Séance n° 12-bis

Rudiments de programmation

Le but de cette séance est de voir quelques rudiments de programmation : arguments optionnels, macros avec version étoilée, comparaison de nombres, calculs sur les nombres, etc.

12.1 Utilisation de % pour indenter le code

TeX considère que les sauts de lignes sont des espaces (avec la différence que deux sauts de lignes consécutifs sont équivalents à un changement de paragraphe). Si on veut indenter une macro pour la rendre plus lisible, il faut donc faire attention aux sauts de lignes qui peuvent introduire des espaces parasites indésirables. Par exemple,

```
\newcommand{\macro}[2]{\begin{tabular}{ll}#1&#2\end{tabular}}
```

peut être récrite sous la forme suivant, qui met beaucoup mieux en avant sa structure :

```
\newcommand{\macro}[2]{
  \begin{tabular}{ll}
  #1 & #2
  \end{tabular}
}
```

Pour éviter tout espace parasite, il vaut mieux prendre l'habitude de mettre des % à la fin de chaque ligne ; par exemple,

```
\newcommand{\macro}[2]{%
  \begin{tabular}{ll}%
  #1 & #2%
  \end{tabular}%
}
```

Ici, certains de ces % sont inutiles, mais il vaut mieux en mettre qui ne servent à rien que d'avoir des surprises plus tard. Un cas où il n'est pas nécessaire de mettre des % est lorsque la ligne se termine par une commande qui n'a pas d'argument, par exemple, `\itshape` car alors cette commande mange l'espace (le saut de ligne) qui la suit. Par exemple, dans

```
\newcommand{\enitalique}[1]{%
  \itshape
  #1%
}
```

il n'y a pas besoin de % après \itshape. Ceci n'est pas spécifique à \itshape, mais commun à toutes les commandes ne prenant pas d'arguments [1].

Exercice 1. — Dans le code suivant, manque-t-il des % ? Y en a-t-il des inutiles ? nuisibles ? (On n'a pas vu certaines des commandes utilisées, mais cela n'a pas d'importance pour juger l'opportunité des % utilisés.)

```
\newcommand{\textebidon}[4]{
  \leavevmode%
  \color{#2}
  #1 est de couleur #2, mais pas
  #3 qui est de couleur%
  \textcolor{#4}%
            {#4}.
  \normalcolor
}
```

12.2 Macros longues et courtes

Par défaut, les commandes définies par \newcommand sont *longues*, c'est-à-dire qu'il est possible d'avoir plusieurs paragraphes dans leurs argument. Ce peut être ou ne pas être une bonne chose selon le cas de figure. D'un côté, cela permet d'utiliser la commande dans plus de contextes, mais d'un autre coté, cela pose peut résulter dans des messages d'erreurs moins facilement déchiffrables (l'erreur, au lieu d'avoir lieu au prochain changement de paragraphe, pourra avoir lieu à la fin du document, donc très loin de là où la commande est positionnée dans le fichier source, ce qui la rend moins facilement détectable). Pour obtenir une commande qui n'est pas longue, il suffit d'utiliser \newcommand* au lieu de \newcommand. Par exemple, la commande \abs qui met son argument entre valeur absolue n'a aucune raison d'être longue, donc il vaut mieux la définir avec \newcommand* :

```
\newcommand*{\abs}[1]{\left\lvert#1\right\rvert}
```

Probablement, toutes les macros personnelles qu'on a vu jusqu'ici devraient être courtes, donc définies avec \newcommand* au lieu de \newcommand.

Exercice 2. — Faire une commande \longtextbf qui met son argument en gras et peut prendre plusieurs paragraphes en argument.

12.3 Gestion des espaces dans les commandes

Par défaut, l'espace après une commande est avalé. Par exemple,

```
\LaTeX et HTML, ce n'est pas la même chose
```

donnera « LaTeXet HTML, ce n'est pas la même chose ». Remarquer que le « et » est collé à LaTeX. Mettre plusieurs espaces après \LateX ne changera rien. On peut soit isoler \LaTeX du reste par des accolades vides,

```
\LaTeX{} et HTML, ce n'est pas la même chose
```

soit rajouter à la main l'espace avec \ (antislash suivi d'un espace)

```
\LaTeX\ et HTML, ce n'est pas la même chose
```

On obtiendra alors « LaTeX et HTML, ce n'est pas la même chose ».

