Outils pour la manipulation du rapport d'activité

José Grimm

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

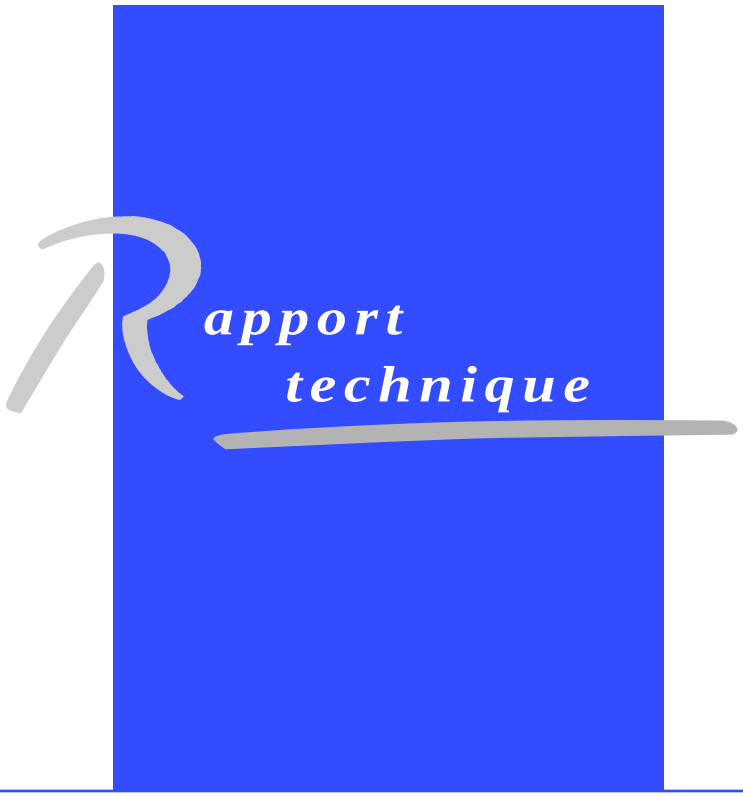
Outils pour la manipulation du rapport d'activité

José Grimm

N° 0265

Juillet 2002

THÈME 4



ISSN 0249-0803

INRIA
SOPHIA ANTIPOLIS

Outils pour la manipulation du rapport d'activité

José Grimm

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet MIAOU

Rapport technique n° 0265 — Juillet 2002 — 131 pages

Résumé : En 1999 de nouveaux outils ont été créés pour la gestion du rapport d'activité de l'Inria, dénommé *raweb*. Ceci se caractérise essentiellement par la notion de « modules » et l'introduction d'un préprocesseur « raweb.pl » qui vérifie la sémantique. Pour le rapport de l'année 2002, le source du rapport peut être du XML.

Nous décrivons dans ce document comment convertir des sources LaTeX vers HTML (génération des anciens rapports, grâce à latex2html) et XML (ce qui se fait via une extension du script raweb.pl, qui devient un traducteur latex vers XML), et nous expliquons comment convertir ce XML en HTML et Pdf. On donnera également la DTD du rapport 2002, ainsi qu'un exemple commenté.

Ce document contient de nombreux exemples de code TeX, qui montrent toute la difficulté qu'il y a à écrire un traducteur général, et les limitations des traducteurs existants, y compris celui décrit dans ce rapport. Il s'agit ainsi d'une base de travail pour un nouveau traducteur, nommé tralics.

Mots-clés : Latex, XML, HTML, MathML, Perl, PostScript, Pdf

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Tools for manipulation of the activity report

Abstract: In 1999, new tools were designed for the *raweb*, the scientific part of Inria's annual report. The LaTeX source is a set of 'modules', and a Perl script, 'raweb.pl' is used to check it. For the year 2002, the input can be XML.

In this document, we explain how to convert the LaTeX source into HTML, using latex2html, and into XML, extenting the Perl script into a latex-to-XML translator. We also explain how to convert this XML into HTML and Pdf. We shall explain the DTD used in 2002, and a complete example.

This document contains a great number of TeX examples that show that it is very hard to write a general translator, and gives the limitations of these translators, including the one described here. It is thus the basis of a new translator, called tralics.

Key-words: Latex, XML, HTML, MathML, Perl, PostScript, Pdf

Chapitre 1

Introduction

1.1 Généralités

L'écriture du rapport d'activité est un exercice imposé chaque année à une centaine de projets de l'Inria. Au début, il s'agissait de produire un document papier, distribué par les centres de documentation. À l'heure actuelle, le résultat est disponible sur le serveur Web de l'Inria, et diffusé avec le rapport annuel sur un CD-ROM, sous la forme d'un ensemble de documents PostScript, Pdf et HTML.

Pour faciliter le traitement, le document source est un texte LaTeX. Il est envisagé pour l'année 2002 d'autoriser un source sous la forme XML. On se propose d'offrir aux rédacteurs une traduction en XML de leur texte de l'année précédente. Un objectif de ce document est de montrer comment faire cette traduction, au moyen d'un certain nombre de traducteurs pilotés par un script Perl. Par ailleurs, on montrera également comment générer du HTML et du Pdf à partir d'un source XML. La version Pdf du rapport 2001, installée sur le serveur Web au mois de mars 2002, utilise les outils décrits dans ce rapport.

Ce document contient, dans le premier chapitre, un certain nombre de constructions TeX un peu compliquées que la plupart des traducteurs automatiques ne savent pas résoudre, et dans les chapitres suivants un certain nombre d'idées d'implémentation. Le dernier chapitre décrit les principes du traducteur utilisé pour le rapport 2001, et est à la base d'un traducteur nettement plus puissant, écrit en C++ et non plus en Perl, qui sera utilisé pour le rapport 2002.

La traduction de LaTeX vers XML doit satisfaire les trois contraintes suivantes
- le résultat est facilement analysable,
- la structure est conservée,
- la traduction est univoque.

La première condition est réalisée automatiquement de par le choix du langage XML, par ailleurs il est possible de mettre de la structure dans un document XML (on peut aussi faire du n'importe quoi, par exemple convertir le LaTeX en dvi, puis de traduire le dvi, ce qui est trivial). La structure que l'on veut conserver est définie par la DTD, dans le cas du rapport d'activité, elle sera donnée dans les chapitres suivants. Évidemment, pour pouvoir conserver la structure, il faut qu'elle apparaisse dans le document source.

On fera dans la suite trois types d'hypothèses sur le document source : des hypothèses techniques (du type, il n'y a pas de `\0` dans le source, ces hypothèses permettent de simplifier un

peu le traducteur), des hypothèses de validité (le document se compile sans erreur et respecte les règles du rapport d'activité telles que décrites sur la page Web), et des hypothèses plus générales : le document respecte les règles non écrites. Nous expliquerons ces règles plus loin, et tenterons de les justifier ; cependant l'argumentaire est essentiellement le suivant.

Supposons que le document source soit A, et ce que voit le rédacteur est B ; de façon précise, disons que B est la version PostScript de A. Notre traducteur fournit une version XML de ce document, à savoir C. On donne un moyen de visualiser ce document, sous forme Pdf, HTML ou PostScript. Notons par D la version PostScript. Dire que la traduction est univoque signifie en gros que deux textes similaires se traduisent en des textes similaires, et deux textes différents en des textes différents. Ce dernier point peut être précisé par l'égalité suivante $B = D$.

Comme dit plus haut, obtenir $B = D$ est trivial : il suffit de traduire en XML la version PostScript du document (cela n'est pas compliqué, car le PostScript généré à partir du dvi est simple), mais ceci ne respecte absolument pas la structure : le document XML sera formé de lignes, et chaque ligne est une suite de caractères, avec des espaces entre les caractères (certains mots peuvent être coupés). Le résultat dépend très fortement de la taille de la page, de la police utilisée, etc.

Notons par p les divers paramètres de taille de page, de police de caractères, d'espacement inter-paragraphe, etc., qui se trouvent dans le fichier de classe du raweb. La traduction XML du document sera, par définition, indépendante de p. La version PostScript en dépend, on aura donc B_p (le source LaTeX traduit en PostScript), et D_p (le résultat XML traduit en PostScript). On va demander l'égalité forte : pour tout p, $B_p = D_p$. Concrètement, cela signifie la chose suivante : soit `\M` la macro qui permet de mettre des mots clés dans le texte, et `<M>` sa traduction en XML. Ces deux commandes seront traduites en PostScript sous la même forme M_p. Il est cependant possible, pour l'utilisateur, de changer la police de caractères utilisée (via un `\def` bien choisi) ; ceci n'est pas possible dans XML.

Si on note par q les paramètres par défaut du rapport d'activité, ce que voit le rédacteur, c'est B_q, et en général, il se débrouille pour que B_q soit beau. Il n'a aucune idée de B_p pour un p différent de q. Prenons un exemple : la document contient une figure avec deux images côte à côte, et une légende qui fait référence à l'image de gauche et l'image de droite. Supposons que les deux images soient de largeur 7 cm. Si on change la largeur de la page, pour avoir une largeur de texte de 13 cm, il n'y a pas de place pour les deux images, et TeX va les placer verticalement (dans certains cas, il n'y a pas assez de place sur la page pour ceci, et la légende disparait), et la légende sera fausse. La bonne façon de faire est de dire que la largeur des images est de 45% de la largeur du texte (ce qui laisse un peu de place entre les images).

On dira dans la suite qu'un document est admissible s'il se compile sans erreur, s'il respecte les règles écrites du rapport d'activité, et également les règles non écrites. En particulier, le document ne doit pas modifier les paramètres p (taille de page, police de caractères, etc.), et ne doit pas en faire un usage explicite (cette notion n'est pas facile à formaliser, voir exemple plus haut). Pour simplifier le traducteur, on demande également d'utiliser une commande de haut niveau plutôt que plusieurs commandes de bas niveau (même si le résultat est exactement le même). On demandera également de respecter l'esprit LaTeX : la commande `\subitem` est censée faire un sous-item dans un index, il ne faut donc pas l'utiliser dans le corps du texte.

On peut donner une autre définition d'un document admissible : c'est un document qui se compile, qui respecte les règles, et qui ne contient que des commandes admissibles, i.e. des commandes reconnues comme telles par le traducteur. En d'autres termes, est admissible un

document qualifié ainsi par l'auteur du traducteur. C'est pour cette raison que nous devons analyser chaque construction douteuse, et expliquer pourquoi nous ne la qualifions pas d'admissible. Mais avant cela, on commencera par expliquer les divers langages utilisés : TEX, Perl et XML.

1.2 TEX et LATEX

1.2.1 Vocabulaire

Soit le fragment de code LATEX suivant.

```
$\left[1=2\right\}$ est une \emph{formule de mathématiques}~!
```

En règle générale, cette formule se traduit par '$[1 = 2\}$ est une *formule de mathématiques* !'. Les quantités du type `\left`, `\right` et `\emph` sont appelées des commandes. Les deux premières sont des primitives TEX, la dernière est une commande utilisateur (en fait, sa sémantique est spécifiée par LATEX). Certaines primitives TEX ont une syntaxe particulière, les commandes LATEX ont en général une syntaxe plus régulière. Le texte entre accolades suivant le `\emph` est appelé l'argument de la commande. Les accolades sont des délimiteurs (de groupe ou d'arguments). Le caractère dollar est spécial dans la mesure où il sert à la fois de délimiteur ouvrant et fermant. La commande `\}` est appelée commande mono-caractère, alors que `~` est un caractère actif (c'est une commande, non précédée d'un backslash).

Le but d'un processeur TEX est de lire et d'interpréter ces commandes, de même que le texte compris entre ces commandes, et de générer un fichier dvi, lequel peut être visualisé et imprimé (directement ou après conversion en PostScript). Il est possible de mettre des marqueurs spéciaux dans le dvi ; ces marqueurs peuvent être des indicateurs de couleur, ou des liens hypertextes (on peut ainsi mettre des marqueurs hypertextes qui deviennent actifs, une fois que le fichier dvi est traduit en PostScript puis en Pdf). On peut insérer un marqueur d'inclusion d'images (si l'image est du PostScript, la traduction en PostScript ne pose pas de problème). Certains logiciels, comme `tex4ht` insèrent des marqueurs spéciaux pour faciliter la traduction en HTML. Le logiciel Ω, qui est une extension de TEX, permet de traduire des formules de mathématique en MathML, il les met dans un fichier spécial. Une autre extension de TEX, pdfTEX, produit du Pdf au lieu de produire du dvi. Il n'y a pas de marqueurs spéciaux, on peut cependant mettre des liens hypertextes, des bookmarks, des mots clés, et des images, à condition qu'elles soient dans le bon format (par exemple Pdf).

Une commande qui n'est pas une primitive sera souvent appelée une macro. Il est possible de rajouter des commandes à TEX. Pour faciliter la gestion, toutes ces commandes sont regroupées en quatre endroits différents :

- dans un fichier de macros prédigérées. Un tel fichier est appelé un format. Le format standard le plus courant est appelé LATEX. On pourrait citer plaintex (le format conçu par Knuth et expliqué dans le TEXbook), amstex (inclus maintenant dans LATEX), context (une variante de LATEX), ainsi que d'autres formats (par exemple musictex pour faire de la musique).
- dans un fichier de classe. Un tel fichier contient toutes les commandes spécifiques à un type de document (livre, article, transparents, rapport d'activité Inria, etc.).
- dans un package. Un tel fichier contient un ensemble d'utilitaires, par exemple pour insérer des images, pour faire de belles mathématiques (amstex est devenu un ensemble de packages), pour écrire en plusieurs colonnes, etc.

– dans le fichier source.

Ainsi, ce document est écrit en LaTeX, il utilise la classe report, charge les packages `RR` (pour la mise en page, et le logo Inria), le package `babel` (pour avoir une typographie à la française), le package `fancyvrb` (pour les exemples de code).

Il est possible redéfinir le comportement des primitives TeX. Par exemple, si l'exemple donné au début de la section est précédé de

```
\catcode`\$=\active\def$#1~{\catcode`\$=3 zut} %$ emacs
```

alors la traduction de la formule est 'zut!'.

Cet exemple est intéressant à plus d'un titre. Il montre que la traduction automatique d'un texte LaTeX est extrêmement compliquée.

On notera tout d'abord que la formule contient quatre signes dollar, dont la moitié est précédée d'un backslash, donc n'est pas un délimiteur mathématique. Les deux autres viennent en paire, c'est du moins ce que croit l'éditeur de texte Emacs. C'est uniquement pour cet éditeur que le caractère dollar été inséré dans un commentaire. Si l'on veut un écrire un traducteur, il faut faire attention à ne pas se laisser abuser par des constructions de ce genre.

La commande `\catcode` est une primitive TeX, elle prend en argument le code interne d'un caractère (ici c'est le code du caractère dollar, donné sous la forme symbolique `` `\$ ``), et modifie le comportement de l'analyseur lexical. D'abord on dit à TeX que le dollar est un caractère actif, et ensuite que c'est un délimiteur mathématique.

La commande `\def` est une primitive TeX qui permet de définir une nouvelle commande (ou de la redéfinir, ou de définir un caractère actif). Ici, on définit le dollar, comme macro avec un argument délimité par un tilde; l'expansion de la macro ignore l'argument, et rend le comportement normal au dollar.

1.2.2 Quelques considérations sur la traduction

Le document à traduire utilise des primitives TeX, et des commandes, qui se trouvent dans quatre endroits différents. Il est clair qu'il faut savoir expanser les macros utilisateur (au moins les plus simples, celles qui ne prennent pas d'argument, on peut être ambitieux et vouloir tout expanser). Si on expanse tout, on en est réduit à traiter uniquement les primitives. On ne peut cependant pas expanser les macros du fichier de classe (si la macro `\motscle` s'expanse en `\textbf{mots clés}~`, il n'y a plus moyen de reconvertir ceci en `<motscle/>`. Ainsi, certaines macros du fichier de classe devront être considérées comme des primitives, et d'autres comme des commandes interdites. Par exemple, `\ra@finpart` est dans le fichier de classe, mais personne n'est censé l'utiliser (cette commande est utilisée par `\end{participants}` pour, entre autres, mettre un point final derrière la liste des participants). Dans la section 1.2.11 on donnera l'exemple de la commande `\verb`, elle utilise une commande auxiliaire `\@sverb`. Cette commande fait également partie de la liste des commandes interdites.

On pourra donc faire l'hypothèse suivante : il y a des primitives (celles de TeX), et des commandes considérées comme primitives (celles de LaTeX, du fichier de classe), et des commandes qui existent a priori, mais que le traducteur ne prendra pas en compte. Comme il y a un bon millier de fichiers de packages possibles, on ne prendra en compte que ceux qui servent le plus souvent, et ces fichiers seront considérés comme faisant partie du noyau LaTeX, et tant pis pour les autres. Finalement, il reste les commandes utilisateur, qu'il faudra expanser.

On ne va pas implémenter toutes les primitives TEX. Ainsi, on ne va pas implémenter la commande `\'`, car cela est excessivement compliqué, on va juste supposer que `\'e` est la même chose que `é`, et que ceci est le caractère Unicode 00E9, à savoir : « latin small letter e with acute ». De même `\k a` sera supposé donner `ą`, le caractère 0105, à savoir « latin small letter a with ogonek » ; plutôt qu'un numéro Unicode, on traduira ceci sous forme d'entité, à savoir `&aogon;`. Le cas de `\'\^e` ne sera pas traité : le caractère Unicode 1EBF, latin small letter e with circumflex and acute, n'existe pas en français, le jour où l'auteur vietnamien Hàn Thế Thành sera cité dans le rapport d'activité, on avisera (cette personne est l'auteur du logiciel pdfTEX, utilisé pour traduire le XML en Pdf).

On interdira aussi la commande `\font\myfont=toto`, et son utilisation `{\myfont \char 217}`. En effet, cela veut dire : utiliser le caractère 217 dans la fonte. On n'a pas envie de traduire la fonte en une fonte utilisable par le logiciel qui va exploiter le document XML (pour HTML, il faudrait faire une image), ni de décrypter les fichiers TFM pour en déduire les kerning et les ligatures.

La traduction des formules de mathématiques pose certains problèmes qui seront simplement évoqués ici. En effet, la gestion des mathématiques est l'un des points forts de TEX, et amstex augmente considérablement la puissance de TEX. Toutes les constructions possibles dans TEX ne le sont pas dans MathML. Notons cependant que presque tous les symboles mathématiques possibles et imaginables sont ou seront dans Unicode. L'un des problèmes concerne la gestion des formules trop grosses. Il est possible dans amstex de couper une formule en deux, la première partie étant alignée à gauche, et le reste à droite. Il y a un moyen de regrouper un ensemble d'équations, via une accolade, chaque équation ayant son propre numéro (on peut demander à l'ensemble d'être numéroté 17, et les sous-équations seront 17a, 17b, etc.). Ce mécanisme est trop subtil pour nous.

L'espacement horizontal dans les formules est géré de façon intelligente par TEX. Dans le cas de $a + b$ ou $a = b$, il y a un certain espace de chaque coté de l'opérateur, espace qui disparait si la formule passe en exposant. Exemple

$$\frac{a+b}{a=b}, \qquad x^{a+b}_{a=b}.$$

Chaque opérateur est typé, et la gestion des espaces dépend du type. Il est de tradition d'omettre des parenthèses autour de l'argument du sinus dans les cas simples, ceci est géré proprement par TEX, comme le montre l'exemple suivant :

$$\sin^2 x + \sin(x)^2,$$

Toute expression peut être considérée comme opérateur, à condition de mettre la bonne commande (`\mathbin`, `\mathopen`, etc.). Pour les formules en ligne, une coupure de ligne ne peut se faire que sur un opérateur de type plus ou égal, à condition que ce soit l'opérateur principal. Ainsi l'expression $f(x^{a+b})$ ne sera jamais coupée ; il est à noter que Netscape n'a aucun scrupule à couper l'équivalent HTML derrière le signe plus.

Comme cela est visible plus haut, les exposants utilisent une police de caractères plus petite, et les exposants dans les exposants sont encore plus petits. Il y a donc trois tailles de police qui entrent en jeu, et quatre modes (dans la mesure où le placement des indices dans une somme dépend de savoir si la formule est dans le texte, ou hors texte). Pour compliquer le tout, un

exposant sous un trait de fraction est placé légèrement plus bas. Comparez le placement de l'exposant dans les divers cas de la formule suivante :

$$\frac{a_2+x^2}{a_2+x^2}=\frac{x_3^2}{x_3^2}.$$

Il n'est pas rare de voir du `\displaystyle` dans une fraction en ligne (pour augmenter la lisibilité de la formule), ou l'équivalent proposé par amstex, à savoir `\dfrac`.

Dans la majorité des cas, l'opérateur est placé avant son argument. Par exemple, pour mettre un point sur une lettre, on dit `\dot x` ; pour mettre un prime après une lettre on dit `x'`. Ceci est la même chose que `x^{\prime}`. On dira `x_2^3` pour mettre un exposant et un indice sur la lettre. Dans les deux cas, la commande est placée après le noyau. On peut toujours accrocher un exposant à un noyau vide, ce qui permet d'obtenir ${}^2x$, mais cela pose de gros problèmes si on veut aligner verticalement les exposants. Il est possible d'encadrer une formule par des parenthèses ayant la bonne taille, par exemple

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

en utilisant '`\left(`' et '`\right)`'. Notons que cela définit implicitement un groupe (voir plus loin). Si on connait la taille, on peut utiliser `\big`, `\Big` ou autres variantes.

La commande `\over`, ainsi que ses variantes, est la commande ayant la syntaxe la plus tordue de TEX : elle permet de faire une fraction, le numérateur est à gauche, le dénominateur est à droite. C'est le groupe courant qui définit le numérateur et le dénominateur. Par exemple `x=\left( a+b \over c+d\right)^2+1` donne

$$x=\left(\frac{a+b}{c+d}\right)^2+1$$

L'ennui avec cette façon de faire est la suivante : supposons qu'au numérateur on ait une macro `\foo`, dont l'expansion dépend de la taille de la police (par exemple, `\foo` donne a/b dans le cas d'une petite police et $\frac{a}{b}$ sinon). Tant que TEX n'a pas vu le `\over`, il ne sait pas s'il s'agit d'un numérateur, et `\foo` ne peut connaître son mode. Il y a une manière de s'en sortir : utiliser `\mathchoice`, et expanser `\foo` dans les quatre modes possibles. Toute formule de mathématique est analysée deux fois : dans une première passe, les macros sont expansées, les types des opérateurs sont déterminés, etc. Dans la seconde passe, TEX regarde les `\mathchoice`, et choisit la bonne valeur. Il sait dans quel mode on est, et il peut donc calculer les espacements horizontaux et verticaux dans les formules. Tout ceci est très lourd. On supposera que les rédacteurs n'utilisent pas `\over` mais l'équivalent LATEX `\frac`.

1.2.3 Options de francisation

Les principaux formats TEX (plain, LATEX, amstex) ont été conçus par des américains ; à l'heure actuelle la majorité des développements autour de TEX est faite par des européens. Ceci a comme conséquence qu'un grand nombre d'utilisateurs éprouvent le besoin d'utiliser des

primitives de base pour écrire en bon français, alors que tout est fait, en principe, pour simplifier la vie.

Dans la version initiale de TEX, il fallait entrer les caractères accentués sous la forme `\'e`, et dire `\c{c}` pour avoir ç. Pour des raisons d'homogénéité, Lamport conseille `\'{e}` (est-ce bien raisonnable ?). Certains auteurs imaginent que la construction é n'est pas portable.

Tous les caractères de la norme iso-8859-1 sont reconnus en standard par LATEX, ce qui ne pose plus que le problème du `\oe` (qui existe aussi en HTML), et, pour l'année 2002, celui du symbole de l'euro. Pour simplifier la gestion des tableaux, il est conseillé d'écrire `\a'{e}` (*no comment*).

Pour mettre en emphase des mots, les américains disent "ceci" ; pour imiter des logiciels de mauvaise qualité, on pourra dire "ceci". En français, on dirait plutôt « cela ». L'éternelle question est : comment entrer ces caractères qui n'existent pas sur mon clavier made in Mexico ? Une solution consiste à entrer deux caractères inférieurs consécutifs. Avec de la chance, on obtiendra «, sinon `<<`, ou alors ¡¡, suivi de ¿¿. Une autre solution consiste à dire `\guillemotleft`. Si on charge les packages de francisation ou d'internationalisation adéquats, on pourra utiliser `\og` ou `\fg`. Notons la différence : `\og x\fg. \guillemotleft y\guillemotright` donne « x ». «y». En d'autres termes, `\og` et `\fg` gèrent correctement l'espacement de part et d'autres des guillemets (ceci dépend des options utilisées). Rappelons que les espaces introduits devant ou derrière certains signes de ponctuation sont des espaces insécables (il ne peut y avoir de coupure de ligne à cet endroit). Remarque : si `<<` donne des guillemets, c'est peut-être parce qu'il s'agit d'une ligature (comme ffi), donc une propriété qui dépend de la fonte, et non une commande TEX.

Que se passe-t-il si on dit `Ah! oh? bof;` ? Essayons : 'Ah ! oh ? bof ;' En fait, suivant les options, un espace est ou non rajouté automatiquement, dans la mesure où les caractères de ponctuation sont actifs (ceci peut poser problèmes pour des packages qui font l'hypothèse contraire). Cependant, certains auteurs préfèrent mettre des espaces explicites, comme dans `Ah\,! oh\,? bof\,;`, ce qui donne 'Ah ! oh ? bof ;'. Comme il n'y a pas d'espace fine dans HTML, ces `\,` vont disparaitre lors de la traduction, ou être remplacés par des espaces insécables.

1.2.4 Définition de commandes

Une définition de commande est typiquement de la forme

```
\def\toto{toto}
\def\titi#1#2{Arguments #2 et #1}
\def\tata#1.#2=#3#{Vu #1 #2 #3}
```

Juste après le `\def`, il y a un nom de commande, la liste des arguments, et le corps de la macro entre accolades. Dans le cas de `\toto` il n'y pas d'argument, et le corps est toto. Dans le cas de `\titi`, il y a deux arguments, nommés `#1` et `#2`. Lors de l'expansion, `#1` fera référence au premier argument et `#2` au second.

Si l'on dit `\titi12`, les arguments de `\titi` seront 1 et 2. Notons qu'un espace est ignoré après le nom de la commande, donc, dans le cas de `\titi AB`, les arguments sont A et B. On peut également dire `\titi A B`, ou `\titi {AB}{CD}`. Dans ce dernier cas, les arguments sont AB et CD ; notons qu'une paire d'accolades disparaît, et donc, dans le cas de `\titi {{AB}} {{CD}}`, les arguments sont `{AB}` et `{CD}`.

Dans cet exemple, les accolades supplémentaires sont des délimiteurs de groupe, elles ne servent à rien. Nous allons voir d'autres cas où elles peuvent servir. En règle générale, mettre des accolades superflues ne change rien ; mais il y a un certain nombre d'exceptions. Par exemple, dans le cas de `\def`, l'argument est le token qui suit (les espaces sont ignorés), on n'a pas le droit de mettre des accolades. Nous verrons plus loin le cas de `\hskip` : il ne faut pas d'accolades. Certaines primitives TEX ont une syntaxe spéciale : dans le cas de `$a \over b+c$`, on construit une fraction dont le numérateur est a et le dénominateur est $b+c$. Il est recommandé d'utiliser la syntaxe `\frac{a}{b+c}` qui met en évidence numérateur et dénominateur. Dans le cas de `\catcode‘\$`, l'argument est le `‘\$`, (le backquote à lui tout seul est un argument incomplet, un exemple d'utilisation de ce mécanisme est donné en section 1.2.11). Une construction du genre `$a^2$` accroche un exposant à un noyau, dans le cas de `$^e$`, le noyau est vide (voir exemples section 1.2.10). Une formule du type `$a^b^c$` est illégale car ambigüe, il faut des accolades. Une formule de la forme `$a'^2$` est spéciale, car équivalente à `$a^{\prime2}$`. Le placement des accents est spécial aussi : pour avoir des double accents, genre $\hat{a}$, il faut utiliser le package amsmath et taper `$\hat{\hat{a}}$` (les accolades sont obligatoires). Pour mettre un accent sur une lettre, en utilisant la primitive TEX, il faut dire quelque chose comme `\accent23a`, et surtout ne pas mettre d'accolades autour de 23.

Dans l'exemple précédent, nous avons vu comment définir une macro sans argument, et une macro avec des arguments normaux. Le troisième exemple est plus compliqué :

```
\def\tata#1.#2=#3#{Vu #1 #2 #3}
```

Entre le nom de la macro et l'accolade qui commence le corps, il y a `#1.#2=#3#`. Si l'on dit `\tata  ok. {3} = xyz {2+1}`, les arguments sont `ok`, ␣`{3}`␣ et ␣`xyz`␣. En effet, le point après le `#1` signifie que le premier argument est la chaine de caractères (de longueur minimale, équilibrée du point de vue des accolades) qui précède un point, le deuxième est la suite qui précède le signe égal. Le dernier argument est tout ce qui précède l'accolade ouvrante (c'est le # final qui dit ceci). Mettre un # devant une accolade ouvrante est très peu fréquent, dans le source LATEX cela n'arrive que quatre fois, un exemple est donné dans la section 1.2.10, ligne 11. Notons que les espaces entre les arguments sont conservés, par contre une paire d'accolades délimitant un argument disparait (s'il n'y avait pas les espaces autour du `{3}`, la valeur de l'argument aurait été 3).

Il est des fois important de savoir quelles sont les accolades qui sont conservées, et celles qui disparaissent. Avec une définition du genre `\def\toto#1{\titi #1}` et `\def\titi#1{{\it #1}}`, dans le cas de `\toto{12}`, seul le 1 est en italique, dans le cas de `\toto{{12}}`, le nombre 12 est en italique. Si on dit `\def\toto#1{\textit{#1}}` et `\def\titi{12}`, après `\toto\titi`, on verra 12 en italique. Dans la première version, notre macro-expanseur expansait les macros dans un ordre aléatoire, et, suivant l'ordre, le 2 sera ou non en italique.

Il est possible de définir des macros dans des macros. Par exemple

```
\def\toto#1{\def\titi##1{#1##1###1####1}}
```

Dans ce cas, l'expansion de `\toto {17}` sera `\def\titi#1{17#1#17##1}`, de sorte que ceci définit une macro à un argument, et l'expansion de `\titi X` sera `17XX7#1` (ce qui donnera une erreur, à cause du # qui reste). Notons que si 17 avait été remplacé par 25, on aurait vu `\def\titi#1{25#1#25##1}`, ce qui provoque une erreur différente : le `#2` est illégal, car `\titi` ne prend qu'un seul argument. Comme on peut le voir, ce genre de constructions est réservé aux experts.

Il existe d'autres commandes qui sont utiles pour résoudre des problèmes non triviaux, à savoir `\expandafter`, `\noexpand`, `\aftergroup`, `\afterassignment`, `\futurelet`, `\csname`, `\xdef`, etc. Toutes ces commandes peuvent poser problème à un traducteur automatique naïf (voir l'exemple paragraphe 1.2.11). Notons également que `\bgroup` et `\egroup` sont des alias pour une accolade ouvrante et fermante. Ainsi, avec la définition `\def\toto#1#2{#1\bgroup \it #2}`, un texte du genre `\toto {a}{b}c` sera valide, b et c seront en italique. Cette utilisation d'alias perturbe également les traducteurs.

1.2.5 Définition de commandes dans LaTeX

La syntaxe recommandée est

```
\newcommand{\toto}{toto}
\newcommand{\titi}[2]{Arguments #2 et #1}
```

Ces deux définitions sont identiques aux définitions TeX données plus haut. Il n'y a pas l'équivalent de `\tata` (qui est considéré comme étant trop dangereux). Par contre on peut dire

```
\newcommand{\tutu}[3][tutu]{ arguments #1 #2 #3}
```

Dans ce cas, `\tutu` est une macro à trois arguments, le premier est délimité par des crochets. Cet argument est optionnel, s'il n'est pas fourni, la valeur par défaut sera tutu. Remarquons que dans le cas de `\tutu[1[2]3]`, le premier argument sera `1[2`, le second sera 3 et le dernier sera le crochet fermant. En général on s'attend plutôt à voir `\tutu[{1[2]3}]xy`. Dans ce cas, l'argument optionnel est '1[2]3', sans les accolades.

Rappelons que dans TeX, les accolades ont deux fonctions : délimiteur d'arguments, et groupes. Dans le cas de `{\it x}`, la commande `\it` demande le passage en italiques. Lors de la fin du groupe, la police utilisée à l'extérieur redevient la police courante (plus généralement, toute définition, affectation, qui n'est pas globale, n'a d'effet qu'à l'intérieur du groupe). Dans LaTeX, il y a moyen de définir des groupes nommés, appelés environnements, de la forme `\begin{x}`, `\end{x}`. Quand LaTeX voit `\begin{x}`, il ouvre un groupe TeX, et définit deux variables : la ligne courante, et le nom. Quand il voit `\end{x}`, il est capable de comparer, et de dire que le groupe commencé à la ligne tant s'est terminé par autre chose. On peut définir un environnement via

```
\newenvironment{toto}[2]{#1\begin{y}#2 !!} {\end{y} !!}
```

Alors `\begin{x}a b c \end{x}` est identique à `a\begin{y}b !! c \end{y} !!` Notons que la substitution des paramètres ne se fait que dans la partie « begin », jamais dans la partie « end ». Il se passe que `\begin{y} ... \end{y}` est équivalent à `{\y ... \endy}` (modulo les tests), où `\endy` est une expression générée via `\csname`. Nous n'expliquons pas ici ce que fait cette macro, sauf que, si `\endy` est une commande inconnue, elle est ignorée. Utiliser cette feature n'est pas recommandé, mais il n'est pas rare de voir du code ressemblant à `\begin{it} blabla\end{it}`, qui met le blabla en italiques.

Dans le cas d'une définition dans TeX on peut rajouter trois préfixes devant un `\def` : `\long`, `\outer` ou `\global`. Les deux premiers servent à la gestion des erreurs, on ne s'en occupera pas. Le dernier dit que la définition est globale. Notons que `\def\toto{}\def\toto{}` est légal. En LaTeX, un new-machin ne peut redéfinir une commande, il faut utiliser pour cela renew-machin. En cas de doute `\ProvideCommand` définit une commande, sauf si elle existe déjà. Pour le raweb, on suppose qu'aucune commande n'est redéfinie, ainsi seule `\newcommand` est autorisée.

1.2.6 Quelques petits exemples

Rappelons que `\foo` et `\;` sont deux commandes, la différence est que dans le premier cas, la commande commence par une lettre, son nom contient toutes les lettres qui suivent, et un espace qui suit est ignoré ; dans le second cas, la commande est formée d'une seule lettre, et les espaces qui suivent ne sont pas ignorés. C'est le `\catcode` d'un caractère qui dit si c'est une lettre. Les deux commandes `\makeatletter` et `\makeatother` changent le `\catcode` du caractère @. Ainsi

```
\makeatletter
  \def\toto@val{}
  \def\toto#1{\def\toto@val{#1}}
  \def\usetoto{\toto@val}
\makeatother
```

utilise une variable `\toto@val`, qui en principe est interdite. Une des hypothèses faites par LaTeX est qu'aucune commande commençant par `c@` n'existe.

Comme dit plus haut, `\catcode‘\$=3` change le `\catcode` du signe dollar. Ce qui suit le `\catcode` doit être le code d'un caractère, suivi d'un signe égal optionnel (avec des espaces optionnels de chaque côté). Si on dit `\def\A{25}`, il faut un signe égal dans `\catcode\A7`, car ceci construit le nombre 257, qui est invalide en tant que référence à un caractère. Pour pallier à ce genre de problème, TeX propose `\chardef\A25`. Dans ce cas, `\A` sera une référence au caractère 25, `\catcode \A7` marche (et est plus efficace, car le nombre 25 n'est lu qu'une seule fois, lors de la définition de `\A`, et non à chaque utilisation). De la même manière `\coundef\A17` fera que `\A` est équivalent à `\count17`, une référence au compteur 17.

Il est possible de définir des macros via `\let`. Cette commande va lire deux tokens, A et B (entre ces deux tokens, il peut y avoir des espaces optionnels, un signe égal optionnel, et s'il y a un égal, un espace optionnel). Le token A doit être définissable (commande ou caractère actif). Ce que fait `\let`, c'est mettre la valeur courante de B dans A. Si B est un caractère, il s'agit de la valeur du caractère (y compris son `\catcode`), si B est une commande, il s'agit de sa signification (si c'est une primitive), ou le code (si c'est une commande utilisateur). On supposera dans la suite avoir dit

```
\let\bgroup={  \let\egroup=} \let\sp=^ \let\sb=_
```

Ainsi `$x\sp\bgroup a+b\egroup$` est identique à `$x^{a+b}$`. On rappellera qu'une liste de tokens (un corps de macro par exemple) contient autant d'accolades ouvrantes que d'accolades fermantes. Si on dit

```
\def\foo{{\catcode‘}=0\egroup}
```

il y a bien le nombre d'accolades qu'il faut. Lorsqu'on évalue `\foo`, la première accolade fermante sera utilisée pour l'argument de `\catcode`, le `\egroup` terminera le groupe (ce que fait `\foo`, c'est changer localement le `\catcode` de l'accolade, donc, pas grand chose). Une des utilisations typiques de `\let` est la suivante :

```
\def\totoA{macro très longue}
\def\totoB{autre macro très longue}
\def\titi#1{\ifx 0#1\let\toto\totoA \else \let\toto\totoB\fi}
```

Ici `\titi` est une commande qui change la valeur de `\toto`, en fonction de la valeur de son argument. Utiliser `\let` plutôt que `\def` est intéressant, car on copie juste un pointeur, plutôt que de parser une définition. Notons également la distinction entre

```
\def\titi#1{\ifx 0#1\let\toto\totoA \else \let\toto\totoB\fi\toto}
\def\titi#1{\ifx 0#1\totoA \else \totoB\fi}
```

Dans le premier cas de figure, on calcule dans le test le nom de la commande à exécuter, et on l'exécute ensuite, dans le second cas, on l'exécute dans le test. Supposons que le test soit vrai, donc que `\totoA` soit évalué. Dans le premier cas le `\fi` est déjà traité, dans le second cas il ne l'est pas, il occupe donc de la place en mémoire (de plus, cela occupe de la place en pile). Quand nous verrons les boucles, ceci prendra une importance considérable.

Une autre utilisation de `\let` est la suivante : supposons que `\B` soit une macro qui s'expanse en `to`. Si on dit `\let\A\B` puis `\def\B{\A\A}`, alors `\B` s'expanse en `\A\A`, donc en `toto`. Si on avait utilisé `\def` à la place de `\let`, on aurait eu une définition récursive, donc une boucle sans fin. Remarque : supposons que `\B` soit une certaine macro, disons `\let\A\B` et `\def\a{\B}`. Si on expanse une seule fois `\A`, c'est comme si on expansait une seule fois `\B`. Si on expanse une seule fois `\a`, le résultat est `\B`, donc si on expansait deux fois `\a`, c'est comme si on expansait une seule fois `\B`. Si on expanse autant de fois que possible, il n'y a pas de différence.

On peut dire `\futurelet\A\B\C`. C'est la même chose que `\let\A= \C\B\C`. On peut mettre un caractère à la place d'une commande dans le cas de `\B` et `\C`. On verra plus loin à quoi cela peut bien servir. On peut dire `\expandafter \A\B`. Dans ce cas, TEX va lire un token `\A`, le mettre de côté, puis expanser le token qui suit (à savoir `\B`). Le résultat de l'expansion sera remis dans la liste des tokens à relire, et `\A` sera remis devant. Au lieu de `\A` et `\B`, on peut mettre n'importe quoi, cependant, si `\B` n'est pas une commande, ou n'est pas une commande qui puisse être expansée, le résultat sera `\A\B`, donc sans intérêt. Considérons l'exemple suivant :

```
\def\toto{\titi}\def\titi{\tata}\def\tata{\tutu}
\expandafter\expandafter\expandafter\def\toto{5}
\let\E\expandafter \E\E\E\def\toto{6}
\def\E{\expandafter} \E\E\E\def\toto{7}
\expandfter\def\toto{8}
```

À la première ligne, on définit trois commandes, `\toto`, `\titi` et `\tata`. Ce que fait la dernière ligne, c'est mettre de côté le `\def`, expanser `\toto`, ce qui donne `\titi`, et remettre `\def\titi` dans le flux. Après cela, `\titi` vaudra 8. La seconde ligne est plus compliquée : il y a trois `\expandafter` de suite. On va donc mettre de côté le second `\expandafter`, et le `\def`, puis expanser `\toto`, ce qui donne `\titi`. On va se retrouver avec `\expandafter`, `\def`, `\titi`. Finalement ceci définit `\tata` comme valant 5. La troisième ligne est plus subtile. Quand TEX voit le `\E`, donc un `\expandafter`, il l'expanse. Il va mettre de côté le `\E`, puis expanser le `\E`, donc le `\expandafter`. Ceci va donner `\E\def\titi`. Finalement, ceci définit `\tata` comme valant 6. Si on considère enfin la quatrième ligne, l'expansion du troisième `\E` est `\expandafter`. On obtient donc `\E\expandafter\def\toto{7}`. Le `\E` dans cette liste ne sert à rien, car `\def` ne peut pas être expansé. Ainsi le code définit `\titi`. Cet exemple montre une distinction subtile entre `\let` et `\def`. S'il y avait eu sept `\expandafter` au lieu de trois, cela aurait défini la commande `\tata`.

Il est possible de construire des commandes contenant n'importe quel caractère : il suffit de mettre ces caractères entre `\csname` et `\endcsname`, comme dans `\csname 1+1=2\endcsname`. Il est ainsi possible de construire dynamiquement des commandes. Par exemple

```
\def\nameuse#1{\csname #1\endcsname}
\nameuse{1+1=2}
```

Notons que la commande précédente ne fait pas grand chose : la valeur par défaut d'une commande construite avec `\csname` est `\relax`, et non une commande indéfinie. Pour définir la commande on peut dire :

```
\def\namedef#1{\expandafter\def\csname #1\endcsname}
\namedef{1+1=2}{vrai}
```

Cet exemple fonctionne dans la mesure où `\csname` est une commande qui peut être expansée (contrairement à `\def` qui ne l'est pas). Un exemple plus réaliste est le suivant :

```
\def\allocate#1{....}
\def\newcount#1{\allocate{ctr}\countdef#1\allocationnumber}
\def\newcounter#1{\expandafter\newcount\csname c@#1\endcsname}
```

On supposera que la commande `\allocate` positionne dans `\allocationnumber` un numéro de compteur unique (si l'argument est `ctr`). Si ce numéro est, par exemple 17, `\newcount\A` va définir `\A` comme équivalent à `\count17`, et `\newcounter{toto}` va définir `\c@toto`. Remarque : dans cet exemple, le but du `\expandafter` n'est pas d'expanser `\csname` avant `\newcount`, car l'ordre n'a pas d'importance, mais d'expanser le `\csname` (qui serait sinon l'argument de `\countdef`, qui n'expanse pas, mais redéfinit, son premier argument).

On verra plus que la commande `\newcounter` fait d'autres choses en plus. Notons que tout ce qui est entre `\csname` et `\endcsname` est expansé. Il y a parfois une différence importante entre ce qui est expansé, et ce qui est évalué. Voilà ce qui peut être expansé : les commandes utilisateurs, les marques (on n'en parlera pas), les instructions conditionnelles, `\expandafter`, `\csname`, `\noexpand` et `\the` (voir plus loin), ainsi que les conversions (`\string`, `\number`, etc), mais pas `\lowercase` (cette commande fait de la magie noire).

On peut définir une commande par `\edef`. Cette construction est plus délicate à manier. A priori, c'est comme `\def`, sauf que le corps de la macro est expansé, non pas une seule fois comme dans un `\expandafter`, mais tant que possible. Voici un exemple

```
\def\A{\B\C} \def\C{1}
{\let\B\relax \global\edef\D\bgroup{\A\noexpand\C\egroup}
```

Au lieu de `\global\edef`, on aurait pu dire `\xdef`, c'est la même chose : on est dans un groupe, et on veut que la définition soit visible hors du groupe. Le corps de la macro commence quand TEX voit une accolade ouvrante. Ainsi `\D` est une commande délimitée par `\bgroup`. Le `\egroup` s'expanse en une accolade fermante, donc termine le corps de la macro (c'est un peu tordu tout cela). Il y a deux moyens dans un `\edef` de supprimer l'expansion d'une commande : on peut mettre un `\noexpand` devant. Ce mécanisme ne fonctionne que pour les tokens qui sont visibles. Dans le cas de `\A`, la première expansion est `\B\C`. Comme la valeur de `\B` est `\relax`, ce token ne peut pas être expansé, et TEX le laisse tel quel (l'astuce est que TEX ne le remplace pas par `\relax`). Le résultat est donc : `\B1\C`. Finalement, le dernier caractère dans la ligne est une accolade qui termine le groupe dans lequel on a temporairement dit que `\B` était `\relax`.

Considérons l'exemple suivant.

```
\def\add#1#2{\edef#1{#1\do{#2}}
\def\cons#1#2{\begingroup\let\@elt\relax\xdef#1{#1\@elt #2}\endgroup}
```

Supposons que `\A` et `\B` soient deux macros vides. Si on dit

```
\add\A x, \add\A y, \add\A z,
\cons\B{ab}, \cons\B{cd}, \cons\B{ef}.
```

on aura `\do{A}\do{B}\do{C}` dans `\A` et `\@elt ab\@elt cd \@elt ef` dans `\B`. Ceci donne deux moyens d'ajouter un élément dans une liste. Notons les trois différences entre `\add` et `\cons`. Si on utilise `\add`, le résultat contiendra des accolades (ce qui rend `\cons` moins gourmand en mémoire). D'un autre côté, `\add` suppose que `\do` est `\relax`, tandis que `\cons` ne fait aucune hypothèse sur `\@elt` ; mais la définition est toujours globale. Une fois qu'on a construit une liste

avec \add, l'utilisation typique est de définir \do, puis d'évaluer la liste. Dans le cas de \cons, on utiliserait plutôt les deux commandes \car pour extraire le premier élément de la liste et \cdr pour avancer dans la liste.

L'un des problèmes avec \edef est le suivant. Si l'expansion du corps donne une sous-expression de la forme \let\A\B, comme \let n'est pas expansé, c'est \A qui le sera. Ceci peut provoquer une erreur incompréhensible si \A est une commande auxiliaire de LaTeX, définie comme suit \def\A\C{}. Pour éviter des difficultés dans la plupart des cas, une commande du type \rm est définie par \def\rm{\protect\prm}. Si \prm apparait dans un \edef, n'importe quoi peut se produire. La commande \protect est normalement définie comme ne faisant rien. Dans certains cas, elle est redéfinie localement comme \noexpand, \string ou \noexpand\protect\noexpand ; dans tous les cas, c'est une commande qui s'expanse et inhibe l'expansion du token qui suit.

