrire_nombre:n PERSO 4
\yh_ecrire_separateur:n PERSO 7–9
\yh_ecrire_si_premier:n PERSO 7, 8
\yh_est_premier_bool PERSO 6
\yh_impair_est_premier:nT PERSO 5

Remarques sur le document et historique des corrections

Remarques

Je tente — comme chacun sait il y a souvent loin de la coupe aux lèvres — de respecter l'orthographe dite *nouvelle*. On ne verra donc plus d'accent circonflexe sur certains i ! Il m'arrive, dans mon ardeur de jeune révolutionnaire, de priver dudit accent certains o ou autres a. On devra à quelque lectrice ou lecteur leur rétablissement.

Je tiens à remercier, d'avance, tout ceux qui prendront la peine de me signaler erreur, omission, impropriété, obscurité et autres maladresses.

Historique des corrections

2. Seule une correction orthographique motive la version 1.2, première version déposée sur le CTAN.
1. La version 1.1 intègre les corrections proposées par François Pantigny — courriel du 21 septembre 2019 — même si je persiste à dépouiller les i de leurs si surannés chapeaux.