1. Sauf certaines qui n'ont qu'un seul caractère qui n'est pas une lettre, comme \% par exemple, qui ne mange pas l'espace qui le suit.

Pour éviter ces problèmes, il est possible d'utiliser le package xspace :

```
\usepackage{xspace}
```

puis de redéfinir la commande `\LaTeX` pour qu'elle utilise la commande `\xspace` :

```
\renewcommand{\LaTeX}{LaTeX\xspace}
```

Ensuite,

```
\LaTeX et HTML, ce n'est pas la même chose
```

donnera bien « LaTeX et HTML, ce n'est pas la même chose ».

Exercice 3. — En utilisant la commande `\xspace`, faire un raccourci pour « si et seulement si » dans le texte suivant

> Une fonction est positive si et seulement si elle ne prend aucune valeur strictement négative. Mais une fonction n'est pas décroissante si et seulement si elle n'est pas strictement décroissante. Cela fait en tout trois si et seulement si.

On peut aussi, au contraire, vouloir dans certains cas forcer à ignorer les espaces après une commande alors que ce n'est pas normalement le cas. Pour faire cela, on peut utiliser `\ignorespaces` ou `\ignorespacesafterend`. Ces commandes sont surtouts utile au début d'un environnement pour éviter un espace parasite si on saute une ligne après le `\begin{...}` et après le `\end{...}` dans un environnement personnel.

Par exemple, si on définit un environnement par

```
\newenvironment{petit}{\small}{}
```

Utiliser

```
Ceci est du
\begin{petit}
texte en petit
\end{petit}
et ceci du texte normal
```

fournira

Ceci est du texte en petit et ceci du texte normal

Noter les deux espaces parasite avant et après les mots « texte en petit ». Pour éviter cela, on pourait taper

```
Ceci est du
\begin{petit}%
texte en petit
\end{petit}%
et ceci du texte normal
```

mais il est possible de tout automatiser avec les commandes `\ignorespaces` ou `\ignorespacesafterend` :

```
\newenvironment{petit}{\small\ignorespaces}{\ignorespacesafterend}
```

Le code précédent donnera alors

Ceci est du texte en petit et ceci du texte normal

En règle générale, c'est donc une bonne idée de penser à utiliser `\ignorespaces` ou `\ignorespacesafterend` dans un environnement personnel.

Exercice 4. — Y a-t-il besoin de `\ignorespaces` et `\ignorespacesafterend` dans l'environnement personnel suivant. Pourquoi ?

```
\newenvironment{questions}{\begin{enumerate}}{\end{enumerate}}
```

Finalement, abordons comment ôter un espace précédent une commande. Quand on tape

```
Bonjour ! Bonjour! Bonjour~! Bonjour\hspace{1em}!
```

on obtient

Bonjour ! Bonjour ! Bonjour ! Bonjour !

On constante que, dans tous les cas, l'espace avant le signe ! est invariant. On ne va pas aborder comment modifier le caractère ! lui-même, mais juste comment définir une commande `\exclam` qui ferait la même chose. On part du caractère ! tout seul. Pour y accéder sans le comportement intelligent précédent, il suffit d'écrire `\string!`. Alors si on fait

```
\newcommand{\exclam}{\string!\xspace}
```

taper

```
Bonjour \exclam Bonjour\exclam Bonjour~\exclam Bonjour\hspace{1em}\exclam
```

donnera

Bonjour ! Bonjour! Bonjour ! Bonjour !

Cette fois-ci, tous les espaces sont pris en compte. Pour avoir un espacement constant avant `\exclam`, on va utiliser `\unskip` qui supprime l'espacement précédent puis rajouter l'espacement à la main avec un `\hspace`. Par exemple,

```
\newcommand{\exclam}{\unskip\hspace{0.167em}\string!\xspace}
```

permettra d'obtenir

Bonjour ! Bonjour ! Bonjour ! Bonjour !