1.2.7 Les variables dans TeX

On appellera, pour simplifier, variable dans TeX, tout objet qui peut être modifié ou consulté. Par exemple l'occupation de la table de hash n'est pas une variable (bien que la valeur finale soit imprimée dans le fichier log). La place qui reste sur la page courante est une variable, mais on n'en parlera pas ici. Certaines variables ont un statut spécial : la commande \mag ne peut être utilisée qu'une seule fois, dans LaTeX certaines quantités ne peuvent être modifiées que avant le \begin{document}, (dans la mesure où LaTeX utilise une copie privée de la variable). Certaines variables ne sont pas toujours visibles : \spacefactor n'existe qu'en mode horizontal, tandis que le numéro de la page courante n'est vraiment connu que lorsque TeX passe à la page suivante. On ne considérera dans ce paragraphe que les commandes et les registres.

Nous avons beaucoup parlé de commandes : \foo est une utilisation, \def\foo est une définition. Dans tous les autres cas, \foo est une définition, et il faut précéder \foo de quelque chose si on veut accéder à sa valeur. Par exemple \the\foo rend la valeur, \showthe\foo montre la valeur. Dans le cas \foo\bar, ceci met la valeur de \bar dans \foo. La valeur d'une variable peut être une entier (par exemple 27), une dimension (par exemple 13,2mm), un ressort (par exemple 3pt plus 2 fill), ou une liste de tokens. Supposons que la valeur de \foo soit un entier. On peut dire \foo=12, ce qui va mettre 12 dans \foo. On peut dire \foo=\bar, si \bar peut se convertir en entier (par exemple, si \bar est une dimension, et vaut 12sp, on mettre 12 dans \foo). On peut dire \foo=‘A, ou \foo=‘\A. Dans ces deux cas, la valeur mise dans \foo sera le code ASCII de la lettre A (ou : le code ASCII de l'unique caractère de la commande \A). Au lieu de dire 17, on peut dire '17 (nombre donné en base 8), ou "17 (nombre donné en base 16). Notons que -'77pt est une dimension valide.

Un ressort est une expression de la forme A plus B minus C. La valeur nominale est A, il s'étire jusqu'à $A + B$, se contracte jusqu'à $A - C$. L'exemple suivant est une erreur :

```
\def\toto{1}\def\foo{min}\def\bar{uscule}
\skip2\toto7 = 3pt \iftrue\foo\fi\iffalse\else\bar\fi
```

Après avoir vu \skip, TeX lit un nombre entier. Ici, c'est 217, car les macros sont expansées au passage. Ensuite, il faut un signe égal optionnel, puis un ressort. La quantité A est obligatoire, ici elle valut 3pt. Les quantités B et C sont optionnelles. Après avoir expansé les \if, \foo et \bar, TeX a vu minus. Il veut une dimension, ou un fill. Il a vu c, c'est une erreur.

Notons quelques subtilités : `\parskip=0pt plus 1filll`, suivi d'une ligne de la forme : Les subtilités de TEX, est une erreur : quand TEX lit le fil, il regarde s'il y a des L qui suivent (il en faut 1, 2 ou 3). Le fait que le L soit en majuscule ou en minuscule n'a pas d'importance, les espaces, non plus. Tout L, à partir du quatrième, provoque une erreur. Si on dit `\dimen0=2\dimen0`, on met le double de `\dimen0` dans `\dimen0`. Si on dit `\count0=2\count0`, on met 2 dans `\count0` (puis autre chose dans `\count0`) On peut dire `\count\count0=3` ceci met 3 dans le compteur dont le numéro est dans le compteur 0. Dans le cas

```
\chardef\foo 12\foo
\count0=3\ifnum \count1>\count0 ...\fi
```

l'occurrence du second `\foo` arrête la lecture des chiffres à mettre dans `\foo`. Par contre, dans la seconde ligne, TEX n'a pas encore fini de lire le nombre à mettre dans `\count0` lorsqu'il évalue le test (qui a donc peu de chances de comparer `\count1` avec 3). Par contre `\input\input` ne marche pas : « such attempts at sabotage must be thwarted ».

Les variables de TEX peuvent être classées en trois grandes catégories : les propriétés des caractères, les registres et le reste (dimension de la page, par exemple). À chaque caractère est associé un certain nombre d'information, par exemple le `\lccode` qui explique comment convertir le caractère en minuscules. et le `\catcode`, le code de catégorie qui explique comment l'analyseur lexical doit traiter le caractère. On parlera des `\lccode` plus loin.

Il y a dans TEX des registres de plusieurs types. On a vu `\dimen0` (qui contient une dimension), `\skip217`, qui contient un ressort, il y a `\box23`, qui contient une boite, `\toks23` qui contient une liste de tokens, `\count17` qui contient un entier. Une liste de tokens est identique à une macro sans arguments (modulo détails techniques). On a vu plus haut une macro qui ajoute un élément à une liste, la liste étant dans une macro. Le code suivant fait de même avec une liste de tokens.

```
\def\addtohook#1#2{#1\expandafter{\the#1#2}}
%   \T\expandafter{\the\T ...}
\newtoks\foo
\addtohook{\foo}{\do{A}}\addtohook{\foo}{\do{B}}\addtohook{\foo}{\do{C}}
```

La commande `\newtoks` définit `\foo` comme `\toks23` (le nombre est différent pour chaque appel de `\newtoks`). Chaque fois que l'on utilise `\addtohook` avec `\T` comme premier argument, ... comme second, le code commenté est exécuté. Le premier `\T` est une affectation. Ce que l'on veut, c'est une liste entre accolades. Le `\expandafter` va évaluer le `\the` d'abord. Ce qu'on aura entre accolades, c'est la valeur de `\T`, puis les .... Rajouter quelque chose en tête de liste est moins facile. Ici le résultat sera `\do{A}\do{B}\do{C}`. Un autre exemple est :

```
  \T\expandafter{\L}
% \xdef\L{...\the\T}
  \xdef\L{\catcode\the\count@=\the\catcode\the\count@\relax\the\T}
```

On suppose ici que `\T` est de la forme `\toks23`, et `\L` est une commande sans argument. Ce que fait le code est la chose suivante : on copie le corps de la macro dans `\T`, puis on dit que `\L`, c'est ... suivi de `\the\T`. Tout est expansé, sauf que le résultat de l'expansion de `\the\T` n'est pas expansé. Ce que fait le code, c'est rajouter quelque chose en tête de la liste `\L` (notons que `\add` expanse complètement la liste `\L`). Dans le cas de figure, si `\count@` contient 65, et que le caractère de code 65 a un `\catcode` de 11, on rajoutera `\catcode65=11\relax` dans la liste.

Il y a plusieurs intérêts à mettre du texte dans une boite. D'abord, si la boite est utilisée plusieurs fois, cela permet de gagner du temps. Par ailleurs, on peut créer une boite dans un

certain contexte, et l'utiliser dans un autre (par exemple, le titre de la section 1.2.11 contient du texte en verbatim, ce texte vient d'une boite créée avant la section). Finalement, mettre du texte dans une boite permet de connaître la taille du texte, et de prendre des décisions en fonction de cette taille.

Finalement, les registres les plus utilisés sont les compteurs. Plutôt que de dire `\count16`, et risquer des conflits avec un package qui utilise le même compteur, il vaut mieux utiliser des compteurs nommés, dont le nom est calculé (la commande `\newcount` est similaire à `\newtoks`, mais construit un nouveau compteur). Dans LaTeX, cela va beaucoup plus loin. Essentiellement `\newcounter{toto}` va allouer un numéro, disons 17, et définir une macro `\c@toto` dont la valeur une référence à `\count17`. Il y a une commande `\value`, et `\value{toto}` est identique à `\c@toto`. Si on veut la valeur du compteur `toto`, il suffit donc de dire `\the\value{toto}`. Si on veut mettre 10 dans le compteur, il suffit de dire `\value{toto}=10`. Comme expliqué plus haut, une fois que TeX a lu les deux chiffres 1 et 0, il va continuer à regarder la suite du texte, pour savoir s'il n'y a pas d'autres chiffres. La syntaxe recommandée est `\setcounter{toto}{10}`. Dans ce cas, il y a un `\relax` implicite derrière le 10, et la question ne se pose pas.

Si on a un compteur X, on peut lui ajouter une valeur Y, multiplier par Z, ou diviser par T. Exemple :

```
\count0\time
\divide\count0 60
\count2=-\count0
\multiply\count2 60
\advance\count2 \time
```

On met dans `\count0` la valeur de `\time`, le nombre de minutes depuis minuit. On divise par 60, on met l'opposé dans `\count2`, qui l'on multiplie par 60, auquel on ajoute `\time`. Ceci donne l'heure et la minute, et montre en passant comment on obtient le reste de la division. Autre exemple :

```
\dimen0=2mm\dimen1=0.2cm
\advance\dimen0 by-\dimen1
\count0=\dimen0
```

Dans cet exemple, on met dans `\dimen0` la différence entre 2 millimètres et 0,2 centimètre, et dans `\count0` la différence, exprimée en `sp`, le résultat est 5 (de l'ordre de deux centièmes de micron).

En LaTeX, il y a une commande `\addtocounter` qui ajoute une valeur à un compteur. Il y `\stepcounter`, qui incrémente un compteur, mais qui excécute aussi le code suivant :

```
\let\@elt\@stpelt  \csname cl@#1\endcsname
```

Un peu d'explication : si `toto` est un compteur, `\cl@toto` est une liste de la forme `\@elt{A} \@elt{B} \@elt{C}`. Le code précédent exécute donc `\@stpelt` avec comme arguments A, B, C, etc., cette commande met 0 dans le compteur donné en argument. Dans le cas où `titi` est l'argument optionnel de `\newcounter{toto}`, on exécute `\@addtoreset{toto}{titi}`, la commande étant définie comme suit :

```
\def\@addtoreset#1#2{\expandafter\cons\csname cl@#2\endcsname {{#1}}}
```

L'intérêt de cette manipulation est la suivante. Si on dit `\newcounter{chapter}`, suivi de `\newcounter{section}[chapter]`, le compteur de section sera remis à zéro chaque fois que le compteur de chapitre est incrémenté. De plus, si on définit le compteur de sous-section comme dépendant du compteur de section, celui-ci sera remis à zéro chaque fois que le compteur de

chapitre ou de section sera modifié. Note : ce n'est pas parce que l'utilisateur ne peut pas définir des commandes avec un argument optionnel à la fin que cela n'existe pas.

Il y une commande supplémentaire `\refstepcounter`, qui fait un tout petit peu plus qu'incrémenter le compteur. Notons d'abord que la sous-section courante est numérotée 1.2.7. Pour avoir le numéro dans la phrase précédente, nous avons mis un label dans la sous-section, et une référence ici. La valeur de la référence est ce que calcule `\refstepcounter` (pour simplifier, on va dire que c'est `\thetoto`, si `toto` est le nom du compteur). Rappelons que `\the\c@toto` est la valeur du compteur, et par défaut `\thetoto` est la même chose. Cependant `\thesubsection` est une macro définie comme `\thesection.\@arabic\c@subsection` : il s'agit donc de la section (qui est le numéro du chapitre suivi du numéro de la section), suivi du numéro de sous-section, en chiffres arabes. Au lieu de `\@arabic\c@toto`, on peut mettre `\arabic{toto}` (qui cache les commandes internes), ou demander à ce que le nombre soit imprimé en chiffres romains, ou en lettres, etc. On reviendra plus loin sur `\label` et `\ref`.

1.2.8 Fontes

Dans la suite de ce document, on se posera souvent la question de savoir quelle est la portée d'une commande de type `\it`. Rappelons que `\it` change la police courante, pour passer en italiques. Il est important de savoir quelle est la police courante. Un logiciel comme `latex2html` utilise beaucoup d'énergie pour résoudre ce problème. Il est extrêmement difficile de lui faire comprendre que `\french` est une commande du même type (surtout si ce que l'on veut, ce n'est pas une balise `<french>`, `</french>`, mais quelque chose de plus subtil, à savoir ajouter des espaces insécables devant les signes de ponctuations comme les points-virgules en dehors des mathématiques).

Nous expliquons dans cette partie comment fonctionne le mécanisme des fontes dans LaTeX. Au niveau le plus bas, on dit `\font\tenit=cmti10`. La commande `\tenit` demandera l'utilisation de la police cmti10, et donc de l'italique. À un niveau supérieur, on dira `\def\it{\fam\itfam \tenit}` (on ne dira pas ici à quoi sert cette commande `\fam`). À un niveau plus méta, on peut définir une macro `\tenpoint` qui définit (entre autres) `\it` ainsi que montré plus haut, et une macro `\twelvepoint` qui redéfinira la même commande dans un corps plus gros.

Dans LaTeX ce mécanisme est généralisé. Une fonte est définie par un ensemble d'attributs, et ces attributs peuvent être changés. Par exemple, on peut remplacer « cm » (computer modern) par « tm » (Times), sans rien changer d'autre. Il y a un attribut d'encodage (qui indique dans quel ordre sont placés les caractères dans la police), un attribut de forme, un attribut de graisse, un attribut de famille et un attribut de taille. Par exemple, `\large` change l'attribut taille de toutes les fontes, `\ttfamily` passe en famille machine à écrire, `\bfseries` passe en gras, `\itshape` en forme italique. Toutes ces commandes changent globalement la fonte courante. Il existe une version qui prend un argument, à savoir `\texttt`, `\textbf`, `\textit`, etc. Notons que les caractères, vus par TeX, sont des rectangles. Si on passe d'une police penchée à une police droite, il vaut mieux insérer `\/` (la commande de correction italique) pour éviter que les caractères ne se rentrent dedans. Cette commande est insérée automatiquement dans le cas de `\textit`. En parallèle à ces commandes, il existe l'équivalent pour le mode mathématique, par exemple `\mathit`. La commande `\it` est comme `\itshape`, sauf que la famille est romaine et graisse normale. Comme on peut le voir, il y a de multiples façons d'écrire un texte en italique.

Les mathématiques posent des problèmes particuliers. Entre b et b il y a peu de différence (on a un b en italique, le même en mode math). Par contre, dans le cas de *problème* et $problème$, la différence est manifeste. Notons que le deuxième mot a été entré comme `$probl\grave eme$`, vu que les accents sont gérés de façon spéciale en mode math. Un exemple de la forme `x$^{\grave{e}me}$` donne $x^{ème}$. Une manière plus rusée serait `x$^{\hbox{ême}}$` donc $x^{ème}$. Notons que l'exposant est en caractère romain, mais trop grand.

La bonne manière de mettre en exposant, en petit, et en romain, du texte est la suivante : `x\textsuperscript{ème}`, ce qui donne x^{ème}. Notons qu'en bon français, on dit 1er, 1re, 1ers, 1res, 2e, 3es, etc., via les commandes `1\ier`, `1\iere`, `1\iers`, `1\ieres`, `2\ieme`, `3\iemes`. Ceci évite toute contorsion ; par ailleurs, on dirait plutôt x^{ième}, et bien entendu x^{e}, si l'on parle du dixième siècle.

On supposera que dans le raweb, il n'y a pas de commandes de bas niveau, du type `\fam` ou `\font`, ni de commandes de changement de fontes globales (composer le texte en Times est interdit).

1.2.9 Les espaces

Dans TEX les espaces sont ignorés après une commande de la forme `\foo` (sauf hack innommable). Toute suite d'espaces est considérée comme un seul espace. Cependant l'opération de transformation de plusieurs espaces en un seul n'est effectuée qu'une seule fois. Ainsi `x␣{␣}␣␣␣␣{␣␣}␣y` contient 5 espaces. De même, dans le cas de `\def\A{␣}`, `\def\B{␣\A␣\A␣}`, `x␣\B␣y` contiendra 4 espaces (vu que `\A` est un espace, et `\B` contient trois espaces). La commande `\space` est en fait définie comme `\A` plus haut.

Tout retour chariot est considéré comme étant un espace (mais une ligne blanche est équivalente à `\par`). Ainsi, les espaces avant le retour chariot, le retour chariot, les espaces qui suivent sont considérés comme un seul espace. Une conséquence subtile est que les espaces en début de ligne sont ignorés, même si la ligne précédente se termine par un commentaire. Il y a deux façons dans TEX d'ignorer explicitement des espaces : ceux qui sont déjà lus, et ceux qui ne le sont pas encore. Il est de tradition de mettre les début et fin d'environnements seuls sur une ligne. Dans le cas de `... \end{x} \end{y} \end{z}`, il y a typiquement trois retours chariots, donc trois espaces. Le mécanisme d'environnements de LATEX demande à TEX d'ignorer au moins deux de ces trois. Rappelons que dans le cas d'une commande à argument optionnel, la recherche des arguments optionnels va lire les espaces. Ainsi, les espaces (et même un retour chariot) sont ignorés après un `\\`.

Les espaces sont ignorés en mode mathématique. En fait, dans le cas de `$x+y=z$`, ce sont les opérateurs plus et égal qui gèrent l'espacement. Certains espaces ne sont pas ignorés : par exemple `~` produit un espace insécable (il interdit une coupure de ligne), `\quad` produit un espace beaucoup plus long, `\,` un espace beaucoup plus court. Notons que `\~` définit un accent ; il n'est pas facile de produire le signe `~`. On voit parfois une horreur du genre `\~{}` dans une URL (i.e. en verbatim).

Pour des espaces plus longs, plutôt que `\,\,\,`, on peut utiliser `\hskip 3cm` ou `\hspace{3cm}` (respectivement syntaxe TEX et LATEX, elle n'est pas la même). On peut aussi utiliser des `\kern`. Dans le TEXbook, il est expliqué comment faire pour éviter de coller TEX au mot qui suit, sachant que l'espace après `\TeX` est ignoré. Les solutions sont les suivantes : `\TeX{} book`, `{\TeX} book` et `\TeX\ book` ; dans dernier cas, il s'agit de la commande backslash suivie d'un

espace (ou de plusieurs). Il y a une autre manière de s'en sortir, c'est de définir par exemple `\def\toto{toto\xspace}`, et d'inclure le package `xspace`. Avec cela le problème est résolu (sauf qu'en français correct, il faudra quand même dire `\toto~!`).

Il est de tradition, en anglais par exemple, de mettre plus d'espace après un signe de ponctuation fort qu'entre des mots. Pour savoir si un point termine une phrase, TEX utilise une heuristique : si le caractère précédent est une majuscule, ce point est considéré comme point abréviatif. Il faut utiliser `\@` si cela n'est pas le cas. Ainsi, dans le cas de `Donald~E. Knuth`, le tilde sert à éviter une coupure de ligne intempestive.

Nous avons dit plus haut qu'une ligne blanche équivaut à un `\par`. Cette commande termine le paragraphe courant ; elle ne fait rien s'il n'y a pas de paragraphe courant (ainsi deux lignes blanches consécutives sont équivalentes à une seule). Pour mettre un peu plus d'espace vertical, on peut utiliser `\vskip` ou `\vspace` (l'équivalent de `\hskip` ou `\hspace`). Il y a trois commandes prédéfinies (dont la taille dépend du corps courant) : `\bigskip`, `\medskip` et `\smallskip`. Il n'y a pas, en TEX de commande qui commence un paragraphe : on peut utiliser `\indent` ou `\noindent` en début de paragraphe. Dans LATEX, il y a une commande \\ qui termine la ligne courante (et commence un paragraphe non indenté, avec un espace interparagraphe nul). Pour mettre plus d'espace, les gens sont tentés de mettre deux \\ consécutifs. Mais ceci ne marche pas. Les rédacteurs ne manquent pas d'imagination pour résoudre le problème. Par exemple :

```
\def\ligne{\protect{\mbox{}\\\mbox{}\indent}}
```

Dans cet exemple, il y a une boite vide sur la ligne terminée par le \\, puis une seconde boite vide, puis une indentation de paragraphe. Dans le cas où `\ligne` est suivie par une ligne blanche, le `\indent` ne sert pas à grand chose. Dans le cas contraire, on peut avoir un paragraphe doublement indenté. Notons également que le `\protect` sert à éviter une expansion prématurée, au cas où l'argument serait mis dans une table des matières, par exemple. Ici, ce n'est pas le cas ; par ailleurs, l'argument du `\protect` est l'accolade, qui n'a nul besoin d'être protégée, la construction est donc doublement absurde.

Entre deux \\ on peut voir des quantités diverses et variées : un ~ (un espace), une boite vide `\hbox` ou `\mbox`, une boite avec un ~ ou un espace dedans, une ligne vide créée via `\centerline`. Certains poussent le vice jusqu'à mettre `\hbox{\par}`. Notons que ceci est équivalent à une boîte vide, mais pose des problèmes ardus à un traducteur automatique.

L'exemple suivant montre la difficulté de la gestion des espaces :

```
... dans
   %%%
   \htmladdnormallink{\url{http://www.dante.net/tf-ngn/}}%
                     {http://www.dante.net/tf-ngn/}.\null{}%
   %%%
~\`A ce jour, ....
```

Le signe `%` à la fin de la troisième ligne ne sert à rien, les espaces entre les arguments d'une commande étant ignorés. Le `\null` crée une boite vide (elle ne sert à rien). Les accolades qui suivent ne servent à rien (sans ces accolades, on n'aurait pas besoin du pour-cent qui suit). Les deux lignes qui définissent le lien hypertexte, et les deux lignes de commentaire autour définissent un texte sans espace. Notons qu'il y a un espace après le « dans », introduit par le retour chariot. Finalement, le ~ sur la dernière ligne est une horreur typographique : ceci est une incitation à mettre le À en fin de ligne, alors que c'est un mot qui débute une phrase.

1.2.10 Expansion conditionnelle

Dans les paragraphes précédents, nous avons vu comment définir une macro `\toto` qui s'expanse en `\titi` et une macro qui s'expanse en `tutu`. Un traducteur peut-il remplacer `\toto` par `\titi` et `\titi` par `tutu` ? La réponse est non, d'une part, parce que si `\foo` est une macro à un argument, `\foo\titi` va voir `\titi`, et donc `tutu` comme argument, tandis que `\foo tutu` va voir `t` comme argument. Une autre raison est l'existence de l'expansion conditionnelle. Le but de cette section est de donner un aperçu de ce que l'on peut faire, en espérant qu'un traducteur automatique ne verra jamais les constructions les plus compliquées.

Il est dans certains cas, très limités, possible de traduire une construction conditionnelle TEX en construction inconditionnelle XML, mais dans la majorité des cas, il faut interpréter la conditionnelle (il n'y a pas de conditionnelles dans XML). On verra cependant dans la section 1.4.4 un exemple important, où une traduction de XML vers XML dépend du contexte. Dans le cas de figure, on a une commande de type `\labelref` qui est utilisée dans un titre de section, qui agit comme `\label` dans le texte, et comme `\ref` dans le cas où le titre est dans la table des matières. Dans ce paragraphe, on va considérer les trois exemples suivants : `\couleur`, `\map` et `\loop`.

La commande `\map` est la plus simple à expliquer : le but du jeu est d'appliquer une commande à des arguments donnés dans une liste. Il y a un exemple dans la section précédente (si on incrémente un compteur X, on veut remettre à zéro des compteurs spécifiés dans une liste), un autre exemple dans la section 1.2.11 : à la troisième ligne il y a `\let\do\toto\dospecials`. On va supposer que la liste est définie par

```
\def\liste{\do{A}\do{B}\do{C}}
```

On a vu plus haut comment une telle liste peut être construite. La macro `\map` est alors triviale :

```
\def\map#1#2{\def\do{#1}#2}
```

Par exemple `\def\foo#1#2{ #1#2 }` et `\map{\foo A}\liste` donne AA AB AC . En règle générale, le premier argument de `\map` est un nom de macro, un unique token. Dans ce cas, on peut remplacer `\def\do{#1}` par `\let\do#1`, ce qui est plus efficace.

L'exemple de la couleur est le suivant. Supposons qu'on ait deux commandes `\enrouge` et `\envert`, qui prennent un argument et l'impriment en rouge et en vert, respectivement. On veut faire une macro `\couleur` à deux arguments : une couleur et un texte, le texte sera imprimée dans la couleur adéquate (dans le cas d'une couleur inconnue, la couleur courante sera utilisée). On va utiliser l'astuce suivante : la commande à utiliser est le nom de la couleur précédée des deux lettres e et n. Le code est trivial

```
\def\couleur#1{\csname en#1\endcsname}
```

Dans le cas de `\couleur{bleu}{texte}`, la commande `\enbleu` est appelée, elle n'existe pas, mais ce n'est pas grave, TEX l'ignore. Le seul problème, c'est que `\couleur{d}{document}` est la même chose que `\end{document}`, ce qui est assez catastrophique.

Nous avons en tête un autre exemple :

```
% \newcommand{\tutu}[3][tutu]{ arguments #1 #2 #3}
\def\tutuaux[#1]#2#3{ arguments #1 #2 #3}
\def\tutu{\ifnextchar[{\tutuaux}{\tutuaux[tutu]}}
```

La ligne commentée est identique aux deux autres. La commande `\ifnextchar` sera expliquée plus loin. Ce qui nous intéresse ici est : comment construire le `[#1]#2#3` à partir du nombre 3 ? Le code LATEX est donné ci-dessous, il convertit aussi `\newcommand{\toto}[3]` en `\def\toto#1#2#3`.

Ce code est un peu indigeste, résultat de plusieurs années de modifications. Nous avons supprimé le token `\l@ngrel@x` (pour simplifier les explications), et le token `\long` (qui ne sert à rien).

```
1  \def \@yargdef #1#2#3{%
2    \ifx#2\tw@
3      \def\reserved@b##11{[####1]}%
4    \else
5      \let\reserved@b\@gobble
6    \fi
7    \expandafter
8      \@yargd@f \expandafter{\number #3}#1%
9  }
10 \def \@yargd@f#1#2{%
11   \def \reserved@a ##1#1##2##{%
12     \expandafter\def\expandafter#2\reserved@b ##1#1%
13     }%
14   \reserved@a 0##1##2##3##4##5##6##7##8##9###1%
15 }
```

Ce code contient plus de caractères spéciaux (`#` et `@`) que n'importe quel autre. Pour faciliter l'explication, on va remplacer `\@yargdef` par `\ydef` et `\@yargd@f` par `\yaux`, renommer `\reserved@a` et `\reserved@b` en `\Ra` et `\Rb`. Par ailleurs, il faut savoir que lorsque TEX lit une définition de commande il remplace `##` par `#` et `#1`, `#2`, `#3` par des tokens spéciaux, on dénotera ceux-ci par `X`, `Y`, `Z`, etc. Voici donc ce qu'a mémorisé TEX :

```
\ydef XYZ->
  \ifx Y\tw@
    \def\Rb#11{[##1]}
  \else
    \let\Rb\@gobble
  \fi
  \expandafter \yaux \expandafter{\number Z}X
\yaux XY ->
  \def \Ra #1X#2#{
    \expandafter\def\expandafter Y\Rb #1X}
  \Ra 0#1#2#3#4#5#6#7#8#9#X
```

La commande `\ydef` prend 4 arguments, le premier est le nom de la commande à définir (`\toto` ou `\tutuaux`), le second est un indicateur (si c'est `\tw@`, on traite le cas d'un argument optionnel), le troisième est le nombre d'arguments, à savoir 3, et le dernier (non visible) est le corps de la macro à définir. Il est important, pour la suite, de savoir que ce corps commence par une accolade.

Les lignes 2 à 6 seront expliquées plus tard. Les lignes 7 et 8 contiennent un `\expandafter`. L'ordre d'évaluation est donc : on expanse d'abord le `\number`, puis le reste. Cet appel à la commande `\number` est en général de peu d'intérêt, mais permet de construire une commande à N arguments, où N est dans un registre (certaines packages utilisent cela). Elle autorise des constructions étranges : `‘\^^C` est une manière légale de dire que la commande prend trois arguments.

Expliquons les lignes 2 à 6. L'idée est simple : si la commande à définir est normal, Y n'est pas `\tw@` et `\Rb` est définie comme une commande avec un argument, qui ne fait rien avec l'argument. Sinon, `\Rb` est définie en ligne 3, et TEX mémorise

```
\Rb X1 -> [#1]
```

Il s'agit d'une commande qui prend un argument, qui est tout ce qui précède le chiffre 1, et qui rend une liste de 4 tokens : les crochets, un # et le chiffre 1.

Expliquons maintenant les lignes 11 et 12. Il y a un `\def`, et TEX mémorise la commande `\Ra`. Comme dit plus haut, le `#1` sera remplacé par un X, et le X qui suit sera remplacé par sa valeur. Pour comprendre les choses, on va mettre W à la place de X (c'est le nombre d'arguments) et `\toto` à la place de Y (c'est la commande à définir). On obtient donc :

```
\Ra XWY# -> \expandafter\def\expandafter \toto\Rb XW}
```

Le # qui reste signifie : le second argument est tout ce qui précède l'accolade ouvrante (celle du corps de la macro à définir). Le premier argument est tout ce qui précède le W. On traitera les cas suivants : W est 0, W est un chiffre entre 1 et 9, W est #, W est autre chose (exercice pour le lecteur : que se passe-t-il si W est # ? que faut-il donner en argument à `\newcommand` pour que W soit un # ?). Pour rendre les choses encore plus simples, soit $s(n)$ la suite `#1#2....#n#`, et $S(n)$ la suite `#n...#9`. La ligne 14 contient

```
\Ra 0#1#2#3#4#5#6#7#8#9#W
```

que l'on peut voir comme étant 0, $s(n-1)$, n, $S(n+1)$, `#W` pour tout n entre 1 et 9. En particulier, si W est un chiffre entre 1 et 9, c'est 0, $s(n-1)$, suivi de W, suivi de $S(n+1)$, `#W`, si W est 0, c'est rien, suivi de W, suivi de $S(0)$`#0`, et si W n'est ni un chiffre, c'est 0, $s(9)$ suivi de W, suivi de rien. Dans le cas où c'est machin suivi de W suivi de bidule, alors les arguments de `\Ra` seront machin pour X et bidule pour Y.

Supposons, pour commencer, que l'on définisse une macro normale. Alors `\Rb` est `\gobble`, et l'expansion de `\Rb XW` consiste à supprimer le premier token de X, et à rajouter W derrière. Si W est 0, X est vide, `XW` est 0, et `\Rb XW` est vide. Sinon, X commence par un 0, que `\Rb` va supprimer. Ainsi, si W est un chiffre entre 0 et 9, le résultat sera `#1#2...#W`. Si W est autre chose, le résultat sera `#1...#9#W`. Ceci va provoquer une seule erreur TEX : You already have nine parameters.

Dans le cas où on définit une commande avec argument optionnel, le résultat sera le même, mais le `#1` initial sera replacé par `[#1]`. Comme il n'y a pas de `#1` initial si W est 0, il se peut qu'il y ait un erreur de forme Paragraph ended before `\reserved@b` was complete, ou `\end` occurred when `\ifx` on line 9 was incomplete, ou autre.

On va proposer dans la suite une autre méthode pour calculer `[#1]#2#3` à partir du nombre $N = 3$. Cette méthode est peut-être moins efficace, mais plus simple à expliquer. L'hypothèse est qu'on dispose d'une macro `\sharp`, à un argument, qui donne `[#1]`, `#2`, `#3`, etc. Avec cette façon de faire, le cas particulier $N = 1$ est identique au cas général. Ce que l'on veut faire, c'est calculer la suite des `\sharp i`, pour i entre 1 et N. Une solution consiste à construire `\sharp1\sharp2...\sharp9`, faire du pattern matching pour avoir les N premiers termes, rajouter le N à la fin, puis à utiliser `\map` pour évaluer le `\sharp`. Une autre manière de faire (plus raisonnable) consiste à utiliser une boucle. Notons que la version originale du code consistait à faire une boucle (en partant de N, et en décrémentant), le `\sharp` étant évalué à la fin, via un `\map`. C'est là que les conditionnelles entrent en jeu.

À part la commande `\ifcase`, la syntaxe des fonctions de tests est de la forme `\ifXX A \else B \fi`. Si le test est vrai alors A sera exécuté, sinon B. La partie `\else` est facultative. Un exemple de `\ifcase` est

```
\ifcase #1 zero \or un \or deux \or trois \or quatre \else autre\fi
```

La commande `\ifnum` compare deux nombres. On peut regarder si le premier est plus petit, plus grand, ou égal au second. Exemple (qui serait plus facile à comprendre s'il y avait un test $\geq$) :

```
\sharp 1\ifnum #1>1 \sharp2\fi ... \ifnum#1>7 \sharp8\fi\ifnum#1>8 \sharp9\fi
```

ou

```
\sharp 1\ifnum #1>1 \sharp2\ifnum ... \sharp8\ifnum#1>8 \sharp9\fi\fi...\fi
```

en empilant les `\if`, ce qui est plus efficace en temps CPU, et moins en mémoire.

On peut faire une boucle comme suit

```
\def\code{\advance\count0 by 1 \sharp\the\count0}
\def\loop{\ifnum\count0<\count1 \code\loop\fi}
```

Dans toute boucle, il y a une initialisation, un test de fin, un incrément, et un code. On n'a pas donné le code de l'initialisation : il faut mettre 0 dans `\count0` et le nombre voulu dans `\count1`. Le test de fin est dans `\loop`, l'incrément et le code sont dans la macro `\code`. En règle générale, on exécute le code avant l'incrément. Si on inverse, il faut initialiser `\count0` et `\count1` à un de plus que précédemment (encore une fois, ce serait plus simple s'il y avait un test $\leq$). On peut alors dire ceci :

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
\let\repeat\fi % this makes \loop...\if...\repeat skippable

\loop \ifnum \count0<\count1 \sharp \the\count0 \advance\count0 by 1\repeat
```

Supposons que `\count0` soit 1, et `\count1` soit 4. La commande `\loop` met dans `\body` tout le code, puis appelle `\iterate`. Ceci évalue `\body`. Le test est vrai, et on exécute `\sharp1`, puis on incrémente le compteur. On met ensuite `\iterate` dans `\next`. Ce que voit TEX, c'est le `\else`, il va jusqu'au `\fi`, puis évalue `\next`, c'est-à-dire `\iterate`. À un certain moment, il y aura 4 dans `\count0`, le test sera faux, et on mettre `\relax` dans `\next`. Notons que la commande `\iterate` est récursive (elle s'appelle elle-même), mais la récursion est terminale : lorsque `\iterate` s'appelle elle-même, c'est via le `\next`, le dernier token dans la liste. Cette récursion terminale ne prend pas de place en mémoire.

La version suivante de `\loop` est donnée par LATEX :

```
\def\loop#1\repeat{\def\iterate{#1\relax\expandafter\iterate\fi}%
  \iterate \let\iterate\relax}
```

On constate que `\body` et `\next` ne sont pas utilisées. La première subtilité est le `\relax`. Rappelons que notre code se termine par `advance\count0 by 1`. Lorsque TEX a vu le 1, il continue à parser pour trouver la fin du nombre. Dans la version originale il y avait un `\let`, ce qui arrête tout. Dans cette version, sans le `\relax`, il y aurait `\expandafter`, ce qui ferait que TEX expanserait le `\fi`, donc ce qui suit, dans le `\ifnum` qui suit, et ceci avant que `\count0` ne soit mis à jour, ce qui est évidemment catastrophique. L'intérêt du `\expandafter` est, bien sûr, de rendre la récursion terminale. La construction `\expandafter\toto\fi` est équivalente à `\let\next\toto\else\let\next\relax\fi\next`. Remarque : si on peut inverser le sens de la boucle (i varie de N à 1, plutôt que de 1 à N), on peut éviter le compteur `\count0` : il suffit de comparer `\count1` à 1, et de décrémenter le compteur à chaque itération.

Il y a plusieurs autres manières de faire des boucles, par exemple :

```
\@whilenum \count0<\count1 \do {\sharp \the\count0 \advance\count0 by 1}
\def\titi{A,B,C}
\@for \toto:=\titi\do{x\toto y}
```

Le dernier exemple suppose que le `\catcode` du double point est normal, cet exemple risque de ne pas marcher dans des documents français.

Parlons maintenant des deux constructions `\if` et `\ifx` : elles comparent des tokens. Dans le cas de `\if`, les macros sont expansées, dans le cas de `\ifx`, elles ne le sont pas. Avec les définitions `\let\B\A` et `\def\C{\A}`, `\ifx\A\B` sera vrai, et `\ifx\A\C` sera faux (sauf si `\A` est comme `\C`, c'est à dire une macro sans argument dont le corps est `{\A}`, cas où `\A` est une macro qui récurse violemment). Dans le cas de `\if\A\B...`, il se produit la chose suivante : si `\A` s'expanse en quelque chose qui a au moins deux tokens qui ne peuvent être expansés, ces deux tokens sont comparés. Si `\A` s'expanse en un unique token celui-ci est comparé au premier token de l'expansion de `\B`, et si `\A` s'expanse en la chaîne vide, c'est comme si on avait dit `\if\B`. Dans le cas où les tokens à comparer ne sont pas des lettres, le résultat est étrange : `\if\par\let` est vrai. Ainsi, pour comparer deux chaines de caractères, il vaut mieux mettre ces chaines dans une macro et utiliser `\ifx`. La commande `\iftrue` est équivalente à `\if00`, le test est toujours vrai. Le source de TEX contient l'exemple suivant `\if\iftrue abc\else d\fi`. Le second test est vrai, le premier test compare donc a et b, il est faux. Quand TEX voit le `\else`, c'est le `\else` du second `\if` (la gestion des conditionnelles est donc compliquée, dans l'exemple plus haut, il manque un `\else` optionnel et un `\fi`).

Définissons maintenant la macro `\couleur` proprement. Rappelons qu'il y a deux arguments, un nom de couleur, et du texte, et qu'on veut le texte dans la couleur adéquate. Elle peut être codée comme suit :

```
\def\couleur#1#2{%
 \def\crouge{rouge}\def\cvert{vert}\def\cc{#1}%
 \ifx\cc\crouge\enrouge{#2}\else\ifx\cc\cvert\envert{#2}\else#2\fi\fi}
```

On peut éviter toutes ces définitions auxiliaires, à condition de les cacher dans une autre macro, disons `\ifstringeq`. Ceci donne alors :

```
\def\ifstringeq#1#2#3#4{%
 \def\tempa{#1}\def\tempb{#2}%
 \ifx\tempa\tempb#3\else#4\fi}

\def\couleur#1#2{%
 \ifstringeq{#1}{rouge}{\enrouge{#2}}
 {\ifstringeq{#1}{vert}{\envert{#2}}{#2}}}
```

On peut se poser la question : pourquoi ne pas faire une commande qui prend deux arguments, les compare, et rend vrai ou faux ? le problème est qu'une commande ne rend pas de valeur. La seule façon d'avoir une valeur de retour est de mettre la valeur dans une variable (une macro, un registre, une token list, une dimension de fonte, etc. ; pour simplifier, on ne parlera que des macros). Cette affectation peut être faite par l'appelant ou l'appelé. Dans cet exemple, c'est l'appelant qui fait l'affectation :

```
\def\couleur#1{%
 \ifstringeq{#1}{rouge}{\let\next\enrouge}
 {\ifstringeq{#1}{vert}{\let\next\envert}{\let\next\relax}}%
 \next}
```

Avec cette nouvelle définition, on peut dire `\couleur{vert}[clair]{texte}`, pourvu que la commande `\envert` accepte un argument optionnel. Ici c'est l'appelé qui fait l'affectation :

```
\def\streq#1#2{%
 \def\tempa{#1}\def\tempb{#2}%
```

```
 \let\egal 0%
 \ifx\tempa\tempb\let\egal1\fi}

\def\couleur#1{%
 \streq{#1}{rouge}%
 \ifx\egal 1\let\next\enrouge\else
    \streq{#1}{vert}%
    \ifx\egal 1\let\next\envert\else \let\next\relax\fi\fi
 \next}
```

Une autre subtilité de TEX est la suivante : si TEX voit `\if AB C\else D\fi`, il compare A et B, si le test est faux, il va chercher le `\else` (sans expanser C, ni les `\if` qui sont dedans). Dans le cas contraire, il va expanser C, jusqu'à trouver le `\fi` ou le `\else`, puis s'il a vu un `\else`, il va sauter ce qui est entre le `\else` et le `\fi`. Le `\fi` qui termine le `\if` n'est pas nécessairement celui qui apparait dans le code plus haut (dans le cas de `\loop\if...\repeat`, il n'y a pas de `\fi` explicite, il est macro-généré). Il y a quelques précautions à prendre : dans le cas de

```
\iffalse \loop\ifnum0=0 étrange\repeat \else Ok\fi
```

TEX est censé évaluer Ok, mais pas le reste ; il y a `\ifnum`, donc TEX doit voir un `\fi`, sinon cela ne marche pas. Ce `\fi` est en fait le `\repeat` (voir le commentaire associé à l'instruction `\let\repeat\fi`). Une version de `\couleur` qui utilise ce truc est :

```
\def\ifstringeq#1#2#3#4{%
   \def\tempa{#1}\def\tempb{#2}%
   \ifx\tempa\tempb\aux{#3}\else\aux{#4}\fi}
\def\aux#1#2\fi{\fi#1}
\def\couleur#1{%
 \ifstringeq{#1}{rouge}{\enrouge}{\ifstringeq{#1}{vert}{\envert}{\relax}}}
```

Dans ce cas de figure, si le test est vrai, on appelle `\aux`, qui est une macro qui va lire le texte jusqu'au `\fi`, évaluer le `\fi`, puis évaluer le troisième argument. Si le test est faux, TEX ne va pas évaluer le `\aux`, ni le `{#3}`, il va voir le `\else`, puis évaluer le second `\aux`. Le `\fi` dans `\ifstringeq` n'est donc jamais évalué.

On va maintenant considérer une variante du problème de la couleur : on veut écrire une macro à 3 arguments A, B et C, qui va lire un token ; si c'est A, on exécute B, sinon C. Il est indispensable de passer par une macro auxiliaire si on veut lire quelque chose. La solution est donc

```
\def\ifnextchar#1#2#3{%
 \let\tempa=#1\def\tempb{#2}\def\tempc{#3}%
 \ifaux
}
\def\ifaux#1{%
  \let\lettoken=#1%
  \ifx\lettoken\tempa\let\tempd\tempb\else\let\tempd\tempc\fi
  \tempd
}
```

Notons que nous avons mis un signe égal dans `\let\tempa=#1`, ce signe est optionnel, il est indispensable pour le cas où le premier argument serait le signe égal. Cette façon de faire a un inconvénient : comment garantir que l'argument est un token ? Récrivons d'abord le code comme suit :

```
\def\ifnextchar#1#2#3{%
 \let\tempa=#1\def\tempb{#2}\def\tempc{#3}%
 \ifaux
}
\def\ifaux#1{\let\lettoken=#1\ifnch}
\def\ifnch{%
  \ifx\lettoken\tempa\let\tempd\tempb\else\let\tempd\tempc\fi
  \tempd
}
```

Ce que l'on voudrait faire, c'est remplacer `\ifaux` par une macro qui lise un token, le mette dans `\lettoken`, et appelle `\ifnch`. On pourrait penser à `\expandafter` ou `\afterassignment`. Mais le problème essentiel est que l'on veut lire un token. En particulier `\ifaux`, suivi de `{x}` ou `x`, voit la même chose. Les spécifications sont : dans le premier cas, ce qui suit la macro n'est pas un x, dans le second c'est un x. Pour cette raison, TEX fournit un moyen de positionner `\lettoken` sans lire le token. Au lieu de `\expandafter` ou `\afterassignment` on utilise `\futurelet`, comme suit :

```
\def\ifnextchar#1#2#3{%
 \let\tempa=#1\def\tempb{#2}\def\tempc{#3}%
 \futurelet\lettoken\ifnch
}
```

Avec cette définition, le code marche. Le vrai code est un peu différent : si le token lu par `\ifnch` est un espace, il est ignoré, et le token suivant est comparé. En particulier, comme \\ est une commande avec un argument optionnel, les espaces qui suivent sont ignorés (normalement les espaces ne sont pas ignorés après une commande de type \\). Une question non triviale est : comment une macro peut-elle lire un espace et rien d'autre ? Comme ceci :

```
\def\:{\toto} \expandafter\def\: {}
```

Voyez-vous l'astuce ? l'espace après double point ne disparait pas. La macro que l'on définit est `\toto` (vu que `\:` s'expanse en `\toto`), il s'agit d'une macro délimitée par un espace.

Revenons à la commande `\map`. Supposons que l'on veuille faire quelque de spécial pour le premier ou le dernier élément de la liste. Par exemple, on a une liste d'objets, on veut séparer les objets par des virgules, et mettre un point à la fin. Une manière récursive de faire est la suivante :

```
\def\foo#1#2\fin{\textit{#1}\ifx#2\fin\fin.\else, \foo#2\fin\fi}
```

le texte `\foo{A}{B et C}{D}\fin` sera le même que
A, *B et C*, *D*.
En réalité, le code plus haut est complètement faux. Dans le cas `\foo{A}{XY}{UV}\fin`, la macro reçoit deux arguments, le premier est A (sans les accolades). Le second est `{XY}{UV}`. Mais dans le cas `\foo{A}{XY}\fin`, le second argument est XY (sans les accolades). Ceci a deux conséquences : d'abord le test risque de comparer X et Y (au lieu de comparer XY et `\fin`). Il vaut mieux remplacer le test par `\ifx\fin#2\fin`. De toutes façons, tester si un argument est vide est tellement compliqué que la première version du code est toujours fausse. La deuxième conséquence est que `\foo` sera appelée récursivement avec `XY` et non `{XY}` comme argument : il y aura donc une virgule entre le X et le Y.

La solution suivante n'a pas les inconvénients cités plus haut : le fait de mettre un `\do` devant les arguments empêche les accolades de disparaitre.

```
\def\foo\do#1#2\fin{\textit{#1}\ifx\fin#2\fin.\else, \foo#2\fin\fi}
\foo\do{A}\do{B}\do{C}\fin
```

On peut se poser la question : pourquoi ne pas itérer ? Ce qui suit l'argument de `\do` est soit un `\do`, auquel cas ce n'est pas le dernier élément, ou alors c'est `\fin`, et c'est le dernier. Consulter ce qui suit l'argument est facile, il suffit d'utiliser `\futurelet`. Il y a une autre solution : au lieu de mettre une virgule après chaque argument sauf le dernier, il suffit d'en mettre une devant chaque argument, sauf le premier. Par exemple

```
\def\do#1{\ifx\premier1\let\premier 0\else , \fi\textit{#1}}
\let\premier 1
\do{A}\do{B}\do{C}.
```

Un autre problème est le suivant : avec `\begin{E}x,y, z\end{E}`, se débrouiller pour avoir un espace après les virgules, et un point final. Dans ce cas la solution est simple : il suffit de redéfinir localement la virgule pour qu'elle imprime une virgule suivie d'un espace, et qu'elle ignore les espaces qui suivent, la fin de l'environnement ajoutant un espace (la partie non triviale est comment éviter la récursion : la virgule s'expanse en virgule qui s'expanse etc.). Un problème du même genre : au lieu de x, y, z on a des commandes `\p`. Dans ce cas, on veut rajouter des virgules, sauf s'il y en a. La solution n'est pas compliquée : la commande `\p`, une fois qu'elle a fini de traiter son argument utilise `\ifnextchar` pour regarder si une virgule suit ; dans ce cas, on n'a pas besoin de redéfinir la virgule (application : gérer les virgules et leur espacement dans les listes de mots clés et liste de participants dans le raweb).

Il y a d'autres constructions possibles. Nous verrons plus loin comment définir un environnement `verbatim`. Avec les mêmes techniques, on peut faire un environnement `comment`, qui ignore le contenu de l'environnement. C'est comme si on disait

```
\newenvironment{comment}{\iffalse}{\fi}
```

On peut noter la construction étrange suivante : `{\ifnum0=‘}\fi`. Dans ce cas, on compare deux nombres, 0 et le code ASCII de l'accolade. Que la comparaison donne vrai ou faux, le résultat de l'expansion est le même : on ne fait rien. Notons que le parseur voit une accolade ouvrante et une accolade fermante, tandis que l'évaluateur ne voit qu'une accolade ouvrante. Le résultat est donc le même qu'un `\bgroup`. En fait, dans certains cas particuliers, ce n'est pas le cas, le lecteur intéressé pourra consulter la référence [4]. On fera l'hypothèse, pour notre traducteur que le source TEX ne contient ni de `\bgroup`, ni de tests comme celui qui précède. On supposera également qu'aucune commande n'utilise d'accent grave pour convertir le caractère qui suit en code ASCII.

Une autre difficulté pour un traducteur est la construction suivante :

```
\def\foo#1{
  \sbox\tempboxa{#1}%
   \ifdim \wd\tempboxa >\hsize
    #1\par
  \else \hbox to \hsize{\hfil\box\tempboxa\hfil}%
 \fi}
```

Dans ce cas, la commande `\foo` prend un argument, le met dans une boite, et compare la largeur de la boite avec la largeur de la ligne courante. Si la boite tient sur la ligne, elle est centrée. Sinon, l'argument est imprimé comme un paragraphe normal. La difficulté principale est, bien entendu, de calculer la largeur du texte.

Une petite curiosité, qui permet de faire des boucles :

```
\def\nlines#1{\expandafter\nlineii\romannumeral\number\number #1 000\relax}
\def\nlineii#1{\if#1m\expandafter\theline\expandafter\nlineii\fi}
```

```
 \def\theline{A}
\nlines{5}
```

Le résultat est : AAAAA. L'idée de base est prendre le nombre, le multiplier par mille, et le convertir en chiffres romains. Si le nombre est 5, le résultat sera mmmmm (cinq M). La suite est une boucle : tant que le caractère courant est un m, appliquer `\theline`, et le `\relax` termine la boucle. La subtilité de l'exemple réside dans le fait qu'il n'y a pas d'effet de bord, c'est de l'expansion pure (il y a plusieurs façons d'obtenir 5 M de suite, mais la plupart utilisent une variable auxiliaire). On peut multiplier l'argument par mille, mais il faut également une variable auxiliaire. Ici, la multiplication est obtenue par concaténation. Il faut un `\number` dans le cas où l'argument est un nombre implicite, par exemple, une référence à un registre. Si l'argument est `\count0` (valeur du compteur 0), l'espace après `#1` va être lu pour trouver le numéro du compteur ; cet espace est donc indispensable. Si maintenant l'argument est `\count1␣` (i.e. contient un espace à la fin), les deux `\number` sont indispensables pour lire les deux espaces.

On peut utiliser le code suivant :

```
\bgroup
  \edef\foo{\ifnum 0<0#1x\else y\fi}\def\bar{x}%
  \ifx\foo\bar
    \global\compteurtheme=#1
  \else \global\compteurtheme=0 \@latex@error{Pas un thème #1}\@eha\fi
\egroup
```

Dans le cas où `#1` contient une suite de chiffres dont un au moins est non nul, le test sera vrai, et `\foo` contiendra x, donc sera égal à `\bar`. Dans le cas où `#1` contient des chiffres, comme précédemment, et suivi d'autre chose, disons X, alors le test sera vrai, `\foo` contiendra `Xx`, et ne sera pas `\bar`. Dans les autres cas, `\foo` contiendra y, et ne sera pas `\bar`. Ce test est utilisé par le style rapport de recherche pour vérifier que `#1` est un nombre (il y a d'autres tests sur le nombre, qui sont effectués en regardant la valeur du compteur `\compteurtheme`). Cet exemple montre qu'un certain nombre de tests fait dans les fichiers de style ne peuvent absolument pas être faits par un traducteur automatique par simple macro expansion. Si le traducteur est en Perl, tester que `#1` est un nombre est trivial : il suffit de faire du pattern matching (est-ce qu'il y a autre chose que des chiffres ? si oui, c'est mauvais ; est ce qu'il y a autre chose que des zéros ? si non, c'est que le nombre est nul).

1.2.11 Un exemple non trivial de macro : `\verb`

Le code qui suit est un exemple, un peu simplifié, d'une commande LaTeX.

```
\def\verb{%
  \bgroup
    \let\do\@makeother \dospecials
    \verbatim@font\@noligs
    \@vobeyspaces \frenchspacing\@sverb}

\def\verb@egroup{\global\let\VBG\@empty\egroup}
\let\VBG\@empty

\def\@sverb#1{%
  \catcode`#1\active
  \lccode`\~`#1%
```

```
13   \gdef\VBG{\verb@egroup\error{...}}%
14   \aftergroup\VBG
15   \lowercase{\let~\verb@egroup}}
```

Notons d'abord que ce code contient deux lignes vides, donc deux `\par`, et que les fins des lignes 5, 7 et 15 induisent un espace. Ces espaces et changements de paragraphe sont ignorés (pourvu que la définition soit hors texte). Les lignes 1, 10, 12 et 13 sont terminées par un `%`, sans cela, on aurait des espaces non voulus dans la macro expansion.

Ceci définit une commande `\verb`, qui ouvre un groupe, via `\bgroup`. À la ligne 3, on exécute `\dospecials`, après avoir redéfini `\do`, ceci change le type de tous les caractères spéciaux pour les rendre normaux (y compris les caractères rendus spéciaux par des packages comme `babel`). À la ligne 4, on demande de passer en police tt, et de supprimer les ligatures prédéfinies, par exemple celle qui transforme << en «). À la ligne 5, on demande à TEX d'interpréter proprement les retours à la ligne, et de passer en mode « espacement français » (i.e. l'espace inter-mot est indépendant des signes de ponctuation). Ensuite on appelle `\@sverb`.

Aux lignes 7 et 8, on définit une commande `\verb@egroup`. Cette macro positionne globalement `\VBG` qui ne fait rien, et puis quitte le groupe. À la ligne 13, `\VBG` est définie comme appelant `\verb@egroup` puis fait une erreur (non explicitée ici).

La commande `\@sverb` est un peu spéciale. Elle prend un argument, disons *c*. Cet argument est un caractère (notons que dans le cas de `\verb{toto}` ou `\verb\titi`, l'argument est `{` ou \, vu que ces caractères ont changé de type). Ce caractère est rendu actif. À la ligne 12, on dit que l'équivalent en minuscule du caractère tilde est ce caractère *c*. Ensuite commence la grosse machinerie.

La commande `\aftergroup\VBG` mémorise la commande `\VBG`. Cette commande est mise en attente, et évaluée à la fin du groupe (celui ouvert par `\bgroup`). Ensuite, on exécute la ligne 15. Ce que fait `\lowercase`, c'est de convertir en minuscules toutes les lettres. Dans notre cas, il y a deux commandes et un tilde. Les commandes ne sont pas converties. En général le tilde n'est pas converti, sauf que, ici, on a dit explicitement que la conversion donne *c*.

Prenons l'exemple de `\verb+\toto+`. Dans ce cas, on aura le signe plus dans *c*. Le résultat du `\lowercase` est donc `\let + \verb@egroup`, et TEX va exécuter ces commandes, ce qui termine l'expansion du `\verb`. La situation est la suivante : tous les caractères sont équivalents à des lettres, sauf le retour chariot qui va provoquer une erreur et le signe +, qui est actif, et dont la définition est donnée ligne 7. Après avoir fini l'expansion du `\verb`, TEX poursuit son travail normalement : ajouter les caractères au paragraphe courant, et interpréter les commandes. Il va voir 5 lettres : backslash, t, o, t et o. Il va ensuite voir un +, caractère actif. Il va donc interpréter la commande associée, à savoir `\verb@egroup`, qui, comme dit, va redéfinir (globalement) `\VBG`, et terminer le groupe. Quand on termine le groupe, toutes les variables définies localement reprennent leur ancienne valeur. En particulier, la police courante est rétablie, le `\catcode` des caractères spéciaux, et celui du +, sont rétablis, de même que la conversion du tilde en minuscules. Ainsi TEX se retrouve dans le même état qu'avant le `\verb`. Comme on termine un groupe, le `\VBG` mis de côté sera évalué, mais il ne fait rien. Donc `\verb+\toto+` donnera `\toto`.

Notons que l'auteur du document aurait pu entrer l'expression précédente sous la forme `\verb-\verb+\toto+-`. Paresseux comme il est, il a juste dit `|\verb+\toto+|`.

Exemple tordu : `\def\truc#1{#1#1} \truc{\verb+x+}++`. Au lieu de `xx++`, ceci donne `x+x+`. En effet, TEX ne change pas les `\catcode` des caractères déjà lus. Ainsi, les caractères délimitant

la fin du `\verb` ne sont pas ceux dans l'argument de `\truc`, mais ceux placés derrière. S'il n'y avait pas ces + à la fin de la ligne, on aurait une erreur de la forme : l'argument du `\verb` contient une fin de ligne illégale.

Autre exemple : `\def\truc#1{{#1}} \truc{\verb+x+}`. Dans ce cas, on a mis des accolades autour de `#1` dans le corps de `\truc`. Regardons de plus près ce qui se passe une fois l'expansion du `\verb` terminée. Rappelons que tous les caractères (sauf le retour chariot et le +) sont normaux, que la police courante est `tt`, qu'il y a un groupe ouvert, et un token dans la pile. TEX va alors lire le `x` et le `+` (celui-ci n'est pas actif, car faisant partie de l'argument de `\truc`). Il va ensuite continuer à lire le corps de `\truc` et voir l'accolade fermante (c'est une accolade normale). Elle va terminer le groupe courant. Ainsi qu'expliqué plus haut, TEX va revenir dans son état normal. Il va exécuter le `VBG` empilé. Or, sa valeur est celle définie en ligne 13 (il s'agit d'une définition globale), et le code en ligne 7 sera exécuté. Ceci va terminer un autre groupe (celui ouvert par l'accolade ouvrante avant l'argument de `\truc`). Une fois le `\verb@egroup` fini, la commande `\error` va être exécutée. Notons la magie de la chose : lorsque l'erreur est signalée, tous les groupes ouverts sont fermés, et TEX se retrouve dans un état propre. Ce que montre cet exemple, c'est qu'avec un peu d'expérience, on peut écrire des macros compliquées avec un rattrapage d'erreur correct. Notons également que mettre du verbatim dans un titre de section est réservé aux experts.

Il y a une variante de `\verb`, c'est l'environnement `verbatim`. Il y a deux manières de procéder : dans la première on dit que chaque caractère est un caractère normal (comme dans le cas de `\verb`), sauf le retour à la ligne, qui s'expanse en `\par`. La commande appelée par l'environnement va prendre un argument délimité par `\end{verbatim}`, imprimer l'argument, puis évaluer le `\end{verbatim}` (il faut jongler avec les `\catcode` pour faire cela proprement). La seconde manière de faire consiste à utiliser une boucle : on utilise une macro qui lit une ligne (un argument délimité par la fin de la ligne). On s'arrête si l'argument est `\end{verbatim}`, on imprime la ligne, et on itère sinon. Avec cette manière de faire, on peut numéroter les lignes.

1.2.12 Encodage de caractères

Cette section est un peu technique, on s'en excusera à l'avance. Dans TEX un caractère est représenté par un nombre entre 0 et 255 (dans Ω, un caractère est codé sur 16 bits). Dans la version originale de TEX, les caractères étaient donnés sous la forme 7bits. Faisons l'hypothèse, raisonnable, que l'encodage interne soit l'ASCII. Alors le caractère A est représenté par le nombre 65. Question : comment manipuler le caractère retour chariot, caractère dont le code est 13 ? Dans le source TEX, il est dit que la construction `\catcode'15=0` fonctionne (ceci signifie que le retour chariot, dont le numéro est donné ici en base 8, se comporte comme un backslash). Rappelons que toute marque de fin de ligne est remplacée par un retour chariot, et les espaces en fin de ligne sont ignorés. Ainsi, manipuler le retour chariot en tant que marque de fin de ligne a des effets non voulus (notons en passant que le nom d'une commande ne peut être distribué sur plusieurs lignes, donc si la fin de ligne est comme un backslash, le résultat sera la commande de nom vide).

La solution proposée par TEX est la suivante : dans le cas d'une construction de la forme `^^M`, si le caractère a un code ASCII x, le résultat est le caractère dont le code ASCII y est $y = x - 64$ si $x \geq 64$, et $y = x + 64$ sinon. Ainsi `^^M` est le caractère de code ASCII 13, c'est le retour chariot, ou contrôle-M, noté traditionnellement `^M`. De même, `^^@` est le caractère nul, de code ASCII 0. Dans tous les cas de figure $0 \leq y < 128$. Lorsque TEX est passé à 8 bits, ce mécanisme a été

étendu : une construction de la forme `^^ab` désigne le caractère dont le code est ab, en base 16 (ici le caractère 171, donc le guillemet ouvrant). Au lieu de a et b, on peut donner un chiffre ou une lettre minuscule entre a et f (de sorte que `^^A` donne toujours contrôle-A). Comme Ω est codé sur 16 bits, on peut dire `^^^^1ebf`, pour désigner le caractère Unicode e accent aigu accent circonflexe.

Si on a un texte qui contient essentiellement des caractères 7bits, et très peu d'autres caractères, l'utilisation de caractères 16bits consomme énormément de place. La phrase précédente contient 159 caractères ASCII et 6 caractères non ASCII. Dans TEX, cela consomme 165 octets, dans Ω, 330. Une manière d'encoder le source, en autorisant tout caractère Unicode consiste à remplacer les caractères 8bits par des séquences de la forme `\'e`. Cela prendrait donc 177 octets. La norme UTF-8 n'en requiert que 171. Expliquons comment cela marche, cf [5].

Tout caractère X, dont le code interne x satisfait $x < 128$ est codé par un seul octet, à savoir X. Dans les autres cas, on utilise plusieurs octets AB, ABC, ou ABCD, dont les valeurs a, b, c et d sont toutes au moins 128. Si la séquence fait k octets, les k premiers bits de a sont 1, le bit suivant est 0. Pour les autres octets, le premier bit est 1, le bit suivant est 0. Les autres bits sont ceux de x. L'algorithme est donc le suivant : écrire x en base 64, ce qui va donner x_1, x_2, x_3, etc. Le premier octet est $x_1 + t$, les autres sont $x_i + 128$. Supposons par exemple $x < 2^{11}$. Écrivons $x = u2^6 + v$. Il y a 5 bits dans u, 6 bits dans v. Alors $a = 128 + 64 + u$ et $b = 128 + v$. Si $x < 2^{16}$, mais est plus grand que 2^{11}, on écrit $x = (u2^6 + v)2^6 + w$. Il faut trois octets, avec $a = 128 + 64 + 32 + u$, $b = 128 + v$, $c = 128 + w$. Dans le cas de caractères 8bits, on peut écrire $x = u2^6 + v$, avec $u < 4$. Si $u = 0$, c'est un caractère 7bits, il suffit d'un octet, sinon u est 1, 2 ou 3, et a est 193, 194 ou 195, et le premier octet est Á, Â ou Ã, et le deuxième octet est $x - 64$. Par exemple é donnera Ã©. Ainsi, un texte iso-latin encodé en UTF-8 a un schéma bien particulier.

Supposons que l'on veuille faire comprendre du Unicode à TEX. Le code qui suit est extrait de `xmltex`, il a été écrit par D. Carlisle. Il faut faire deux choses : expliquer à TEX la signification du caractère Unicode 233, et lui faire comprendre que Ã© est la représentation UTF-8 de ce caractère. Toute manipulation des caractères Unicode, qu'ils soient codés en UTF-16, UTF-8, iso-latin2, etc., passe par une représentation intermédiaire sous la forme `\8:Ã©`, i.e., une commande dont le nom est `8:` suivi de la représentation UTF-8 du caractère.

Pour faciliter la compréhension du code qui suit, considérons ceci :

```
\def\toto#1{%
  \catcode`#1\active
  \begingroup
  \lccode`\~ = `#1%
  \lowercase{\endgroup\def~{\titi}}}
```

Supposons que l'argument de `\toto` soit une lettre, par exemple A (si l'argument n'est pas un caractère, une erreur sera signalée). La première ligne du code va rendre la lettre A active. Ensuite on ouvre un groupe. On dit ensuite `\lccode`\~ = `A`. Ceci veut dire : l'équivalent en lettre minuscule du caractère tilde est la lettre A (en fait, TEX met 65 en position 126 dans une certaine table, le fait que tilde ne soit pas une lettre et que A soit une lettre majuscule importe peu). La dernière ligne est spéciale. Ce que fait `\lowercase`, c'est de prendre un argument, délimité par des accolades, et rendre cet argument (sans les délimiteurs), en remplaçant toutes les lettres par leur équivalent minuscule. Le résultat est ici `\endgroup\def A{\titi}`. Notons l'astuce : on suppose que le caractère tilde est actif lors de la définition. Le résultat de `\lowercase` est alors un caractère A actif. C'est donc un argument valide pour le `\def`. Notons que le `\endgroup`

finit le groupe courant : on oublie donc la conversion du tilde en minuscules. On peut mettre `\uppercase` à la place de `\lowercase`, et utiliser `\uccode` au lieu de `\lccode`.

Ce que l'on veut faire, c'est construire les caractères ABCD, connaissant les codes internes a, b, c et d. On va dire `\uccode ‘.=a, \uccode‘!=b, \uccode‘,=c, \uccode ‘;=d`, suivi d'un `\uppercase{.!,;}`. Pour calculer a, b, c et d, on utilise le code suivant.

```
\gdef\XML@utfeight@a#1{
     \@tempcnta\count@
     \divide\count@64
     \@tempcntb\count@
     \multiply\count@64
     \advance\@tempcnta-\count@
     \advance\@tempcnta"80
     \uccode‘#1\@tempcnta
     \count@\@tempcntb}
```

Ce que fait ce code est le suivant : soit x la valeur du compteur `\count@`. On calcule le quotient et le reste de la division par 64. Le quotient est remis dans `\count@`. On ajoute 128 au reste, et on met cela dans le compteur `\@tempcnta`. Ceci donne le dernier octet (ou l'octet suivant) dans une suite. Cette valeur sera le `\uccode` de l'argument de la commande. On considère ensuite :

```
\gdef\XML@utfeight@b#1#2#3#4{
     \advance\count@"#10\relax
     \uccode‘#3\count@
     \uppercase{\gdef\XML@tempa{#2#3#4}}}
```

Rappelons que le premier octet d'une séquence, de longueur k (2, 3 ou 4), est ce qui reste dans `\count@` plus t, où t est un nombre qui commence par k bits égaux à 1. Si y vaut respectivement 12, 14 ou 15, on a $t = 16y$, donc si `#1` est C, E ou F, c'est y en base 16, et `"#10\relax` est la quantité t à ajouter. On a donc maintenant dans `\count@` la valeur du premier octet. On met dans le `\lccode` de `#3` cette valeur. On appelle ensuite `\uppercase`.

La commande suivante prend un argument délimité par un point virgule.

```
\gdef\XML@charref#1#2;{
  \begingroup
  \uppercase{\count@\if x\noexpand#1"\else#1\fi#2}\relax
  \ifnum\count@<"80\relax
    \uccode‘\~\count@
    \uppercase{
    \ifnum\catcode\count@=\active
      \gdef\XML@tempa{\utfeightay~}
    \else
      \gdef\XML@tempa{~}
    \fi}
  \else\ifnum\count@<"800\relax
     \XML@utfeight@a,
     \XML@utfeight@b C\utfeightb.,
  \else\ifnum\count@<"10000\relax
     \XML@utfeight@a;
     \XML@utfeight@a,
     \XML@utfeight@b E\utfeightc.{,;}
   \else
     \XML@utfeight@a;
     \XML@utfeight@a,
```

```
     \XML@utfeight@a!
     \XML@utfeight@b F\utfeightd.{!,;}
    \fi
    \fi
  \fi
  \endgroup}
```

Supposons que l'argument soit `233;`. Alors `#1` sera 2, `#2` sera 33, le test sera faux, et on met 233 dans le compteur `\count@`. Si l'argument est `xE9`, on aura `x` pour `#1`, E9 pour `#2`, et on va mettre `"E9`, donc 233 aussi dans `\count@`. Notons le `\uppercase` dans cet exemple : si l'argument est `Xe9`, le test sera également vrai. La suite est un test sur le nombre de bits, donc sur le nombre d'octets qu'il faut créer. Si le considère le cas de 233, il faut deux octets, qui sont Ã et ©. La magie du `\uppercase` va faire que `\XML@tempa` sera défini comme `\utfeightbÃ©`. Dans le cas d'un caractère 7 bits, par exemple A, le résultat sera `\utfeightay A` si A est actif, et A sinon.

Pour des raisons qui seront expliquées plus loin, l'expansion de `\utfeightb` dépend du contexte. Une manière de définir cette commande est la suivante.

```
\def\unprotect@utfeight{
  \let<\XML@lt@markup
  \let&\XML@amp@markup
  \def\utfeightax##1{
    \csname 8:\string##1\endcsname}
  \let\utfeightay\utfeightax
  \let\utfeightaz\utfeightax
  \def\utfeightb##1##2{
    \csname 8:##1\string##2\endcsname}
  \def\utfeightc##1##2##3{
    \csname 8:##1\string##2\string##3\endcsname}
  \def\utfeightd##1##2##3##4{
    \csname 8:##1\string##2\string##3\string##4\endcsname}}
```

Considérons finalement la procédure suivante :

```
\gdef\UnicodeCharacter#1#2{
   \begingroup
   \def\active{\catcode\count@}
   \XML@charref#1;
   \expandafter\expandafter\expandafter
   \expandafter\expandafter\expandafter
   \expandafter
    \gdef\XML@tempa{#2}
  \endgroup}
```

Notons la redéfinition de `\active` à la troisième ligne. Dans ce cas `\ifnum\catcode\count@=\active` sera vrai. En d'autres termes, si on passe 65 comme premier argument (c'est le code ASCII de la lettre A), le résultat sera `\utfeightay A`, que A soit actif ou non. La quatrième ligne du code définit une commande `\XML@tempa`. Ensuite il y a un grand nombre de `\expandafter`, il y a en 7. Comme expliqué à la section 1.2.6, le token `\XML@tempa` sera expansé trois fois. La première expansion est `\utfeightb Ã©`, la seconde est `\csname 8:Ã©\endcsname` et la troisième est la commande `\8:Ã©`. C'est cette commande qui sera définie, la valeur étant le second argument de `\UnicodeCharacter`.

Si on veut parser une expression XML de la forme `<balise att=val>`, il faut ranger `val` dans une table indexée par `att`. Ce que fait TEX est essentiellement `\def\att{val}`. Le nom

de la commande est construite par des `\csname` (comment faire autrement ?). Supposons que le nom `att` contienne par exemple un é (ou n'importe quel caractère Unicode). Ce que l'on va faire, c'est tout convertir en UTF-8. On va donc se débrouiller pour que `\utfeightb Ã©` s'expanse en Ã©. Ceci est fait par le code suivant.

```
\def\utfeight@protect@chars{
  \let\utfeightax\string
  \let\utfeightay\string
  \let\utfeightaz\string
  \def\utfeightb##1##2{
    ##1\string##2}
  \def\utfeightc##1##2##3{
    ##1\string##2\string##3}
  \def\utfeightd##1##2##3##4{
    ##1\string##2\string##3\string##4}}
```

Le lecteur intéressé pourra voir qu'il y a deux autres définitions dans le fichier `xmltex.tex`, une pour le cas où l'objet serait dans un `\edef`, et une pour le cas où il serait dans un `\write`.

Rappelons que le but du jeu est de lire des expressions de la forme Ã©. Ce qu'on va faire, c'est rendre actifs tous les caractères (ceux qui peuvent être le premier octet d'une séquence). On utilisera une boucle pour ce faire. Le code qui suit est un peu arrangé pour être plus lisible. On suppose que `\CountA` est un compteur.

```
\gdef\utfeightloop{
  \uccode`\~\count@
  \expandafter\uppercase\tempa
  \advance\count@ 1
  \ifnum\count@<\CountA
  \expandafter\utfeightloop
  \fi}
```

Ce que fait le code est clair : pour tout entier n, entre `\count@` inclus et `\CountA` exclus, on exécute `\tempa`, dans lequel le caractère tilde a été remplacé par le caractère de code n. Soit maintenant le code suivant :

```
    \begingroup
    \count@="C2  % 128+64+2
    \CountA="E0  % 128+64+32
    \gdef\tempa{{ \xdef~##1{\noexpand\utfeightb\string~##1}}}
    \utfeightloop
%
    \count@="E0
    \CountA="F0 % 128+64+32 + 16
    \gdef\tempa{{ \xdef~##1##2{\noexpand\utfeightc\string~##1##2}}}
    \utfeightloop
%
    \CountA="F4  % 128+64+32+16 +4
    \gdef\tempa{{ \xdef~##1##2##3{\noexpand\utfeightd\string~##1##2##3}}}
    \utfeightloop
    \endgroup
```

Il y a trois boucles. Notons la fin de la dernière boucle : ce code permet de traiter le cas de premier octet de tout caractère Unicode X avec $2^8 \leq x < 2^{20}$. Par exemple Ã est défini comme `\utfeightb Ã`, où le Ã est un caractère passif (à cause du `\string`).

Le cas des caractères 7bits est plus tordu. Le code est le suivant.

```
1  \count@0
2  \catcode0=13
3  \gdef\XML@tempa{
4   \begingroup
5     \uccode0\count@
6    \uppercase{\endgroup
7      \edef^^@{
8        \ifnum\catcode\count@=11 %
9          \noexpand\utfeightay\else\noexpand\utfeightax\fi
10       \noexpand^^@}
11     \expandafter\edef\csname 8:\string^^@\endcsname{\string^^@}}
12  \ifnum\count@<127\advance\count@1 \expandafter\XML@tempa\fi}
13 \XML@tempa
```

Ce code définit une macro `\XML@tempa` et l'exécute, pour chaque n dans le compteur `\count@`, entre 0 et 127 inclus. Au lieu d'utiliser le caractère tilde comme d'habitude, on utilise le caractère nul, entré sous la forme `^^@`. Plutôt que de dire `‘^@@`, on utilise le nombre 0. Supposons, pour fixer les idées, que $n = 65$. La boucle traite donc le caractère A. À la ligne 11, on définit une commande de nom `\8:A` comme valant A. Rappelons que le caractère nul est actif (ligne 2), et `\uppercase` remplace `^^@` par un A actif; les deux `\string^^@` rendent un A non actif. Le code entre les lignes 7 et 10 définissent le caractère actif A. Notons la subtilité : a priori, le caractère A n'est pas actif, mais le `\edef` voit un A actif. Ce que l'on fait, c'est définir une commande, qui sera associée à A, le jour où il sera actif. En fait, le code regarde le `\catcode` du caractère à la ligne 8. Si A est de type lettre (ce qui est le cas), le résultat sera `\utfeightay`, sinon, `\utfeightax`. Une fois, ce code exécuté, on remplace `\utfeightax` par `\utfeighty` pour les caractères retour chariot, `^`, `_`, `~`, `%`, `$`, `#`, `{`, `}` et `\`. On utilise `\utfeightaz` pour les caractères `<` et `&`.

Pour les caractères `{}\_^`, on utilise le code suivant :

```
\begingroup
\catcode‘\"=12\relax
\gdef\ActivateASCII#1{
  \uppercase{\count@"0\if x\noexpand#1\relax\else\count@#1\fi\relax}
  \toks@\expandafter{\nfss@catcodes}
       \xdef\nfss@catcodes{
       \catcode\the\count@=\the\catcode\the\count@\relax\the\toks@}
  \toks@\expandafter{\XML@catcodes}
     \xdef\XML@catcodes{
       \catcode\the\count@\active\the\toks@}
  \fi}
\endgroup
```

On passe à la procédure un numéro de code, la procédure enregistre dans `\nfss@catcodes` l'ancien `\catcode` et dans `\XML@catcodes` le nouveau (qui est actif). Notons la grosse astuce dans la ligne qui contient `\uppercase`. Ce que fait le `\uppercase`, c'est rendre `x5C` et `X5c` équivalents. Dans ce cas, le test sera vrai. Rappelons que `\if x#1` compare x et le premier caractère de `#1`. Si le test est vrai, on va se retrouver avec une expression de la forme `\count@"05C\relax`, ce qui met 5c (donné en base 16) dans `\count@`. Si par contre l'argument est 32, le test sera faux, ce qui va donner `\count@"0` suivi de `\count@32`, ce qui met 32 dans le registre, après avoir mis 0 (en base 16). À comparer avec `\XML@charref`.

1.2.13 Génération dynamique de commandes

L'un des problèmes d'un traducteur automatique concerne les macros qui lui sont inconnues. Nous avons montré plus haut la syntaxe de `\def` et de `\newcommand`, et donné quelques exemples. Il est clair que vérifier la validité d'un document est impossible, sans exécuter TEX (ou quelque chose qui lui ressemble comme un clone).

Dans la section 1.2.1 nous avons changé le `\catcode` du dollar. Avec cette modification, le premier dollar qui suit (et qui est évalué) a un comportement étrange, tous les autres dollars sont normaux. Si l'on veut vérifier le document, il faut vérifier que ce premier dollar est conforme à son usage. Oui, mais où est la première utilisation de ce dollar ? Peut-être dans la commande qui formate la page de titre (surprenant, mais véridique).

Notre objectif est de valider partiellement un document de type rapport d'activité. On supposera que le document se compile sans erreur. Une validation partielle signifie, par exemple, qu'on regardera que l'introduction est bien avant la conclusion, et non le contraire. On peut être plus ambitieux : valider complètement et traduire, sans utiliser LATEX.

Une des questions intéressantes est : qu'est un document valide ? On pourrait donner la définition suivante : c'est un document qui se compile sans erreur, et qui peut être visualisé et imprimé (certains visualisateurs ont des problèmes lorsque le document fait plus de 2^{15} pages). Si l'on veut une table des matières, placée en début de document, il faudra au moins trois passes : dans la première passe, on verra que le chapitre 17 commence à la page 359. Lors de la deuxième passe, on mettra cette information dans la table des matières. Cependant, comme la table des matières fait trois pages, le chapitre commencera à la page 362 (ou 363, suivant les cas). Il faut donc une nouvelle passe. Il n'est pas possible dans TEX de demander de faire deux passes sans génération de dvi, et de générer le dvi uniquement lors de la dernière passe. Ainsi, des informations doivent être lues et écrites sur disque dans un ou plusieurs fichiers auxiliaires. Ces fichiers peuvent contenir des commandes qui définissent d'autres commandes dynamiquement. En fait, les fichiers auxiliaires sont lus deux fois : au début et à la fin (ainsi LATEX peut nous avertir si la table des matières est ou non à jour). Les commandes définies dans ce fichier ont un comportement différent dans les deux cas.

Il y a une autre façon de résoudre le problème de pagination évoqué plus haut : il suffit de numéroter 1 la première page du texte (les pages précédentes pourraient être numérotées avec des chiffres romains). Ce document contient 131 pages, plus trois pages de garde, qui ont un numéro, mais qui n'apparait pas. Notons que, s'il y a deux pages numérotées 3, une référence à la page 3 est ambigüe.

Il arrive assez fréquemment que des modifications dans le fichier source (changement des packages) rendent les fichiers auxiliaires illisibles (avec en prime, une erreur incompréhensible). Il se peut aussi que des packages soient plus sensibles que d'autres ; imaginons une commande `\myref` qui dise : voir l'équation page N, sauf si N est la page suivante, auquel cas le résultat serait : voir l'équation page suivante. Comme les deux textes ont une longueur différente, il se peut que, si l'équation est sur la page suivante, insérer le texte long la fasse passer sur la page d'après et vice-versa. Dans ce cas de figure, l'algorithme ne se stabilisera pas : toute exécution de LATEX donnera un résultat différent. Un tel document est-il valide ? Oui, mais il nécessite une intervention manuelle, ce qui n'est pas l'objectif.

1.3 Perl

Le traitement du rapport d'activité est fait par des procédures standard (LaTeX, `latex2html`, etc), mais nécessite des pré et post-processeurs écrits en Perl.

Un exemple typique de procédure est

```
sub remove_idrefs{
 1 while s/\\9(\d+)!/$idtable{$1}/g;
}
```

Ici on définit une procédure, de nom `remove_idrefs`. Elle ne prend par d'arguments (ou alors, ils sont ignorés). Le corps de la procédure contient une seule instruction. Celle-ci est de la forme `A while B`. On aurait pu mettre à la place `while(B) {A}` (en rajoutant des délimiteurs supplémentaires). Dans les deux cas, le résultat est le même : tant que B n'est pas faux, exécuter A.

Dans l'exemple qui nous intéresse, l'évaluation de B rend un entier. Par convention un entier est faux si, et seulement si, il est nul. Une chaine de caractères est fausse si elle est vide (idem pour les listes). Une expression du type `$x{$y}` signifie l'élément associé à `$y` dans la table d'association `%x` (c'est autre chose que `$x`). L'évaluation d'une telle quantité est considérée comme fausse s'il n'y a rien dans la table à la position voulue (et c'est une erreur si on veut faire autre chose avec l'objet indéfini).

Dans l'exemple, la valeur de A est 1. Évaluer 1 rend 1, ceci n'est pas intéressant. Le code plus haut signifie donc : évaluer B, tant que cette évaluation rend autre chose que faux.

La quantité B est de la forme `s/C/D/g`. Cela veut dire : substituer C par D globalement. Les règles de substitution sont subtiles. Considérons donc un cas simple. On suppose que l'expression dans laquelle se fait la substitution est 11212. Dans le cas de `s/12/22/` on remplace 12 par 22, et on obtient 12212. Dans le cas de `s/12/22/g`, on remplace tous les 12 par des 22, on obtient 12222. Notons qu'il reste un 12 non substitué, dans la mesure où Perl ne revient pas en arrière. Il faut une boucle `while` si l'on veut faire des substitutions sur du texte généré par les substitutions précédentes.

Dans l'exemple, on remplace `\\9(\d+)!` par `$idtable{$1}`. Dans les expressions C et D, Perl fait de l'évaluation (ou de l'interpolation), autrement dit, considère que certains caractères ou suites de caractères ont des significations spéciales. En général, on rend un caractère normal, en le faisant précéder d'un backslash. Dans l'expression C, on rend un signe particulier (backslash, parenthèse, crochets, signes plus, étoile, etc) normal en le faisant précéder d'un backslash, on rend une lettre ou chiffre spécial en le faisant précéder d'un backslash.

Dans notre cas, le pattern C est : un backslash, suivi du chiffre 9, suivi d'un groupe (de numéro 1), suivi d'un point d'exclamation. Le groupe contient `\d+`. Ceci signifie : des chiffres, en nombre arbitraire non nul. Il peut y avoir plusieurs groupes dans un pattern. On peut référencer un groupe via `\1`, `\2`, etc. Dans la deuxième partie, on peut également référencer un groupe, via `$1`, `$2`, etc. Autrement dit, la procédure remplace `\918!` par l'élément en position 18 dans la table. Le résultat de la substitution est le nombre d'occurences substituées. Ainsi, si au moins une substitution se fait, on recommence le tout (peut-être qu'il y a du `\9` dans l'élément en position 18 dans la table).

Soit la procédure

```
sub new_idref{
 ++$idrefs;
 $idtable{$idrefs} = $_[0];
 "\\9$idrefs!"    # $emacs
}
```

Cette procédure contient un commentaire destiné à faire croire à Emacs qu'il y a un nombre pair de dollars, et donc n'est pas intéressant. La procédure prend un argument (l'élément en position 0 de la liste `@_`, obtenu via `$_`) incrémente une variable globale, et si le résultat est 18, positionne l'argument dans la table `%idtable` en position 18, et rend `\918!`.

Les deux procédures décrites sont donc intimement reliées l'une à l'autre. Ce genre de technique permet de cacher temporairement des bouts d'expression.

Soit la procédure

```
sub remove_brace{
 1 while s/\{([^\{\}]*)\}/++$id;"\\3$id!$1\\4$id!"/egs;
}
```

Dans ce cas, le pattern est formé d'une accolade ouvrante, d'un groupe, d'une accolade fermante. Le groupe contient `[^\{\}]*`. Ceci signifie : n'importe quel caractère, à l'exclusion des accolades, en nombre arbitraire. En d'autres termes, le pattern matche un texte délimité par des accolades ne contenant pas d'accolades. Le E après le dernier slash dit que le deuxième membre est une expression à évaluer (et non à interpoler). Ainsi, le remplacement est `"\\3$id!$1\\4$id!"` où `$id` a été incrémenté. Cette expression est interpolée (les `$1` et `$id` sont remplacés par leur valeur). Exemple.

```
{ \begin{x} ... \end {x} }
```

La première substitution donne

```
{ \begin\31!x\41! ... \end \32!x\42! }
```

Une deuxième substitution donne

```
\33! \begin\31!x\41! ... \end \32!x\42! \43!
```

Encore plus fort :

```
sub remove_env{
   s/\\(begin|end)\s*\\3(\d+!)(.*)\\4\2/normalise($1,$3)/egs;
   1 while s/\\5([^\?]*)\?([^\?]*)\\6\1\?/
        ++$id; "\\7$id!$1!$2\\8$id!$1!"/egs;
}
```

La première expression substitue une expression qui commence par un backslash suivie d'un groupe (qui contient begin ou end), suivi d'espaces optionnels, suivi d'un backslash, suivi de 3, suivi d'un groupe, formé d'une suite non vide de chiffres et un point d'exclamation, suivi d'un autre groupe (qui contient n'importe quoi), suivi de backslash, quatre, et de ce qui est dans le deuxième groupe ; une telle expression sera remplacée par l'appel de la fonction `normalise`. Pour simplifier, on suppose que `normalise`(x, y) rend `\5y?` (si x est begin) et `\6y?` sinon.

La deuxième expression dans la procédure va remplacer une suite qui commence par `\5x?` et qui se termine par `\6x?` (avec le même x), et qui ne contient pas de point d'interrogation, par la même suite, mais où `\5x?` et `\6x?` sont remplacés par `\7n!x!` et `\8n!x!` (où n est un nombre unique).

Ainsi, à l'aide de toutes ces procédures, on peut remplacer

```
\begin{x} A \begin{x} B \end{x} C \end{x}
```

par

```
\738!x! A \737!x! B \837!x! C \838!x!
```

et donc apparier les début et fin d'environnement.

Remarque. Toutes ces procédures font des hypothèses implicites. La première est que le texte ne contient pas les marqueurs spéciaux (i.e. backslash suivi d'un nombre). On peut raisonnablement supposer que la commande `\3` n'est pas utilisée dans le document, mais il n'est pas interdit de voir `\\3`. Ainsi, la première précaution consiste à masquer les `\\`. Notre algorithme ne fonctionne pas si un nom d'environnement contient un point d'interrogation ou un point d'exclamation (qui sert de délimiteur de fin de nombre). Ceci n'est grave, les noms des environnements standards LaTeX étant formé de lettres (avec une astérisque optionnelle). Par contre, le texte ne doit pas contenir de point d'interrogation, ce qui signifie qu'il faut les masquer.

Supposons que l'on veuille remplacer tous les A par des B dans les environnements de nom y dans une chaine donnée `$x`. On commence par dire `$x=~ m/Y/`, où Y est un certain pattern. Typiquement Y sera de la forme `\\7(\d+!y!)(.*)\\8\1` ; ceci va positionner deux variables `$1` et `$2` (vu qu'il y a deux groupes) et `$‘`, `$&`, `$’` (ce qui précède l'expression matchée, l'expression trouvée, et ce qui est derrière). Comme on va faire du pattern matching (qui va modifier ces variables globales), on va sauvegarder des résultats partiels, `$x="$‘\\7$1"`, `$y="\\8$1"` et `$tmp=$&`. Il suffit ensuite de dire `$tmp =~ s/A/B/g` et de concaténer les morceaux `"$x$tmp$y"`.

Dans aucune des trois procédures décrites plus haut, nous n'avons dit quelle était l'expression modifiée par le pattern matching. Il s'agit de la variable `$_`, une variable globale. Abuser de variables globales est dangereux. Heureusement, on peut se mettre dans un mode dans lequel la syntaxe des variables globales est spéciale (par exemple `$::idx`, nous n'avons pas utilisé cette syntaxe dans les exemples, pour les rendre moins durs à déchiffrer). Dans le cas de la fonction `remove_braces`, on a une substitution de la forme `s/A/B/`. Dans cet exemple, il vaut mieux remplacer A par une variable globale (à des fins de réutilisation). Notons que Perl évalue les variables dans un pattern. Comme les patterns doivent être compilés (traduits en automates à état fini), un pattern variable doit être recompilé à chaque utilisation. Il y a cependant un moyen de dire qu'il est constant (modifier les variables apparaissant dans les patterns supposés constants est donc une opération dangereuse).

L'exemple qui suit sera utilisé dans la suite.