On obtient bien exactement la même chose que si on avait utilisé !.

Exercice 5. — En utilisant `\unskip`, reproduire les commandes `\og` et `\fg` à l'identique. On peut accéder au caractère "«" avec `\guillemotleft` et à "»" avec `\guillemotright`. On pourra aussi utiliser `\nobreak` juste avant une espace pour la rendre insécable.

Exercice 6. — Faire une commande `\pourcent` qui est toujours précédée d'une espace insécable.

12.4 Argument optionnels

Il est possible d'avoir des arguments optionnels dans les nouvelles commandes et environnements qu'on défini. Pour cela, il faut, après le crochet contenant le nombre d'argument, mettre la valeur par défaut de l'argument optionnel (il ne peut y en avoir qu'un, qui est toujours `#1` ; le premier argument obligatoire sera donc `#2`, le second `#3`, etc.). Par exemple,

```
\newcommand{\strong}[2][red]{\textcolor{#1}{#2}}
```

fait une commande `\strong` qui met par défaut son argument en rouge, sauf si on spécifie une autre couleur entre crochet. Par exemple,

CODE	RÉSULTAT
`\strong{important}`	important
`\strong[blue]{moins important}`	moins important

Ceci fonctionne aussi avec les environnements, mais attention, l'argument ne peut être utilisé dans le code de fin de l'environnement.

Exercice 7. — Faire une commande `\nom` qui prend un argument optionnel (le prénom) et un argument obligatoire (le nom) et qui imprime le prénom suivi du nom en petites capitales. On pourra utiliser la commande suivante qui teste si son argument est vide et exécute son deuxième argument si c'est le cas et son troisième argument dans le cas contraire.

```
\newcommand{\ifempty}[3]{\ifx#1\\\\#2\else#3\fi}
```

On peut vouloir plusieurs arguments optionnels, et pas seulement en première position. Pour cela, on peut utiliser le package `xargs`. Sa documentation, entièrement en français, est disponible à l'adresse suivante :

`http://tug.ctan.org/tex-archive/macros/latex/contrib/xargs/xargs-fr.pdf`

La documentation de ce package est bien faite, donc nous nous contenterons de juste donner un petit exemple des possibilités. Si on veut faire une commande avec deux arguments optionnels (en position 2 et 4) et trois arguments obligatoires (cinq arguments en tout, donc), on utilisera

```
\newcommandx{\phraseidiote}[5][2=chat,4=gouttière]{Le #1 a mangé le #2 #3 dans la
                                                   #4 sur le #5}
```

Ensuite, on peut utiliser la commande ainsi :

CODE	RÉSULTAT
`\phraseidiote{chien}{gris}{toit}`	Le chien a mangé le chat gris dans la gouttière sur le toit
`\phraseidiote{chien}[rat]{gris}{toit}`	Le chien a mangé le rat gris dans la gouttière sur le toit
`\phraseidiote{chien}{gris}[niche]{toit}`	Le chien a mangé le chat gris dans la niche sur le toit
`\phraseidiote{chien}[rat]{gris}[niche]{toit}`	Le chien a mangé le rat gris dans la niche sur le toit

12.5 Comparer des nombres

Pour comparer des nombres, on peut utiliser la commande `\ifnum`. Sa syntaxe est la suivante :

```
\ifnum 2=3
  vrai%
\else
  faux%
\fi
```

Pour accéder à la valeur d'un compteur, on utilise `\value`, comme dans

```
\ifnum \value{page}=1
  première page%
\else
  deuxième page%
\fi
```

Au mieux de tester l'égalité, on peut aussi tester si deux nombres sont strictement inférieurs ou strictement supérieurs :

```
\ifnum \value{page}<2
  vrai%
\else
  faux%
\fi
```

et

```
\ifnum \value{page}>1
  vrai
\else
  faux
\fi
```

Exercice 8. — Redéfinir `\thechapter` pour que `Chapitre \thechapter` affiche « Chapitre premier » pour le premier chapitre et « Chapitre II », « Chapitre III » pour les suivants.