```
1  sub make_abstract {
2      local($_) = @_;
3      $_ = &translate_environments("$_");
4      $_ = &translate_commands($_);
5      local($env_id) = " CLASS=\"ABSTRACT\"" if ($USING_STYLES);
6      join('','\n<H3>$abs_title:</H3>\n"
7          , (($HTML_VERSION > 3)? "<DIV$env_id>" : "<P>")
8          , "<I>", $_ , "</I>",
9          , (($HTML_VERSION > 3)? "</DIV>" : "</P>")
10         , "\n<P>");
11   }
```

Comme on le voit, il n'y a pas de marqueurs spéciaux devant les variables globales. On notera également le `&` devant les noms de procédures (ils sont facultatifs). Il y a deux utilisations de la construction `A ? B : C` qui rend B si A est vrai et C si A est faux. Il y a une commande `join`, elle permet de concaténer des chaines avec un séparateur. La cinquième ligne est un peu

spéciale : elle est de la forme `local(x) = y if z;`. Il faut l'interpréter comme suit : on définit une liste de variables locales (la liste contient un seul élément). La valeur de la n-ième variable est la valeur du n-ième élément de la liste qui se trouve derrière le signe égal. Or, derrière le signe égal, il y a un test : si z est vrai, alors y. Ce que fait Perl, c'est d'évaluer le test. S'il est faux, le résultat sera la liste vide. S'il est vrai, comme y n'est pas une liste, il sera transformé en une liste de longueur 1. Finalement, dans le cas où la liste n'est pas assez longue, elle est étendue avec des chaines vides.

Comme on le voit, Perl est un langage très puissant, très dense, et il est très facile de faire des erreurs ; si ces erreurs sont obtenues par évaluation de procédures construites au vol, le debug peut être assez compliqué.

Quelques notes sur la complexité. Il y a deux façons différentes de remplacer les A (suivis d'espaces optionnels) par des B (suivi d'un espace unique), à savoir

```
s/A\s*/B /g;      1 while s/A\s*/B /;
```

Si on savait a priori que chaque A est suivi exactement d'un espace, on pourrait utiliser `s/A/B/g`, ou la procédure `tr`. Dans ce dernier cas, on connait la complexité : à la fois en temps et en espace, elle est proportionnelle à la longueur T de la chaine. Dans le cas qui nous intéresse, on va supposer que T est la taille initiale et très peu différent de la taille finale. La complexité en temps de la première substitution est T, car chaque caractère est examiné une seule fois. S'il y a x caractères A, et qu'il sont plus ou moins uniformément répartis, la complexité en temps de la boucle est essentiellement en xT.

La complexité en espace est plus difficile à estimer. On va faire l'hypothèse suivante : dans le cas d'une substitution unique, la complexité (la mémoire allouée) est égale à la taille du résultat. Dans le cas de la boucle, cela donnerait donc xT. Dans le cas d'une substitution multiple, on va supposer que Perl met les résultats intermédiaires dans un tampon, et qu'il double la taille de ce tampon si nécessaire. À la fin, on aura un tampon de taille y avec $y/2 \leq \mathrm{T} \leq y$, et au pire y octets ont été alloués en plus pour les autres tampons. Ceci donne une taille de au plus 4T. Comme il faut recopier le tampon dans le résultat, ceci nous donne 5T. Finalement, on peut supposer que dans le cas d'une substitution multiple la taille mémoire allouée est proportionnelle au résultat final, indépendante du nombre de substitutions.

En conclusion : il vaut mieux faire une substitution globale qu'une boucle, on y gagne à la fois en temps et en espace.

Considérons maintenant le problème suivant : on a des A suivis de B, et on veut faire quelque chose avec les caractères situés entre les A et les B. On peut utiliser une procédure de la forme

```
sub toto{
   my $res = ""; # $ emacs
   while(/A/) {
     $res .= $`; $_ = $';
     /B/; $_ = $'; $res .= fct($');
   }
   $res . $_;
}
```

En règle générale, au lieu d'avoir des lettres A et B, on a des quantités un peu plus compliquées. Le pattern B peut dépendre de A de façon compliquée (il suffit de mettre un `if` dans le code). Dans le cas où B se déduit facilement de A, on peut utiliser la procédure suivante, qui fait le même travail :

```
sub titi{
  s/A(.*?)B/ fct($1)/ eg;
}
```

Notons d'abord que la complexité temporelle des deux procédures est la même : chaque caractère du code est analysé une seule fois. La complexité spatiale de `titi` est essentiellement proportionnelle à la taille T du source. Dans le cas de `toto`, pour chacune des deux lignes de la boucle, on a deux variables `$res` et `$'`, dont la longueur totale est de l'ordre de T. Ainsi la complexité spatiale de `toto` est de l'ordre de xT, où x est le nombre d'occurrences de A.

Supposons maintenant que la chaine B soit constante. Une application est la suivante : on veut traiter `\begin{citation}` et `\end{citation}`. Une solution est la suivante : on utilise `split` pour découper le texte en morceaux, B étant le délimiteur (ce qui suit ne marche que si le texte se termine par un B, dans le cas contraire, il faut faire un peu attention). Pour chaque morceau, on cherche le A, puis on applique la procédure `fct`. La complexité en espace de la procédure est maintenant 2T. Le coût en espace du découpage est de l'ordre de T, il faut recoller les morceaux, ce qui coûte également T. À ce total de 4T, il faut ajouter l'espace utilisé par les cellules de liste : il en faut 2N, où N est le nombre d'occurrences de B. Si on suppose qu'une cellule coûte 8 octets, et que $16\mathrm{N} \leq \mathrm{T}$ (dans notre cas de figure, on a $30\mathrm{N} \leq \mathrm{T}$, car A et B font 30 caractères en tout), on obtient, au pire, 5T. La morale est la suivante : l'utilisation de `split` a en gros le même coût que la procédure `titi`.

Supposons maintenant que l'objectif soit de déplacer les C qui se trouvent entre A et B, et les mettre derrière B. Pour simplifier, on va supposer qu'il n'y a qu'un seul C entre un A et un B (s'il y en a plusieurs, la meilleure stratégie consiste à utiliser une procédure de type `toto`). On peut faire simplement :

```
s/A(.*)C(.*)B/A$1$2BC/g;      # (S1)
```

En fait, ce code est erroné, il faut utiliser plutôt

```
s/A(.*?)C(.*?)B/A$1$2BC/g;      # (S2)
```

Notons que (S2) ne marche que s'il y a un C entre chaque A et B : dans le cas de ABCACB, le résultat est ABACBC. Ce que l'on cherche à faire, en fait, c'est de remplacer `\caption{...\label{...}...}` par `\caption{...}\label{...}`. On suppose que des procédures du type `remove_braces` et `remove_env` ont été appliquées. Ainsi, la vraie substitution est

```
s/\\7(\d+!caption!)(.*)(\\4\d+!label!(.*)\\8\1/.../g;      # (S3)
```

(le lecteur remplira les points de suspension). Cette procédure réalise l'objectif, mais est une catastrophe du point de vue temps de calcul. Pour comprendre pourquoi, prenons un exemple plus simple

```
s/\\7(\d+!caption!)(.*)\\8\1/.../g;      # (S4)
```

Il s'agit d'une variante de `titi`, avec `(.*?)` remplacé par `(.*)`. La différence essentielle est que, dans un cas, on cherche la chaine la plus longue (contenant n'importe quoi), et dans l'autre cas, la chaine la plus courte. On fait l'hypothèse que `\\8\1` n'apparait qu'une seule fois dans le texte. Ainsi chercher cette chaine de gauche à droite, ou la chercher de droite à gauche revient au même. Du point de vue du temps de calcul, pour comparer les deux méthodes, il faut savoir si la chaine est plutôt à gauche, ou plutôt à droite. Dans notre cas, les `\caption` ne sont pas emboîtés (s'ils l'étaient, il faudrait une procédure récursive), la fin du `\caption` est donc plutôt à gauche. Pour fixer les idées, supposons qu'il y ait a `\caption`, de longueur moyenne b, et que le texte ait une longueur totale c. On va supposer également qu'il y a d `\label` en dehors des captions.

Si la recherche des fins de `\caption` se fait de gauche à droite, le temps de recherche est ab. Le temps de recherche des `\caption` est de l'ordre de $c - ab$ (rappelons que Perl ne cherche pas dans la partie déjà cherchée, ou le résultat de la substitution). En d'autres termes, le temps total est c, indépendant du nombre de `\caption`. Si la recherche se fait de droite à gauche, la situation est différente. Supposons les `\caption` uniformément répartis. Le i-ième sera en position ic/a, sa fin en position $ic/a + b$, donc $c - ic/a - b$, en comptant à partir de la fin, et cette quantité est le temps de recherche. Il faut sommer sur tous les i, ce qui va donner $ac/2 - b$. Si on ajoute le temps de recherche des `\caption`, on obtient $ac/2 - 2ab + c$. Essentiellement, le temps de recherche est le produit du nombre de `\caption` par la longueur totale du texte.

On peut maintenant expliquer ce qui se passe dans le cas de (S3). Considérons le `\caption` numéro i. Il est placé en position ic/a. Le texte qui suit a une longueur $c(a - i)/a$. Il y a donc $d(a - i)/a$ labels derrière lui qui sont en dehors de la `\caption`. Pour chacun de ces labels, on cherche la fin du `\caption` (l'ordre de recherche n'a pas d'importance, on ne le trouvera pas). S'il s'agit du k-ième label en partant de la fin, le coût de recherche est kc/d. S'il y a j labels à traiter, en sommant, on obtient $j^2x/(2d)$, donc $(a - i)^2/(2a^2)$. En sommant en i, on obtient $acd/6$. En d'autres termes, le coût est le produit de la longueur du texte, par le nombre de `\caption` et par le nombre de `\label` en dehors des `\caption`. N'importe quel autre algorithme est préférable.

On peut maintenant se poser la question de savoir si les autres procédures sont optimales. Considérons d'abord le cas de `remove_braces`. Le pattern ressemble à (S4) : on cherche une accolade ouvrante, un texte de longueur maximale sans accolades et une accolade fermante. Note : la recherche va toujours se faire de gauche à droite, remplacer maximale par minimale ne change rien. Supposons d'abord qu'il n'y a pas d'accolades. La complexité est T en temps, et 0 en espace. Supposons maintenant qu'il n'y a pas d'accolades imbriquées. Le temps de traitement est le même, sauf que le texte entre accolades doit être copié deux fois (une fois dans `$1`, une autre dans `\\3$id!$1\\4$id!`). En gros c'est au pire 2T. Comme on a fait l'hypothèse que la substitution globale coûte 5T, ceci donnera 7T. Considérons finalement le cas de `{a{b{c}d}e}` : trois niveaux d'accolades. Quand Perl voit l'accolade devant le a, il essaie de traiter celle-ci. Il va scanner a, puis voir l'accolade ouvrante, ce qui est mauvais. Il va donc backtracker, et considérer l'accolade suivante. Dans le pire des cas, il va scanner la quantité a deux fois (s'il est intelligent, il risque même de se rendre compte que l'accolade suivante est à la position courante). La même chose se passe pour b. Dans tous les cas, le texte est scanné au pire deux fois. En conclusion : s'il y a N niveaux de d'accolades, le temps total de la procédure est au pire 2NT, et la mémoire allouée 7NT. Comme N est relativement faible, on ne peut pas espérer beaucoup mieux.

En ce qui concerne, `remove_env`, il est clair que la première substitution devrait contenir `(.*?)` au lieu de `(.*)`. La deuxième substitution est de la forme `\\5X\?X\\6\1\?`, où X matche n'importe quoi (de longueur maximale) qui ne contient pas de point d'interrogation. La recherche se fait de gauche à droite car X doit être suivi d'un point d'interrogation (directement dans le premier cas, avec `\\6\1` comme séparateur sinon). Ainsi, la procédure `remove_env` a essentiellement la même complexité que `remove_braces`. Elle est en fait plus lente, à cause du `\1`. En effet, supposons que `\1` soit `toto`. Comment fait Perl pour trouver la chaine 'totox' dans un texte ? il utilise un automate à état fini (dans ce cas l'automate a 6 états, chaque état est défini par une sous-chaine initiale de 'totox'. Si on est dans l'état où on a vu 'toto', et que le caractère suivant est un t, on passe dans l'état 'tot'. De même, si on veut chercher toto ou titi, on utilise un automate à 7 états. Avec cet automate, chaque caractère est examiné une seule fois. Mais

je ne pense pas que que Perl crée un automate pour \1 (i.e. pour chaque fin d'environnement). Ainsi la recherche est plus coûteuse (elle dépend de la longueur des noms d'environnements).

1.4 Le langage XML

Le langage Perl est, comme vu plus haut, particulièrement difficile à interpréter : si l'on voit `$x`, cela peut être la variable `$x`, ou la liste `@x` ou le tableau `%x`. Les choses sont plus simples dans TEX, on sait distinguer les commandes et le texte, mais ce n'est pas toujours facile de trouver les arguments d'une commande.

Le langage XML (eXtended Markup Language) ne souffre pas de ces problèmes : il est particulièrement facile à parser, mais ce n'est pas un langage de programmation, juste un langage de description. Prenons l'exemple suivant

```
<math>1=2</math> est <emph> une formule <color val="red"> fausse </color>.</emph><p/>
```

Les termes entre crochets, du type `<math>` sont appelées des balises. `<math>` est une balise ouvrante, `</math>` une balise fermante, `<p/>` est une balise vide (ouvrante et fermante). Dans le cas de `val="red"`, on parle d'attributs, avec un nom et une valeur. Un élément est tout ce qui est compris entre une balise ouvrante et la balise fermante associée. Un document XML est dit bien formé, si à toute balise ouvrante est associée une balise fermante, et ceci pour le document global, et pour le contenu de chaque élément (il y a d'autres règles que nous ne spécifierons pas ici).

On peut associer une DTD à un document XML. Cette DTD explique quels sont les attributs (noms et valeurs) autorisés pour chaque élément, et donne des contraintes pour le contenu de chaque élément. Si les règles sont vérifiées, on dira que le document est valide.

1.4.1 DTD locale

Soit l'exemple suivant, un extrait de la référence [1]

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE TEI.2 SYSTEM "teixlite.dtd" [
3     <!ENTITY dash "&#x2010;">
4     <!ENTITY oelig "&#x0153;">
5     <!ENTITY Galantes SYSTEM "verlainefg.xml">
6  ]>
7  <TEI.2>
8  ...
9  &Galantes;
10 </body>
11 </text>
12 </TEI.2>
```

Il s'agit d'un document XML bien formé et valide, à condition de remplacer les ... par la partie omise, et à condition que les deux fichiers externes référencés existent bien.

Dans cet exemple, on utilise `&Galantes;` et `&#x153;`. Dans les deux cas, il s'agit d'une entité, quelque chose qui commence par `&` et se termine par un point-virgule, qui sera remplacé par sa valeur. Dans le second cas, il s'agit d'une caractère Unicode, le caractère numéro 339. Les lignes 3, 4 et 5 utilisent une balise spéciale `<!ENTITY>` pour définir une entité. À la ligne 4, on dit que

`&oelig;` est `&#x153;`, donc `\oe`. À la ligne suivante, on dit que `&Galantes;` est le contenu d'un fichier externe.

À la ligne 2, on utilise une balise spéciale `<!DOCTYPE>`, elle définit la DTD ; en l'occurrence, il s'agit d'une DTD qui se trouve dans un fichier externe, et qui est localement étendue par la partie entre crochets (ici, on ajoute trois entités). On dit également que le document contient un élément racine, à savoir `<TEI.2>`.

Finalement la première ligne du fichier utilise une balise spéciale `<?xml?>`, avec une syntaxe régulière. Elle dit que le document est un document XML, conforme à la norme XML version 1.0, est que le fichier est encodé en iso-latin1 (rappelons que l'encodage XML est Unicode, donc 16bits, ce qui oblige à faire une déclaration pour tous les textes non codés en ASCII 7 bits).

1.4.2 DTD globale

Voici un extrait d'une DTD :

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!ENTITY lt  "&#x26;#x3C;" >
3
4  <!ENTITY % mathml PUBLIC "mathml" "mathml2.dtd">
5  %mathml;
6
7  <!ENTITY  ier "<hi rend='sup'>er</hi>">
8
9  <!ELEMENT raweb (accueil, (nocitestar|nocite)*,
10    (moreinfo? , composition, presentation,
11    fondements?, domaine?, logiciels?, resultats,contrats?,international?,
12    diffusion?, biblio)) >
13
14 <!ENTITY % bibliostuff "bnote|bauteurs|beditеur|btitle|borganization|
15      bschool|byear|bmonth|url|bseries|bnumber|bvolume|bedition|
16      binstitution|baddress|bpages|bhowpublished|bbooktitle
17      |bpublisher|bjournal|bchapter|btype">
18
19 <!ELEMENT citation (%bibliostuff;)*>
20 <!ATTLIST citation key CDATA #IMPLIED
21                   id ID #REQUIRED
22                   type CDATA "truc"
23                   toto CDATA #FIXED "toto" >
24
25 <!ELEMENT cite EMPTY>
26 <!ATTLIST cite ref IDREF #REQUIRED>
27
28 <!ELEMENT bnote (#PCDATA|xref|hi)*>
```

Par rapport à un fichier XML standard, il y a trois balises supplémentaires `<!ELEMENT>`, `<!ATTLIST>` et `<!ENTITY % toto>`. Ce dernier type de balises est utilisé aux lignes 4 et 14 ; on définit une entité `%toto;` qui s'utilise comme `&toto;`, mais dont la portée est limitée à la DTD. À la ligne 4, on définit `%mathml;`, et on l'utilise en ligne 5. Ces deux lignes demandent simplement l'inclusion du fichier `mathml2.dtd` (qui peut inclure d'autres fichiers). La définition de la ligne 2 est un peu

spéciale : `<` équivaut à `&#x26;#x3C;` Ceci sera évalué une première fois en `&#x3C;`, puis une seconde fois, ce qui donne le caractère <.

À la ligne 7, on définit une entité `&ier;`, dont la valeur est `<hi rend='sup'>er</hi>`. Rappelons qu'en TEX, la bonne façon de dire 1er est `1\ier`, en XML on pourra dire `1&ier;`.

La ligne 9 définit l'élément `raweb`, qui est de la forme (A, B, C). Ceci signifie que l'élément `<raweb>`, pour être valide doit contenir un A, suivi d'un B, suivi d'un C. Ici A est `<accueil>`, B est de la forme $(u|v)*$, donc un u ou un v, en nombre arbitraire. De même C est de la forme $(x, y, z, ...)$, un C sera donc un x, suivi d'un y, d'un z, etc. Les éléments suivis d'un point d'interrogation sont facultatifs. Ainsi un document valide aura la forme suivante : `<accueil>`, des `<nocite>` ou `<nocitestar>`, un `<moreinfo>` optionnel, un `<composition>`, un `<presentation>`, un `<fondements>` optionnel, ..., un `<diffusion>` optionnel, et finalement un `<biblio>`.

À la ligne 19, on dit qu'un élément `<citation>` est une suite quelconque (dans n'importe quel ordre) de `<bnote>`, `<bauteur>`, etc. À la ligne 28, on dit qu'un élément `<bnote>` est formé d'éléments `<xref>`, `<hi>` avec du texte. Par exemple

```
<bnote>1&iere; édition disponible sur <xref url='http:www.inria.fr/'></bnote>
```

est valide. Finalement, à la ligne 25, on dit que l'élément `<cite>` est vide.

Le reste de la DTD spécifie les attributs des éléments. Par exemple `<citation>` a 4 attributs, `key`, `id`, `type` et `toto`. Trois de ces attributs peuvent avoir une valeur quelconque (`CDATA`), l'attribut `id` doit être un identificateur (`ID`), ce qui impose quelques restrictions : des chiffres, lettres, tirets sont OK, mais les espaces, signes +, etc., sont interdits, de même qu'un chiffre initial. Il y a une autre restriction : l'identificateur doit être unique. À la ligne 26, on dit que l'élément `<cite>` a un attribut `ref` de type `IDREF`, ce qui signifie que la valeur doit être une identificateur (si la valeur est `toto`, il doit exister un élément qui a un attribut de type `ID` dont la valeur est `toto`. En toute logique, c'est élément est un `<citation>`, mais cette contrainte n'est pas testée par XML).

Certains attributs sont marqués `#REQUIRED` : la valeur est obligatoire, d'autres sont marqués `#FIXED` : la valeur de l'attribut `toto` de `<citation>` peut être omise, si elle ne l'est pas, elle doit nécessairement être toto. Dans le cas de `type`, il y a une valeur par défaut, dans le cas de `key`, il n'y a pas de valeur par défaut. On peut aussi énumérer l'ensemble des valeurs possibles.

1.4.3 Introduction à XSL

Reprenons l'exemple donné plus haut, en lui rajoutant une DTD locale :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE test [
   <!ELEMENT math (#PCDATA) >
   <!ELEMENT emph (#PCDATA|color|emph)* >
   <!ELEMENT color #PCDATA >
   <!ELEMENT p EMPTY>
   <!ELEMENT test (#PCDATA|math|emph|p)* >
   <!ATTLIST color val CDATA #IMPLIED>
]>
<test>
  <math>1=2</math> est
    <emph> une formule <color val="red"> fausse </color>.</emph><p/>
</test>
```

Supposons qu'on veuille transformer ce texte en un texte équivalent avec
- l'élément `<math>` est inchangé,
- il n'y a pas de paragraphe,
- au lieu de `<color>`, il y `<textred>` et `<textblue>`,
- au lieu de `<emph>` il y a `<rm>` et `<it>`.

Ces transformations peuvent être faites via un processeur XSLT. Ce processeur applique des règles écrites en XML. Les deux premières transformations sont exécutées par le code suivant :

```
<xsl:template match="math">
  <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="p">
</xsl:template>
```

La deuxième règle dit : pour tous les éléments `<p>`, les remplacer par rien du tout (ou par un espace ? ceci n'est pas clair). La première règle dit : pour chaque élément `<math>`, le remplacer par l'évaluation de `<xsl:copy>`, qui rend une copie du nœud sélectionné. Il y a un mécanisme, XPath, qui permet de sélectionner un arbre, un attribut, etc. Le cas le plus simple est `"."`, c'est l'élément courant, `@toto`, c'est l'attribut toto, `note[3][@type='x']` choisit le troisième enfant du nœud courant, s'il est de type `note`, et s'il a un attribut `type` dont la valeur est `x`.

Dans le cas de la couleur, on va faire un test en fonction de l'attribut `val`. Il y a deux conditionnelles dans xsl : les tests simples, par `<xsl:if>` (si la condition est vraie, alors faire telle action), ou `<xsl:choose>`, dans laquelle on peut mettre des `<xsl:when>`, qui sont testés à tour de rôle, et une valeur par défaut (optionnelle). Comme le montre l'exemple, toutes les balises non reconnues par XSLT sont laissées dans le texte. On appelle `<xsl:apply-templates>`, qui transforme l'arbre (comme on n'a pas dit lequel, il s'agit du contenu de l'élément courant).

```
<xsl:template match="color">
  <xsl:choose>
    <xsl:when test="@val='red'">
      <textred> <xsl:apply-templates/></textred>
    </xsl:when>
    <xsl:when test="@val='blue' or @val='bleu'">
      <textblue> <xsl:apply-templates/></textblue>
    </xsl:when>
    <xsl:otherwise>
       <xsl:apply-templates/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

La dernière transformation est plus compliquée : il s'agit de convertir les `<emph>` en `<rm>` ou `<it>`. On supposera que le mode par défaut est `<rm>`, et `<emph>` échange `<it>` et `<rm>`. Le code utilisé ici va appeler explicitement une procédure, qui va calculer la police à utiliser. Il s'agit d'un nom, que l'on convertit en balise via `<xsl:element>`. Pour le fun, on montre comment ajouter un attribut.

```
<xsl:remplate match="emph">
   <xsl:variable name="mode">
     <xsl:call-template name="find_mode"/>
   </xsl:variable>
```

```
   <xsl:element name="$mode">
      <xsl:attribute name='automatically-generated'>yes</xsl:attribute>
      <xsl:apply-templates/>
   </xsl:element>
</xsl:template>
```

La procédure suivante va calculer le mode. On pourrait regarder la parité du nombre d'ancêtres de type `<emph>` de l'élément courant. Nous allons procéder de manière plus compliquée. D'abord, on va utiliser un paramètre (dans le cas où la procédure est appelée sans argument, la valeur sera celle de la déclaration). S'il n'y a pas d'ancêtre `emph`, on rend la valeur du paramètre (via `<xsl:value-of>`). Dans le cas contraire, on introduit une variable locale `aux` qui va contenir le contraire de `result`, et on va appeler récursivement la fonction, avec comme valeur du paramètre cette valeur `aux`, et comme point d'application l'ancêtre.

```
<xsl:template name="find_mode">
  <xsl:param name="result">it</xsl:param>
  <xsl:choose>
    <xsl:when test="ancestor::emph">
      <xsl:variable name="aux">
        <xsl:choose>
           <xsl:when test="$result= 'it'>rm</xsl:when>
           <xsl:otherwise>it</xsl:otherwise>
        </xsl:choose>
      </xsl:variable>
      <xsl:call-template name="find_mode" select="ancestor::emph">
        <xsl:with-param name="result" select="$aux"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$result"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Notons que la syntaxe XSL est régulière : c'est du XML pur, tandis que la partie XPath (les attributs comme `test`, `select`, etc) a une syntaxe spéciale.

1.4.4 Formattage

Notre objectif principal est de traduire un source LaTeX en XML. Par ailleurs, on a du source XML, produit soit par notre traducteur, soit par d'autres moyens. Ce source doit être traduit en HTML et en Pdf. La première conversion se fait aisément via une feuille de style XSLT. On suppose que le document est un ensemble de modules, chaque module va donner une page Web. La seule difficulté est de produire une table des matières, et de chainer proprement les pages WEB.

Pour simplifier le travail, on suppose que chaque module a un attribut `html` dont la valeur est le nom de la page HTML. Dans XSL, il est facile de trouver le module précédent et le module suivant, puis d'extraire l'attribut `html`. Si les modules sont regroupés par sections, la recherche se complique un peu, mais ne pose pas de problème particulier. Pour avoir une table des matières, il faut formater deux fois les titres des modules.

Le problème se complique pas mal si on veut générer du Pdf. En effet, on ne peut plus dire qu'un module fait une page : certains modules sont plus longs, d'autres moins longs. Par ailleurs on veut une présentation correcte, c'est-à-dire décomposer le texte en lignes, travail qui est fait par le navigateur dans le cas de HTML. On se propose ici d'utiliser TEX. Une solution envisageable consiste à expliquer à TEX le fonctionnement de toutes les balises. Par exemple, une balise `<module>` deviendrait `\begin{module}`. On va utiliser une autre technique : une feuille de style va convertir certaines balises, on va donc faire comme si TEX ne connaissant pas `\em`, mais uniquement `\rm` et `\it`, et demander au préprocesseur de convertir `<em>` en ce qu'il faut, voir plus haut.

Avec cette façon de faire, la feuille de style effectue un travail considérable, qui pourrait être fait par TEX ; autrement dit, toutes les commandes définies dans le fichier de classe `raweb.cls` (que l'utilisateur ne voit pas) vont être placées dans une feuille de style, interprétées, et le résultat donné à TEX. Par exemple, le fichier de classe spécifie l'indentation des paragraphes. Il y a 3 stratégies : toujours indenté, jamais indenté, ou indentation de tous sauf le premier paragraphe. Ce que va faire le formateur, c'est de calculer l'indentation de chaque paragraphe, et TEX va obéir les règles à la lettre.

De façon précise, on utilise XSL-FO. Une section pourrait être transformée en

```
<fo:block text-align='center' space-after='6pt' space-before='12pt'
   space-before.precedence='0' space-after.precedence='3'> title
</fo:block>
<fo:block text-indent='2cm' space-after='7pt' space-before.minimum='6pt'
   space-before.optimum='8pt'  space-before.maximum='10pt'
  text-align='justify'> texte
</fo:block>
```

Cet exemple montre comment gérer l'espacement entre un titre et le corps du texte, et entre les divers paragraphes. Comme il y a du jeu (on donne des valeurs minimales et maximales), et des priorités, le formateur doit s'en sortir sans trop de problèmes. Notons que le titre pourrait interdire les sauts de pages juste après lui. Si on veut une table des matières, il faut formater le titre avec des règles différentes (on centre rarement un titre dans une table des matières).

Typiquement, la table des matières va contenir

```
<fo:block text-align='justify'>
   <fo:simple-link internal-destination="N4">1. Chapitre Toto</fo:simple-link>
   <fo:leader leader-length.minimum='12pt' leader-length.optimum=40pt'
      leader-length.maximum='100%' leader-pattern='dots'/>
   <fo:page-number-citation ref-id='N4'/>
</fo:block>
<fo:block text-align='justify' start-indent='10mm'>
   <fo:simple-link internal-destination="N5">1.1. Section truc</fo:simple-link>
   <fo:leader leader-length.minimum='12pt' leader-length.optimum=40pt'
      leader-length.maximum='100%' leader-pattern='dots'/>
   <fo:page-number-citation ref-id='N5'/>
</fo:block>
```

Dans cet exemple nous avons mis 2 items, il y en a général nettement plus. La différence entre les deux items est que la section est indentée par rapport au chapitre. On aurait pu utiliser une police différente, un corps plus petit, etc. Entre le titre du chapitre et le numéro, il y a des points conducteurs. On demande que la longueur minimale soit de 12 points (éviter de coller le texte et le numéro). Le texte contient deux chaines de caractères N4 et N5. On suppose qu'il y a deux

labels ce nom, au début du chapitre et de la section. Ces chaines servent deux fois : une fois pour mettre dans la table des matières le numéro de page, et une fois pour mettre un lien hypertexte vers la page. Notons par contre que le numéro du chapitre et de la section sont codés en dur dans le document FO. Celui-ci est calculé par une procédure qui ressemble typiquement à

```
<xsl:number level='multiple' count='chapter|section' format='1.1'/>
```

Remarque : dans le raweb, les numéros des sections sont codés en dur : la section résultats nouveaux sera numérotée 6, même si c'est la première.

Les deux identificateurs N4 et N5 utilisés plus haut peuvent être générés automatiquement par XSL-FO, ou alors, on peut utiliser le fait que les auteurs mettent souvent un `\label` dans leur texte LaTeX après un début de chapitre ou de section. Le convertisseur LaTeX vers XML, défini dans ce document accroche le label au titre. Le code XML sera de la forme

```
<section> <head id='titi'>truc</head> ... </section>
```

Le formattage du titre de la section donnera donc

```
<fo:block id='titi'>Section 1.1 truc</fo:block>
```

Autrement dit, le titre de la section est formaté deux fois : une fois pour la table des matières (avec un attribut `internal-destination`), et une autre fois, dans le texte avec un attribut `id`, les valeurs doivent être les mêmes.

On peut également spécifier des formats de page : on peut dire qu'un début de chapitre doit commencer sur une belle page (impaire), quitte à mettre une page vide devant ; on peut dire que le format des pages paires et impaires est différent (par exemple, pour avoir une marge extérieure de 3cm et intérieure de 2cm, la valeur des marges de gauche et de droite dépend de la parité de la page). La procédure suivante dit que l'en-tête des pages impaires doit contenir un certain titre, et le numéro de la page :

```
<fo:static-content flow-name="xsl-region-before-right">
  <fo:block text-align="justify" font-size="{$bodySize}">
         <fo:inline> <xsl:value-of select="$PRID"/> </fo:inline>
         <fo:leader rule-thickness="0pt"/>
         <fo:inline> <fo:page-number/> </fo:inline>
  </fo:block>
</fo:static-content>
```

Un problème non trivial concerne la page de titre. Nous avons truandé, en ajoutant deux balises, une début, et une à la fin (le contenu de cette balise est le numéro du thème). L'interprétation par TeX de ces balises est la suivante : il faut placer le logo INRIA en tête du document, écrire « Institut national ...», placer le logo RA2001 en bas de page, le numéro du thème doit être placé juste au dessus du logo. Tout le reste de la page de titre doit être justifié verticalement dans l'espace qui reste. Le code est le suivant :

```
\def\foratheme#1{
  \vskip8cm \vfil
  \ra@atxy(70mm,174mm) {\hbox to 72mm{%
       \hrulefill\hspace{8mm}TH\'EME \uppercase{#1}%
            \hspace{8mm}\hrulefill}}}

\def\foinria{%
 \setbox0\hbox to 14cm{%
     \noindent\hskip3cm\hfill
     {\fontencoding{T1}\fontfamily{ptm}\fontseries{m}%
   \fontshape{n}\fontsize{10pt}{12pt}\selectfont
```

```
        INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE}%
      \hskip-5cm\hfill}%
  \null\vskip0.7cm \leavevmode\hskip-3.5cm\box0\null\vskip2cm\vfil}

\AtBeginDocument{
  \let\@texttop\@usemyatxy
  \ra@atxy(55mm,173mm){\includegraphics{LogoRA2001}}%
  \ra@atxy(7.8cm,2.5cm){\includegraphics[width=5.7cm]{Logo-INRIA-couleur}}%
}
```

Quelques explications : `\foratheme` fait un `\vskip8cm` (c'est la hauteur du logo) et un `\vfil` pour centrer verticalement. Le reste consiste à placer le thème (dans la police courante) au bon endroit. Ce que fait `\foinria`, c'est de placer « Institut national ...», avec la police qui va bien, dans une boite. On commence la page avec un espace vertical, la boite centrée horizontalement, un peu d'espace et un `\vfil`. Le reste concerne le placement des images.

1.4.5 Les mathématiques et XML

Il est possible en théorie d'écrire n'importe quelle formule de mathématiques avec un nombre très restreints de signes, d'après Bourbaki il suffit de τ, $\vee$, $\neg$ et $\square$, mais cela devient extrêmement lourd. Par exemple $\neg \vee \neg \vee \neg AB \neg \vee \neg BA$ s'abrège en $A \iff B$. On passe ainsi d'une notation préfixe non ambigüe à une notation mixte (l'opérateur peut être devant, derrière, au milieu, en plusieurs morceaux), qui nécessite des parenthèses pour être non-ambigüe. Parfois, il y a tellement de parenthèses que pour augmenter la lisibilité, on met d'autres signes à la place (crochets, accolades, etc.) même si ces signes ont une signification différente. Dans une formule de la forme

$$\forall x, y \in \mathbb{R} \qquad (x+y)^2 = x^2 + 2xy + y^2,$$

il faudrait mettre deux quantificateurs ($\forall x \in \mathbb{R}$ et $\forall y \in \mathbb{R}$), il y a un opérateur de multiplication implicite dans le cas de $2xy$, et un opérateur d'exponentiation implicite dans le cas de x^2. Modulo ces détails, tout le monde peut comprendre la formule. Dans certains cas, la notation x' désigne la dérivée temporelle de la fonction $x(t)$, ou la dérivée formelle de x (si on travaille dans une algèbre différentielle), ou une expression qui n'a rien à voir avec x ; l'interprétation dépend donc du contexte.

Avant que les espaces vectoriels ne soient considérés comme des structures simples, les gens mettaient des flèches au dessus des vecteurs, comme $\vec{X}$, par analogie avec la notation $\overrightarrow{AB}$, qui est le vecteur défini par les deux points A et B. Cette flèche ne veut rien dire, c'est juste un signe qui dit : attention, ce n'est pas un objet comme les autres.

Il y a une grande variété de signes possibles. Par exemple $\leq, \leqq, \lesssim, \lessdot, \precsim, \leqslant, \lessapprox, \lll$ sont des variantes de « plus petit que », $\twoheadrightarrow, \rightarrowtail, \rightrightarrows, \rightarrow, \Rightarrow, \mapsto, \hookrightarrow$ sont des variantes de flèche vers la droite. En ce qui concerne les lettres, on les tire d'alphabets divers et variés : grec comme α, Γ, $\digamma$, hébreu comme $\aleph$, $\beth$ et $\gimel$, gothique comme $\wp$. Autres symboles : $\varkappa$, ϖ, ℓ, $\hbar$, $\hbar$. Tous les symboles définis plus haut ont un nom et numéro Unicode, par exemple le caractère 2AAE « equals sign with bumpy above » ressemble à $\eqsim$, le caractère 2252 « approximately equal to or the image of » est $\fallingdotseq$, le caractère 211C « black-letter capital r » est $\Re$, etc. À partir du moment où les caractères sont dans Unicode, on peut les inclure dans un document XML sans problème.

Il y a une norme, MathML, qui permet d'inclure des mathématiques dans un document XML, sous deux formes.

```
<mrow>
  <msup>
    <mfenced>
      <mrow>
        <mi>a</mi>
        <mo>+</mo>
        <mi>b</mi>
      </mrow>
    </mfenced>
    <mn>2</mn>
  </msup>
</mrow>
```

et

```
<mrow>
  <apply>
    <power/>
    <apply>
      <plus/>
      <ci>a</ci>
      <ci>b</ci>
    </apply>
    <cn>2</cn>
  </apply>
</mrow>
```

Ces deux exemples sont tirés de la page Web de présentation de MathML. Ils représentent tous deux la formule $(a+b)^2$. Dans le deuxième cas (*content markup*), on dit qu'on applique la fonction puissance à deux arguments, le premier étant la somme de deux termes a et b (déclarés variables), le second argument étant le nombre deux. Le logiciel qui interprète la formule va convertir l'opérateur plus en signe +, et ajouter les parenthèses s'il le faut. Un logiciel de calcul formel pour générer très facilement ce code MathML à partir d'une expression de la forme `power(a+b,2)`. Dans le cas de la formule TEX `f(x+y)` on a un problème : s'agit-il de l'application de la fonction f à la somme $x+y$, ou s'agit-il d'une multiplication implicite ? C'est pour cette raison que notre traduction génère du code de la première forme (*presentation markup*). Dans l'exemple on a deux objets $(a+b)$ et 2, l'un est placé un peu plus haut que l'autre. L'objet $(a+b)$ n'est rien d'autre que $a+b$, avec des délimiteurs autour (ces délimiteurs sont définis comme attributs de `<mfenced>`, la valeur par défaut étant les parenthèses). On remarquera que a et b sont dans des `<mi>` dans un cas dans des `<ci>` dans l'autre, alors que le nombre 2 est dans un `<mn>` ou un `<cn>`. Le signe d'addition est donné dans un cas sous la forme `<plus/>`, dans l'autre cas sous la forme `<mo>+</mo>`.

D'après la documentation MathML, un élément `<mn>` est un nombre, avec les exemples suivants :

```
<mn> 2 </mn>
<mn> 0.123 </mn>
<mn> 1,000,000 </mn>
<mn> 2.1e10 </mn>
<mn> 0xFFEF </mn>
<mn> MCMLXIX </mn>
<mn> twenty one </mn>
```

Quand notre traducteur voit $f(1,234)$ que doit-il faire ? considérer qu'il s'agit de la fonction f appliqué à 1 virgule quelque chose, ou f appliqué au nombre mille et quelque ? ou f appliqué appliqué à deux arguments, le premier étant 1 ? Le lecteur attentif verra que TEX a choisi la dernière solution, il y a un espace après la virgule. En français, on dirait 1,5 pour 3/2. Certains logiciels de francisation ne mettent pas d'espace après la virgule par défaut, rien que pour ce cas. Cependant, dans les textes scientifiques, la notation universelle est : mettre un point pour séparer la partie entière de la partie décimale, et ne pas mettre de virgule. Il est à noter que `\hskip 0,3cm` est accepté par TEX, mais non reconnu par `latex2html`.

L'élément `<mi>` est censé contenir un identificateur, par exemple :

```
<mi> x </mi>
<mi> D </mi>
<mi> sin </mi>
<mi mathvariant='script'> L </mi>
<mi></mi>
```

Notre traducteur ne génère jamais de `<mi>` vide (en fonction de quoi ?), ni d'éléments du type sin. Une tradition fort ancienne (et qui a tendance à se perdre) veut que les lettres majuscules soient écrites en caractères droits, les lettres minuscules en lettres penchées (italique). Il y a bien sûr des exceptions : le L script ($\mathcal{L}$) est très différent du L normal. Un certaine nombre de fonctions (du type sinus) ont une abréviation standard (comme sin), et cette abréviation s'écrit en lettres droites ; certaines abréviations comme Im, Ker, Hom, etc., commencent par une lettre majuscule. On écrira, en TEX, $\sin x$ ou $\sin(x)$ (notez la gestion des espaces). Notre traducteur convertit `\sin` en un `<mo>` plutôt qu'un en `<mi>`.

Tout ce qui n'est pas un `<mi>` ni un `<mn>` est un `<mo>`. Il y a quatre utilisations : opérateur, clôture, séparateur ou accent. Le terme de clôture est une traduction maladroite de l'anglais *fence*. Dans le premier exemple, il y a un élément `<mfenced>`. Le contenu de cet élément est identique à :

```
<mrow> <mo>(</mo> <mi>a</mi> <mo>+</mo> <mi>b</mi> <mo>)</mo> </mrow>
```

Il se passe que le traducteur MathML vers Pdf (décrit dans la section suivante) n'interprète pas les deux expressions de la même manière ; ceci nous cause des soucis, comme expliqué dans le chapitre 3. Néanmoins, les attributs `open`, `close` et `separators` de `<mfenced>` sont convertis en `<mo>`. Dans l'expression $(a+b)$, les parenthèses sont des clôtures, le plus est un séparateur. Dans le cas de $\hat{x}$, le chapeau est un accent. Un opérateur peut avoir plein de propriétés, par exemple sa taille peut varier en fonction du contexte, le placement des indices en dépend, l'espacement entre les objets dépend du type d'opérateur, etc. L'élément `<mrow>` sert à grouper des termes. Par exemple, dans le cas de x_{a+b}, il faut mettre l'indice $a+b$ dans un `<mrow>`.

Il y a dans MathML des éléments `<mover>`, `<munder>` et `<munderover>` qui permettent d'aligner verticalement deux ou trois objets, des éléments `<msub>`, `<msup>` et `<msubsup/>` qui permettent de mettre des indices et des exposants, `<mfrac>`, `<mroot>`, `<msqrt>` qui permettent de mettre des fractions ou des racines. Il y a `<mmultiscripts>`, qui permet de mettre des exposants ou indices à gauche (il n'y a pas l'équivalent en TEX de cette construction). Il est possible de mettre du texte dans des mathématiques, via l'élément `<mtext>`.

Il y a un élément `<mstyle>` que l'on n'utilisera pas. Essentiellement cet éléments a des attributs qui sont visibles (hérités) par tous les descendants. Supposons par exemple qu'on ait une parenthèse dans un `<mrow>` (ou un `<mfenced>`). La taille de la parenthèse est la taille de

l'expression complète, sauf si on demande explicitement (i.e. via un attribut de la parenthèse ou du `<mstyle>`) une taille maximale. Ce mécanisme ne marche pas dans `xmltex`.

Il y a un élément `<mpadded>`, exemple :

```
<mrow>
  <mpadded width="0"> <mi> R </mi> </mpadded>
  <mspace width="0.3em"/>
  <mtext> | </mtext>
</mrow>
```

Le résultat est $\boldsymbol{R}$, quelque chose qui ressemble vaguement à $\mathbb{R}$, en beaucoup plus laid. Cette construction n'est pas implémentée dans la version actuelle de `xmltex`, et notre traducteur ne l'utilise pas. Il y a un élément `<mphantom>` (non reconnu par `xmltex`) qui permet de faire des fantômes. Nous n'avons pas eu l'occasion de rencontrer ce genre de construction dans le rapport d'activité (c'est une construction très prisée de `latex2html`).

Il y a moyen de faire des tables, plutôt qu'on long discours, on renverra le lecteur aux exemples donnés dans le chapitre 3.

1.5 TEX et XML

Nous avons vu en section 1.2.11 comment on peut définir une macro `\verb` qui fait des choses compliquées. On pourrait penser à rendre le caractère < actif, de sorte qu'il comprenne `<toto>`. Il faut également traiter les cas de `</toto>`, `<toto x='y' z="t">` et même `<toto/>`. Pour pouvoir parser un fichier XML, il faut en plus traiter les entités, de la forme `&lt;` ou `&#x01234;`. Tout ceci étant faisable, cela a été implémenté dans `xmltex` et PassiveTEX. Un exemple d'utilisation de cet outil est donné dans [1].

1.5.1 `xmltex`

Il s'agit d'un parseur XML, écrit en TEX, dont nous allons donner brièvement le fonctionnement. La lecture du source est intéressante, pour ceux qui comprennent les `\catcode`, `\expandafter`, etc. et le lambda-calcul. Supposons que l'on donne l'exemple 1.4.1 à `xmltex`. Que va-t-il faire ? Il faut d'abord interpréter la balise `<?xml?>`, car celle-ci indique l'encodage du document source. Il faut ensuite traiter la balise `<!DOCTYPE>`, car c'est elle qui dit quoi faire (c'est bien beau de parser, mais on veut du Pdf ou du dvi à la fin). La DTD ne sert pas vraiment, ce qui est important ce sont les actions associées aux balises. C'est pour cela qu'un fichier `xmt` est lu, le nom du fichier correspondant à la DTD. Il faut cependant parser la DTD locale, car l'élément `&oelig;` n'est pas connu à priori.

Le traitement du `&` est simple : il est suivi de certains caractères et d'un point virgule. Dans le cas de `&lt;`, on regarde la valeur d'une certaine macro (dont le nom dépend du nom de l'entité). Dans le cas de `&#12;` ou `&12;`, on convertit l'argument en un entier en base 10, puis en caractère Unicode, format UTF8, puis on regarde dans une table.

Le traitement de `<toto>` est, en gros, le suivant. D'abord `<toto/>` est considéré comme `<toto></toto>`, ce qui simplifie un peu les choses. Ensuite, quand on voit une balise ouvrante, on traite les attributs. Il y a une partie non triviale, concernant les espaces nominaux (par exemple `<xsl:if>` est `if` dans l'espace nominal, dont `xsl` est une abréviation, il peut y avoir

plusieurs abréviations différentes. Il faut normaliser cela. On appelle ensuite la commande TEX `\E:toto` ; on appelle `\E:/toto` quand on voit la balise fermante.

Le lien entre le parseur et l'évaluateur est un peu délicat : on ne peut pas tout parser, puis tout évaluer, cela prendrait trop de place en mémoire. Par ailleurs, certains environnements amsmath doivent voir explicitement la fin (si la fin de l'environnement est donné par `</toto>`, il faut d'abord remplacer la balise XML avant de pouvoir lancer la commande TEX qui traite la balise ouvrante).

1.5.2 Fichiers xmt

Pour produire un dvi ou un Pdf à partir d'un document XML, il faut définir une action `\E:toto` et `\E:/toto` pour chaque balise toto. Pour faciliter les définitions, une syntaxe spéciale est prévue. Les commandes sont à placer dans un fichier `titi.xmt`, où `titi` est le nom de la DTD.

Pour définir une entité comme `&amp` ou `&#174`, on dira

```
\XMLentity{amp}{\utfeightaz&}
\UnicodeCharacter{174}{\ifmmode \circledR \else \textregistered \fi}
```

On peut dire `\XMLelement{toto}{AL}{B}{E}` : ceci définit l'élément `<toto>`. Ici B est la commande à exécuter quand on voit `<toto>` et E quand on voit `</toto>`. L'argument `AL` est une liste de spécification d'attributs (voir plus loin). Si B est `\xmlgrab`, alors E reçoit le contenu de l'élément comme argument (dans les autres cas, B et E ne reçoivent pas d'argument). Voici un exemple tiré du traitement des mathématiques dans fotex. Comme dit plus haut, la fin de l'environnement `gather*` doit être explicite car amsmath fait deux passes sur le texte.

```
\XMLelement{fotex:displaymath}
   {}
   {\begin{displaymath}}
   {\end{displaymath}}

\XMLelement{fotex:eqnarray}
 {}
 {\xmlgrab}
 {\begin{gather*}#1\end{gather*}}
```

On peut dire `\XMLattribute{A}{B}{C}` dans la partie `AL` de `\XMLelement`. Dans le cas de `<toto A='18'>`, on peut récupérer la valeur de l'attribut `A` via la commande B. Si la valeur de l'attribut n'est pas spécifiée, la valeur de C sera utilisée à la place. On peut dire `\XMLnamespaceattribute{N}{A}{B}{C}`. Dans ce cas, pour chaque attribut de nom A dans le namespace N (quel que soit l'élément), la commande B permettra de récupérer la valeur de l'attribut.

On peut se poser la question la valeur de l'attribut couleur est-elle rouge ou rose pâle ? Nous avons vu plus haut que comparer des chaines de caractères dans TEX est non trivial. Ici le problème est pire, car on ne sait pas dans quel encodage est codé le source (il est transformé en UTF-8, mais on ne va pas traduire le â en UTF-8). Pour cette raison, on peut dire `\XMLname{A}{B}`. Ceci associe à la commande B le nom XML A. On peut aussi dire : `\XMLstring B<>A</>`, dans ce cas, `A` est une expression XML quelconque. Les conversions sont réalisées lors de cette déclaration. Pour mettre un â dans le texte, il suffit de dire `&ahat;`.

Exemple (extrait du fichier fo)

```
\XMLnamespaceattribute{fo}{active-state}{\FOactivestate}{}
\XMLname{fo:list-item-body}{\FOListItemBody}
```

Dans cet exemple, `\FOactivestate` va contenir la valeur de l'attribut `active-state` dans l'espace nominal `fo`, si l'élément courant en a un, et sera vide sinon (pour les curieux, cet attribut est un moyen permettant par exemple de changer la couleur des liens déjà visités). Si une macro désire savoir si son parent est `fo:list-item-body`, elle pourra comparer le nom de son parent avec la macro `\FOListItemBody` (comparer les caractères est risqué, à cause des problèmes d'encodages). Pour les curieux, une liste est une suite d'items, chaque item ayant un label et un corps, l'équivalent LaTeX du label est l'argument optionnel de `\item`, le corps est ce qui suit cet argument optionnel.

Autre exemple :

```
\XMLstring\jg@lt<>&lt;</>
\XMLstring\jg@gt<>&gt;</>
\XMLstring\att@PREFIX<>prefix</>
\XMLelement{m:mo}
  {\XMLattribute{form}{\XML@mathmlform}{inline}}
  {\xmlgrab}
  {\ifx\XML@mathmlform\att@PREFIX
    \mathop{{\operator@font #1}}%
   \else \def\jg@tck{#1}
     \ifx\notinover B\gdef\toto{#1}\else
     \ifx\jg@tck\jg@gt\string>\else
     \ifx\jg@tck\jg@lt\string<\else
     #1\fi\fi\fi
   \fi
   }
```

Il s'agit d'une variante du traitement des mathématiques. Il y a deux tests. Ici on dit : quand on voit `<m:mo>` (un `<mo>` dans l'espace nominal MathML) il faut récupérer l'élément complet. Il y a un attribut `form` optionnel, dont la valeur par défaut est `inline`. Si la valeur de l'attribut est `prefix`, l'élément sera précédé d'un `mathop`. Le résultat TeX, dans le cas de `<mo form=prefix>sin</mo>`, est le suivant : $\sin x$ et $\sin(x)$. Nous avons mis un argument, avec et sans parenthèses, pour montrer la gestion des espaces.

Il y a deux subtilités : pour une raison peu claire la traduction XML de `$a<b$` donne $a\langle b$, ce qui n'est pas beau. Ainsi un `&lt;` dans un `<mo>` sera traduit comme un vrai signe inférieur. La deuxième subtilité concerne le placement des accents. Pour placer l'accent x sur la lettre y, il faut que x soit un `<mo>`. Ce que l'on fait, c'est mettre dans une variable globale `\toto` le contenu de l'élément.

Finalement, on peut dire

```
\XMLelement{list}{
 \XMLattribute{type}{\listtype}{unordered}}
 {\csname List\listtype\endcsname}
 {\csname endList\listtype\endcsname}

\XMLelement{item}
 {}
 {\csname Item\listtype\endcsname}
 {}
```

La traduction TEX de

```
<list type='XX'> <item/> A <item/> B </list>
```

est

```
 \ListXX \ItemXX A \ItemXX B \endListXX
```

Si on définit les diverses commandes comme il faut, ceci devient, pour un XX bien choisi :

```
 \begin{itemize} \item A \item B \end{itemize}
```

Chapitre 2

Le rapport 2001

2.1 Introduction

Le rapport d'activité d'un projet est constitué d'un fichier source LaTeX, de trois fichiers de bibliographies, d'illustrations, et de deux fichiers LaTeX auxiliaires (pour les fiches-projet).

Le rapport est traduit en HTML et PostScript, et mis sur le web. Nous expliquerons d'abord la syntaxe des sources, parlerons d'outils de traduction. On mentionnera parfois une traduction possible en XML, celle-ci sera conforme à une DTD ad-hoc, la vraie DTD sera définie dans le chapitre suivant.

2.2 Les sources LaTeX

Ces fichiers LaTeX doivent obéir à des règles de syntaxe et de sémantique, telles que définies sur la page Web de l'Inria. Le texte qui suit est une adaptation de cette page qui montre la syntaxe, ce qui est le point important pour la traduction vers HTML ou XML ; nous ne parlerons de sémantique que lorsque cela est nécessaire.

Dans cette section, nous utiliserons les définitions suivantes
- *mot* : une suite de caractères (lettres, chiffres, tirets) ;
- *mot restreint* : un mot ne comportant pas de lettres accentuées (caractères ASCII uniquement) ;
- *mot étendu* : un mot pouvant contenir des caractères bizarres ;
- *ligne* : une suite de mots, ne comportant pas de commandes spécifiques raweb, mais pouvant contenir des mathématiques simples, et des appels de note ;
- *paragraphe* : une suite de lignes, pouvant contenir des mathématiques hors texte, des illustrations, etc, mais pas de commandes de sectionnement ;
- *texte général* : une suite de paragraphe, avec des commandes de sectionnement de niveau `\subsubsection` et inférieur, toujours sans commandes spécifiques raweb.

2.2.1 Fiches Projet

Il s'agit de deux documents, l'un en français, l'autre en anglais, ayant la même structure : trois paragraphes, introduits par `\subsubsection*`, chaque paragraphe peut contenir des environne-

ments `itemize`, on peut mettre des références HTML via `\href`. Toutes les autres commandes LaTeX qui ne génèrent pas du texte sont interdites.

2.2.2 Document principal

Le document principal est défini par

raweb = '\documentclass{ra2001}'
en-tête
'\begin{document}'
'\maketitle'
modules*
'\loadbiblio'
'\end{document}

L'en-tête du document est formée de deux parties. D'une part, une partie libre, qui ne contient pas de texte, mais uniquement des commandes de la forme `\usepackage[options]{package}` et des définitions de commande utilisateur. Il y a une partie imposée.

en-tête = partie libre
'\projet{' projet '}{' alt-abrégé '}{' nom-explicité '}'
'\theme{' thème '}'
'\typeprojet{' type '}'
'\localisation{' lieu '}'
projet = *mot-restreint* (en lettres capitales)
alt-abrégé = *mot-étendu* (peut être vide)
nom-explicité = *ligne*
thème = *mot-restreint* (un chiffre, une lettre)
type = *mot*
lieu = *mots*

Dans la version 2002, la sémantique de la localisation va changer. Il faudrait dire `\UR{xx}`, où `xx` est une, ou plusieurs macros, de la forme `\URSophia`.

2.2.3 Définition d'un module

Un module est donné par la définition suivante :

module = '\begin{module}{' section '}{' nommod '}{' titre '}'
interface
synopsis ?
corps ?
'\end{module}'
section = sec1 |sec2 |sec3 |sec4 |sec5 |sec6|sec7|sec8 |sec9
nommod = *mot-restreint*
titre = *phrase*

2.2.4 Interface d'un module

L'interface d'un module est définie par :

interface = (participants | kw | moreinfo | glossaire)*
moreinfo = '\begin{moreinfo}' *paragraphe* '\end{moreinfo}'
kw = '\begin{motscle}' *mot*+ '\end{motscle}'
glossaire = '\begin{glossaire}' glo+ '\end{glossaire}'
glo = '\glo{' *mots*+ '}{' *paragraphe* '}'
participants = '\begin {participants}' pers+ '\end{participant}'
pers = '\pers{' prénom '}' ('[' particule ']') ? '{' nom '}' ('[' info ']') ?
prénom = *mot*+
particule = *mot*+
nom = *mot*+
info = *ligne*

2.2.5 Synopsis et corps

synopsis = '\begin{abstract}' *texte général* '\end{abstract}'
corps = '\begin{body}' *texte général* '\end{body}'

Notons que `\moduleref [année]{projet}{section}{nommod}` permet de faire une référence à un module défini par les paramètres listés. L'année et le projet sont optionnels. Notons que `nommod` identifie à lui seul un module, le champ `section` est redondant. Si le nom du module est vide, ceci fait une référence à la section.

2.2.6 Première section

Cette section ne contient qu'un seul module, lequel ne contient ni corps ni résumé, mais
sec1mod = catperso+
catperso = '\begin{catperso }{' type '}' pers+ '\end {catperso}'
type = *mot*+

2.2.7 Bibliographie

Il y a trois fichiers de bibliographie, appelons-les A, B et C. La commande `\footcite` permet de faire des citations à des références dans le fichier A, la commande `\cite` aux deux autres fichiers. La bibliographie contient toutes les références citées, plus celles du fichier B, même si elles ne sont pas citées.

2.2.8 Remarques

Les spécifications du RA2001 sont les mêmes que celles du rapport 1999. Entre temps, le traitement a légèrement évolué, certains idées n'ont pas été implémentées, il y a donc quelques divergences par endroits. Nous citons dans ce paragraphe quelques points.

En principe, un module de nom toto, dans la section « Résultats nouveaux », du projet miaou, de l'année 2001 donne une page HTML nommée
`http://www.inria.fr/rapportsactivite/RA2001/miaou/resul_toto.html`
Pour faire en sorte que le nom de la page tienne sur 32 caractères, on tronque le nom de la section sur 5 caractères, et le nom du module sur 21 caractères. Avec cet algorithme on peut facilement référencer un module d'un projet quelconque, d'une année quelconque. Pour simplifier, le script

raweb.pl teste simplement si le nom tronqué est unique (a priori, on pourrait avoir deux modules avec les mêmes noms dans des sections différentes ; ceci est interdit, et est utilisé par le traducteur XML).

Il y a en principe une page HTML par section, contenant les liens vers les modules de la section, et une page principale contenant des liens vers les modules et sections du document complet ; ces pages seront appelées la table des matière. Rappelons que la première section ne contient qu'un seul module, et que souvent la deuxième n'en contient également qu'un seul. Pour éviter une table des matières triviale (être obligé de cliquer deux fois pour voir le contenu), on utilise le principe suivant : un module tout seul dans une section peut avoir un titre vide, dans ce cas il y a une indirection de moins dans la table des matières. Ce mécanisme étend celui de `\paragraph` et `\paragraph*` : dans le second cas, le paragraphe n'est pas numéroté, et n'apparait pas dans la table des matières. Nous pensons abandonner ce mécanisme dans la version XML ; ceci simplifie beaucoup le traitement, mais peut donner des résultats un peu moins jolis.

Pour avoir sur le WEB un résultat plus joli, on a introduit des frames. Ainsi, à chaque module, se trouvent associés au moins 4 pages HTML. Se pose donc la question : vers quoi doit pointer le lien généré par `\moduleref` ? Par ailleurs, on peut obtenir des noms plus long que 32 caractères (à cause des extensions supplémentaires). Pour contourner cette difficulté, nous avons choisi, pour la version XML, de ne pas accoler le nom de la section au nom du module. Ceci ne change rien en ce qui concerne la traduction de `\moduleref`, mais simplifie de beaucoup la traduction de `\ref` (autrement dit, la génération du lien HTML lors de la conversion XML vers HTML devient beaucoup plus simple). En ce qui concerne les frames proprement dits, on ne sait à l'heure actuelle, s'il y aura des frames dans le rapport 2002 ; ceci doit cependant être transparent au niveau XML.

D'après la définition donnée plus haut, un module est formé d'une interface, d'un synopsis et d'un corps. La notion de synopsis est un peu floue. Comme elle est associée à un environnement `abstract`, les auteurs considèrent le synopsis comme un résumé (on peut ainsi voir un résumé devant un texte de 5 lignes, un résumé qui n'a pas de rapport avec le texte qui suit, un résumé qui contient la première phrase du texte, ou une introduction à la suite). Dans la version 2001 du rapport d'activité, le résumé est composé dans une police spéciale (bien que certains forcent le résumé à être en italiques). Dans la version XML, on se propose de supprimer cette notion de synopsis/résumé/abstract. En fait, on ignore complètement les `\begin`, `\end` suivi de `body` ou `abstract`. Aisni un module devient : une interface, suivi de texte.

L'un des problèmes de la traduction de LaTeX en XML est que les auteurs ne respectent pas toujours les règles. On peut ainsi voir des mots clés après le synopsis, ou des `\pers` dans une liste. Nous pensons raisonable d'édifier des règles beaucoup plus strictes, mais en moins grand nombre.

2.3 Conversion en HTML

Il existe plusieurs types d'outils pour convertir un document LaTeX en HTML (voir par exemple [3]). Ceux-ci peuvent être adaptés plus ou moins facilement pour produire du XML, et prendre en compte des classes de document baroques (i.e. non standard).

- Les outils de type `Hévéa` ou `tth` sont constitués à la base d'un parseur de commandes LaTeX. Ils sont rapides, difficiles à adapter (en général, il faut modifier le source), et le

traitement des effets de bord n'est pas facile. Le terme « effet de bord » sera expliqué plus loin.
- Les outils de type `tex4ht` utilisent LaTeX pour traiter le source, et insèrent des balises dans le fichier `dvi`. Ainsi toutes les classes de document, et toutes les commandes utilisateur sont prises en compte automatiquement. Nous n'avons pas assez d'expérience sur ce type d'outils pour savoir s'il est facile de les adapter pour générer du XML.
- Les outils de type `latex2html` utilisent des règles de réécriture. Ils sont facilement paramétrables (il suffit d'ajouter du code `perl` ou `lisp`), mais lents et lourds.

Une des difficultés majeures pour adapter un traducteur LaTeX vers HTML en un traducteur vers XML est le problème de la fermeture implicite des balises, qui est autorisée en HTML, interdite en XML.

Nous donnons ici quelques transformations de source, de LaTeX vers XML ou HTML, et nous signalons quelques difficultés au passage.

La première version du rapport d'activité a été traduite en HTML via un processeur SGML. Depuis nous avons utilisé `latex2html`, qui a toujours donné des résultats satisfaisants, assez pour ne pas investir dans d'autres logiciels (il y a toujours un travail non négligeable d'adaptation). La lenteur de la traduction (argument souvent utilisé par la concurrence) n'est pas un problème : même si la traduction met une minute, ceci est négligeable devant le travail de correction, relecture, etc.

Nous décrivons dans cette section comment fonctionne `latex2html`, puis montrons quelques exemples. Remarquons que `latex2html` est un script perl de 260 pages (plus un grand nombre d'autres fichiers), alors que notre traducteur est nettement plus petit (60 pages dans la version 2, une centaine dans la version 3). Le traducteur `tralics`, est par contre, équivalent en taille à `latex2html`.

2.3.1 Principe général de `latex2html`

Essentiellement, on part d'un document LaTeX, et on remplace toutes les commandes par des balises HTML. Ceci se fait en plusieurs passes. Certaines parties peuvent être trop compliquées à traduire, dans ce cas on délègue le travail à TeX, et le résultat sera une image. Si l'expression à traiter est

```
\begin{equation} \toto\hbox{\ref{titi}}\end{equation}
```

il faut que le fichier vu par TeX contienne
- la définition de la macro `\toto`, si elle n'est pas expansée,
- tous les packages nécessaires à l'expansion de `\toto`,
- le `\label` associé au `\ref`,
- les valeurs actuelles des compteurs, pour pouvoir numéroter l'équation.

Gérer le contenu du fichier TeX est donc non trivial (il y a un préambule faisant une centaine de lignes de TeX). Chaque macro utilisateur doit être à la fois interprétée et mise de coté. Dans le cas où `$\alpha^2$` apparaît deux fois dans le fichier source, une seule image sera créée, et référencée deux fois, mais dans le cas précédent, il faut deux images, car le numéro d'équation n'est pas le même. Dans un certain nombre de cas, le source TeX généré est incorrect (au fur et à mesure de l'évolution de `latex2html`, ce phénomène est de plus en plus rare). Il est donc possible que `latex2html` demande 17 images, et que TeX en fournisse 16 ou 18. Par ailleurs, si on fait une modification mineure dans le source, on n'a pas envie de reconvertir toutes les images

(c'est ce qui prend souvent le plus de temps). Toutes ces raisons expliquent la complexité de `latex2html`.

La première chose que fait `latex2html`, c'est de remplacer les `\input` par le contenu du fichier (sauf ceux qui sont dans un environnement `verbatim`, bien sûr), dans notre cas, la commande `\input` est interdite, de même que les environnements `verbatim`, ce qui simplifie nettement le travail (dans la version 3 du traducteur, les environnements du type `verbatim` sont traduits). Le texte est alors découpé en sections, et chaque section est traitée séparément, les résultats des diverses sections sont concaténés, et un post-traitement est appliqué (celui-ci génère par exemple la table des matières). Pour chaque package ou classe de document ou option de classe, un fichier Perl est chargé en mémoire (par exemple `raweb.perl`, à ne pas confondre avec `raweb.pl`).

Remarquons que l'algorithme de découpage en sections implique qu'on ne peut pas utiliser de commande qui s'expanse en `\subsection`, et demander en même temps qu'une subsection soit une page WEB : il y a un problème de synchronisation. Ainsi, la transformation de `\RAstartmodule` en `\subsection` est faite dans le fichier `raweb.perl` par une procédure qui manipule le texte brut.

Il y a deux phases importantes : le marquage et la macro-expansion. Pour associer les accolades ouvrantes avec les accolades fermantes, `latex2html` les remplace par `<<N>>` où N est un numéro (la même chaine désigne l'accolade ouvrante et fermante). Lorsque le texte a été traduit, il est remplacé par `<#N#>`. Nous utiliserons dans la suite un autre algorithme de marquage : `\nN!`, où N est un numéro, et n un indicateur (qui n'est pas le même pour la balise ouvrante et la fermante). Le pattern typique de `latex2html` est `<<(\d+)>>([\s\S]*)<<\1>>`, tandis que le nôtre est `\\3(\d+)!(.*)\\4\1!` (la différence entre un point et `[\s\S]` concerne le traitement des retours à la ligne).

Une des difficultés de la traduction de TEX vers HTML ou XML est illustrée par le problème suivant : un \ en fin de ligne est interprété par TEX comme s'il était suivi par un espace. Or `latex2html` ne voyant pas d'espace ne le traduit pas. Dans certains cas, le retour à la ligne disparait. Si ce qui suit est une balise, `latex2html` voit un backslash suivi d'un signe inférieur, et râle en disant qu'il ne sait pas traduire `\<`.

2.3.2 Exemples de traduction

Exemple 1

```
\begin{abstract} Ceci est un \textit{r\'esum{\'{e}}~! \end{abstract}
```

Si on passe ceci à `latex2html`, la procédure `do_env_abstract` est appelée. Celle-ci va appeler `make_abstract`, procédure qu'on a modifié comme suit :

```
sub make_abstract {
    local($_) = @_;
    $_ = &translate_environments("$_");
    $_ = &translate_commands($_);
    local($env_id) = " CLASS=\"ABSTRACT\"" if ($USING_STYLES);
    join('',"\n<H3>$abs_title:</H3>\n"
        , (($HTML_VERSION > 3)? "<DIV$env_id>" : "<P>")
        , "<I>", $_ , "</I>",
        , (($HTML_VERSION > 3)? "</DIV>" : "</P>")
        , "\n<P>");
  }
```

Le résultat est :

```
<H3>Résumé :</H3>
<DIV CLASS="ABSTRACT"><I>
Ceci est un <i>résumé</i> !
</I></DIV>
```

Dans la dernière version de `latex2html`, le titre du résumé est calculé par le code suivant :

```
 if ((defined &do_cmd_abstractname)||$new_command{'abstractname'}) {
     local($br_id)=++$global{'max_id'};
     $title = &translate_environments("<<$br_id>>\\abstractname<<$br_id>>");
 } else { $title = $abs_title }
```

Cet exemple montre bien le fonctionnement de `latex2html` : il y a deux passes, dans la première, on traduit les environnements, et dans la seconde, on traduit les commandes. On voit donc déjà une première difficulté : l'ordre de traduction est imposé. Cet exemple montre également comment on peut paramétrer le résultat : suivant la version de HTML utilisée, on peut utiliser des divisions ou des paragraphes, on peut on non utiliser des feuilles de style, et la valeur de `abs_title` est utilisée pour le mot « Résumé ». Par ailleurs, dans la dernière version de `latex2html`, ce nom est également obtenu, soit via une procédure Perl, soit via une redéfinition de `\abstractname` dans le source LaTeX.

Le fichier `raweb.perl` contient 24 procédures de traitement de commandes LaTeX, 3 procédures de traitement d'environnement, et quelques autres procédures additionnelles.

L'exemple contient un certain nombre de difficultés. D'une part, il y a une demande de passage en italiques qui est inopérande, car tout le texte est en italique. Le rédacteur aurait dû utiliser `\emph` au lieu de `\textit`. À la réflexion, ceci ne fait que déplacer le problème : la police choisie pour le résumé ne devrait pas être codée comme `<I>` dans le HTML, mais dans la feuille de style. Le traducteur, ne connaissant pas la police courante, ne peut donc traduire le `\emph`, sauf à le traiter comme `\textit`.

Nous avons par ailleurs supprimé le tilde. En effet, nous aurions dû mettre un espace insécable à la place. Cependant, les extensions de francisation de LaTeX insèrent automatiquement un espace insécable devant les signes de ponctuation comme le point d'exclamation, ce qui fait que la plupart des auteurs ne le font pas explicitement ; ce qui signifie que pour être propre, il faut un postprocesseur.

L'exemple montre également deux façons différentes d'avoir des lettres accentuées. Il y a deux autres variantes `r{\'e}sum\'{e}`. Pour normaliser les lettres accentuées, on s'était proposé d'utiliser un préprocesseur tel que `recode`.

Une traduction XML serait plutôt :

```
<abstract> Ceci est un <it>résumé</it> ! </abstract>
```

Une autre traduction possible :

```
<div name='abstract'> <head>Résumé</head> Ceci est un <it>résumé</it> ! </div>
```

Cette façon de faire ressemble un peu au résultat HTML, sauf qu'il ne contient pas les paramètres des polices de caractères (qui sont en principe dans une feuille de style, ou équivalent).

Exemple 2

```
\newcounter{toto}
\newcommand\foo{\refstepcounter{toto}}
Ce texte fait référence à \ref{titi} défini ici \foo\label{titi}
```

La commande LaTeX `\newcounter` définit un compteur nommé `toto`. La commande `\foo` a deux effets : le compteur est incrémenté, et sa valeur est mise dans la variable globale `\@currentlabel`. Notons que la valeur du compteur est un entier, tandis que la variable globale contient une suite de caractères, et son calcul peut être paramétré (si le compteur est 5, la variable peut contenir cinq ou V, par exemple). Comme on utilise la valeur par défaut, la valeur est l'équivalent en lettres arabes du nombre. La commende `\foo` peut aussi avoir comme effet de remettre à zéro tous les compteurs qui dépendent de `toto`.

La commande `\label` mémorise deux quantités : la page courante, et la valeur de `\@currentlabel`. Il est possible de référencer ces deux valeurs via `\pageref` et `\ref`. Notons que traduire `\pageref` en HTML n'a pas grand sens. Ce que fait `latex2html`, c'est de mettre un lien hypertexte vers la position du `\label`, et d'indiquer une valeur, dans le cas de `\ref`, c'est la valeur de `\@currentlabel`, et dans le cas de `\pageref`, c'est une puce. Si on génère du Pdf via le package hyperref, la situation est plus compliquée. Un `\label` mémorise 5 informations. Dans l'exemple précédent, il y a le titre de la section courante, un marqueur spécial, `toto.1`, et la chaine vide.

Dans le cas où la valeur du compteur est 17, la traduction en HTML de l'exemple précédent est du type :

```
Ce texte fait référence à <a href="#titi">17</a> défini ici <a name="titi"></a>
```

La difficulté majeure est d'obtenir la valeur (le nombre 17). Ce que fait `latex2html` c'est de lire le fichier auxiliaire (il suppose qu'on n'utilise pas le package hyperref, car la syntaxe est différente). On peut essayer d'avoir une traduction sans demander à LaTeX de calculer le label. Pour cela, il faut d'abord savoir quel est le compteur à utiliser. Supposons que cela soit `\toto`. La valeur du compteur est `\thetoto` (le nombre en chiffres arabes par défaut ; si le défaut n'est pas utilisé, c'est que la commande `\thetoto` a été redéfinie, on peut traiter ce cas). La valeur du label est le résultat de `\p@toto` appliqué à ceci (suite à un bug de conception, `\p@toto` ne peut pas faire grand chose, sauf ajouter un préfixe devant, il est possible de modifier ce comportement, comme expliqué dans les sources, mais on supposera que le rédacteur ne connaisse pas cela). On va donc faire comme si on savait calculer `\@currentlabel` à partir du compteur `toto`.

La question essentielle est donc de savoir quel est le compteur qui crée le `\@currentlabel`, sachant que cette variable est locale au groupe. On va faire l'hypothèse suivante : si un `\label` est juste derrière une sous-section (mais avant une sous-sous-section), c'est le compteur de sous-section. Lors de la traduction en XML, on va raccrocher le label à la sous-section ou à son titre. Si le `\label` est dans une formule de mathématiques, c'est le compteur d'équations. Si le `\label` est derrière un `\caption` dans une table ou une figure, c'est le compteur de table ou de figure (on supposera par ailleurs que la table ou la figure continent un seul `\caption`, donc a un numéro unique).

L'exemple précédent ne rentre pas dans le cadre prévu plus haut. Notre traduction va faire comme si la commande `\foo` n'existait pas (dans la version initiale du traducteur, on refusait de traduire `\newcounter`, et par ailleurs, les manipulations des compteurs).

Exemple 3

```
\pers{Jean}{Dupont}[{\sc dr} Inria]
```

La traduction en HTML de ce code est

```
Jean Dupont [<small>DR</small> Inria] %
```

Comme on peut le voir, il y a une certaine perte d'information. Il est impossible de distinguer le nom du prénom. Nous reviendrons plus tard sur l'utilisation de la commande `\sc` : elle demande de mettre toute la suite (mais jusqu'où ?) en petites capitales. La bonne manière de faire est de dire `\testsc{dr}`. Comme la gestion des petites capitales est un peu délicate dans HTML, le traducteur traduit tout en majuscules et insère des commandes `<small>` aux endroits adéquats. Par exemple `\textsc{Dupond}` sera traduit en `D<small>UPOND</small>` ce qui empêche une recherche plein texte du mot « Dupond ».

La traduction XML proposée est

```
<pers pre="Jean" nom="Dupont"> <sc>dr</sc> Inria </pers>
```

Il n'y a ici pas de perte d'information.

Remarque : soit les deux personnages historiques suivants : Gustaf de Laval et Henri de La Vaux. Si on les classe par ordre alphabétique, le premier sera classé avec les D, le second avec les L. Ceci est dû au fait que la particule « de » a un comportement différent suivant la nationalité. La commande `\pers` a un argument optionnel `[part]`, qui fait partie du nom, et qui n'est pas utilisé pour le classement.

Nous avons initialement prévu d'utiliser `ltx2x` pour traduire le source en HTML. Celui-ci refuse les arguments optionnels de ce type. On a donc décidé d'utiliser un préprocesseur pour accoler la particule au nom. Dans la version actuelle, ce problème ne se pose plus. Par contre, il y a un autre :

```
\pers{Jean}{Dupond\footnote{Homonyme de Jean Dupont}}
```

On en peut pas mettre de note dans un attribut, la construction est donc incorrecte. Ce que va faire le préprocesseur, c'est de convertir le texte précédent en

```
\pers{Jean}{Dupond}[Homonyme de Jean Dupont]
```

(on supprime la note, car cela ne fait que distraire le lecteur).

Exemple 4

```
\subsection{toto}
```

La traduction HTML de cette commande est du type `<H3>toto</H3>`, sachant que, en général, il faut faire une table des matières, ce qui impose la mise à jour d'information répartie ailleurs. Notons également que cette commande met à jour les compteurs de sous-section (et peut-être d'autres).

Supposons que l'on veuille générer le code XML suivant :

```
<subsection> <titre> toto </titre>
```

Se pose immédiatement la question de savoir où mettre la balise fermante `</subsection>`. Notons que la sous-section se termine lors du début d'une autre sous-section, du début de commande de sectionnement de niveau supérieur, ou la fin du document. En HTML, on peut omettre la balise fermante `</p>` associée à une balise ouvrante `<p>`. Le placement par défaut utilise des règles de ce type. Ceci n'est pas possible dans XML. On donnera dans la suite trois solutions à ce problème.

Il reste un autre point, maltraité par notre traducteur : le changement de police de caractères. En effet, s'il y a un `\it` dans la section, la fermeture de la section entraine la fin du `\it` ce qui est une erreur.

Exemple 5

```
$x^2+\delta_3=\int f(t)\,dt$
```

La différence essentielle entre les divers traducteurs LaTeX vers HTML consiste en leur gestion des mathématiques. Hévéa suppose par exemple que le navigateur Web possède toutes les polices mathématiques et propose ceci :

```
<I>x</I><SUP><FONT SIZE=2>2</FONT></SUP>
+<FONT FACE=symbol>d</FONT> <SUB><FONT SIZE=2>3</FONT></SUB>
=<FONT SIZE=4><FONT FACE=symbol>ò</FONT></FONT>
<I>f</I>(<I>t</I>) <I>dt</I>
```

`latex2html` produit soit une image pour toute la formule, soit une image pour chaque morceau intraduisible. Par exemple :

```
<SPAN CLASS="MATH"><I>x</I><SUP>2</SUP> + <IMG
 WIDTH="18" HEIGHT="29" ALIGN="MIDDLE" BORDER="0"
 SRC="img1.png"
 ALT="$ \delta_{3}^{}$"> = <IMG
 WIDTH="15" HEIGHT="33" ALIGN="MIDDLE" BORDER="0"
 SRC="img2.png"
 ALT="$ \int$"><I>f</I> (<I>t</I>) d<I>t</I></SPAN>
```

Cet exemple montre toute la subtilité de `latex2html`. Le fait que le d dans le `dt` ne soit pas en italiques n'est pas clair : est-ce un bug, une feature ? Dans le cas de x^2, Hévéa met l'indice en plus petit, `latex2html` ne ne fait pas, ce qui nous parait préférable (mettre des exposants trop petits diminue la lisibilité). Notons la gestion des espaces : Hévéa met dans espaces là où il y en a dans le source. Le seul espace est donc celui qui précède le f, qu'on a mis pour séparé le `\int` du `f` (espace qui est normalement ignoré par TeX). Par contre `latex2html` est nettement plus rusé : il met des espaces de part et d'autre des signes plus et égal (ce qui est normal) et en met un après le f, ce qui n'est pas logique. Les deux traducteurs remplacent l'espace fine par une espace insécable (il n'y a pas d'espace fine dans HTML). Notons que la première image de `latex2html` ne contient pas `\delta` mais `\delta_{3}^{}` (la raison de ce choix n'est pas claire). Nous verrons plus loin quel est l'intérêt d'ajouter des accolades autour du chiffre 3.

Une des manières de traduire l'expression précédente en XML, plus précisément, en MathML, consiste à faire comme `latex2html` : imprimer l'expression dans un fichier temporaire, et appliquer un autre logiciel, par exemple Ω. On obtient ainsi :

```
<math>
  <mrow>
    <msup>
      <mi> x </mi>
      <mn> 2 </mn>
    </msup>
    <mo> + </mo>
    <msub>
      <mi> &delta; </mi>
      <mn> 3 </mn>
```

```
      </msub>
      <mo> = </mo>
      <mo> &int; </mo>
      <mi> f </mi>
      <mrow>
        <mo> ( </mo>
        <mi> t </mi>
        <mo> ) </mo>
      </mrow>
      <mi> d </mi>
      <mi> t </mi>
    </mrow>
  </math>
```

Notons que l'espace fine a disparu. Par ailleurs, cet exemple montre que MathML est très verbeux.

Notre traducteur (version 1) donne ceci

```
<math>
  <msup> <mi>x</mi> <mn> 2 </mn> </msup>
  <mo> + </mo>
  <msub> <mi>&delta;</mi> <mn> 3 </mn> </msub>
  <mo> = </mo> <mo> &int; </mo> <mi>f</mi>
  <mo> ( </mo> <mi>t</mi> <mo> ) </mo> <mspace width='3pt'/>
  <mi>d</mi> <mi>t</mi>
</math>
```

Par rapport au code de Ω, on notera les différences suivantes : le `(t)` donne un `<mrow>` dans Ω, ce qui est joli, mais si on avait mis `(t(`, Ω n'aurait pas vu la parenthèse fermante, et aurait ignoré (sans rien dire) le reste de la formule. L'autre remarque est la suivante : il n'y a pas d'espace dans les éléments `<mi>`. Il y a une astuce : la conversion de MathML vers Pdf utilise une fonte romane si l'élément `<mi>` contient plus d'un caractère, une police italique sinon. Plutôt que de changer ceci, on préfère s'en servir : la traduction de `\mathrm{x}` donne un élément `<mi>` avec un espace.

Exemple 6

```
\begin{figure} \includegraphics{toto.eps} \end{figure}
```

Le problème essentiel pour la conversion en HTML de ce genre de commande est la traduction de l'image postscript en un format d'image reconnu par HTML. Pour la traduction en XML, nous avons décidé de ne pas traduire l'image. Toute la difficulté réside dans la sémantique de l'environnement `figure` (et de son associé, `table`).

Un chapitre entier est dévolu dans [2] pour expliquer le comportement des flottants. Faire la liste de tous les problèmes potentiels prendrait trop de place. On verra dans le chapitre suivant quelles sont les solutions proposées.

2.4 Le script raweb.pl

2.4.1 Principe de fonctionnement

Le script raweb.pl est divisé en cinq parties : la première partie est une phase de préparation, on y définit quelques tables, et on interprète les arguments du script ; la deuxième partie concerne la lecture du source TEX ; elle est suivie par une phase de tests et un post traitement. Finalement, le script s'occupe également de la traduction en XML, cette dernière partie sera expliquée dans le chapitre suivant.

Pour ne pas alourdir les notations, on supprimera les préfixes `$` devant les variables Perl. Ainsi `$::modlist{$::modno}` sera transcrit en `modlist{modno}`. Il y a une table d'association `todo` qui contient les champs suivants `all`, `merge`, `ps`, `split`, `check`, `parse`, `html` et `xml`. Cette table indique ce que le script doit faire. Initialement, tous les champs sont 0, après analyse des paramètres, un et un seul champ aura une valeur de 1 : ce sera l'objectif. Une phrase de type : « si l'objectif est xml ... » signifie : si `todo{xml}` est 1. Les objectifs principaux sont : `check` (test de syntaxe), `ps` (test et génération de PostScript), `html` (test, génération de PostScript et de HTML), et `xml` (test, et génération de XML). Les autres options sont moins utiles.

Remarque : le script est optionel, il est remplacé par le traducteur `tralics`, qui au passage, fait ce qui est décrit dans cette section.

2.4.2 Phase de préparation

Dans toute la suite, on supposera que le document à traiter s'appelle `safir`, et qu'il contient quelque chose de la forme :

```
\projet{SAFIR}{\textsc{safir}}{Le projet modèle}
\theme{2b}
\typeprojet{Projet}
\localisation{Sophia Antipolis}
\begin{module}{fondements}{le-module}{Titre-du-module}
Le texte du module.
\end{module}
```

Il y a une variable `projet` qui contient 5 champs : les 3 premiers sont SAFIR, safir et SAFIR, les deux autres étant « SAFIR » et « Le projet modèle ». Pour des raisons historiques, il y a une redondance : le premier argument de la commande `\projet` doit être en majuscules, et doit être le nom du fichier. La procédure `see_projet` est appelée pour traiter la commande `\projet`. Elle appelle trois fois `find_field` pour trouver les arguments, puis `ignore_text` (voir plus loin), et elle remplit la variable `projet`. Au début de chaque module, on vérifie que la commande `\projet` a été effectivement vue.

Il y a une table `global_info` qui contient les champs suivants `localisation`, `theme`, `type`, `UR`, `theme_url`, `loc_url`, `projet_url`, `ps_url` et `pdf_url`. La procédure `see_aux_info` traite les trois commandes `\localisation`, `\theme`, `\typeprojet`, elle appelle `find_field` puis `ignore_text`, et remplit les 3 premiers champs de la table. À partir de la localisation, on en déduit l'UR via une heuristique (qui ne marche que pour les projets localisés dans une unité de recherche unique). Cette heuristique a été supprimée en 2002. On calcule ensuite les URL du thème, de l'unité de recherche, et du projet. On calcule aussi les URL de la version PostScript et Pdf du document final. Ceci permet donc de remplir complètement la table.

Il y a une table `modlist` qui est la liste de tous les modules. En position N dans la table se trouve une liste de longueur 5 : le numéro de la section dans laquelle se trouve le module, le nom du module, le titre du module, le nom complet du module, et finalement le contenu. Le nom complet du module donné en exemple plus haut est `safir_fondements_le-module`, un fichier TEX de ce nom va être créé, et va contenir le contenu du module.

Il y a une table `MS` qui contient l'état global de notre automate. Les champs `before` et `after` contiennent le texte (hors module) qui est avant ou après le `\begin{document}` (sauf les commandes expliquées plus haut). Normalement `before` ne doit contenir que des `\usepackage` ou des définitions de commandes, `after` ne doit contenir que des `\cite`, `\nocite`, et un environnement `moreinfo`. Si l'objectif est `xml`, tout le reste est ignoré.

Il y a un champ `state` qui dit si la ligne courante est avant le `\begin{document}`, après ou dans un module. Il y a un champ `changed`, qui est vrai uniquement si la ligne courante va modifier `state` (parce qu'elle contient le début ou la fin d'un module). Il y a un champ `content` qui est le contenu actuel du module, utilisé lors de la lecture. Finalement, trois champs `name`, `section` et `first_line` contiennent le nom du module, le numéro de la section et le numéro de la première ligne du module.

Il y a quatre tables pour gérer les sections. La table `nom_section` contient 3 en position « fondements », (fondements est un des arguments autorisés de `\module` ou `\moduleref`). La table `sec_acc` est l'inverse de la précédente, la table `sec_nums` contient `fonde` en position 3 (c'est le nom tronqué sur 5 caractères), et finalement, la table `sec_names` contient le titre « Fondements scientifiques ». La procédure `check_mod`, qui interprète `\begin{module}` récupère les trois arguments, le premier étant la section sous forme symbolique. Elle vérifie que c'est un nom correct (notons qu'une valeur vide est autorisée, sauf pour le premier module), et remplit `MS{section}`. La procédure compte aussi le nombre de modules dans les sections 1 et 2 : en effet, il faut un seul module en section 1, et s'il n'y a pas de modules en section 2, c'est le contenu de la fiche projet en français qui sera mis à la place (pour l'année 2001 uniquement).

2.4.3 Traitement des arguments

Le script regarde les arguments les uns après les autres. S'il l'argument commence par un tiret, il est interprété comme suit :

- verbose : on positione `verbose` à vrai, le script sera verbeux.
- debug : on positionne `stop_on_error` à faux (certaines erreurs ne sont pas fatales).
- all : on positionne `todo{all}` à vrai.
- check : on positionne `todo{check}` à vrai.
- split : on positionne `todo{split}` à vrai.
- merge : on positionne `todo{merge}` à vrai.
- ps : on positionne `todo{ps}` à vrai.
- xml : on positionne `todo{xml}` à vrai.
- html : on positionne `todo{html}` à vrai.
- nocaptions : on positionne `todocaption` à vrai.

Les options suivantes prennent un argument (noté toto dans la description)

- dir : on positionne `RAWEBDIR` à toto.
- latexname : on positionne `LATEXNAME` à toto.
- latex2htmlname : on positionne `LATEX2HTMLNAME` à toto.

- dvipsname : on positionne `DVIPSNAME` à toto.
- htmloptions : on positionne `HTMLOPTIONS` à toto.
- latexoptions : on positionne `LATEXOPTIONS` à toto.
- dvipsoptions : on positionne `DVIPSOPTIONS` à toto.

Tout argument du script qui ne commence pas par un tiret est considéré comme le nom du fichier à traiter. Celui-ci doit être `safir2001` ou `safir2001.tex`. Comme dit plus haut, il doit y avoir adéquation entre le nom du fichier et le nom du projet. Les variables suivantes sont calculées

- `infile` : nom complet du fichier, safir2001.tex ;
- `no_ext` : fichier sans extension, safir2001 ;
- `no_year` : fichier sans l'année, donc safir ;
- `year` : l'année, 2001 ;
- `raclass` : la classe de document `ra2001`.

C'est une erreur si le nom de fichier n'est pas donné, est donné plusieurs fois. C'est une erreur de donner une option non reconnue. C'est une erreur de donner deux options qui remplissent la table `todo` (si aucun objectif n'est donné, on utilise `check`). C'est également une erreur si le fichier source n'existe pas.

2.4.4 Post traitement

Un certain nombre de procédures sont appelées, une fois que le texte est lu et vérifié. Dans cette section, on ne considère pas le cas où l'objectif est xml, car ceci sera décrit dans le chapitre suivant.

La procédure `mkcfg` est appelée si l'objectif est all, ps ou html. Elle crée un fichier `hyperref.cfg`, qui permet de mettre un certain nombre d'informations dans le fichier Pdf (auteur du document, etc). Ce fichier contient en particulier l'ensemble des mots clés, qui a été collecté par la procédure `do_indexing`. Malheureusement, on ne peut lire ce fichier via `xmltex`, ce qui fait que, si l'objectif est xml, le fichier `hyperref.cfg` doit être détruit (on espère quand même résoudre ce problème un jour).

La procédure `copy_and_exec` est appelée si l'objectif est all, ps, merge ou html. Elle lit le fichier `.latex2html-init` du répertoire RAWEBDIR et le copie dans le répertoire courant, avec quelques modifications (on insère le nom du projet, son UR, etc, y compris la valeur de HTMLOPTIONS s'il y en a une). La procédure crée un répertoire `icons` et y met des liens symboliques pour les fichiers d'icônes. Elle crée aussi d'autres liens symboliques utiles. Finalement, on crée un fichier `safir.refer.aux`, puis on appelle LaTeX une première fois (ce qui donne entre autres, les fichiers `safir.fb.aux` et `safir.aux`). On lance bibtex sur les trois fichiers .aux, puis LaTeX deux fois. Ensuite, on appelle dvips. Trois procédures `call_latex`, `call_bibtex` et `call_dvips` sont utilisées, les traces des exécutions sont concaténées dans le fichier safir2001.log.

Pour générer le fichier postscript, on appelle DVIPSNAME avec DVIPSOPTIONS comme arguments (normalement DVIPSNAME est simplement dvips). Si DVIPSOPTIONS ne contient pas la chaine `%s`, on rajoute `%s.dvi -o %.ps` derrière. Dans tous les cas `%s` sera remplacé par le nom du fichier. Ainsi, avec les bonnes options, le résultat PostScript sera imprimé plutôt que copié sur disque.

Pour générer le fichier dvi, on appelle LATEXNAME avec LATEXOPTIONS et le nom du fichier. Normalement LATEXNAME est latex, et LATEXOPTIONS est `--interaction=batchmode` ; une exception est faite pour des versions vieillottes du moteur TeX.

La procédure `merge_bib` est appelée si l'objectif est `all`, `merge` ou `html`. Les trois fichiers bbl générés par bibtex sont lus et concaténés. On supprime les lignes contenant `\setcounter`, `\end {thebibliography}`, `\etalchar`, et on remplace `\begin {thebibliography}` par `\removebracket` (c'est même plus subtil que cela). Puis on insère un unique `\begin`, `\end`, et définition de `\etalchar`. On renumérote tous les `\bibitem` numérotés. Finalement, le résultat est imprimé dans le fichier `safir.main.bbl`. Remarque : si l'objectif est xml, il y a une procédure similaire, mais beaucoup plus simple, pour concaténer les trois fichiers bbl en en seul.

2.4.5 Lecture du fichier source

Il y a une variable globale `line` qui est incrémentée chaque fois qu'une nouvelle ligne est lue dans le fichier source. La procédure `append_line` est une procédure auxiliaire qui va concaténer avec `$_` de nouvelles lignes jusqu'à obtenir une accolade (si la fin de fichier est trouvéee c'est une erreur). Les lignes vides, ou qui commenencet par `%` sont ignorées.

La procédure `find_field` appelle `append_line` pour commencer. C'est une erreur si le résultat ne commence pas par une accolade ouvrante (précédée par des espaces éventuels). Elle appelle ensuite `append_line`, tant que nécessaire pour trouver l'accolade fermante associée à l'accolade ouvrante. Elle rend le texte entre accolades, met la partie non lue dans `$_`. Note : dans le cas de `{toto\}` ou `{\toto%}`, le résultat est `toto\` ou `\toto%`, ce qui est une erreur. La procédure est utilisée pour trouver les arguments de `\projet`, etc, ainsi que dit plus haut, et les arguments de l'environnement `module`. Ceci impose donc quelques légères contraintes sur le texte.

La procédure `ignore_text` provoque une erreur si la ligne courante n'est pas vide ou ne commence pas par un signe `%`. Elle est appelée avec comme argument ce qui suit un `\projet`, etc.

La procédure `read_a_file` lit le fichier source. Elle posisitionne `MS{state}` à 0, et ne met rien dans `start_of_line` (que l'on abrégera en SOL). Il y a deux autres variables `unprocessed` et `unprocessed_l`, que l'on abrégera en U et UL dans la suite. La ligne courante est `$_`, elle est interprétée comme suit :

- si elle commence par `%`, elle est ignorée.
- si elle commence par `\documentclass`, on vérifie que ce qui suit est bien la valeur de `raclass`,
- si elle commence par `\begin{document}`, on provoque une erreur si `MS{state}` n'est pas 0, et on change la variable en 1. On appelle `ignore_text`.
- si elle contient `\end{document}`, on arrête tout. La lecture du source est finie.
- si la ligne courante contient `\begin` ou `\end`, on vérifie d'abord qu'il y a une accolade sur la ligne. On appelle `find_field` pour trouver le nom de l'environnement. Supposons qu'on ait vu X, qui est `\begin{env}` ou `\end{env}`. On met dans U ce qui précède X, dans `$_` ce qui suit X, et dans UL la concaténation de U et X. Si le nom de l'environnement est `module`, on positionne `MS{changed}` à vrai.
  Quand on voit `\begin{env}`, on empile le nom dans une pile. Quand on voit `\end{env}`, on dépile et on teste : c'est une erreur si la pile est vide, ou si les noms diffèrent. C'est aussi une erreur si la pile est non vide quand on voit `\end{document}`.
- si le ligne contient `\begin` ou `\end`, mais que l'environnement n'est pas `module`, on remplace SOL par la concaténation de SOL et UL, on recommence les tests sur la ligne courante. Si l'environnement est `module`, on appelle `check_mod` ou `end_mod`, puis `check_line`.

– Dans tous les autres cas, on appelle `check_line` (en concaténant SOL et `$_`).

La procédure `check_line` ignore les lignes qui commencent par `%`, `\maketitle` ou `\loadbiblio`. Elle provoque une erreur si la ligne contient des commandes interdites, soit des commandes qui commencent par RA, utilisées en interne par le script, soit des commandes du type `\input`, `\include`, `\section` et `\subsection`. Elle appelle `see_projet` ou `see_aux_info` si la ligne courante contient `\projet`, `\theme`, `\typeprojet` ou `\localisation`. Dans le cas où `todocaption` est faux, elle remplace `\caption{x}` par `\caption[y]{x}` où y est un identificateur unique (ceci pour contourner un bug dans `latex2html`).

Quand on voit `\begin{module}`, la procédure `check_mod` appelle `check_line` sur la concaténation de SOL et U (tout ce qui précède le `\begin`) on incrémente le compteur de modules, puis on appelle `find_field` trois fois, pour avoir les arguments. On vérifie que le nom du module est correct (uniquement des lettres, des chiffres, et de tirets), et on met le résultat dans `MS{name}`. On tronque le nom sur 21 caractères, et on vérifie l'unicité du nom tronqué. Supposons que le projet soit P, la section S, le nom N, le nom tronqué T, et le titre t. On remplit `MS{content}` de la façon suivante : dans le cas xml, on met `{P}{S}{N}`, sinon on met `\HTMLset{current_module_name}{S/T}` puis `\RAstartmodule{P}{S}{N}`. Remarquons que `\HTMLset` est ignoré par LaTeX, mais son résultat utilisé par raweb.perl, le fichier chargé en mémoire par `latex2html`. Par ailleurs, dans le cas xml, il manque le `\begin{module}`, qui sera rajouté plus tard (la raison de ceci sera expliqué plus loin) (remarque : toutes ces subtilités ont disparu dans `tralics`). Si le titre t est vide on ajoute `%\relax` dans le fichier. On rajoute (sauf dans le cas xml) un environnement `rawhtml` (du code HTML ignoré par LaTeX) qui contient juste un commentaire. Finalement, on rajoute `\RAlabel{P@s@N}` où s est le nom de la section (rappelons que S est le numéro de la section). Une fois tout ceci fait, on remplit les 4 premiers champs de `modlist{modno}`, à savoir la section, le nom du module, le titre et le nom complet.

Quand on voit `\end{module}`, la procédure `end_mod` appelle `check_line` sur la concaténation de SOL et UL. Après cela, la valeur de `MS{content}` est le contenu du module courant. Dans le cas où l'objectif est xml, celui-ci sera copié dans le dernier slot de `modlist{modno}`. Si l'objectif est all, ps, split ou html, on appelle `do_indexing` sur le contenu du module, et le résultat est copié dans le fichier M.tex (où M est le nom complet du module). Sauf si l'objectif est merge ou parse, le module est testé via `do_all_checks`. Note : dans le contenu du module, chaque ligne est précédée par un numéro de ligne, celui-ci disparait lorsque le résultat est copié (dans le slot ou le fichier), mais est conservé pour la phase de vérification. (Notons que `tralics` a besoin des numéros de ligne lors de la traduction).

La procédure `fiche_projet` vérifie que les deux fichiers `safir-fp.fr.tex` et `safir-fp.en.tex` existent. Elle lit la fiche projet en français, supprime le `\documentclass`, le `\begin{document}`, et le `\end{document}`, et met le résultat de coté, pour usage ultérieur. Elle remplace également `\href` par `\htmladdnormallink`.