12.6 Commande pour les siècles

Le but de ce § 12.6 est de faire une macro appelée `\siecle` qui prend en argument un chiffre (arabe) non nul et retourne le siècle en chiffres romains petites capitales suivi de « après J.-C. » ou de « avant J.-C. » selon que le nombre soit positif ou non. Voici divers résultats de cette macro :

CODE	RÉSULTAT
`\siecle{-5}`	vᵉ siècle avant J.-C.
`\siecle{-1}`	ıᵉ siècle avant J.-C.
`\siecle{0}`	renvoie une erreur
`\siecle{1}`	ıᵉʳ siècle après J.-C.
`\siecle{2}`	ıɪᵉ siècle après J.-C.

Exercice 9. — Faire une commande `\siecle` qui prend en argument un argument strictement positif et affiche ce nombre en petites capitales avec un ᵉ à côté suivi de « après J.-C. ». On pourra créer un compteur (voir l'aide-mémoire sur le site) et stocker la valeur de l'argument dans ce compteur puis afficher ce compteur nombre romain mis en petites capitales. La commande pour mettre en exposant du texte est `\up`.

Il faut maintenant s'assurer que lorsque l'argument vaut 1, on ait « ıᵉʳ » et non « ıᵉ ». On utilise pour cela la commande `\ifnum` vue dans le § 12.5. De même, pour envoyer une erreur lorsque le siecle vaut 0, il suffit d'utiliser la commande

```
\newcommand{\messageerreur}[2][LaTeX]{\GenericError{}{#1 Error : #2}{}{}}
```

Par exemple, on l'utilise en disant `\messageerreur{le numéro de siècle ne doit pas être nul}`. Finalement, il faut aussi prendre en compte un argument négatif ; pour cela, il suffit de tester si le nombre est négatif et de le rendre positif le cas échéant en rajoutant un signe − devant.

Exercice 10. — Faire la commande `\siecle` telle que décrite au début de ce § 12.6.

Exercice 11. — Que faut-il modifier dans le code précédent pour que les siècles s'affichent en majuscules au lieu de petites capitales ?

12.7 Rudiments de calcul avec TeX

Pour faire des calculs, voir, selon les besoin, les packages `calc`, `fp`, `xlop` voire même `tikz`.

On peut faire des calculs basiques (les quatre opérations) sur des entiers avec un résultat entier (donc attention à la division : 5/3 donnera 1, c'est-à-dire la partie entière de 5/3). La commande s'appelle `\numexpr`, et voici un exemple typique d'utilisation

```
Le produit des entiers de 1 à 5 vaut \number\numexpr 1*2*3*4*5\relax.
```

(Noter l'utilisation de `\number` devant `\numexpr` pour éviter un message d'erreur du genre « You can't use '`\numexpr`' in horizontal mode » et le `\relax` pour marquer la fin du calcul ; à la place de `\number`, on peut aussi utiliser `\the`.) Le résultat de cette ligne de code est

Le produit des entiers de 1 à 5 vaut 120.

On va maintenant voir comment utiliser `\numexpr` pour faire quelques macros simples mais utiles.

Exercice 12. — Sachant que `\year`, `\day` et `\month` (éventuellement précédés de `\number` ou de `\the`) donnent respectivement l'année, le mois et la date du jour, faire une macro `\CalculeAge` qui calcule automatiquement l'age à partir de l'année de naissance (`#1`), le mois de naissance (`#2`) et le jour de naissance (`#3`). Une telle macro est utile par exemple pour mettre son age sur un CV ou calculer le nombre d'années depuis la mort de quelqu'un.

Exercice 13. — Faire une macro `\cloner` qui prend deux arguments : le premier est un symbole (par exemple, *) et le deuxième est le nombre de fois que l'on veut répéter ce symbole. La macro pourra être codée de la façon suivante : si son deuxième argument est < 1, elle ne devra rien faire ; si son deuxième argument est ≥ 1, elle affichera le symbole puis appelera récursivement la macro `\cloner` mais avec un deuxième argument diminué de 1 (ainsi, `\cloner{*}{5}` sera la même chose que `*\cloner{*}{4}`).