La procédure `check_presentation` ne fait rien s'il y a au moins un module en section 2. Dans le cas contraire, elle fait comme s'il y avait un module dans cette section, dont le contenu est `\begin{module}{presentation}{presentation}{présentation}`XX`\end{module}` où XX est le contenu de la fiche projet en français (pour l'année 2001 uniquement).

2.4.6 Génération du résultat

La procédure `print_mods` crée le résultat. Il y a trois parties : le début, une action par module, et une fin. Le principe est le même dans le cas où on génère du xml, mais les détails changent, ceux-ci seront expliqués dans le chapitre suivant.

Au début, on imprime dans le fichier `safir_declarations.tex` tout ce qui se trouve dans `MS{before}` (donc les `\usepackage` et les définitions de commande). On imprime dans le fichier `safir_declbis.tex` tout ce qui est dans `MS{after}`, et quelques informations. Exemple :

```
\nocite{*}
\def\RAprojet{SAFIR}
\def\RAprojetlow{safir}
\RAstartprojet{2b}{Projet}{\textsc{safir}}
 {Systèmes Algébriques et Formels pour l'Industrie et la Recherche}{Sophia Antipolis}
\begin{htmlonly}
  \title{Systèmes Algébriques et Formels pour l'Industrie et la Recherche}
  \author{\textsc{safir}
  \rm\\[1cm]Rapport d'activité de l'année 2001\\[1cm]
  {\it \htmladdnormallink{Sophia Antipolis}
  {http://www.inria.fr/inria/organigramme/fiche_ur-sop.fr.html}}\\[1cm]
   Thème :
  \htmladdnormallink{2b}{http://www.inria.fr/recherche/equipes/projets_theme2.fr.html}
  \\[1cm]\htmladdnormallink{Page de présentation du projet}
  {http://www.inria.fr/recherche/equipes/safir.fr.html} -
  Rapport d'activité au format
  \htmladdnormallink{PostScript}
  {http://www.inria.fr/rapportsactivite/RA2001/safir/safir.ps.gz}
  ou
  \htmladdnormallink{PDF}{http://www.inria.fr/rapportsactivite/RA2001/safir/safir.pdf}}
  \maketitle
\end{htmlonly}
\mytableofcontents
\begin{moreinfo}bla bla \end{moreinfo}
```

On écrit alors dans le fichier `safir.tex` quelque chose comme :

```
\documentclass{raweb}
\input{safir_declarations}
\begin{document}
\input{safir_modules}
\end{document}
```

Finalement, on construit un fichier `safir_modules.tex` qui ressemble à :

```
\input{safir_declbis}
\def\RAmysection{composition}
\section{Composition de l'équipe}
\RAlabel{SAFIR@\RAmysection@}
\input{safir_composition_en-tete.tex}
\def\RAmysection{presentation}
\section{Présentation et objectifs généraux}
\RAlabel{SAFIR@\RAmysection@}
\input{safir_presentation_presentation.tex}
...
\begin{htmlonly}
```

```
  \input{safir.main.bbl}
\end{htmlonly}
\begin{latexonly}
  \section{Bibliographie}
  \loadbiblio
  \bibliography{safir2001}
\end{latexonly}
```

L'algorithme est le suivant : pour chaque section, on compte le nombre de modules dans la section. S'il y en a au moins un, on imprime `\section` et `\RAlabel`. On imprime ensuite un `\input` pour chaque module dans la section.

2.4.7 Les tests

Une fois que le texte est complètement lu, on vérifie qu'il ne manque rien (thème, type de projet, localisation). Sur la partie `MS{after}` on appelle `check_new_command` et `check_preamble`. La première procédure teste qu'il n'y a pas de `\def`, `\let`, `\newcommand`, `\newenvironment`, ni de `\newif`, `\newcount` et autres. La second procédure vérifie qu'il n'y a pas de macros `\pers`, `\subsubsection`, `\paragraph`, `\subparagraph` ni d'environnements `participants`, `motscle`, `glossaire`, `body` et `abstract`.

Pour chaque module, on appelle `check_new_command` et `check_module_ref` (cette dernière procédure vérifie les `\moduleref` mais pourrait être améliorée).

La grosse procédure de test est `check_pers_env`. Elle commence par récupérer les environnements `motscle`, `catperso` et `participants`. On vérifie que les environnements ne sont pas emboîtés, qu'il n'y a d'environnements `catperso` que dans les modules de la première section. On vérifie que les `\pers` sont uniquement dans un environnement `participant` ou `catperso`. On appelle la procédure `check_kw` dans le cas d'un environnement `module`, et la procédure `check_catperso` dans les autres cas.

Une procédure auxiliaire `remove_braces` remplace les bouts de texte type `{...}` et `[...]` par `\2N!` et `\3N!` où N est un nombre unique via un mécanisme similaire à celui décrit dans 1.3. Dénotons ces motifs par b et B. Dénotons par p le motif : b, B optionnel, b, B optionnel ; il est remplacé par `\4N!`. C'est une erreur s'il reste des `\pers` après cette substitution. C'est une erreur si le dernier argument (optionnel) de `\pers` commence par une virgule.

La procédure `check_catperso` vérifie qu'il y a bien un argument, dans le cas de l'environnement `catperso`. Sinon, on vérifie que, dans le cas de `catperso`, il n'y a que de `\pers`, et dans les autres cas, qu'il n'y a que des `\pers` séparés par des virgules. Toute la complexité du code est due au fait que l'on veut un message d'erreur compréhensible. La procédure imprime également un warning dans le cas ou l'environnement `participants` est au pluriel, et ne contient qu'un participant, et vice-versa.

La procédure `check_kw` prend en argument un environnement `motscle`. Elle vérifie que l'environnement n'est pas vide, et envoie un warning s'il y a des commandes LaTeX dans les mots. En fait, on aimerait éviter les mots clés du genre H^∞, (max, +), TOTO (police bizarre). Pour traiter le cas de « `\oe uvres compl{\‘e}tes` », une procédure auxiliaire est utilisée pour transformer le `{\‘e}` en è. Elle échoue cependant pour le `\oe` (l'encodage utilisé étant iso-latin1).

2.5 L'outil ltx2x

Une première tentative de traduction du source LaTeX en XML est faite avec cet outil. Nous expliquerons dans la suite pourquoi on y a renoncé. Les idées de traduction expliquées ici serviront quand même dans la suite.

2.5.1 Principe de fonctionnement

Il s'agit fondamentalement d'un parseur. Le source LaTeX est découpé en unités lexicales, et une action peut être associée à chaque unité en fonction de son type. Si par exemple `\foo` est déclarée comme une commande à deux arguments, on peut spécifier une action pour le début et la fin de la commande, et de chaque argument (donc six actions en tout). Si chaque action consiste à imprimer une balise, on traduira trivialement `\foo{10}{11}` en

```
<foo><arg1>10</arg1><arg2>11</arg2></foo>
```

Parmi les actions possibles, on peut demander d'ignorer la valeur d'un argument, ou rediriger l'impression dans un buffer (une variable globale). On peut également lire un buffer. Ainsi, il est tout aussi facile de générer

```
<foo><arg2>11</arg2><arg1>10</arg1></foo>
```

bien que le texte source soit lu et traité de gauche à droite.

En plus des commandes de type `\foo`, on peut spécifier une action pour des commandes de type mono-caractère (type `\~`), pour les caractères actifs (type `~`), pour les environnements, pour le début et la fin du document.

2.5.2 Syntaxe des commandes TeX

Il est possible en TeX de définir une commande comme suit

```
\def\foo(#1,#2){[#1,#2]}
\def\bar#1#2{[#1,#2]}
```

Dans ce cas `\foo(10,11)` et `\bar{10}{11}` donneront [10,11], tandis que `\foo(1,2)` et `\bar 12` donneront [1,2]. Dans le cas de la commande `\bar`, les accolades sont optionnelles si l'argument est un token unique (par exemple, un chiffre, une lettre, un nom de commande). Cependant `ltx2x` ne reconnait pas cette syntaxe abrégée. Par ailleurs, la syntaxe des commandes de type `\foo` n'est pas reconnue (sauf dans des cas particuliers de schémas de commandes LaTeX où une paire (x, y) désigne les coordonnées d'un point).

Une tradition veut que les arguments optionnels d'une commande soient donnés entre crochets. Dans `ltx2x`, cet argument optionnel peut être le premier ou le dernier. Notons que la commande `\pers` a deux arguments optionnels, le deuxième et le dernier. Cette commande ne rentre pas dans le cadre général (ni dans la liste des exceptions). Pour traiter cette difficulté, le préprocesseur accole la particule au nom, et ce que voit `ltx2x`, c'est une commande à trois arguments, le dernier étant optionnel.

Supposons que l'on veuille traduire `x_{3}^{2}` sous la forme `x<sub>3</sub> <sup>2</sup>`, Ceci se fait facilement en définissant `^` et `_` comme étant des caractères actifs à un argument. Par contre un grand nombre de rédacteurs sont allergiques à la multiplication des accolades inutiles, et entrent l'expression précédente sous la forme `x_3^2`. Dans ce cas, `ltx2x` est impuissant. Il faut donc utiliser un préprocesseur pour ajouter les accolades (comme le fait `latex2html`, voir

exemple 5). Notons que MathML demande `<msubsup> x 3 2 </msupsub>`, ce qui est impossible à faire. Comme `ltx2x` est un parseur, pas un évaluateur, il ne sait traiter les `\ifx`, et autres conditionnelles de TEX.

2.5.3 Fermeture de balises

Supposons que l'on veuille traduire `\subsection{toto}` par `<subsection><sname>toto</sname>`. La question est alors de savoir à quel endroit placer la balise fermante (voir exemple 4 section 2.3.2). Dans `ltx2x`, il est possible de dire que `\subsection` est une commande de sectionnement de niveau 4. Si `\section` est une commande de sectionnement de niveau 3 (donc inférieur), l'appel à `\section` insèrera automatiquement la balise fermante de la sous-section (de façon précise : le code associé à la fin de la commande `\subsection` sera exécuté).

Considérons l'exemple

```
\begin{module}\subsubsection 1 \paragraph 2 \paragraph 3\end{module}
\begin{module}\subsubsection 4 \paragraph 5\end{module}
```

Pour simplifier, nous avons omis les arguments de l'environnement module, et des commandes subsubsection et paragraphe. La traduction sera

```
<module><subsubsection>1 <paragraph>2</paragraph><paragraph>3</module>
<module></paragraph></subsubsection><subsubsection> 4<paragraph>5</module>
...</paragraph></subsubsection>
```

Dans cet exemple les ... correspondent à du texte inconnu (si le deuxième module n'est pas suivi d'une commande de sectionnement, c'est la fin du document qui forcera l'émission des deux balises fermantes). Le document XML ainsi construit n'est pas bien formé : il faudrait qu'un module soit défini comme étant une commande de sectionnement, mais cela n'est pas possible dans `ltx2x`. Pour résoudre le problème, nous allons ruser un peu.

Le principe de base est qu'un module est une sous-section, et que le document principal ne contient pas de `\subsection`, et qu'un module ne peut contenir de commandes de sectionnement de niveau supérieur. Un préprocesseur ajoute `\subsection` devant chaque module. On obtient ainsi

```
<subsection><module><subsubsection>1 <paragraph>2</paragraph><paragraph>3
</module></paragraph></subsubsection></subsection><subsection>
<module><subsubsection> 4<paragraph>5</module>
...</paragraph></subsubsection></subsection>
```

On utilise alors un postprocesseur, qui déplace les `</module>` pour les placer juste devant la `</subsection>` qui suit. Ensuite on supprime les `<subsection>` et les `</subsection>`.

2.5.4 Paragraphes

Il n'y a pas dans TEX de commande de début de paragraphe, juste une commande de fin de paragraphe (explicite sous forme `\par`, ou implicite sous la forme d'une ligne blanche). On se propose de traduire sous la forme `</par><par>` ces changements de paragraphes. Pour obtenir un document bien formé, il faut jongler un peu. En particulier, on va ajouter `</par>` devant chaque balise de sectionnement, et `<par>` derrière chaque balise.

L'exemple suivant

```
\begin{module}\subsubsection 1\par2 \paragraph 3 \par 4\paragraph 5
\par 6\end{module} \par
\begin{module}...
```

donnera :

```
</par><subsection><par></par><module><par>
</par><subsubsection>
<par>1</par><par>2 </par><paragraph><par>3</par><par>4</par>
</paragraph><par></par><paragraph><par>5</par><par>6</par></module>
</par></paragraph><par></par></subsubsection><par></par></subsection>
<par></par><subsection><par></par><module><par>...
```

Compliquons l'exemple comme suit. Le premier module commence par

```
\begin{module}\par \participants\par\motscles\par\glossaire\par
```

où nous avons mis une commande à la place des environnements participants, mots-clés et glossaire. Pour simplifier, on notera par `<participants/>` la traduction de `\participants`. De même que pour les commandes de sectionnement, on va encadrer avec des `</par>` et `<par>`. On obtient alors

```
</par><subsection><par></par><module><par>
</par><par></par><participants/><par>
</par><par></par><motscles/><par>
</par><par></par><glossaire/><par>
</par><par></par><subsubsection>
<par>1</par><par>2 </par><paragraph><par>3</par><par>4</par>
</paragraph><par></par><paragraph><par>5</par><par>6</par></module>
</par></paragraph><par></par></subsubsection><par></par></subsection>
<par></par><subsection><par></par><module><par>...
```

On déplace alors `</module>` comme cela est dit plus haut, on élimine les `<subsection>` et `</subsection>`, on élimine les paragraphes vides. On obtient alors

```
</par>
<module><participants/><motscles/><glossaire/>
  <subsubsection>
    <par>1</par><par>2 </par>
    <paragraph><par>3</par><par>4</par></paragraph>
    <paragraph><par>5</par><par>6</paragraph>
  </subsubsection>
</module><module><par>...
```

Pour que le résultat soit bien formé, il faut supprimer le `</par>` avant le premier module et le `<par>` après le dernier module.

Il s'avère que des `\par` peuvent trainer un peu partout dans le document. Il est donc nécessaire d'encadrer par des balises `</par>`, `<par>` un grand nombre d'autres balises, à savoir `<catperso>`, `<pers>`, `<participants>`, et d'autres. Note : quelle que soit la version du traducteur, il faut supprimer les paragraphes vides, ne pas en créer : une construction du type `\par\par` est fréquente dans TEX.

2.5.5 Autre traitement des paragraphes

La solution donnée précédemment suit le principe général suivant : on rajoute des commandes `\subsection`, on traduit le tout en XML, on déplace les `</module>`, on supprime les balises

`subsection`, et on supprime les paragraphes vides. Cette solution est un peu lourde ; de plus il y a un problème : si la section ne contient pas de sous-section, mais une sous-sous-section, on verra quand même la balise de fin de sous-section. Pour pallier à cet inconvénient, on peut demander au préprocesseur de remplacer les commandes de sectionnement par des débuts d'environnement, et de placer les fins d'environnement au bon endroit (ceci contourne le bug de `ltx2x`). L'algorithme est le suivant : on découpe le texte en morceaux, chaque commande de sectionnement définissant un endoit où couper. À chaque morceau on associe un niveau (1 pour sous-section, 2 pour paragraphe, et 3 pour sous-paragraphe). Pour chaque morceau de niveau 3, on insère la fin de sous-paragraphe, et on colle ce morceau au morceau précédent. Pour chaque morceau de niveau 2, on insère la fin de paragraphe, et on colle ce morceau au morceau précédent. Pour chaque morceau de niveau 1, on insère la fin de sous-section, et on colle ce morceau au morceau précédent. On verra dans le prochain chapitre une autre solution encore plus simple, qui utilise une pile.

2.5.6 Mathématiques

La traduction d'une expression simple comme `$x^2_{a+b}=\ddot y$` est

```
<math>
  <msubsup> <mi>x</mi>
   <mrow> <mi>a</mi> <mo> + </mo> <mi>b</mi> </mrow>
    <mn> 2 </mn> </msubsup>
  <mo> = </mo>
  <mover accent='true'> <mi>y</mi> <mo>&die;</mo></mover></math>
```

(l'élément `&die;` veut dire diérèse, qui est l'une des fonctions du tréma). Faire cette traduction via `ltx2x` est clairement impossible, c'est pour cela que l'on utilise un autre traducteur. Si l'on veut utiliser `ltx2x`, il faut extraire les formules, et les cacher. Une première solution est la suivante. Soit l'exemple :

```
\\$1$ et $$a$$ et \[x\] \\\$
$$a \mbox{b $c$} \mbox{$$d$$}$$
```

Une traduction en XML simplifié est

```
<newline/> <math>1</math> et <math display="true"> a </math>
et <math display="true"> x </math> <newline/> <dollar/>
<math display="true"> a <text> b <math>c</math> </text>
<text><math></math> d<math></math> </text></math>
```

Notons d'abord les points suivants : quand TEX voit deux signes dollars consécutifs, ils les considère comme début de formule hors-texte, sauf si TEX est dans un mode horizontal restreint (voir [4], page 287). Ainsi la formule `\mbox{$$d$$}` est une erreur logique : le d n'est pas en mode mathématiques, il y a par contre deux formules vides de part et d'autre. La seconde difficulté est qu'on peut mettre du texte dans des maths, mais pas de maths dans du texte dans des maths. Nous expliquons dans cette section comment trouver les formules de maths, pas comment les traiter. On utilisera en fait une autre méthode.

L'algorithme de marquage consiste à remplacer d'abord `\\` et `\$` par des codes spéciaux, puis les `$$` et `$` par `\3N` et `\4N`, où N est un nombre. Ceci donne :

```
\10.\41.1\42. et \31.a\32. et \[x\] \10.\20.
\33.a \mbox{b \43.c\44.} \mbox{\34.d\35.}\36.
```

(notons le point qui suit le N, sinon, il y a aurait un problème avec `\411`). On met des délimiteurs ouvrants et fermants en fonction de la parité de N, ce qui donne :

```
\\\(1\) et \[a\] et \[x\] \\\$
\[ \mbox{b \(c\) \mbox{\]d\[}\].
```

La traduction de la deuxième ligne est erronnée, mais nous avons dit plus haut que cela n'était pas grave. Les formules de mathématiques sont alors copiées quelque part, et remplacées par une référence. La résultat de la traduction en XML de la première ligne est

```
<newline/><nomath id=1/> et <nodisplaymath id=2/> et
<nodisplaymath id=3/>  <newline/> <dollar/>
```

Les références sont ensuite remplacées par la traduction de ces formules en MathML. Remarque : la phrase « où N est un nombre » utilise une expression mathématique triviale. Dans ce cas de figure (cas où la formule contient une seule lettre), le préprocesseur remplace `\(x\)` par `\simplemath{x}`, ce qui se traduit par `<simplemath>x</simplemath>` et qui peut être transcrit en HTML sous la forme `<i>x</i>`.

Remarque : comme d'habitude, il faut s'assurer que lorsqu'on remplace une commande par une autre, que cette autre commande n'existe pas déjà dans le document source. Si nous avions remplacé le dollar par `\toto`, on aurait pu avoir un problème avec tous les documents définissant `\toto`.

2.5.7 Les tableaux

Soit le code suivant

```
\newenvironment{trait}{
\hspace*{-.9cm}
\begin{longtable}{|p{\textwidth}}  }{
\end{longtable}}
```

Ce code est extrait d'un rapport. Interpréter ce bout de code est non trivial. En gros, l'environnement `trait` peut contenir un paragraphe. Ce paragraphe est formaté on utilisant la largeur courante (argument `p{\textwidth}`), et il y a un trait à gauche (c'est le `|`). Tout ceci est décalé à gauche de telle sorte que le texte soit justifié avec les autres paragraphes (et le trait se retrouve dans la marge. La traduction d'un tel environnement serait donc

```
<par trait-marge="yes"> le texte </par>
```

Expliquons d'abord ce que fait TEX quand il voit un tableau. L'exemple suivant est tiré du source de TEX.

```
\tabskip 2pt plus 3pt
\halign to 300pt{u1#v1&
                 \tabskip 1pt plus 1fil u2#v2&
                  u3#v3\cr
   a1& \omit &\vrule\cr
   \noalign {\vskip 3pt}
   b1\span b2\cr
   \omit &c2\span \omit\cr}
```

Le résultat est un tableau à trois lignes et trois colonnes. Le `&` sépare deux cellules dans une même ligne, le `\cr` indique la fin d'une ligne. Comme on peut mettre des tableaux dans des tableaux, se pose la question de savoir à qui appartient le `&` ou le `\cr`. La construction `{\ifnum0=‘}\fi` est un moyen très tordu pour mettre `&` dans un groupe, mais pour qu'il soit quand même vu hors du groupe par le formatteur du tableau. Le tableau n'est pas joli, on va le montrer quand même.

u1a1v1 u3lv3
u1b1v1u2b2v2
u2c2v2

La première ligne du tableau est le modèle : pour chaque colonne, il y a une expression, de la forme `u#v`. Ce que va mettre TEX dans le tableau, c'est `uxv` où `x` est la quantité donnée dans le tableau. La commande `\tabskip` précise l'espacement entre les colonnes. Notons que le modèle spécifie une seule valeur, les autres proviennent de la valeur donnée avant le tableau. La commande `\omit` veut dire : utiliser `x` au lieu de `uxv`. La commande `\span` veut dire : prendre le texte des 2 éléments avant et après, les traiter, regrouper le résultat, et mettre cela dans deux cellules. Il y a un `\multispan` qui prend un argument N. Si N est 3, le résultat est le même que `\omit \span \omit\span\omit`. Le `\vrule` permet de mettre un trait vertical de la bonne hauteur. Le `\noalign` permet d'insérer du matériel entre deux lignes.

Ce mécanisme a été jugé trop compliqué pour LATEX, et on a quelque chose de plus simple. En fait, à part les cas les plus simples, le mécanisme de LATEX a la même complexité.

D'abord, on changement de ligne se traduit par `\\` au lieu de `\cr`. Si `\\` a un argument N optionnel, il s'agit d'un espace vertical à insérer entre deux lignes, (donc `\noalign\vskip`). Pour une raison incompréhensible, si l'espace est positif, un insère un `\vrule` de largeur 0, de hauteur 0, profondeur $c + N$, où c est une constante, dans la cellule courante. L'idée est : si un veut des traits verticaux délimitant le tableau, il ne faut surtout pas utiliser `\noalign`, cela interromprait les traits. Il se passe que `\\[3mm]` n'augmente pas de 3 mm l'espace entre deux lignes. Par ailleurs, on veut souvent un trait horizontal, qui est fait par `\hline`. Notons que deux `\hline` consécutifs donne un `\vskip` dans un `\noalign` entre les deux traits. Certains trouvent cela étrange (les traits verticaux sont interrompus).

Le modèle dans LATEX est également différent. On peut dire r, c ou l, si on veut aligner à droite, ou centrer ou aligner à gauche. On peut mettre des barres si on veut un trait vertical. Il n'y a pas de `\span`, ni de `\multispan`, mais un `\mutlicolumn` qui est comme `\multispan`, sauf qu'il y a un argument modèle. Exemple :

```
\begin{tabular}{||r|lc|}
\hline a & b &c\\
\hline\hline
aa &\multicolumn{2}{c|}{bc}\\\hline
\end{tabular}
```

a	b c
aa	bc

Si on oublie le `&` devant le `\multicolumn`, on aura une erreur étrange : misplaced `\omit`. Au lieu de `\tabskip X`, on dira `@{X}`. On peut dire `p{X}` si on veut que la colonne courante soit de largeur X. Une extension permet même de dire `>{u}c<{v}`, là où dans TEX on dirait `\hfil u#v\hfil`.

L'objectif est de traiter ceci :

```
\begin{eqnarray*}
  A&=&B\\
  D&=&E\\
\end{eqnarray*}
```

qui donne ceci :

$$\begin{aligned} A &= B \\ D &= E \end{aligned}$$

Il s'agit d'un tableau à quatre colonnes (la dernière est le numéro d'équation, il n'y en a pas dans ce cas), composé en mode mathématique. L'exemple ne le montre pas, mais la première colonne est alignée à droite, la seconde est centrée, et la dernière alignée à gauche. La traduction XML peut être :

```
<eqnarray star="true">
 <ligne> <elt>A</elt> <elt>=</elt> <elt>B</elt></ligne>
 <ligne> <elt>D</elt> <elt>=</elt> <elt>E</elt></ligne>
</eqnarray>
```

Cette traduction est facile à obtenir, on peut paramétrer `ltx2x` pour que `&` donne `</elt><elt>` et `\\` donne `</elt></ligne> <ligne> <elt>`. Le `\\` final peut poser problème : le post processeur rectifie (ce problème existe même sans utiliser `ltx2x`). Un petit détail : il faut tout composer en mode mathématique, c'est à dire parser les cellules, avant de demander à `ltx2x` de tout reparser ensuite.

Autre exemple

```
\newcommand{\R}{\ensuremath{\mathbb{R}}}
\begin{eqnarray*}
\left\{\begin{array}{lcl}
\dot{x} & = & Ax+g(x,u)\\
 y & = & Cx \\
 \multicolumn{3}{l}{x\in \R^n}
\end{array}
    \right.
\end{eqnarray*}
```

qui donne ceci :

$$\left\{\begin{array}{lcl} \dot{x} & = & Ax+g(x,u) \\ y & = & Cx \\ x \in \mathbb{R}^n \end{array}\right.$$

Comme le montre l'exemple, on ne peut pas faire l'hypothèse qu'un `eqnarray` se termine toujours par des `\\`. Par ailleurs il y a des tableaux dans des tableaux. Le tableau interne se traduit schématiquement comme ceci :

```
<array>
   <modele> <col align="left"/> <col align="center"/> <col align="left"/> </modele>
   <ligne> <elt>A </elt><elt> = </elt><elt> B</elt></ligne>
   <ligne><elt> D </elt><elt> = </elt><elt> E</elt></ligne>
   <ligne< <multicolumn nb="3" align="left"/> C </elt></ligne>
</array>
```

On n'entrera pas trop dans les détails, car `ltx2x` ne fait que la moitié du travail. Il faut aussi un post-traitement, pour supprimer les `<modele>`.

La traduction de la formule est donnée ci-dessous.

```
<math>
 <mtable>
  <mtr>
   <mtd columnalign='right'><mfenced open='{' close='.'>
    <mrow>
     <mtable>
      <mtr>
       <mtd columnalign='left'><mover accent='true'> <mi>x</mi><mo>&dot;</mo>
           </mover></mtd>
       <mtd><mo>=</mo></mtd>
       <mtd columnalign='left'> <mi>A</mi> <mi>x</mi> <mo>+</mo> <mi>g</mi>
              <mo>(</mo> <mi>x</mi> <mo> , </mo> <mi>u</mi> <mo>)</mo> </mtd>
      </mtr>
      <mtr>
       <mtd columnalign='left'><mi>y</mi></mtd>
       <mtd><mo>=</mo></mtd>
       <mtd columnalign='left'> <mi>C</mi> <mi>x</mi></mtd>
      </mtr>
      <mtr>
       <mtd columnspan = '3' columnalign='left'>
        <mrow><mi>x</mi><mo>&Element;</mo><msup><mi>&Ropf;</mi><mi>n</mi></msup>
        </mrow></mtd>
      </mtr>
     </mtable>
    </mrow>
   </mfenced>
  </mtd></mtr>
 </mtable>
</math>
```

Chapitre 3

La version XML

3.1 La DTD

On donnera la liste des éléments de la DTD avec leurs attributs, la liste des entités prédéfinies, et un exemple commenté.

3.1.1 Les éléments

Pour définir les contenus des éléments, on posera les définitions suivantes :
- *mots* : il s'agit de caractères, sans éléments ;
- *texte très restreint* : des mots, et des éléments `<hi>` ;
- *texte restreint* : des mots et des éléments `<anchor>`, `<ref>`, `<xref>`, `<ident>`, `<code>`, `<kw>`, `<hi>`, `<term>` et `<formula>` ;
- *texte général* : idem avec en plus `<cit>`, `<label>`, `<list>`, `<note>`, `<figure>` ou `<table>`.

Certaines valeur d'attributs sont restreintes : les valeurs des attributs `rows` et `cols` doivent être des nombre entiers ; la valeur de l'attribut `target` d'un élément `<ref>` doit être un IDREF, autrement dit, si on dit `target='toto'`, il doit y avoir un élément (et un seul) qui a un attribut `id='toto'`. La valeur d'un attribut `id` doit être un ID, un identificateur unique. Presque tous les éléments peuvent avoir un attribut `id`, nous listerons les éléments pour lesquels cet attribut sert à quelque chose.

Les éléments `table`, `figure` et `formula` ont un attribut qui les rendent en-ligne ou hors-ligne. Si l'élément est hors-ligne il ne peut apparaitre dans la liste indiquée plus haut, ni dans un élément `<p>`.

Les éléments possibles sont les suivants :

Éléments de base Sauf indication contraire, ces éléments contiennent du texte (voir définitions plus haut), mais pas de paragraphe.

`<code>` : contient des *mots*. Cet élément sert en principe à inclure un exemple de programme informatique.

`<ident>` : contient des *mots* (en général un seul). Cet élément sert à mettre un identificateur (un nom de programme par exemple ?).

`<anchor>` : vide. A un attribut `id` obligatoire, qui permet d'utiliser `<ref>`.

`<ref>` : contient du *texte général*. Cet élément a un attribut `target` quasi obligatoire. Si l'attribut `target` vaut `toto`, l'élément crée un lien interne vers l'objet qui a `id='toto'` comme identificateur unique.

`<xref>` : contient du *texte général*. Cet élément permet de faire un lien externe vers une URL définie par la valeur de l'attribut `url`.

`<cit>` : contient un `<ref>`, qui est un lien vers la bibliographie. Il y a un attribut, `rend`, si la valeur est `foot`, il s'agit d'une citation en note de bas de page. Cet attribut est ignoré pour l'instant.

`<note>` : contient du *texte général* ou des paragraphes `<p>`. A un attribut `place`. Le mettre à `foot` crée une note de bas de page. Sinon le texte est mis entre parenthèses.

`<head>` : contient du *texte général*. Sert à mettre un titre dans une section, ou une légende dans une table ou une figure.

`<keywords>` : contient des `<term>`. Permet de faire une liste de mots clés.

`<term>` : contient du *texte restreint* (un mot clé).

`<hi>` Contient du *texte général*. Il y a un attribut `rend`, voir l'exemple plus loin. Le contenu est imprimé dans une police particulière.

`<list>` : Le contenu d'un tel élément est soit une suite d'`<item>`, soit une suite de `<label>` plus `<item>`. Il y a un attribut `type` qui peut prendre l'une des quatre valeurs suivantes : `gloss`, pour faire un glossaire, `simple` pour une liste simple, `ordered` pour une énumération, ou `description` (dans le cas où il y a des `<label>`).

`<label>` : Contient du *texte restreint*. Ceci sert dans les listes.

`<item>` : Contient du *texte restreint* ou des `<p>`. C'est un item dans une liste.

`<table>` : l'élément `<table>` contient un `<head>` optionnel (qui est la légende), et une suite de `<row>`. Cet élément peut contenir un attribut `rend`. Si la valeur est `inline`, il s'agit d'une table dans une table ou une figure ; dans ce cas la table n'est pas numérotée. Dans le cas contraire, on peut mettre un attribut `id` et faire une référence à la table.

`<row>` : cet élément contient une liste de `<cell>`. Il s'agit d'une ligne dans une table. Les attributs possibles sont :

- `top-border`, si vrai les cellules de la ligne courante sont précédées d'un trait horizontal,
- `bottom-border`, si vrai, les cellules de la ligne courante sont suivies d'un trait horizontal.
- `space-before` (ne marche pas...)
- `role`. Si la valeur est `label`, les éléments de la ligne sont en gras.

`<cell>` : cellule de tableau. Contient du *texte général*. Les attributs possibles sont :

- `cols` : indique le nombre de cases du tableau occupé par la cellule, par défaut c'est 1 ;
- `right-border` : si vrai, il y a un trait vertical à droite ;
- `left-border` : si vrai, il y a un trait vertical à gauche ;

- `halign` : peut être `left`, `right` ou `center`. Ceci dit comment le texte est aligné dans la cellule.
- `role` : si la valeur est `label`, le contenu de la cellule est en gras.

`<figure>` : contient un `<head>` optionnel (une légende) et des `<p>`. Les attributs possibles sont les suivants :

- `rend` : la valeur est `inline` ou `array`. Dans le cas où la valeur est `inline`, il s'agit d'une sous figure, elle n'est pas numérotée, on ne peut pas mettre de `id`. Dans le cas où la valeur est `array`, il s'agit d'une figure compliquée (avec des sous-figures dedans).
- `file` : cet attribut doit avoir une valeur, dans tous les cas où l'attribut `rend` n'est pas `array`. C'est le nom de l'image.
- `id` : cet attribut peut être utilisé pour référencer la figure (sauf si la valeur de `rend` est `inline`).
- `width` : cet attribut spécifie la largeur de l'image. De même que pour les attributs qui suivent, il ne sert que si l'attribut `file` est positionné.
- `height` : spécifie la hauteur.
- `scale` : spécifie un facteur d'expansion.
- `angle` : spécifie un angle de rotation.
- `framed` : si vrai, l'image est mise dans un cadre.

`<formula>` : contient `<math>` ou `<simplemath>`. Attributs :

- `type` : le type peut être `inline` (valeur par défaut) ou `display`.
- `id` : si le type est `display`, et cet attribut a une valeur, alors la formule est numérotée, et on peut faire une référence à la formule.

`<math>` : est défini par MathML.

`<simplemath>` : contient des *mots*. Normalement, doit contenir un seul caractère.

Structuration On donne les éléments dans l'ordre d'inclusion, le plus petit d'abord.

`<p>` : contient du *texte général*. Il s'agit d'un paragraphe. Les attributs sont les suivants :

- `noindent` : si `true`, le paragraphe n'est pas indenté.
- `spacebefore` : spécifie l'espacement vertical avec le paragraphe précédent.
- `rend` : si la valeur est `centered`, le paragraphe est centré.

`<div4>` : contient un `<head>`, suivi d'un corps, qui est une suite d'éléments `<cit>`, `<p>`, `<formula>`, `<table>`, `<figure>`, `<list>` ou `<note>` dans un ordre quelconque, en nombre arbitraire. Les formules, tables et figures doivent être hors-ligne. C'est la division de plus bas niveau, numérotée 1.2.3.4.5.

`<div3>` : contient un `<head>`, suivi d'une interface, suivi d'un corps. Le corps est le même que dans le cas de `<div4>`, mais on a le droit à des `<div4>` en plus. L'interface est `<moreinfo>`, `<keywords>` ou `<participants>` (dans un ordre quelconque, en nombre arbitraire). La division est numérotée 1.2.3.4.

`<div2>` : comme `<div3>`, mais le corps peut contenir des `<div3>` ou `<div4>`. La division est numérotée 1.2.3.

`<module>` : comme `<div3>`, mais le corps peut contenir des `<div2>`, `<div3>` ou `<div4>`. Il y a un attribut obligatoire `html`, qui donne le nom de la page `html` à construire. Un module est numéroté 1.2.

Bibliographie Les éléments décrits ici contiennent en général du *texte très restreint*, sauf indication particulière.

`<baddress>` : adresse de la maison d'édition.

`<bauteurs>` : auteurs de l'ouvrage (c'est une suite de `<bpers>`.

`<bbooktitle>` : titre du livre (si la référence est un chapitre de livre)

`<bchapter>` : numéro de chapitre, le cas échéant.

`<bedition>` : numéro d'édition.

`<bediteur>` : le ou les éditeurs (personne qui choisit les textes à mettre dans un livre).

`<bhowpublished>` : précision supplémentaire si le document est publié de façon étrange.

`<binstitution>` : institution (cas d'un rapport technique).

`<bjournal>` : journal dans lequel est paru un article.

`<bmonth>` : mois de publication.

`<bnote>` : note associée à la référence.

`<bnumber>` : numéro (de revue, de rapport, etc).

`<borganization>` : organisateur (d'une conférence par exemple).

`<bpages>` : numéros des pages.

`<bpublisher>` : l'éditeur (société d'édition).

`<bschool>` : école (cas d'une thèse ou rapport de stage).

`<bseries>` : nom d'une série.

`<btitle>` : titre de l'ouvrage.

`<btype>` : type (thèse de doctorat, etc.)

`<bvolume>` : volume.

`<xref>` : (la syntaxe de `<xref>` est donnée plus haut.)

`<byear>` : année de publication.

`<citation>` : contient tout ce qui précède. Chaque élément doit être donné au plus une seule fois. Il y a un attribut `from` qui est l'un de `year` (bibliographie de l'année), `refer` (bibliographie de référence) ou `foot` (autre). Il y a un attribut `id` qui est un label qui permet de faire une référence. Il y a un attribut `key` qui est la clé (valeur affichée par la référence). Il y a aussi un attribut `type` (article, phdthesis, etc).

`<bpers>` : l'élément est vide. Il y a 4 attributs `prenom`, `nom` (obligatoires), `part`, `junior` (facultatifs).

Éléments spécifiques au raweb Ces éléments sont en gros les mêmes que ceux du chapitre précédent.

`<accueil>` : cet élément contient `<theme>`, `<typeprojet>`, `<projet>`, `<projetdeveloppe>`, un élément `<UR>`. Il y a un attribut `html` obligatoire qui est le nom du projet (en lettres minuscules).

`<UR>` : Cet élément contient une liste non vide d'unités de recherche. Il y a 6 unités de recherche : `<URSophia/>`, `<URRocquencourt/>`, `<URRhoneAlpes/>`, `<URRennes/>`, `<URLorraine/>` et `<URFuturs/>`. Ces 6 éléments sont vides.

`<theme>` : contient des *mots*. On suppose que le thème est formé d'un chiffre et d'une lettre. Dans les autres cas, le résultat peut être catastrophique.

`<typeprojet>` : contient des *mots*. Doit être projet, avant-projet, action, ou tout autre mot désignant le type de l'équipe.

`<projet>` : contient du *texte très restreint*, c'est le nom court du projet.

`<projetdeveloppe>` : contient du *texte très restreint*. C'est le nom long du projet, en français.

`<moreinfo>` : contient une suite de `<p>`. On peut mettre cet élément dans l'interface d'un module (ou d'une autre division). On peut également le mettre dans la `<composition>` ou `<raweb>`. Dans ces deux cas, cet élément sera mis dans la section composition de l'équipe.

`<pers>` : contient du *texte très restreint* ou une `<note>`, ou peut être vide. Il y a deux attributs obligatoires `nom` et `prenom`, le nom et le prénom de la personne, le reste contenant des informations sur la-dite personne.

`<catperso>` : contient un `<head>` et une liste de `<pers>`. L'élément `<head>` contient une catégorie de personnel, il est suivi par les personnes de cette catégorie.

`<participants>` : contient une liste de `<pers>`. Idem pour `<participantes>`, `<participante>` et `<participant>`.

`<composition>` : contient une liste de `<catperso>`, éventuellement précédé d'un `<moreinfo>`.

`<presentation>` : contient des modules. La sémantique de cette section, ainsi que des sections qui suivent, est la même que celle du RA 2001.

`<fondements>` : contient des modules.

`<domaine>` : contient des modules.

`<logiciels>` : contient des modules.

`<resultats>` : contient des modules.

`<contrats>` : contient des modules.

`<international>` : contient des modules.

`<diffusion>` : contient des modules.

`<raweb>` : c'est l'élément racine. Il est formé des éléments `<accueil>`, un `<moreinfo>` optionnel, `<composition>`, `<presentation>`, `<fondements>`, `<domaine>`, `<logiciels>`, `<resultats>`, `<contrats>`, `<international>`, `<diffusion>`, `<biblio>`. Certains éléments sont facultatifs, on conseille de tout mettre.

3.1.2 Entités

La DTD définit les entités suivantes : `&ier;`, `&iers;`, `&iere;`, `&ieres;`, `&ieme;`, `&iemes;`, pour dire facilement 1er, 3es, etc. On définit aussi `&numero;` et `&Numero;` pour dire n° et N°. Un grand nombre d'entités sont définies et chargées par MathML.

3.1.3 Exemple

Nous donnons ici un exemple commenté de fichier raweb valide.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<!DOCTYPE raweb SYSTEM 'raweb.dtd' [
   <!ENTITY safir "SAFIR">
   <!ENTITY logo "Logo-INRIA-couleur">
]>
<raweb year='2001'>
```

Remarques : l'année est optionnelle. Dans la DTD locale, on a défini 2 entités, qui seront utilisées dans la suite.

```
<accueil html="safir">
  <theme>2a</theme>
  <typeprojet>Ex-projet</typeprojet>
  <projet>&safir;</projet>
  <projetdeveloppe>Systèmes Algébriques et Formels
     pour l'Industrie et la Recherche</projetdeveloppe>
   <UR> <URSophia/> <URFuturs/> </UR>
</accueil>
```

Il est important de noter que l'attribut `html` de l'élément `<accueil>` soit le nom du projet en lettres minuscules. Normalement, le type de projet commence par une lettre majuscule (le reste est en minuscules). Dans cet exemple, nous faisons comme si le projet était bilocalisé.

```
  <moreinfo>
    <p>SAFIR était un projet commun à l'INRIA, au CNRS et à
       l'université de Nice-Sophia Antipolis, etc.</p>
    <p>Depuis quelques années,
      il sert de mod&egrave;le pour le rapport d'activit&#x000E9;.</p>
  </moreinfo>
```

L'élément `<moreinfo>` est optionnel. Si donné, il va se retrouver en première page du texte. Nous avons dans cet exemple trois façons différentes de mettre des accents : en 8bits iso-latin1, sous forme d'une entité `&egrave`, et sous forme `&#x000E9;`. Il est recommandé de ne mettre que du texte (pas d'images, pas de formules de math, etc) dans cet élément.

```
<composition id="section:composition">
<catperso>
  <head>Responsable scientifique</head>
  <pers prenom="Stephen" nom="Watt">
    Professeur, université de Western Ontario (Canada)</pers>
</catperso>
<catperso>
  <head>Responsable Permanent</head>
  <pers prenom="Manuel" nom="Bronstein">DR</pers>
</catperso>
<catperso>
  <head>Assistante</head>
  <pers prenom="France" nom="Limouzis">TR, à temps partiel dans le projet</pers>
</catperso>
<catperso>
  <head>Personnel Inria</head>
  <pers prenom="José" nom="Grimm"></pers>
  <pers prenom="Yves" nom="Papegay">
       CR, à mi&amp;ndash;temps dans le projet à partir du 1/7/98</pers>
</catperso>
<catperso>
  <head>Ingénieur expert</head>
  <pers prenom="Olivier" nom="Arsac"></pers>
</catperso>
<catperso>
  <head>Personnel UNSA</head>
  <pers prenom="Frédéric" nom="Eyssette">Maître de conférences</pers>
</catperso>
<catperso>
  <head>Chercheur doctorant</head>
  <pers prenom="Raphaël" nom="Bomboy">
      allocataire moniteur normalien, à partir du 1/11/98</pers>
</catperso>
<catperso>
  <head>Chercheur post-doctorant</head>
  <pers prenom="Patrick" nom="Dutto">
     Post Doctorant <hi rend="sc">genie</hi>, à partir du 1/12/98</pers>
</catperso>
<catperso>
  <head>Stagiaire</head>
  <pers prenom="Karine" nom="Berra">
    du 1<hi rend="sup">er</hi> juillet au 1&amp;ier; août 1998
    <note place='foot'>Une note</note>
  </pers>
</catperso>
</composition>
```

Dans la composition de l'équipe, il y a des <catperso>, ils sont classés dans l'ordre suivant : le responsable scientifique, le responsable permanent, l'assistante de projet, le personnel permanent, le personnel temporaire. Cet exemple montre l'utilisation de l'entité –, on s'attend plutôt à voir un tiret normal. À l'intérieur d'un <pers> on peut mettre une note en bas de page, ou utiliser l'élément <hi>. Il y a ici trois utilisations de cet élément (il est implicite dans &ier;), une fois pour dire GENIE, et deux fois pour dire 1er.

```
<presentation id="section:presentation">
<module html="presentation">
 <head>Présentation</head>
  <p>L'objectif pour l'équipe de ce projet est la conception, l'évaluation et
     l'optimisation des algorithmes fondamentaux en informatique, dans les domaines
     de la transmission, du stockage et du traitement de données non numériques.
  </p>
  <div2>  <head>Axes de recherche</head>
  <list>
   <item> Algorithmes de la communication et protocoles pour réseaux, algorithmes
    de bases de données,  algorithmes du calcul formel et du traitement de
    données symboliques, algorithmes de recherche  efficace (hachages
    adaptatifs, recherche multi-dimensionnelle).
   </item>
   <item>La composante évaluation de performance et optimisation fine des
     algorithmes est particulièrement développée, elle repose sur des méthodes
     d'analyse combinatoire et d'analyse asymptotique complexe. </item>
   <item>Développement des outils d'analyse automatique fondés sur le calcul formel.
   </item>
  </list>
  </div2>
  </module>
</presentation>
```

La section <presentation> est formée a priori d'un module (mais il peut y en avoir plus). Dans cet exemple, nous avons mis un module. On s'attend typiquement à ce que ce module soit constitué d'un <p>, suivi de deux ou trois <div2> (axes de recherches, relations internationales, relations industrielles). Nous avons mis une seule division pour simplifier. Cet exemple montre aussi comment mettre une liste d'item. On verra d'autres listes dans la suite. Le module dans cette section est réduit à son strict minimum : pas de participants (cela concerne le projet dans son ensemble), pas de mots clés (mais on pourrait en mettre), et pas de glossaire.

```
  <fondements id="section:fondements">
    <module  html="fonde1">
      <head>Algorithmes en algèbre différentielle</head>
      <keywords>
        <term>algèbre linéaire</term>
        <term>algèbre différentielle</term>
        <term>équations différentielles</term>
      </keywords>
      <list type="gloss"><head>Glossaire</head>
        <label>raweb</label>
        <item>pour « Rapport d'activité sur le web », le sigle à la mode</item>
        <label>XML</label>
```

```
      <item>le langage d'implémentation du raweb</item>
    </list>
```

La section fondements scientifiques contient un certain nombre de modules. On n'y a mis qu'un seul. Cet exemple montre comment introduire des mots clés. Comme cela concerne a priori le projet en entier, on ne met pas de participants. Cet exemple montre aussi l'utilisation d'un glossaire. La suite de la section est :

```
  <p>Une formule de matématiques</p>
  <formula type="display" id='math1'>
     <math><mi>E</mi><mo>=</mo><mrow><mn>m</mn><msup><mi>c</mi><mn>2</mn></msup></mrow>
     </math>
  </formula>
  <p noindent='true'>
   et une autre :
   <formula><math>
    <mrow>
      <msubsup>
        <mo> &int; </mo> <mn> 0 </mn> <mo> &infin; </mo>
      </msubsup>
      <mo form='prefix'> sin </mo>
      <mfenced>
        <mrow> <mi> &alpha; </mi> <mi>x</mi> </mrow>
      </mfenced>
      <mi> d </mi>
      <mi>x</mi>
    </mrow>
   </math></formula>.
 </p>
 <p spacebefore='2cm' rend='centered'> Les fontes mathématiques :
   <formula> <math> <mi mathvariant='monospace'>A</mi><mi>&Aopf;</mi>
      <mi>&Ascr;</mi> <mi mathvariant='sans-serif'>A</mi>
       <mi mathvariant='bold'>A</mi><mi mathvariant='normal'>A</mi><mi>A</mi></math>
   </formula>
 </p>
 </module>
</fondements>
```

Il y a trois paragraphes : le premier n'est pas indenté, car précédé d'une liste. Le second le n'est pas, car il y a un attribut `noindent`. Le troisième est centré (il aurait été indenté sans l'attribut `rend`). Ce paragraphe est nettement séparé du paragraphe qui précède.

Nous avons mis une formule de mathématique simple, et une formule plus compliquée qui est : $\int_0^\infty \sin(\alpha x)dx$. Notons que toute formule a un champ `type`, dont la valeur est `inline` (valeur par défaut) ou `display`. Dans le second cas, la formule sera hors texte, elle sera numérotée s'il y a un champ `id`. Il y a une autre formule qui montre les diverses manières d'écrire la lettre A.

```
  <domaine id="section:domaine">
    <module html="domaine1">
      <head>titre bidon</head>
      <moreinfo> <p> Un moreinfo à titre d'exemple </p>
          <p> Deuxième paragraphe </p>
        </moreinfo>
      <p>  Du texte bidon </p>
```

```
    </module>
  </domaine>
```

La section domaine contient des modules. On a mis un module bidon.

```
  <logiciels id="section:logiciels">
    <module  id="log-raweb" html="logiciel1">
      <head>Le raweb</head>
      <participants>
        <pers prenom="José" nom="Grimm">correspondant</pers>
        <pers prenom="Marie-Pierre" nom="Durollet"></pers>
      </participants>
      <keywords>
        <term>XML</term>
        <term>LaTeX</term>
      </keywords>
      <p>Le script raweb.pl sert à construire le rapport d'activité.
  Il est décrit dans la référence <cit><ref target="cite:JG2001"></ref></cit>.
  La traduction de XML en Pdf est réalisée par <hi rend='tt'>xmltex</hi>, voir
  <cit rend='foot'><ref target="footcite:CGR2000"></ref></cit>
  Le script est disponible sur le
    <xref url="http://www.inria.fr">serveur de l'inria</xref>
 </p>
    </module>
  </logiciels>
```

La section logiciel contient également des modules. On a mis des participants (noter que l'auteur principal est noté correspondant). Il y également des mots clés. Dans cet exemple, il y a trois liens : un vers la bibliographie (en mode normal), un second (en note), et un lien externe vers le serveur WEB de l'inria.

```
<resultats id="section:resultats">
 <module html="res1">
  <head>Le raweb</head>
  <p>Dans ce paragraphe il y a une petite image
     <figure rend="inline" file="&logo;" width='2cm'></figure>.
   Dans la figure <ref target='Logo'></ref> on a la même image, en plus
   grand. Ce module contient la table <ref target='cm'></ref>,
   et fait référence à la formule  <ref target='math1'></ref>.
   Le code qui suit a été généré par raweb.pl décrit dans la section
   <ref target='log-raweb'></ref>.</p>

   <figure id='Logo' rend='array'>
     <head>Deux versions du logo Inria</head>
     <p><table rend='inline'>
         <row>
          <cell> <figure file='&logo;' width="5cm" rend='inline'/></cell>
          <cell> <figure file='&logo;' width="6cm" rend='inline'/></cell>
         </row>
         <row><cell>(a) le logo en 5cm </cell>
              <cell> (b) le logo en 6cm</cell></row>
     </table></p>
   </figure>
```

Cet exemple montre comment insérer des images. Si l'élément `figure` contient l'attribut `rend` avec la valeur `inline`, on insère une image (l'attribut `file` donne le nom, on peut positionner la largeur). Dans le cas contraire, il s'agit d'un flottant, en dehors d'un paragraphe. La légende est dans l'élément `<head>`. Notons l'utilisation d'une table pour aligner les sous-légendes.

```
<table id='cm' rend='frame'><head>Un carré magique d'ordre 3</head>
  <row role='label'>
   <cell/><cell> colonne 1</cell><cell>colonne 2</cell><cell>colonne 3</cell></row>
  <row><cell role='label'>ligne 1</cell><cell> 8</cell><cell>1</cell><cell>6</cell></row>
  <row><cell role='label'>ligne 2</cell><cell>3</cell><cell>5</cell><cell7</cell></row>
  <row><cell role='label'>ligne 3</cell><cell>4</cell><cell>9</cell><cell>2</cell></row>
</table>
```

Ceci montre comment mettre un tableau (il est flottant, en dehors d'un paragraphe). Il y a un cadre autour de chaque cellule. Les cellules de la première ligne et de la première colonne sont en gras.

```
   <table id='cm2'><head>Le carré modifié</head>
    <row top-border='true'>
      <cell left-border='true' right-border='true' halign="right">texte long</cell>
      <cell right-border='true' halign="center">1</cell>
      <cell right-border='true' halign="left">6</cell>
   </row>
   <row top-border='true'>
     <cell left-border='true' right-border='true' halign="right">3</cell>
     <cell right-border='true' halign="center" role='hi'>5</cell>
     <cell right-border='true' halign="left">texte long</cell>
   </row>
   <row bottom-border='true' top-border='true'>
     <cell left-border='true' right-border='true' halign="right" cols='2'>
            4  9</cell>
     <cell right-border='true' halign="left">2</cell>
   </row>
   </table>
 </module>
</resultats>
```

On donne ici une seconde table. Pour chaque cellule, on spécifie l'alignement horizontal. On précise également s'il faut mettre un trait à gauche, à droite, en haut, en bas. Il y a des cellules sur plusieurs colonnes. La cellule du milieu est sur fond grisé.

```
  <contrats id="section:contrats">
    <module  html="contrat1">
      <head>titre bidon</head><p>texte bidon.</p>
    <figure file="&logo;"><head>Le logo version originale.</head></figure>
    </module>
  </contrats>
```

Une autre section, qui contient une image, réduite à son strict minimum.

```
  <international id="section:international">
    <module  html="int1">
      <head>titre bidon</head><p>
    Test des références :
   module <ref target='r1a'></ref> et <ref target='r1b'></ref>
```

```
   sous-section <ref target='r2a'></ref> et <ref target='r2b'></ref>
   paragraphe <ref target='r3a'></ref> et <ref target='r3b'></ref>
   sous-paragraphe <ref target='r3a'></ref> et <ref target='r4b'></ref>
   </p>
  <p> Liens vers les sections :
<ref target='section:composition'/>,
<ref target='section:presentation'/>,
<ref target='section:fondements'/>,
<ref target='section:domaine'/>,
<ref target='section:logiciels'/>,
<ref target='section:resultats'/>,
<ref target='section:contrats'/>,
<ref target='section:international'/> et
<ref target='section:diffusion'/>.
Liens vers des item <ref target='it1'/>, <ref target='it2'/> et
<ref target='it3'/>.
    </p>
    </module>
  </international>
```

Encore une autre section. On a mis des liens vers le contenu de la section suivante. On a également des liens vers l'ensemble de toutes les sections, sauf la bibliographie, et vers les items de l'énumération qui suit.

```
  <diffusion id= "section:diffusion">
    <module  html="diff1" id='r1a'>
      <head id='r1b'>titre de module</head>
      <div2 id='r2a'>
        <head id='r2b'>titre de sous-module</head>
        <div3 id='r3a'>
          <head id='r3b'>Titre de paragraphe</head>
          <div4 id='r4a'>
            <head id='r4b'>Titre de sous paragraphe</head>
            <p>Un paragraphe.
Texte en <hi rend='it'>italique</hi>, en <hi rend='bold'> bold</hi>,
en <hi rend='bold'> <hi rend='it'>italique et gras</hi> </hi>,
en <hi rend='UL'>souligné</hi>,
en <hi rend='sup'>haut</hi>,
en <hi rend='sub'>bas</hi>,
en <hi rend='BO'>italique souligné</hi>,
en <hi rend='sc'>Petites Capitales</hi>,
en <hi rend='small'>plus petit</hi>,
en <hi rend='large'>plus grand</hi>,
en <hi rend='tt'>machine à écrire</hi>,
en <hi rend='courier'>machine à écrire</hi>
et en <hi rend='gothic'> gothique</hi>.
            </p>
            <list type='description'>
              <label>aa</label>
              <item>item 1</item>
              <label>bb</label>
              <item>item 2</item>
              <label>cc</label>
```

```
            <item>item 3
               <list type='ordered'>
                <item id ='it1'>sous-item 1</item>
                <item id ='it2'>sous-item 2</item>
                <item id ='it3'>sous-item 3</item>
              </list>
             </item>
           </list>

         </div4>
       </div3>
     </div2>
   </module>
 </diffusion>
```

Cet exemple montre tous les niveaux de division possibles. Il montre tous les changements de police de caractères possibles. On montre aussi deux types de liste, en plus des listes standard (du type `\itemize` et glossaire).

```
 <biblio>
   <citation key="cle" id="cite:JG2001" type="techreport" from="year">
     <bauteurs>
       <bpers prenom="José" nom="Grimm"/>
     </bauteurs>
     <btitle>Outils pour la manipulation du rapport d'activité</btitle>
     <bnumber>Non encore connu</bnumber>
     <binstitution>Inria</binstitution>
     <byear>2002</byear>
 <xref url="http://www-sop.inria.fr/miaou/Jose.Grimm/raweb/desc.ps.gz"></xref>
   </citation>
<citation from ="foot" key="CGR00" id="footcite:CGR2000" type="inproceedings">
 <bauteurs>
   <bpers prenom="D." part="" nom="Carlisle" junior=""/>
   <bpers prenom="M." part="" nom="Goossens" junior=""/>
   <bpers prenom="S." part="" nom="Rahtz" junior=""/></bauteurs>
  <btitle>De XML à Pdf avec <hi rend='tt'>xmltex</hi> et Passive TeX</btitle>
  <bbooktitle>Cahiers Gutenberg</bbooktitle>
  <bnumber>35-36</bnumber>
  <bpages>79-114</bpages>
  <byear>2000</byear>
  </citation>
</biblio>
```

La bibliographie contient deux références : une référence de l'année, et une référence en note de bas de page. Normalement, on n'est pas censé mettre des références non publiées.

```
</raweb>
```

Ça se termine comme cela.

3.2 Les modifications du script raweb.pl

Le traducteur de LaTeX vers XML existe en trois versions. Dans la première version on utilisait les outils comme `ltx2x`, `recode` et Ω. Dans la deuxième version (utilisée pour traiter le rapport d'activité 2001), on n'utilise plus aucun outil externe. Dans la troisième version, on traite aussi les documents du type rapport de recherche. C'est essentiellement cette version qui sera décrite ici, mais on expliquera les différences par rapport à la version précédente. Il y a une quatrième version, nommée `tralics`, écrite en C++, qu'on ne décrira pas.

Rappelons que le script raweb.pl lit les fichiers sources, vérifie la syntaxe, découpe le texte en modules, et le convertit. Dans le chapitre précédent, nous avons vu comment générer du PostScript et du HTML. On explique ici comment générer du XML. On supposera, pour fixer les idées, que l'on veut convertir le rapport d'activité du projet safir, ou un rapport de recherche de nom desc. Quand on lance LaTeX sur le rapport d'activité 2001 de safir, obtiendra le fichier `safir2001.dvi` (ou les extensions .ps, .pdf, etc). Le script `raweb.pl`, on mode normal, donne lui `safir.dvi` (ou les extensions .ps, .pdf, etc). La conversion en XML donnera `safir.xml`. Pour ne pas écraser les fichiers dvi cités plus haut, la conversion du XML en dvi (ou Pdf) rajoute un préfixe w devant le nom (on aura donc `wsafir.pdf`).

Si on passe l'option `-RR` au script, celui-ci va ignorer la phase de test, et lire le document source. Dans le cas contraire, le texte a été découpé en modules. Dans la version deux, on convertissait chaque module séparément. Dans la version trois, on regroupe l'ensemble des modules (dans des environnements `\RAsection`). En plus des modules, on a une partie qui précède le `\begin{document}`. De cette partie, on extrait les `\nocite`, un `moreinfo` optionnel, et on les rajoute avant le premier module. Il y a aussi le préambule, (nom et type du projet, etc), qui va donner un bout de texte LaTeX à traduire. Le reste de cette partie est ignoré (sauf que les définitions des commandes sont évaluées). L'élément principal est `<raweb>`. Il a un seul attribut, à savoir l'année, quantité qui est extraite du nom du projet. Essentiellement, ceci produit deux choses : des attributs pour l'élément principal, `<RR>`, et un préambule (auteur, titre, résumé, mots clés, etc). Chaque élément du préambule est traduit, et mis de côté, le résultat est inséré dans le document résultat quand on voit `\makeRR` ou `\makeRT`. Dans les deux cas, la structure du XML résultat est la même : il y a un préambule, le corps du document, et la bibliographie.

3.2.1 La bibliographie

Pour chaque fichier bibliographique, on crée un fichier `.aux`, qui contient par exemple :

```
\bibdata{safir_foot2001.bib}
\citation{*}
\bibstyle{rawebxml}
```

Quand on lance bibtex, on obtient un fichier `.bbl` qui contient des quantités de la forme suivante.

```
\citation {GR99a}{cle}{article}
\bauteurs{\bpers\RAo J.\RAb \RAb Garrigue\RAb \RAf \bpers\RAo D.\RAb \RAb
  R{\'e}my\RAb \RAf }
\cititem{btitle}{Extending {ML} with semi-explicit higher-order polymorphism}
\cititem{bjournal}{Journal of Functional Programming}
\cititem{bnumber}{1/2}
\cititem{bvolume}{155}
```

```
\cititem{byear}{1999}
\cititem{bpages}{134--169}
\url{ftp://ftp.inria.fr/INRIA/Projects/cristal/iandc.ps.gz}
\endcitation
```

Pour des raisons trop compliquées à expliquer, il faut substituer `\RRo` par `{`, `\RRf` par `}` et `\RAb` par `}{`. Ainsi `\bpers` est une commande à quatre arguments. Dans l'exemple, les deuxième et quatrièmes sont vides, les autres sont J. et Guarrigue. Une petite manipulation ajoute deux arguments à la commande `\citation`. Le premier est un type, qui peut être `year`, `refer` ou `foot`, le second est vide ou `foot`.

Rappelons qu'il y a trois fichiers de bibliographie pour le rapport d'activité : la bibliographie de l'année, la bibliographie de référence, et la bibliographie en note. On en déduit les arguments supplémentaires de `\citation`. Dans le cas d'un rapport de recherche, il peut y avoir plusieurs fichiers de bibliographie, ils sont tous considérés comme fichiers de l'année. Rappelons les règles : une `\citation` est utile si elle vérifie les propriétés suivantes : soit le type est `foot`, et on a vu `\footcite{cle}`, soit le type est `year`, et on a vu `\cite{cle}`, `\nocite{cle}` ou `\nocite{*}`, soit le type est `refer`. Le même algorithme fonctionne pour les rapports d'activité et de recherche. Si l'on dit `\cite{toto}`, le traducteur va créer un `<ref>` dans un `<cit>`. Le `<cit>` a un attribut `target`, dont la valeur est la clé de citation (précédée par `cite` ou `footcite`), et normalisée (les caractères invalides sont remplacés par un tiret). Si F est cette clé, on mémorise dans `citation{F}` le numéro de la balise `<cit>`. Ceci permet de savoir quelles sont les citations utiles.

Il y a deux difficultés : d'une part `\cite{a+b}` et `\cite{a-b}` sont traduits tous les deux de la même manière. On espère que cela ne se produit pas trop souvent, car cela impose une modification manuelle du source. L'autre difficulté est due à bibtex, qui est semi-insensible à la casse. Autrement dit, on peut dire `\cite{toto}` et définir `TOTO` dans la bibliographie, à condition que cela ne donne pas de conflit. Or, avec notre façon de faire, bibtex ne voit pas de conflit, car on dit `\citation{*}`. C'est donc au traducteur de gérer ce problème. On le fait comme suit. Chaque voit que l'on voit une entrée bibliographique, avec une clé F, on rajoute la valeur (tout ce qui est entre `\citation` et `\endcitation`) dans la table `references{F}`, et on ajoute F dans la table `refaux{f}`, où f est F converti en lettres minuscules. On positionne `bibused{F}` à vrai si la citation est utile.

Une fois lu l'ensemble des fichiers de bibliographie, on considère toutes les références utiles. Soit F une clé. Si `citation{F}` est non vide, il n'y a pas de problème. Sinon, on convertit F en lettres minuscules, ce qui donne f. On regarde `refaux{f}`. Si cette liste est vide, la référence est indéfinie. Si cette liste a plus d'un élément, il y a un conflit. Sinon, soit G l'unique élément de la liste. Ce qu'il faut faire, c'est dire que G est utile, et remplacer F par G partout, en fait dans toutes les balises `<cit>`, et on sait que les balises à modifier sont celles dont le numéro est dans `citation{F}`.

Une fois ce travail effectué, on a la vraie liste des citations à traduire : on construit la liste de tous les `references{F}` pour lesquels `bibused{F}` est vrai, et on traduit ceci en XML.

3.2.2 Découpage en tokens

Rappelons les règles précises utilisées par TEX pour convertir un source en une suite de tokens. Il y a une variable d'état `state`, qui peut prendre trois valeurs : N pour newline, M pour mid-line et S pour skip-blanks. Cette variable gère le traitement des espaces. Chaque fois que TEX lit une

nouvelle ligne dans un fichier, il enlève la marque de fin de ligne et les espaces à la fin de la ligne, puis rajoute sa marque de fin de ligne. Celle-ci peut être vide, mais par défaut c'est le caractère retour chariot. Finalement TEX se place dans l'état N. Note : notre traducteur fonctionne sous Unix, cas où la marque de fin de ligne est le retour à la ligne. Ce que fait notre traducteur, c'est supprimer les `^M` et `^J`, et de mettre `^J` à la place.

Chaque caractère a un `\catcode`, le code de catégorie, qui est un entier entre 0 et 15. Tout caractère de catégorie 15 est invalide, et tout caractère de catégorie 9 est ignoré (on suppose qu'il n'y en a pas). Tout caractère de catégorie 14 est un début de commentaire, on supposera que c'est le caractère `%`. Dans ce cas, TEX passe simplement à la ligne suivante (il va donc se placer dans l'état N, et lire le caractère de la ligne suivante). Quand TEX voit un caractère de catégorie 5, typiquement un retour à la ligne et retour chariot, il passe à la ligne suivante ; cependant, si l'état est N, le résultat est `\par`, si l'état est M, le résultat est un espace, et si l'état est S, le caractère est ignoré, et TEX va lire un caractère de la ligne suivante (dans tous les cas, l'état devient N). Quand TEX voit un caractère de catégorie 10 (typiquement un espace), le résultat dépend de l'état. Si l'état est N ou S, le caractère est ignoré. Si l'état est M, TEX rend un espace, et passe dans l'état S. Ceci a pour conséquence que les espaces en début de ligne sont toujours ignorés (sauf si on change leur `\catcode`), ceux en fin de ligne sont également ignorés, indépendamment du `\catcode`.

Quand `\TeX` voit un caractère de catégorie autre que 0 et 7, il passe à l'état M, et rend le caractère. Un caractère de catégorie 7 (typiquement `^`) indique un début d'exposant. Si deux caractères identiques de catégorie 7 se suivent, TEX contracte ces caractères, et ce qui suit en un seul (par exemple `^^M` donne le caractère de numéro 13). Si cette contraction ne peut pas se faire, TEX passe dans l'état M, et rend le caractère. Si la contraction se fait, c'est comme si TEX avait vu ce caractère, et il recommence. Par exemple, si on dit `^^5e^ab`, TEX va construire le caractère de code 5e (en base 16), ce qui est un chapeau, et on obtient `^^ab`, ce qui donne un guillemet. Cette même construction est utilisée lors de la lecture des macros, et `\^^41bc` est identique à `\Abc`.

Quand TEX voit un caractère de catégorie 0 (typiquement le backslash), il construit une commande. Modulo l'astuce du `^^` expliquée plus haut, si le premier caractère de la commande est une lettre (de catégorie 11), le nom de la commande est formé de tous les caractères de catégorie 11, et l'état est S. Sinon, le nom de la commande est formé d'un seul caractère, et TEX passe dans l'état M (sauf si ce caractère est de catégorie 10, cas où l'état devient S).

Un caractère de catégorie 8 (typiquement le `_`) est une commande indiquant un début d'indice. Un caractère de catégorie 13 est un caractère actif (typiquement c'est le `~`, suivant les options les lettres accentuées sont des caractères actifs). Un caractère de catégorie 1 (typiquement `{`) désigne un début de groupe, un caractère de catégorie 2 (typiquement `}`) désigne une fin de groupe, un caractère de catégorie 3 (typiquement `$`) désigne un changement de mode mathématique. Un caractère de catégorie 4 (typiquement le `&`) est utilisé dans les tableaux, et un caractère de catégorie 6 (typiquement le #) est utilisé dans les tableaux et définitions. Les autres caractères ne sont pas associés à une commande : si on dit 'a=b', TEX positionne les trois caractères dans le résultat. Cependant les lettres ont par défaut la catégorie 11, et les autres, la catégorie 12. La seule différence concerne les noms de commandes.

3.2.3 Valeur d'un token

L'exemple suivant montre une subtilité des `\catcode`.

```
\dimen0 = 12pt
\def\strip@pt#1pt{#1}
```

Lorsque TEX lit une dimension, il lit un nombre (ici 12), puis une unité (ici pt). Dans le cas où le caractère p est actif, il y a une commande associée, et le p sera expansé. Sinon, le `\catcode` du p n'a aucune importance (bien sûr, si le p est un caractère invalide, ignoré, début de commentaire, ou début de commande, le lecteur ne rendra pas de token p). Une fois qu'on a mis 12pt dans `\dimen0`, on peut récupérer cette valeur via `\the\dimen0`. Ceci va rendre `12pt`, une liste de 4 tokens, chaque token étant un caractère de catégorie 12. On ne peut pas appliquer `\strip@pt` à ce résultat : en effet, pour que `\strip@pt` soit une commande valide, il faut que les caractères p et t soient de catégorie 11 (lettre), et donc les `pt` après le `#1` sont de catégorie 11, qui ne vont pas matcher les `pt` de catégorie 12. Il faut utiliser de la magie noire (i.e. `\uppercase`) pour avoir une définition de `\strip@pt` qui fonctionne. Ce que montre cet exemple, c'est que TEX mémorise le caractère et son code de catégorie. Un token sera donc, soit un nom de commande, soit une paire caractère plus catégorie.

Considérons l'exemple suivant

```
\def\foo#1{#1x#1} \def\bar#1{ $#1$}
\expandafter\foo\bar y
\foo\bar y
```

Lorsque TEX lit la première ligne, il construit les tokens suivants : `\def`, `\foo`, `#`, 1, accolade, `#`, 1, x, `#`, 1, accolade. Il va définir la commande `\foo`, en y recopiant essentiellement tout ce qui suit, mais les `#` seront traités de façon spéciale : `#1` devient un unique token, référence au premier argument, tandis que `##` devient un unique token (il n'y a pas de `##` dans l'exemple). La première ligne contient une définition de `\bar`. Il y a également deux espaces (qui risquent de donner deux espaces dans le résultat).

Expliquons la deuxième ligne. Le `\expandafter` force la lecture de `\foo` et `\bar`. L'état est S, il reste un espace et le y non lus. L'expansion de `\bar` force la lecture de son argument, c'est y (notons que l'espace devant le y disparait car on est dans l'état S). Après avoir expansé `\bar`, les tokens lus sont donc : `\foo`, espace, dollar, y, dollar. L'argument de `\foo` est le dollar (l'espace est ignoré, car c'est un argument normal), et le résultat de l'expansion est donc : dollar, x, dollar, y et dollar. Comme il y a un nombre impair des dollars, ceci va faire une erreur.

Considérons maintenant la dernière ligne. L'argument de `\foo` est `\bar`, l'espace qui suit le `\bar` n'est pas lu, mais TEX est dans l'état S. L'expansion de `\foo` donne les tokens `\bar`, x et `\bar`. La première expansion de `\bar` donne : espace, dollar, x, dollar. La second expansion va lire un token, donc lire l'espace (et l'ignorer), et construire un token avec y. L'expansion sera espace, dollar, y, dollar. Au final, ceci donne deux formules de mathématiques, avec un espace devant chacune.

Le problème qui se présente est que l'on n'a pas envie de construire des tokens : on voudrait faire de la substitution simple dans le texte. Le logiciel `latex2html` essaie de faire attention à la gestion des espaces (il insère des espaces après les noms des macros si ce qui suit est une lettre) Dans certains cas, il se trompe ; en particulier ici : il se plaint de ne pas connaitre `\barx`. Le logiciel hévéa ne fonctionne pas mieux dans ce cas : Parsing of argument failed.

Dans la version deux du traducteur, nous avons résolu le problème des espaces de la façon suivante : on remplace tous les `\foo` (éventuellement suivi d'espaces par `\1foo!`. Les trois tokens les plus utiles pour analyser le texte (le signe dollar et les accolades), sont remplacés par `\21!`, `\22!` et `\23!`, les caractères spéciaux par `\3#!`, `\3&!`, `\3<!`, `\3>!` et `\3~!`. Il y a deux exceptions `\00!` et `\01!` désignent `\\` et `\!`. Par conséquent, tout token spécial est de la forme : un backslash, un chiffre, du texte, un point d'exclamation, le texte ne contenant ni backslash ni point d'exclamation. Il faut utiliser un prétraitement pour tenir compte des changement de `\catcode` (i.e. du texte en verbatim). Dans la version actuelle, le texte est découpé en une suite de tokens, comme dans TEX, le token `\foo` donne `\foo`, tout simplement.

On va maintenant expliquer comment TEX manipule les tokens. D'abord, un token est défini par trois quantités A, B et C, il y a une relation entre ces trois quantités. Dans le cas où le token est un caractère, la valeur B est le `\catcode`, la valeur C est le numéro du caractère. Les quantités B et C ont plusieurs noms possibles, par exemple type et valeur ou type et sous-type. Dans le cas d'un caractère, on parle plutôt de type et valeur (si le type, i.e. le `\catcode` est 1, le caractère se comporte comme début de groupe, indépendemment de la valeur, si le `\catcode` est 11, l'action associée est « mettre la valeur dans le dvi »). On a $A = 2^8B + C$.

Posons $E = 2^{12}$. On a $A < E$ pour tout token qui est un caractère non actif. Dans le cas d'un caractère actif, de code x, on a $A = E + x$. Posons $F = E + 256$. Si on a $A < F$, il s'agit d'un caractère (éventuellement actif), et A modulo 256 est la valeur du caractère. Ce que fait `\lowercase`, c'est essentiellement la chose suivante : pour tout token dans la liste, si $A \geq F$, ne rien faire, sinon calculer A modulo 256, ce qui donne x. On regarde la valeur de x dans la table `lccode`. Si la valeur est 0, on ne fait rien, sinon on remplace A par $A - y$. Compris ? Les autres tokens sont de la forme `\foo`. Ce que fait TEX c'est calculer la position dans la table de hash, disons n. On aura dans ce cas $A = F + n$. Si la commande a un caractère, la position est le code ASCII de ce caractère (ce que fait TEX dans le cas de `\catcode ‘\$=3`, c'est prendre la valeur A de `\$`, et de soustraire F). Dans le cas contraire, la position de la commande dans la table de hash est calculée via un hashcode. Il existe une valeur maximale pour n, ce qui va donner $A < G$ pour un certain G.

Il y a dans TEX une grosse table, appelée la table des équivalents, `eqtb`. Elle est divisée en 6 régions. Pour chaque token A avec $A \geq E$, la valeur du token est là-dedans. Si $E \leq A < F$, c'est la valeur d'un caractère actif, si $F \leq A < G$ c'est la valeur d'une commande, et si $A \geq G$, c'est autre chose. Il y a là dedans la table des `\catcode`, la table des `\lccode`, la table de `\count`, etc. Pratiquement toutes les variables s'y trouvent. La valeur d'une variable est formée de 3 champs : L, B, et C. En règle générale, L est un entier 8 bits, B un entier 8 bits, et C un entier 16bits : ça tient sur un entier 32bits. Dans le cas de `\count0` par exemple, il faut 32bits pour représenter le nombre, et la valeur est codée sur 2 mots (la valeur dans un mot, L et B dans un autre). Pour des objets plus grands, la valeur est un pointeur vers la mémoire principale.

Expliquons ce qui se passe quand on dit `\count3=17`. Le lecteur va rendre un token A pour la commande `\count`. L'expanseur va regarder ce qu'il y a dans `eqtb` en position A. Si `\count` n'a pas été redéfini, les valeurs de L, B et C seront 1, 90, et 0. Le macro-expanseur ne va rien faire. L'évaluateur, lui va regarder le type (ici 90, qui signifie `\count`, `\dimen`, `\skip` ou `\muskip`). Il regarde s'il y a un `\global` qui précède (il n'y en a pas dans ce cas). Puis un entier sera lu, ici 3. Ensuite TEX regarde à nouveau le type et le sous-type, pour savoir ce qu'il faut faire. En particulier, il faut remplir le compteur 3, c'est une adresse dans `eqtb`, qui est obtenue en ajoutant 3 à l'adresse du premier registre. Soit A' cette adresse. Le sous-type va demander de lire un entier

(précédé par un signe égal optionnel). Ici cet entier sera 17. Il s'agit donc de mettre localement (car il n'y a pas de `\global`) la valeur 17 dans `eqtb` à la position A'. À cette position il y a L', B' et C'. Il faut remplacer le C' par 17. On peut maintenant expliquer à quoi sert la valeur L ou L' c'est le niveau de définition. Si on est dans un groupe d'ordre N, le niveau courant est $N+1$ (le niveau 1 est en dehors de tout groupe, le niveau 2 dans un groupe, le niveau 3 dans un groupe dans un groupe, etc.). Si L' est le niveau courant, rien ne se passe (la valeur à sauvegarder est déjà dans la pile). Si L' est 0, on définit un objet qui était indéfini, en sortie du groupe, il faut rendre cet objet à nouveau indéfini. Dans les autres cas, il faut restaurer l'ancienne valeur. Dans ce deux cas, TEX met quelque chose dans la pile, et ce quelque chose sera dépilé lors de la sortie du groupe courant ; la valeur de L' sera remplacée par la valeur du niveau courant. Dans la cas où la définition est globale, il n'y a pas de sauvegarde, et le nouveau L' sera 1. Notons que si on est dans un groupe, et que l'on fasse $2N$ affections, en alternant les définitions locales et globales, ceci va empiler N valeurs. Notre traducteur ne fait pas ces sauvegardes, cela prendrait trop de temps. Par contre `tralics` traite ce problème correctement.

Dans l'exemple qui précède, on n'a pas utilisé la valeur B' : c'est la position dans la table qui donne le type (ici, c'est un entier). Si on dit `\def\foo#1{}`, le principe est identique : on calcule l'adresse A' de `\foo`, et on remplit C' avec un pointeur vers le corps de la définition (dans le cas de `\let\bar\foo`, on copie juste un pointeur, et on met à jour un compteur de référence). L'ancienne définition est sauvegardée. La valeur de B' est un code (par exemple 111), qui dit que c'est une commande utilisateur (ce code peut être différent, parce que `\def` peut être précédé de `\long` ou `\outer`).

Si on veut un traducteur qui fonctionne dans tous les cas, et qui donne les mêmes résultats que TEX, il faut, d'une manière ou d'une autre, implémenter ce mécanisme. En particulier, si on dit `\let\foo\bar`, il faut que `\foo` reprenne son ancienne définition en sortie du groupe. Si `\bar` est une commande utilisateur, cela est facile. Sinon, c'est plus compliqué (si on savait que `\bar` n'est pas redéfini, on pourrait implémenter cela comme `\def\foo{\bar}`, cela fonctionne presque toujours). Il y a deux cas de figure : `\let\foo={`, suivi d'un changement de `\catcode` de l'accolade, et `\let\foo\bar` suivi d'une redéfinition de `\bar`.

Une réponse partielle à la question est la suivante. La valeur d'un token qui est un caractère est ce caractère, si son `\catcode` est standard, sinon c'est `\\-cat-x-y`, où x est le `\catcode`, et y est le caractère. La valeur de `\foo` est `\foo`. Dans le cas où on dit `\let\foo\bar`, et `\bar` est une primitive, la valeur sera `\\-let-bar-N`, où N est le code interne de la commande, et si `\bar` n'est pas définie, cela sera `\\-let-bar`. Les codes internes utilisés par notre traducteur n'ont rien à voir avec ceux de TEX. On s'en sert pour accélérer la traduction. Quand on crée un token `\\-let-bar-N`, et qu'on l'évalue, on se sert uniquement de la valeur de N. Mais en cas d'erreur, ou dans le cas de `\string`, il faut bien récupérer le nom de la commande. On pourrait faire comme TEX, à savoir, pour toute primitive, du type `\par`, entrer deux fois la chaîne : une fois dans la table de hash, et une fois dans le code de la commande `print` (qui est utilisée pour imprimer un nom de commande, et également par `\string`). C'est une idée intéressante, mais abandonée : la bonne structure de donnée est une liste de tokens, et on essaie de la représenter sous forme de chaine de caractères, au prétexte que, la liste contenant essentiellement des caractères, une chaine est plus efficace. La *raison d'être* du traducteur `tralics`, un traducteur sans pattern matching, écrit en C++, est celle-ci, à savoir implémenter proprement les définitions, y compris les `\let`.

3.2.4 Traduction du verbatim

Dans la section 1.3, il y a un bout de texte de la forme `\738!x!`. Quand cette section a été écrite, on utilisait ce genre de construction pour coder `\begin{x}`. Maintenant cela ne sert plus, mais a fait planter notre traducteur, lors de la première tentative de traduction de ce document.

Il y a en fait deux mécanismes qui entrent en jeu. D'une part, il s'agit de lire un texte sans interprétation, d'autre part, il faut générer du code XML qui donne, comme résultat, ce qu'il y a dans le source. Le paragraphe qui contient l'expression `\738!x!` contient une remarque : la traduction ne marche pas si le texte contient des quantités du type `\3`. Naïvement, on pensait que ne pas utiliser la macro `\3` était suffisant. Dans la version 3 du traducteur, cette restriction disparait : il est parfaitement possible de définir la commande `\3`. Le hic, est que le texte XML *généré par le traducteur* ne doit pas contenir de backslash suivi de chiffre. Cette restriction n'existe pas dans `tralics`, qui ne fait pas de pattern matching dans le XML généré.

On s'en sort en traduisant les backslash sous la forme `&#92;`. Bien entendu, il faut traduire < et & sous la forme `\<` et `&`.

La traduction du verbatim n'est pas trop compliquée : si on voit `{`, `}`, `%`, `$`, `&`, `\`, on rend `\{`, `\}`, etc. (en espérant que ces commandes ne soint pas redéfinies, problème qui ne se pose plus dans `tralics`). Si on voit `'`, `‘`, `-`, `<`, `>` ou `~`, on rend `\string'`, `\string‘`, etc. La raison fondamentale est la suivante : si on traduit `\verb+--+` par `--`, et que l'on passe ceci à TEX pour générer du Pdf, il va remplacer les deux signes moins consécutifs par un tiret long. La traduction effective contient un espace de longueur 0 entre les deux caractères. Finalement la traduction de l'espace dépend du contexte : cela peut être un espace insécable, ou un ␣.

Une commande du type `\verb` va construire une liste de tokens, et mettre cette liste dans un `\texttt`. Dans le cas d'un environnement `verbatim`, chaque changement de ligne donne un `\par\noindent`. Il y a une option qui permet de numéroter les lignes.

On fait également quelque chose dans le cas de `\DefineShortVerb{\+}` : après cette commande, on peut dire `+toto+` au lieu de `\verb+toto+`, ce qui rend nettement plus simple la rédaction d'un document comme celui-ci. On traite aussi le cas de `\SaveVerb{xx}+\toto+`, et `\UseVerb{xx}`. Ces commandes ne sont pas standard LATEX mais définies dans le package `fancyvrb`. La traduction est la suivante : on interprète la définition comme si on avait vu `\catcode‘\+=16`. Autrement dit, on invente un nouveau type de code de catégorie. Quand on voit un + de catégorie 16, c'est comme si on avait vu `\verb+` (autrement dit, le lecteur, s'il voit un caractère de catégorie 16, remet le caractère dans le flux, et rend `\verb`). Ce que fait `\SaveVerb` n'est pas compliqué : il lit le `xx`, le token qui suit, et expanse le `\verb`. Le résultat est mémorisé quelque part, et utilisé par `\UseVerb`.

3.2.5 Préliminaires

On crée dans le répertoire courant le fichier `wsafir.tex`, qui contient :

```
\def\xmlfile{safir.fo}
\def\LastDeclaredEncoding{T1}
\input{xmltex.tex}
\end{document}
```

Si on lance latex (ou pdflatex) sur ce fichier, il va convertir le fichier `safir.fo` en dvi (ou en Pdf).

On crée dans le répertoire courant, un lien vers le fichier `fotex.cfg` qui contient les lignes suivantes :

```
\frenchspacing
\selectlanguage{french}
\DeclareNamespace{fo}{http://www.w3.org/1999/XSL/Format}
\XMLelement{fo:RATHEME}
  {}{\xmlgrab}{\foratheme{#1}}
\XMLelement{fo:INRIA}
  {} {}{\foinria}
```

et un fichier `wsafir.cfg`, un lien vers le fichier `raweb-cfg.sty`. Ces deux fichiers vont être lus automatiquement lorsque le résultat XML, traduit en fotex, sera traduit en Pdf. Le premier fichier définit deux balises, le second explique à TEX comment traiter ces balises (en fait, c'est juste une indirection). La justification de ces fichiers est expliquée en section 1.4.4. Il y a un autre point important : c'est expliquer à TEX que le document est en français.

3.2.6 Tables de translitérations

On a besoin de trois tables pour la suite. Par exemple, pour traduire :

```
Le $\lambda$ calcul est de la 1$^{iere}$ importance \'{a} l'Inria
```

ces trois tables sont utilisées. La première explique comment traduire en MathML des formules triviales (lettres grecques, etc). La seconde table sert à traduire les accents. Nous avons initialement pensé à utiliser `recode` pour traduire les lettres accentuées en caractères 8 bits, mais cela n'est pas suffisant. Ainsi pour traiter le cas de `\'c`, `\c s`, qui n'existent pas dans la norme iso-8859-1, mais dans la norme Unicode sous la forme `&cacute;`, `&scedil;`, nous avons décidé de traiter tous les accents. Notons également, au contraire de LATEX, que `recode` ne comprend pas la syntaxe `\a'e`. La dernière table est utilisée pour corriger un certain nombre d'horreurs typographiques (voir table 3.1), et servait à corriger certaines déficiences de Ω ; il y a un bout de pattern matching pour réduire la taille de la table.

3.2.7 Prétraitement

Le prétraitement consiste en quatre étapes : d'abord on s'occupe des environnements verbatim, il y a une phase de mise aux normes, la gestion des espaces, puis le balisage.

La phase de mise aux normes est faite par le prétraitement, mais pourrait également l'être par le post-traitement. Il y a certains cas où ce traitement est douteux, du point de vue sémantique, mais, qu'on le fasse avant ou après, le résultat est le même. D'abord on remplace `---` et `--` par `\entity{mdash}` et `\entity{ndash}`. Notons que `-` n'est pas une macro : il s'agit de ligatures (quelque chose qui dépend de la police de caractère courante). Cette transformation est a priori invalide en mode mathématique. On supposera également que personne n'utilise `\--` (ceci autorise une coupure de mot avant le tiret). On remplace tous les ` `` ` et `''` par des « et », et on remplace tous les << et >> par des « et ». On supposera donc que personne n'utilise `\<<` (une vieille manière de mettre des guillemets), ni `\''` (ce qui met un accent sur une apostrophe). Remarquons que `x''` sera traduit comme `$x»$`, ce que TEX risque de comprendre comme $x\check{}$. Pour résoudre ce problème, la phase de traitement des mathématiques remplace les guillemets fermants par des `''`. Finalement, on insère un espace insécable derrière un guillement ouvrant, sauf s'il y a déja un espace (insécable ou non).

source	résultat	modifications
`X$^{er}$`	X^{er}	X^{er}
`X$^e$`	X^{e}	X^{e}
`X$^{ieme}$`	X^{ieme}	X^{e}
`X$\&$`	X&	X&
`$3^{\mbox{ièmes}}$`	$3^{\text{ièmes}}$	3^{es}
`$61^e$`	61^{e}	61^{e}
`X${}^\star$`	$X^{\star}$	X^{*}
`$5^{e}$`	5^{e}	5^{e}
`X$^{th}$`	X^{th}	X^{th}
`X$^{-8}$`	X^{-8}	X^{-8}
`$6^{th}$`	6^{th}	6^{th}
`${2}^{\mbox{\scriptsize nd}}$`	2^{nd}	2^{nd}
`${6}^{\mbox{\scriptsize ème}}$`	$6^{\text{ème}}$	6^{e}
`X$^{\grave{e}me}$`	$X^{\grave{e}me}$	X^{e}
`X$^{\grave{e}re}$`	$X^{\grave{e}re}$	X^{re}
`$27^{\mbox{ème}}$`	$27^{\text{ème}}$	27^{e}
`$25^{\mbox{ème}}$`	$25^{\text{ème}}$	25^{e}
`X$^{\textstyle ème}$`	X^{me}	X^{e}
`$\Sigma{}^{{\rm it}}$`	Σ^{it}	Σ^{it}
`X$^{\text{ème}}$`	$X^{\text{ème}}$	X^{e}
`X$^{\mbox {ème}}$`	$X^{\text{ème}}$	X^{e}
`$1^{\text {ère}}$`	$1^{\text{ère}}$	1^{re}
`$^{133}$X`	${}^{133}X$	${}^{133}X$
`$^{99}$X`	${}^{99}X$	${}^{99}X$
`N$^o$`	N^{o}	N^{o}
`$\ddot {\textrm {o}}$`	ö	ö
`$\ddot {\rm o}$`	ö	ö
`X$^{\rm e}$`	X^{e}	X^{e}
`$3^{e}$`	3^{e}	3^{e}
`$14^{th}$`	14^{th}	14^{th}

TAB. 3.1 – Table des formules spéciales. Elle associe à une formule A une expression B. On montre la valeur de la formule, (avec les dollars autour), la traduction LaTeX et la traduction XML. On a parfois mis un X devant la formule pour montrer l'espacement vertical.

Dans la version 3 du traducteur, il n'y a plus de balisage, et dans `tralics` il n'y a plus de prétraitement.

3.2.8 Traitement des mathématiques, partie 1

L'un des problèmes avec Ω est qu'il ne sait pas traiter le cas de `\dot x'^2_3`. Notre traducteur donne :