Exercice 14 (plus difficile). — Faire une commande `\blabla` qui prend un argument optionnel tel que : `\blabla` est la même chose que `\blabla[25]` et que `\blabla[n]` Affiche n fois le mot bla, avec une majuscule au début, un point à la fin et, si $n < 1$, n'affiche rien. *Indication.* On pourra définir une macro auxilliaire `\auxblabla` qui fait une partie du travail.

Exercice 15. — Faire une macro appelée `\fact` dont le premier argument est un nombre et qui renvoie la factorielle de ce nombre. Rappelons que, récursivement, la factorielle est définie par : $n! = 1$ si $n < 2$ et $n! = n \times (n-1)!$ sinon.

Exercice 16. — En utilisant le fait que, d'après les règles de calculs de TeX, le résultat de n/m multiplié par m ne vaut n que si n est divisible par m, faire une commande `\ifdivisibleby` qui teste la divisibilité par un certain nombre. Les arguments de cette commande devront être les suivants :

```
#1 = nombre dont on veut tester la divisibilité
#2 = nombre par lequel on teste la divisibilité
#3 = code à exécuter si le nombre est #1 est divisible par #2
#4 = code à exécuter si le nombre est #1 n'est pas divisible par #2
```

Par exemple, `\ifdivisibleby{n}{2}{oui}{non}` affichera « oui » si `n` est divisible par `2` et affichera « non » sinon.

12.8 Noms de macros interne

Certaines macros ne sont pas prévues pour être directement utilisées par l'utilisateur final, mais en pratique, on peut en avoir besoin. Ces macros utilisent un `@` dans leur nom, ce qui fait qu'on ne peut pas les utiliser normalement. Par exemple, la commande `\@title` vue la séance n° 6 est une commande interne et donc on a dû utiliser

```
\makeatletter
\fancyhead[LO,RE]{\@title}
\makeatother
```

La macro `\makeatletter` permet de rendre le caractère `@` utilisable dans une macro tandis que `\makeatother` restaure le comportement usuel. Il faut bien faire attention à toujours les utiliser ensemble dès qu'une commande utilise un `@`.

12.9 Faire des macros avec variantes étoilées

La commande interne `\@ifstar` permet de tester si une commande est suivie d'une étoile et appeler d'autres commandes le cas échéant. Par exemple, la commande `\section`, qui a une variante étoilée, est définie [2] de la façon suivante :

2. C'est bien sûr juste le schéma de la définition ; les définitions réelles sont un peu plus complexes que cela.

```
\makeatletter
\newcommand{\section}{\@ifstar\section@star\section@nostar}
\newcommand{\section@star}[1]{...}
\newcommand{\section@nostar}[2][]{...}
\makeatother
```

Le package mathtools propose une commande `\DeclarePairedDelimiter` qui s'utilise comme suit :

```
\DeclarePairedDelimiter{\abs}{\lvert}{\rvert}
```

On peut ensuite utiliser la commande `\abs` de la manière suivante avec le résultat suivant :

CODE	RÉSULTAT
`$\abs{\dfrac{1}{2}}$`	$\lvert\frac{1}{2}\rvert$
`$\abs*{\dfrac{1}{2}}$`	$\left\lvert\frac{1}{2}\right\rvert$
`$\abs[\big]{\dfrac{1}{2}}$`	$\big\lvert\frac{1}{2}\big\rvert$
`$\abs[\Big]{\dfrac{1}{2}}$`	$\Big\lvert\frac{1}{2}\Big\rvert$
`$\abs[\bigg]{\dfrac{1}{2}}$`	$\bigg\lvert\frac{1}{2}\bigg\rvert$
`$\abs[\Bigg]{\dfrac{1}{2}}$`	$\Bigg\lvert\frac{1}{2}\Bigg\rvert$

L'exercice suivant propose de programmer une telle macro à la main.

Exercice 17. — Reproduire la commande `\abs` décrite précédemment. On rappelle que `\big`, `\Big`, `\bigg` et `\Bigg` s'utilisent comme `\left` et `\right`. La variante étoilée de la macro devra utiliser `\left` et `\right`.