```
<msubsup> <mover accent='true'> <mi>x</mi> <mo> &dot;</mo></mover>
   <mn> 3 </mn>
   <mrow> <mo> &prime;</mo> <mn> 2 </mn></mrow>
</msubsup>
```

On explique ici comment cela fonctionne. D'abord, il y a une grosse table. Elle associe à une lettre x la quantité `<mi>x</mi>`, à un chiffre 2 la quantité `<mn> 2 </mn>`, à un opérateur, type + la quantité `<mo>+</mo>`. Cette table contient les lettres grecques, des symboles divers et variés (il y en a beaucoup). On peut aussi trouver des commandes spéciales : par exemple _ et ^ (codés tels que), le prime (codé sous la forme `prime`), la commande `\dot` codée sous la forme `accent &dot;`, la commande `\frac`, codée sous la forme `cmd frac`.

On appellera dans la suite noyau toute expression XML valide (i.e. tout sauf _, ^, `prime`, `accent xxx` et `cmd yyy`). Il y a une précédure `mk_kernel` qui prend 5 arguments x, a, p, i et e (un noyau, un accent, un nombre de primes, un indice, un exposant). La procédure rend un noyau. Absolument tout (sauf le noyau) peut être vide. S'il y a un accent, on commence par le mettre sur le noyau. S'il y a un ou des primes, on le rajoute à l'exposant (s'il n'y a pas d'exposant, le prime devient l'exposant, sinon on crée un `<mrow>`, voir exemple). On supposera qu'il y a au plus 4 primes, et on utilise `&prime;`, `&Prime;`, `&tprime;` ou `&qprime;`. S'il y a des indices ou des exposants, le résultat sera un `<msub>`, `<msup>` ou `<msubsup>`.

Il y a une procédure `translate1` qui appelle `mk_kernel` pour tous les noyaux. Cette procédure considère une suite de tokens, et la balaie de gauche à droite. Chaque fois qu'on voit un noyau x, éventuellement précédé d'un accent a (de la forme `acc xxx`), on regarde s'il y a des `prime` derrière (on les compte, disons qu'il y en a p), ou un _ suivi de i, ou un ^ suivi de e, on appelle `mk_kernel`. Tous ces tokens seront remplacés par le résultat de l'appel. Il peut y avoir des erreurs (par exemple `\dot \frac 12`, accent suivi d'une commande, ou `x ^\frac12`, chapeau non suivi d'un noyau). Les commandes sont laissées dans le texte. Notons que `x^2^3` est une erreur, car le `x^2` est bien convertit en noyau, mais ce noyau n'est pas considéré quand on regarde la suite. Par contre `x^{2^3}` et `{x^2}^3` sont acceptés, car l'expression entre accolades est un noyau. Notons que `$^e$` est également une erreur, mais est une exception dans la table 3.1.

Il y a une procédure `translate2`, qui est appellée juste après. Elle suppose qu'il n'y a pas d'accents, de prime, de souligné, de chapeau. Elle traite les commandes, de gauche à droite comme plus haut.

- Cas de `\frac a b` : le résultat est `<mfrac> a b </mfrac>`.
- Cas de `\stackrel a b` : le résultat est `<mover> a b </mover>`.
- Cas de `\underset a b` : le résultat est `<munder> b a </munder>`.
- Cas de `\overline a` : le résultat est `<mover accent='true> a <mo>&OverBar;</mo> </mover>`.
- Cas de `\root a b` : le résultat est `<mroot> a b </mroot>`.
- Cas de `\sqrt a` : le résultat est `<msqrt>a</msqrt>`.

Notons que `\frac \sqrt a 1` est une erreur.

Il y une procédure qui finit le travail : la liste des tokens est concaténée, et en général mise dans une `<mrow>`. Exception : si la liste a un seul terme, ou si on est en mode externe (au toplevel). La procédure `tokenize` prend en argument une expression mathématique, la convertit en liste de tokens, applique les procédures précédentes.

3.2.9 Traitement des mathématiques, partie 2

Comme vu plus haut, on sait traiter une expression mathématique à condition de pouvoir la convertir en une liste de tokens. Pour ce faire, on ajoute une accolade fermante à la fin du texte (elle sert de délimiteur), et on appelle `next_token` qui rend le token suivant, pour ainsi trouver les tokens les uns après les autres. Si on voit une commande, disons `\foo`, on regarde dans la table, si elle n'y est pas, c'est une erreur. Si c'est un un espace, il est ignoré, si c'est une autre caractère, on regarde dans la table, s'il n'y est pas c'est une erreur. Si on voit une accolade ouvrante, on la lit, et on appelle `tokenize` (vive la récursion). Dans le cas de `\frac{1}{a + b}`, il va se passer la chose suivante : il y a trois tokens, la commande `frac`, et les deux expressions entre accolades. Rappelons ce que fait `tokenize` : cette procédure calcule d'abord la liste des tokens. Pour la première accolade, il y a un seul token `<mn>1</mn>`, et pour la seconde, il y en a trois, `<mi>a</mi>`, `<mo>+</mo>` et `<mi>b</mi>`. On appelle `translate1` et `translate2` sur ces deux listes, dans ce cas il n'y a rien à faire. Finalement, la liste des tokens est remplacée par un token unique, dans le premier cas, la liste contient un token, il n'y a rien à faire, dans l'autre cas, il y a trois tokens, on les met dans un `<mrow>`. La traduction de l'expression est donc, au final : `<mfrac><mn> 1 </mn> <mrow> <mi>a</mi> <mo> + </mo> <mi>b</mi> </mrow></mfrac>`

Il y a des exceptions. L'un des problèmes est que les gens ne respectent pas la syntaxe : on peut voir `${\mathbb X}$` au lieu de `$\mathbb{X}$`, des fois il n'y a pas d'accolades du tout. On essaie de traiter à peu près tous les cas. La traduction de `\mathcal{X}` est `<mi>&Xscr;</mi>`, celle de `\mathbb{X}` est `<mi>&Xopf;</mi>`. Si dans un `\mathcal` ou `\mathbb`, il y a plusieurs lettres, on traduit chaque lettre, et on met le tout dans un `<mrow>`. S'il y a autre chose que des lettres majuscules, (disons lettres minuscules et chiffres), ces caractères sont traduits normalement. La traduction de `\mathsf{X}`, `\mathtt{X}`, `\mathbf{X}` est `<mi font-variant='xx'>X</mi>` où la valeur de l'attribut est respectivement sans-serif, monospace et bold. Ici l'argument X peut être une lettre ou une suite de lettres. Dans le cas de `\it`, `\textit` ou `\mathit`, on ne fait rien.

Il y a un certain nombre de commandes qui sont ignorées, par exemple `\nonumber`, `\nolinebreak`, `\displaystyle`, `\mathbin`, `\limits` et d'autres. Note : dans le rapport 2001, il y a trois occurences de la commande `\limits`, dans deux cas sur trois, le `\limits` ne sert à rien. Il faut traiter de façon spéciale les caractères <, > et ~ (qui sont actifs). Par ailleurs la commande `\ensuremath` est ignorée (on est en mode mathématique).

Il y a des commandes non implémentées : en particulier `\big` et ses variantes.

La procédure principale est `cv_math`. Elle reçoit un argument x et le traite. La valeur de l'argument, et le résultat est imprimée dans le fichier de trace. En cas d'erreur, la procédure rend l'élément `</error>`. À l'origine, cette procédure copiait son argument dans un fichier pour être traité par Ω. Note : dans le cas de `$x''--$`, le préprocesseur remplace les double quote par des guillemets, et le double tiret par un tiret long ; la procédure commence par faire le contraire, on obtient donc un double prime, suivi de deux signes moins, i.e. $x'' - -$.

3.2.10 Traitement des mathématiques, partie 3

Il y a une procédure `TE_handle_inner1`. Son comportement est le suivant : si l'expression à traiter est dans la table, le résultat est le contenu de la table. Si l'argument est vide (modulo les espaces), le résultat est vide. La procédure traite les `\left` et les `\hbox`, et s'il n'y en a pas, appelle `cv_math`, la procédure expliquée plus haut. Il y a une subtilité : dans certains cas, il faut pouvoir cacher une expression. Ceci se fait comme suit : on met l'expression dans une table, en position N, où N est un certain entier, et on rend `\9N!` (par exemple `\917!`). Une expression de la forme `\917!` est donc une expression MathML déjà traduite. Elle reste inchangée par `TE_handle_inner1` et les procédures appelées par celles-ci.

Supposons que la formule à traiter soit de la forme `A \left x B \right y C`. Les deux quantités x et y doivent être de la forme : `\{`, `<`, `[`, `(`, `\lbrace`, `\langle` (ou l'équivalent fermant), ou `|` ou `..` On choisit le premier `\left` et le dernier `\right` (c'est une erreur si B n'est pas équilibré du point de vue accolades). On traduit B via `TE_handle_inner1`, ce qui donne $B1$. On considère `<mfenced open='x' close='y'>B1</mfenced>`, et on cache cette expression, ce qui donne $B2$. On traduit alors `A B2 C` via `TE_handle_inner1`. Exemple `$\left \rangle x \right[^2$`. Dans ce cas de figure, il se passe la chose suivante : l'expression A est vide, l'expression B est `x` et l'expression C est `^2`. La traduction de B est triviale, et ce que l'on donne à `cv_math` sera donc `\926!^2`, qui est facile à traduire. Le résultat est :

```
<msup> <mfenced open='&rang;' close='['><mrow><mi>x</mi></mrow></mfenced>
<mn> 2 </mn> </msup>
```

Supposons que la formule soit de la forme `A \hbox{B} C` (le `\hbox` peut être remplacé par un `\mbox` ou un `\text`). On va traduire le `\hbox` comme expliqué ci-dessous, cacher le résultat, et rappeler `TE_handle_inner1`. Par exemple `\frac{10}{\hbox{to to}^2}` donne

```
<mfrac> <mrow> <mn> 1 </mn> <mn> 0 </mn> </mrow>
  <msup> <mrow><mtext>to</mtext><mspace width='0.5em'/><mtext>to</mtext></mrow>
         <mn> 2 </mn> </msup></mfrac>
```

La traduction de `${a+ \mbox{ $ c = d $ et } =f }$` est

```
<mrow> <mi>a</mi> <mo> + </mo> <mrow><mspace width='0.5em'/><mi>c</mi> <mo> =
</mo> <mi>d</mi><mrow><mspace width='0.5em'/><mtext>et</mtext>
<mspace width='0.5em'/></mrow></mrow> <mo> = </mo> <mi>f</mi> </mrow>
```

La traduction de `$a+ \mbox{ $ c = d $ et } =f $` est :

```
<formula type='inline'>
<math><mi>a</mi> <mo> + </mo></math></formula> <formula type='inline'>
<math><mi>c</mi> <mo> = </mo> <mi>d</mi></math></formula> et <formula type='inline'>
<math><mo> = </mo> <mi>f</mi></math></formula>
```

Notons la différence entre ces deux traductions. Dans le second cas, on a une expression complète, de la forme `$A\mbox{B}C$`, et le `\mbox` n'est pas caché à l'intérieur d'accolades, ni d'environnements, ni de `\left`, `\right`. Dans ce cas, on traduit l'expression complète comme si c'était `$A${B}$C$`. Dans l'exemple, les accolades ne servent pas, mais il faut les garder quand même, et c'est comme si on traduisait `$a+$ $c=d$ et $=f$` (on a mis uniquement les espaces qui servent vraiment). Dans le premier exemple, on a mis des accolades autour de l'expression complète, pour éviter cette simplification. À part le fait que l'expression est dans un `<mrow>`, le résultat devrait être le même. Il ne l'est pas, pour la raison suivante : dans le cas de `$a+$`, l'opérateur + est postfixe, il n'y a pas d'espace autour. Dans le premier cas, il est infixe, il y a donc un espace de chaque côté, en plus du `<mspace>`. La raison pour laquelle on distingue ces

deux cas est que cela permet de mettre, par exemple, un `\it` dans un `\mbox` dans le second cas (mais pas dans le premier).

Comme le montre l'exemple, dans certains cas, un `\hbox` sort du mode mathématique. S'il y reste, l'argument x est traité comme suit. S'il n'y a ni commande, ni espace, le résultat est le contenu de la boite, dans un `<mtext>`. Si x est vide, le résultat est vide, si x est un espace, le résultat est un `<mspace>`. Si x contient un dollar, disons est de la forme `a$b$c`, on appelle `TE_translate_math` sur tout ce qui suit le dollar. Ceci va traduire la formule de math, et laisser ce qui suit le dollar. Ce qui précède et ce qui suit la formule est traité comme le contenu d'une boite. Les trois morceaux traduits seront mis dans une `<mrow>`. On va maintenant supposer que ce qui reste, ce sont des caractères. Toute suite de caractères ne contenant pas d'espaces sera mis dans un `<mtext>`, les blancs donneront des `<mspace>`, et le tout sera mis dans un `<mrow>`.

La procédure `TE_trivial_math` est appelée dans le cas des expressions de la forme x. Il y a plusieurs cas : si la formule ne contient que des chiffres, on rend ce nombre. Si la formule contient juste une lettre, on rend un élément `<formula>` qui contient un `<simplemath>`. Ainsi, la traduction en HTML du résultat de la traduction en XML de `$x$` pourra être `<i>x</i>`. Si la formule contient une lettre grecque, du type λ ou π, le résultat sera `&lambda;` ou `&pi;`. Si la formule est dans `nomathtable`, le résultat sera la traduction de ceci. Sinon, on regarde s'il y a des `\hbox` ou `\mbox` non cachés dans la formule, et si la formule ne contient ni `\begin`, `\end`, `\left` ou `\right`. Dans ce cas, on utilise la technique expliquée plus haut pour supprimer les boites. Dans tous les autres cas, on ne fait rien.

Il y a une procédure `TE_convert_math` qui prend deux arguments, une formule x, et un mode m. Si $m = 0$, on traite une expression du type `$x$`, si $m = 1$, on traite une expression du type `$$x$$`, et sinon, on traite une sous-formule. Dans le cas $m = 0$, on teste d'abord si `TE_trivial_math` peut faire la conversion. Supposons que ce ne soit pas le cas. On commence par extraire tous les `\label` de la formule, pour les mettre derrière. S'il y a des environnement `array`, `pmatrix`, etc, dans la formule, on les traduit, et on les remplace par des `\\9N!`. On appelle `TE_handle_inner1`. Ceci donne une formule MathML. Si le mode est 3, on rend cette formule, sinon, on met la formule dans un élément `<math>` et on met cet élément dans un élément `<formula>` (avec l'attribut `type` positionné à `inline` ou `display`).

La version 4 du traducteur utilise les idées expliquées plus haut, avec une différence de taille : il n'y a pas lieu de cacher quoi que ce soit, et l'algorithmique est plus subtile (on a failli dire, plus compliquée, mais ce n'est pas le cas).

3.2.11 Traitement des mathématiques, partie 4

Résumé des épisodes précédents : on sait convertir une expression de mathématiques isolée, à condition qu'il n'y ait pas de commandes utilisateur.

La procédure `TE_translate_math` procède comme suit : elle considère le texte courant, et expanse les macros utilisateur au passage. Elle s'arrête de scanner le texte si elle voit, à un niveau zéro d'accolades et d'environnement, un `$`, et `&` ou un `\\` ou si elle voit une accolade ou un groupe fermant. On considère que `\(` et `\[` sont des accolades ouvrantes, et que `\)` et `\]` sont des accolades fermantes. Le texte lu est traduit par `TE_convert_math`.

Il y une commande `math_aux` à deux arguments m et p. Cette procédure appelle la procédure précédente, avec le mode m, et elle lit un token. C'est une erreur si le token n'est pas p. La traduction de la commande `\(` est triviale : on appelle `math_aux` avec comme argument 0 (pour

le mode) et `\)` pour p. La traduction de `\[` est tout aussi triviale. Dans le cas de `$` on regarde s'il y a un second dollar, et on applique la commande `math_aux`. Les environnements `math`, `equation`, `displaymath` sont traités de même. La traduction des environnements `array`, `pmatrix`, etc, est plus délicate ; l'idée est de traduite chaque cellule l'une après l'autre, via `TE_translate_math`, la fin de la cellule étant soit le `&`, soit un `\\` soit la fin de l'environnement. La traduction de ces environnements sera expliqué plus loin.

3.2.12 Manipulation de texte

Certaines des procédures décrites ici positionnent un code d'erreur, d'autres rendent `error` dans certains cas, on n'en parlera pas. Les procédures, sauf exception, manipulent le texte courant, et en enlève un bout au début, la macro expansion peut rajouter du texte en tête. On dira qu'un texte est équilibré s'il contient autant d'accolades ouvrantes que de fermantes, on supposera bien sûr que les accolades ouvrantes précèdent les accolades fermantes. Pour trouver un texte équilibré, avec un délimiteur optionnel x, on procède comme suit : soit A le résultat partiel. Tant qu'on n'a pas fini, on découpe le texte en B, suivi d'une accolade ou x, suivi de C. Si entre B et C on a une accolade, on incrémente ou on décrémente le compteur d'accolades. On a fini si le compteur est 0, et si x se trouve entre B et C, ou s'il n'y a pas de x, et que le compteur est 0 (après avoir vu une accolade fermante). Dans ce cas, on rend la concaténation de A et B, et le texte courant devient C. Dans le cas contraire, on remplace A par la concaténation de A, B et ce qui est entre B et C, le texte courant devient C, et on continue.

Les deux procédures `TE_skip_group` et `TE_skip_bracket` cherchent un texte équilibré, la première sans délimiteur, la seconde avec un crochet fermant comme délimiteur. Note : si le texte courant est `{x}y`, la procédure `TE_skip_group` va rendre `{x` et un code de retour. Les procédures qui appellent celle-ci vont regarder le code, et supprimer l'accolade en tête.

La procédure `TE_mac_arg` enlève les espaces initiaux. Si le texte commence par une accolade ouvrante, elle appelle `TE_skip_group`. Si le texte commence par `\1foo!`, elle rend ceci. C'est une erreur si le texte commence par un backslash. Sinon, la procédure rend le premier caractère du texte. La procédure `TE_next_arg` est similaire, sans code de retour. La procédure `TE_next_exparg` est similaire, mais le résultat est traduit en XML. La procédure `TE_next_optarg` appelle `TE_skip_bracket` si le texte commence par un crochet ouvrant, sinon elle ne rend rien (mais dans tous les cas, les espaces initiaux sont supprimés). La procédure `TE_next_expoptarg` fait de même, mais le résultat est traduit en XML.

3.2.13 Macro expansion

La procédure `TE_next_token` prend un argument x. Elle supprime les espaces initiaux. Si $x = 0$, elle regarde si le caractère qui suit est un crochet ouvrant. Dans ce cas, elle appelle `TE_skip_bracket`, sinon elle rend le marqueur spécial `{`. Dans les autres cas, si le texte commence par une accolade ouvrante, elle appelle `TE_skip_group`. Dans les autres cas, c'est une erreur si $x = 1$, ou si le texte ne commence pas par un nom de commande, du genre `\1foo!`, et dans ce cas, le résultat est cette commande.

La procédure `TE_get_name` prend un argument x. Elle appelle `TE_next_token(2)`. Si $x = 0$, c'est une erreur, sauf si le résultat est formé de lettres uniquement, avec un `*` optionnel à la fin.

Si $x = 2$, on accepte n'importe quelle macro, mais si $x = 1$, on accepte uniquement les macros composées uniquement de lettres. Dans ces deux cas, si la macro est `\1foo!`, le résultat sera `foo`.

La procédure `TE_get_nbargs` prend un argument x. Si $x = 0$, elle suppose que le texte est de la forme `#1#2#3` etc, et retourne le nombre de `#`. Dans le cas contraire, elle appelle `TE_skip_bracket`. S'il n'y a pas de crochets, le résultat est 0, sinon la valeur qui est entre les crochets (c'est une erreur si cette valeur n'est pas un chiffre).

Il y a trois procédures `TE_see_new_command`, `TE_see_new_def` et `TE_see_new_env`, dont le code est essentiellement le suivant :

```
TE_get_name(1); TE_get_nbargs(1); TE_next_token(0);TE_next_token(1); ""
TE_get_name(2);TE_get_nbargs(0); { ; TE_next_token(1);""
TE_get_name(0);TE_get_nbargs(1);TE_next_token(0);TE_next_token(1); TE_next_token(1);
```

Ces trois procédures calculent 5 valeurs. Elles sont appelées dans le cas où on voit `\newcommand`, `\def` et `\newenvironment`. La première valeur est N, c'est le nom. Notons qu'on autorise `\def\:`, mais pas `\newcommand\:`. La deuxième valeur est n, le nombre d'arguments. La troisième valeur est b, l'argument optionnel. Notons que s'il n'y a pas d'argument optionnel (et c'est le cas de `\def`), ceci est codé comme une accolade ouvrante. La valeur suivante est b, le corps de la macro. Il y a également c, la fin de l'environnement (cette quantité est vide dans le cas de `\def` et `\newcommand`). Les quatre quantités n, a, b et c sont rangées dans une table, la table des macros, à la position N.

Note : `\def\foo{x}` et `\newenvironment{foo}{x}{}` sont donc considérés de la même manière, contrairement à ce que fait `latex2html`, LaTeX, et `tralics`. Quand on voit `\newtheorem{x}{y}`, on fait comme si on avait vu `\newenvironment{x}{\egroup\par\textbf{y} }{\par\bgroup}`. Remarquons que `\bgroup` et `\egroup` sont interdits en principe. Le résultat de l'expansion de `\begin{x}toto\end{x}` est donc le même que `{}\par\textbf{y} toto\par{}`.

La procédure `TE_expand_mac` sert à expanser les macros. Elle fonctionne comme suit : on récupère les paramètres n, a, b et c associés à la macro. La quantité c est ignorée. Il y a n arguments à récupérer. Chaque argument est obtenu via `TE_mac_arg`. Dans le cas du premier argument, si a n'est pas le marqueur spécial `{`, il s'agit d'un argument optionnel. Si le texte commence par un crochet, on utilise `TE_skip_bracket` pour trouver l'argument, et s'il n'y a pas de crochet, la valeur de l'argument est a. On remplace alors dans une copie de b chaque `#i` par la valeur du i-ième argument. Cette copie de b est le résultat de l'expansion, elle est ajoutée en tête du texte courant.

La procédure la plus importante est `TE_expand`. Cette procédure ne fait rien si le texte courant ne commence pas par `\1foo!`. Dans le cas où `foo` est un accent, on appelle `TE_accent`. Les accents de base sont l'accent aigu, l'accent grave, l'accent circonflexe et le tréma. Il y a aussi `\u`, `\v`, `\c`, et `\a`. Dans ce dernier cas, l'accent est le token qui suit le `\a`. On utilise `TE_mac_arg` pour trouver la lettre sur laquelle mettre l'accent, et on regarde ensuite la table des accents. Dans le cas où `\foo` est une macro utilisateur, on l'expanse, comme dit plus haut (et on appelle `TE_expand` à nouveau). Si on a par contre `\begin{bar}`, et que `\bar` est une macro utilisateur, on fait comme si on avait vu `{\bar`, i.e., on expanse `\bar`, et on ajoute une accolade devant. Finalement, dans le cas de `\end{bar}`, si `bar` est un environnement connu, on prend le champ c de la macro, et on l'insère dans le texte courant, suivi d'une accolade fermante. Note : on utilise le fait que les arguments de `\begin{bar}` ne sont pas visibles par `\end{bar}`.

3.2.14 La traducteur

La procédure `TE_translate` est le traducteur XML, elle reçoit un argument qui devient le texte courant. Elle traduit tout, morceau par morceau, et concatène les résultats partiels. Si le texte ne contient pas de backslash, la traduction est triviale : c'est le texte. Supposons donc qu'il y ait un backslash. Tout ce qui précède le backslash est ajouté au résultat partiel. On suppose donc que le texte commence par un backslash. On appelle `TE_expand`. Si le texte ne commence pas par un backslash, on continue. Sinon, c'est une commande à traduire. Après avoir traduit ce texte, et inséré le résultat dans le résultat partiel, on continue. Les divers cas à traiter sont :

- cas de `\\` : on appelle `TE_backslash`,
- cas de `\!` : on ne rend rien,
- cas de `\begin{foo}` : on appelle `TE_beg_env`,
- cas de `\end{foo}` : on appelle `TE_end_env`,
- cas de `\foo` : on appelle `TE_macros`,
- cas de `<`, `>` ou `~` : on rend `<`, `>` ou ` `,
- cas de `&` : on appelle `TE_ampersand`,
- cas de `#` : erreur,
- cas de `$` : on appelle `TE_math`,
- cas de `{` : on appelle `push(0,"")`,
- cas de `}` : on appelle `pop(0)`.

3.2.15 Du code qui ne sert plus

L'algorithme décrit dans ce paragraphe était implémenté dans la version un du traducteur, nous l'avons remplacé par un autre mécanisme.

L'idée est de remplacer les accolades ouvrantes et fermantes par des quantités du type `\3N!` et `\4N!`, où le numéro N est le même pour l'accolade ouvrante que l'accolade fermante. Une fois ceci fait, l'implémentation de `TE_skip_group` devient triviale, de même que la recherche de texte équilibré avec séparateur. Comme cela est expliqué en section 1.3, on remplace `\begin{toto}` et `\end{toto}` par quelque chose de la forme `\7N!` et `\8N!` et on utilise une version de `skip_group` qui saute aussi par dessus les environnements. Avec cette procédure sous la main, on peut résoudre facilement la question : quelle est l'accolade fermante non appariée à une accolade ouvrante. Par exemple dans le cas de `{{a \it b {c}}d}`, si on considère ce qui suit le `\it`, la réponse est : la seconde. Ainsi, il devient trivial de remplacer cela par `{{a \textit {b {c}}}d}`. Dans la version actuelle du script, on utilise la pile pour ce faire.

Dans la première version du script, on traitait les formules de maths comme suit : pour chaque dollar, on utilise une recherche de texte équilibré avec délimiteur. Partant de

```
$ a { b } { c $ d {$ e {} f $ } $ g} { h } $ i $ j $
```

on obtient :

```
\( a { b } { c $ d {$ e {} f $ } $ g} { h } \) i $ j $
```

puis :

```
\( a { b } { c \( d {$ e {} f $ } \) g} { h } \) i $ j $
```

et finalement :

```
\( a { b } { c \( d {\( e {} f \) } \) g} { h } \) i \( j \)
```

Pour simplifier l'exemple, on a mis `\(` au lieu de `\begin{math}` (en fait, on a plutôt un `\7N!`, où N est un nombre unique, le même pour le début et la fin). Dans la version actuelle, on trouve le dollar fermant en comptant les accolades (ce qui n'est pas toujours correct).

Dans le paragraphe 2.5.3, nous avons montré comment `ltx2x` pouvait insérer automatiquement des balises fermantes de sous-sections, etc. Cependant, si un module contient un paragraphe, mais pas de sub-sub-section, il y aura quand même la balise fermante de sub-sub-section. Pour pallier à ce problème, le traitement est fait en Perl, comme suit. On remplace d'abord `\x` par `\5y?` où x est subsubsection, subsubsection*, paragraph, paragraph*, subparagraph, et subparagraph*, et y est 2, 3, 4, 5, 6 et 7, respectivement.

Le texte est alors découpé en blocs, chaque `\5` délimitant un bloc. On procède en trois passes, dans la première, on aura $N = 6$ et $M = 7$, dans la seconde $N = 4$, $M = 5$, et dans la dernière $N = 2$ et $M = 3$ (on traite donc les sous-paragraphes), puis les paragraphes, puis les sous-sous-sections.

Pour chaque bloc qui commence par `\5N?` ou `\5M?`, on lui rajoute `\6N?` ou `\6M?` à la fin. On colle le bloc au bloc précédent (s'il y en a un). À la fin, tous les blocs sont recollés, le M retranscrit en subsubsection, etc.

Prenons un exemple. Pour simplifier, on met S à la place de `\subsubsection`, P à la place de `\paragraph` et p à la place de `\subparagraph`. Partons de

```
a \p b \P c \P d \S e \p \it f \p g \P g
```

Le marquage donne (la barre verticale délimite les blocs)

```
a | \56? b | \54? c | \54? d | \52? e | \56? \it f | \56? g | \54? h
```

La première passe donne

```
a  \56? b \66? | \54? c | \54? d | \52? e  \56? \it f \66?  \56? g \66? | \54? h
```

La deuxième passe donne

```
a  \56? b \66?  \54? c \64? \54? d \64? | \52? e  \56? \it f \66?  \56? g \66?
\54? h \64?
```

La troisième passe donne

```
a  \56? b \66?  \54? c \64? \54? d \64?  \52? e  \56? \it f \66?  \56? g \66?
\54? h \64? \62?
```

Si on décode ceci, on obtient

```
a  \begin{subparagraph} b \end{subparagraph}
   \begin{paragraph} c \end{paragraph}
   \begin{paragraph} d \end{paragraph}
   \begin{subsubsection} e  \begin{subparagraph} \it f \end{subparagraph}
       \begin{subparagraph} g \end{subparagraph}
       \begin{paragraph} h \end{paragraph}
   \end{subsubsection}
```

Dans la nouvelle version, tout ceci est fait au moyen d'une pile.

3.2.16 La pile

Dans le paragraphe précédent, nous avons montré l'utilité de la pile. Elle est formée d'un pointeur de pile et de deux listes P et F. Dans P il y a du code XML qui servira plus tard et dans F, il y a un nom. La procédure `push` prend deux arguments, a et b qui seront mis dans F et P. Quand on voit une accolade ouvrante, on appelle `push` avec deux arguments : 0 et la chaine

vide. La procédure `pop` prend un argument a. C'est une erreur fatale si la pile F ne contient pas a. Sinon, elle rend le sommet de pile P (et décrémente le pointeur de pile). Par exemple quand on voit une accolade fermante, on apppelle `pop` avec 0 comme argument.

Le résultat de la traduction en XML est codé de la façon suivante. Au lieu de

```
<toto attribut='quelque chose'> aa <titi machin="bidule"/> bb </toto>
```

on verra

```
\733!toto! aa \478!titi! bb \833!toto!
```

et les attributs sont dans une table, en position 33 et 78, respectivement. La procédure `TE_new_id` prend deux arguments x et y, et rend les deux balises (ouvrantes et fermantes) d'un élément de nom x dont la liste des attributs est y. La procédure `TE_new_id0` rend la balise (unique) d'un élément vide.

La procédure `TE_id_stack` prend deux arguments x et y, et crée une balise. Elle met la balise fermante dans la pile (le pointeur de pile n'est pas changé), et rend la balise ouvrante. La procédure `TE_fonts0` prend un argument z, et appelle `TE_id_stack`, avec comme arguments `hi` pour x, et `rend='z'` pour y. Cette procédure est appelée pour traduire les macros du genre `\it`. Note : dans le cas de `{\it a\par b}`, la fin de l'italique est forcée par la commande `\par`, car, comme nous allons le voir, `\par` va forcer un `pop`. Remarque : le code HTML `<i> a<p> b </i>` est invalide, certains navigateurs mettent le b en italiques, d'autres non. Avec notre façon de faire, le résultat est correct. Par ailleurs, `\par` pourrait empiler à nouveau le `hi` qu'il vient de dépiler (voir le code de l'environnement `center`). Ce mécanisme n'était pas possible dans l'ancienne version du script.

La procédure `TE_push1` prend trois arguments m, x et y. Elle crée d'abord une nouvelle balise, et empile la balise fermante (en mettant m dans le frame F). Elle rend la balise ouvrante. La procédure `TE_arg1` prend un argument x. Elle appelle `TE_push1` avec comme arguments x, x et la chaine vide. Elle appelle `TE_next_exparg`, puis `pop(x)`. Les trois résultats sont concaténés. Quelques commandes sont traduites par ce moyen, d'autres commandes utilisent ce moyen pour traduire les arguments. Il y a une version de cette commande qui crée deux balises enboîtées. Il y a une version, `TE_arg1_head`, avec argument par défaut (si l'argument est vide, on met « (Sans titre) » à la place). Une autre variante est `TE_fonts` : au lieu de créer une balise x sans attributs, elle crée une balise de nom `hi` avec l'attribut `rend=y`. Cette variante est utilisée pour traduire les commandes du type `\textit`.

Deux procédures `TE_push_par` et `TE_pop_par` existent pour empiler ou dépiler un élément du type `p`. Il y a aussi l'équivalent avec `cpar` au lieu de `par`, dans ce cas le nom de l'environnement dans F est `cp` au lieu de `p` et l'élément a un attribut `rend='centered'`. La traduction de `\begin{center}` est alors : `TE_pop_par`, suivi de `TE_push_cpar`, et la traduction de `\end{center}` est `TE_pop_cpar`, `TE_push_par` (le résultat est la concaténation des deux appels de procédure). La traduction de `\par` est plus compliquée : en général c'est `TE_pop_par`, `TE_push_par`. Mais si la pile contient `cp`, c'est `TE_pop_cpar`, `TE_push_cpar`, et si la pile contient `caption`, `participant` ou `catperso` on ne fait rien. Remarque : dans le cas de `caption`, le comportement est-il OK ?

Il y a une procédure `TE_push_par_hack`, à 3 arguments x, y et z. Cette procédure appelle `TE_pop_par`, puis `TE_push1(x,y,z)` puis `TE_push_par`. Il y a une variante sans le `pop` initial, une variante sans le `push` final, et bien sûr les procédures inverses avec `pop` à la place de `push`.

La variante qui ne dépile pas le <p> est appelée quand on voit \begin{figure} ou \begin{table}, dans les deux cas, un argument optionnel est lu et supprimé, il s'agit de l'argument qui positionne la table ou la figure. Elle est aussi appelée dans le cas de \footnote. Il y a cependant une subtilité. La traduction de \fotnote{toto} est

```
<note place='foot'><p>toto</p></note>
```

Le post processeur supprime les balises <p> (dans le cas où la note ne contient pas de changement de paragraphes), car le résultat n'est pas très joli. En fait, la balise <p> autorise un changement de page, et dans certains cas, on se retouve avec un marqueur à la fin de la page, et la note sur la page suivante.

La variante qui n'empile pas de <p> est appelée au début d'un \begin{motscle} ou catperso ou participant (avec un e ou un s optionnel dans le nom). Rappelons que dans ces environnements, les paragraphes sont ignorés. Dans le cas de l'environnement catperso, il y a un argument, qui est traité par TE_arg1_head. Dans le cas de motscle, la situation est plus compliquée. Pour chaque mot clé x, on génère <term><kw>x</kw></term>. Pour trouver les mots clés, on considère le texte courant, du moins la partie précédant le \end{motscle}, et on cherche un texte équilibré, avec une virgule comme séparateur. Les espaces autour des mots clé sont supprimés, et le résultat est traduit. Remarque : le \end{motscle} doit être explicite dans le texte, il ne peut pas venir d'une macro expansion.

L'expansion de \pers est réalisée comme suit : on récupère d'abord les 4 arguments (deux des arguments sont optionnels), il s'agit du prénom, de la particule, du nom et de la partie info. Dans le cas où le nom contient un \footnote, tout ce qui suit le \footnote sera mis dans la partie info (avec une virgule comme séparateur, si elle n'est pas vide). Le \footnote disparait. S'il y a une particule, on l'accole au nom. Une virgule après le \pers disparait. Le résultat sera alors <pers prenom='a' nom='b'>c</pers>, où a, b et c sont les résultats de la traduction du prénom, nom et info.

La procédure TE_push_par_hack est appelée quand on voit \begin{xx}, où xx est description, itemize, enumerate ou glossaire. Le nom de l'environnement est list, et l'attribut est type, avec une valeur respectivement égale à description, simple, ordered et gloss. Elle est appelée dans le cas de moreinfo. Dans les trois premiers cas, la fin de l'environnement est traité de façon spéciale : on dépile un <p>, un <item>, un <list>, et on empile un <p>. Ceci est justifié par le traitement de \item : quand on voit un \item, on dépile <p>, puis <item> s'il y en a un dans la pile ; on regarde ensuite s'il y a un argument optionnel, s'il y en a un, on appelle TE_arg1 avec label comme argument ; on empile alors <item> et <p>. Note : un environnement itemize sans \item est une erreur (c'est aussi le cas dans LaTeX). Remarque : la procédure appelée n'est pas TE_arg1 (qui va lire un argument obligatoire, mais une variante, qui va lire un argument optionnel). Exemple :

```
\begin{itemize} \item toto \par titi \item[tata] tutu \end{itemize}
```

La traduction XML sera

```
<list type='bulletted'>
<item><p>toto </p> <p>titi </p></item>
<label>tata</label><item><p> tutu </p></item></list>
```

Notons que le code XML donné plus haut n'est pas valide : on ne peut pas mélanger des \item avec argument optionnel et des \item sans argument optionnel. La commande \glo{x} est traduite comme s'il y avait \item[x]. Ainsi, la traduction de

```
\begin{glossaire} \glo{toto}{titi\par etc}\glo{tutu}{tata}\end{glossaire}
```

sera

```
</p><list type='gloss'><p> </p><label>toto</label><item><p>titi</p>
<p>etc</p></item><p></p><label>tutu</label><item><p>tata</p></item><p></p></list><p>
```

Notons qu'il y a plein de paragraphes vides, qui seront supprimés dans la suite. Ce qu'on verra dans le XML, au final, est :

```
<list type='gloss'> <head>Glossaire</head>
   <label>toto</label> <item><p>titi</p> <p>etc</p></item>
   <label>tutu</label> <item><p>tata</p></item>
</list>
```

(le lecteur attentif pourra se poser la question : d'où sort le `<head>` ? en fait, nous ne savons pas si dans la version définitive, il y a aura ou non un `<head>`, c'est pour cela qu'on n'en a pas parlé).

3.2.17 Autres environnements

Nous avons vu dans les sections 2.5.4 et 3.2.15 deux façons de traiter les commandes `\paragraph`, `\subparagraph` ou `\subsubsection`. Nous donnons ici le code réellement utilisé : il est simple. On dépile d'abord un `<p>`. On dépile ensuite les divisions de niveau inférieur ou égal, pourvu qu'il y ait quelque chose à dépiler. On empile alors la division courante `div2`, `div3` ou `div4`, appelle `TE_arg1_head` pour traduire le titre, et on empile un `<p>`. Le caractère `*` après le nom de ces commandes est ignoré (dans LaTeX, ce caractère signifie : ne pas numéroter la section, ne pas la mettre dans la table des matières ; dans la version XML, toutes les divisions sont numérotées et dans la table des matières). Quand on arrive à la fin d'un module, la pile peut contenir un nombre variable de quantités. En fait, on dépile tout. Avec cette façon de faire, le résultat XML sera toujours bien formé.

Quand on voit un début de module, on commence par récupérer les trois premiers arguments (vu les contraintes, on n'a pas besoin de les traduire). On crée une balise de la forme :

```
<module projet="A3" type="3" html="modname" id="mod:modname">
```

Remarquons que le nom du module apparait 2 fois. On traite le titre du module via `TE_arg1_head`, et on empile un `<p>`.

3.2.18 Autres commandes

Quand on voit une commande genre `\toto`, il y a plusieurs cas à considérer. Le cas le plus simple concerne les commandes qui ne font rien. Voici la liste `\protect`, `\relax`, `\small`, `\leavevmode`, `\null`, `\xspace`, `\noindent`, `\nopagebreak`, `\-`, `\@`, `\/`. Un autre cas simple est celui où la traduction est triviale, voir les tables 3.2 et 3.4. Nous avons expliqué plus haut la gestion des polices. Voir la table 3.3. Certaines commandes prennent un argument et appellent `TE_arg1`, voir la table 3.5. La commande `\mbox` est un peu spéciale : elle rend son argument, sans mettre la balise `mbox` autour dans les deux cas suivants : soit s'il n'y a pas de commande à l'intérieur, soit si l'argument est une simple image. Dans les autres cas, on se sait pas quoi faire. Dans certains cas, l'expansion est un élément vide, l'argument est un attribut voir table 3.6.

Dans le cas de `\label` et de `\ref`, on construit un attribut de type `ID`. Il y a certaines contraintes. On remplace tout caractère qui n'est pas une lettre, un chiffre, un tiret, un souligné ou un deux-point par un souligné. Si le texte commence par un chiffre, on rajoute un souligné devant. Dans le cas de `\moduleref`, on ne considère que le dernier argument. S'il est toto, le

résultat XML	source LaTeX
&lstrok;	\l
&numero;	\numero
&numero;	\no
&ier;	\ier
&iere;	\iere
&iers;	\iers
&ieres;	\ieres
&ieme;	\ieme
&iemes;	\iemes
&copy;	\copyright
&trade;	\texttrademark
«	\guillemotleft
»	\guillemotright
	\quad
	\space
	\qquad
LaTeX	\LaTeX
\$	\$
TeX	\TeX
	\,
§	\S
_	_
&amp;	\&
\{	\{
\}	\}
#	\#
%	\%
...	\dots
...	\ldots
oe	\oe
å	\aa
␣	\␣

TAB. 3.2 – Table des commandes sans argument

avec argument	sans	attribut
\emph	\em	emph
\textsc	\sc, \scshape	sc
\underline		underline
\textsf	\sf	sansserif
\textsl	\sl	it
\textrm	\rm	roman
\texttt	\tt	tt
\textbf	\bf	bold
\textit	\it	it
\textsuperscript, \RAsup		sup
\RAsub		sub

TAB. 3.3 – Table des fontes. Le résultat est un élément <hi>, avec un attribut rend dont la valeur est donnée dans la table.

macro	balise	attribut
\hline	hline	
\bigskip	vspace	val='12pt'
\smallskip	vspace	val='6pt'
\medskip	vspace	val='3pt'
\centering	centering	
\hfill	hfill	
\hfil	hfil	
\vfill	vfill	
\vfil	vfil	
\centering	centering	

TAB. 3.4 – Table des commandes sans argument. La balise résultat n'existe pas dans la DTD.

commande	balise
\bauteurs	bauteurs
\bediteur	bediteur
\fbox	fbox
* \mbox	mbox
* \caption	caption
\RAmathn	mn
\RAmatho	mo
\RAmathi	mi

TAB. 3.5 – Table des commandes simples à un argument. Le cas de mbox est spécial, l'argument optionnel de caption est ignoré.

commande	balise	attribut
+ \label	mylabel	id
+ \ref	ref	target
\RAnomath	nomath	id
\RAnodmath	nodmath	id
\RAnosmath	nosmath	id
* \hspace	hspace	val
* \vspace	vspace	val

TAB. 3.6 – Table des commandes simples à balise vide. Un traitement spécial est fait pour les commandes marquées d'un signe plus. Pour celles marquée d'une étoile, si la commmande est suivie d'une étoile, on rajoute l'attribut supplémentaire `star='yes'`. La balise mylabel est traitée par le post processeur.

résultat sera `<ref target='mod:toto'>` (chaque module génère un label implicite, qui a cette forme).

Le même traitement est fait pour les références bibliographiques. En fait, quand on voit `\cite`, ou `\nocite` ou `\footcite`, on procède comme suit. Tout d'abord, s'il y a un argument optionnel, on l'enlève (que peut-on faire avec ? en fait `tralics` fait ce qu'il faut). On récupère l'argument traduit, et on le découpe en morceaux, la virgule étant le séparateur. Pour chaque item, on enlève les espaces de part et d'autre. On applique le traitement expliqué plus haut. Dans le cas de `\footcite{toto}` ou `\cite{titi}`, le résultat sera

```
<cit rend='foot'> <ref target='footcite:toto'/></cit>
<cit><ref target='cite:titi'/></cit>
```

Dans le cas de `\nocite`, on construit juste le `<ref/>`. Dans les trois cas, la valeur du `target` sera sauvé dans une table (voir section 3.2.1). Dans le cas de `\nocite{*}`, on positionne une variable à vrai. La commande `\citation` prend trois arguments, a, b et c (voir plus haut, un exemple). On construit un élément `<citation>`, à 4 attributs. Il y a un attribut `from`, dont la valeur est x, y ou z suivant que la référence bibliographique vienne de la bibliographie de l'année, de la bibliographie générale, ou celle en note. Le deuxième argument b est la clé de citation, on ajoute `footcite:` ou `cite:` devant, et on applique la même transformation que pour les `\cite`. Le résultat sera donc de la forme :

```
<citation from ="z" key="MT96" id="cite:McTe96" type="inproceedings">
```

Remarques : si le texte contient `\label{10}` et `\label{_10}`, ces deux labels seront transformés en la même valeur, ce qui est une erreur. Il en est de même s'il y a `\label{a/b}` et `\label{a+b}`. Par ailleurs, supposons que la bibliographie contienne deux références `a/b` et `a+b`, et qu'une seule soit référencée dans le texte. Le même problème se pose, alors qu'on pourrait l'éviter, si on s'y prenait autrement.

L'expansion de `\url{toto}` est `<xref url='toto'>toto</xref>`. L'argument n'est pas traduit. Notons cependant que `~` et `#` (qui sont codés comme `\3~!` et `\3#!`) sont transcrits en `~` et `#`. Si l'expansion de `\url` se fait dans un environnement `hanl`, un fait comme si c'était `\texttt`. L'expansion de `\htmladdnormallink` est la suivante : on traduit les 2 arguments dans un environnement `hanl`. Si le résultat est A et B, on créera `<xref url='B'>A</xref>`. Les ` ` dans B seront remplacés par des `~`.

3.2.19 Traitement des figures

La commande `\includegraphics` prend deux arguments : un argument optionnel, x et un autre y. Le résultat est `<graphics file='y'>`. L'argument optionnel x donne des attributs supplémentaires. Pour cela, on le découpe, suivant les virgules. Si on voit `angle=A`, on rajoute `angle='A'` (sauf si la valeur A est 0). Si on voit `scale=A` ou `height=A`, on fait de même. En principe, on fait de même avec `width=B`. Cependant, la quantité B peut être `\textwidth`, `\linewidth` ou `\columnwidth`, quantité que l'on supposera égale à 15cm. Les trois macros citées plus haut ne sont pas traduites. En effet, la traduction de `\textwidth=30cm` consiste à changer la valeur de la largeur du texte à 30cm. Par contre, B est utilisée dans le cas d'une affectation, et c'est bien la valeur de la variable qu'il faut utiliser. Notons que `0.45\linewidth` est un argument valide : c'est un peu moins que la moitié de la largeur du texte.

La traduction de `\subfigure[toto]{titi}` est

```
<subfigure><leg>toto</leg><texte>titi</texte></subfigure>
```

La situation est la suivante :

(a) `\includegraphics[x]{y}` se traduit en un `<graphics>` ;
(b) `\subfigure[x]{y}` se traduit en `<subfigure>` ;
(c) `\caption[x]{y}` se traduit en `<caption>y</caption>` (l'argument optionnel est ignoré) ;
(d) `\label{x}` se traduit en `<mylabel id='x'/>` ;
(e) `\begin{figure}[x]y\end{figure}` se traduit en `<figure>y</figure>`. L'argument de position x n'est pas traduit.

On fera l'hypothèse dans la suite que dans l'environnement `figure` il y a des images, une légende, et un label. La légende et le label sont optionnels. Les règles de LaTeX n'imposent pas cela : on peut mettre autre chose que des figures, on peut mettre plusieurs légendes, etc. Il y le cas simple (une image) et le cas compliqué. Dans le cas compliqué, on supposera que la structure est de la forme suivante

```
\begin{figure}[htbp]
  \begin{center}
     IM1 \hspace{3cm} IM2 \\
     IM3 \hfil IM4
  \end{center}
  \caption{Légende} \lavel{le-label}
\end{figure}
```

Il peut y avoir du `\centerline` pour centrer la chose. La légende peut ou non être dans le `center`. Dans l'exemple, le résultat XML contiendra deux paragraphes. On suppose que cela veut dire : deux lignes d'images. Le cas tordu est celui où on a une ligne d'images, et une ligne de légende. Dans ce cas, il y a des problèmes d'alignement vertical. Pour résoudre ce problème, les gens mettent en général une table à la place du `center`, ou alors utilisent `\subfigure`.

Dans XML, la sémantique de `figure` est la suivante : Si `figure` n'a pas l'attribut `rend='array'` cela signifie qu'il s'agit d'une image unique, définie par la valeur de l'attribut `file`. S'il y a l'attribut `rend='inline'`, l'image est dans le texte ; dans le cas contraire, il peut y avoir un élément `<head>`, qui est la légende. À part cela, le contenu est vide. Une image peut contenir des attributs de taille.

Si `figure` a un attribut `rend='array'`, cela signifie que l'objet sera placé hors texte, il peut y avoir un élément `<head>` (qui est la légende), et des éléments `<p>`, lesquels vont contenir les images ou tables d'images. Dans la section 3.1.3, on a donné un exemple de ces 3 cas. Ce que

donne le traducteur n'est pas conforme du tout à ce qui est dit plus haut. C'est pour cela qu'un post processeur est appliqué.

Le traitement du postprocesseur est comme suit. Il y a plusieurs cas à considérer. Supposons d'abord qu'il y a une table dans le résultat. Dans ce cas de figure, on va convertir tous les `<graphics/>` en `<figure>`, donc changer le nom de l'élément, et ajouter l'attribut `rend='inline'`. On va extraire un `<caption>`, le renommer en `<head>`, et le mettre au début. On va extraire un `<mylabel>`, et rajouter son `id` aux attributs de la figure. On va extraire tous les `<array>` (il peut y en avoir plusieurs), et les mettre dans un `<p>`. S'il reste des choses, le résultat sera mis dans une balise `<junk>`.

Supposons maintenant qu'il y a des `subfigure` dans la figure. Pour chaque paragraphe, on va extraire la liste des subfigures. Tout le reste sera du `<junk>`. On va construire un tableau comme suit : chaque paragraphe donne deux lignes ; la première ligne est formée des éléments `<texte>` des subfigures, et la deuxième est formée des éléments `<leg>`, chacun de ces éléments étant précédé de (a), (b), etc. Une fois ceci fait, on est ramené au cas précédent.

Troisième cas : pas de tableau, pas de subfigure, au moins deux images. On va faire comme si c'était un tableau, et traduire à la louche les commandes de type `\hfil`, `\hspace`, etc. L'algorithme utilisé est le suivant : on convertit les dimensions en points, puis on considère qu'un espace fait 4 points. Ainsi, un `\hspace` sera convertit en un certain nombre d'espaces. Si la valeur est négative, on laisse le `\hspace`. Les `\hfill` se transforment en un seul espace.

Le dernier cas est lorsqu'il n'y a qu'une seule figure. On essaie de faire au mieux.

3.2.20 Traitement des tableaux

Le traitement de l'environnement `table` est similaire à celui des figures, mais on fait une hypothèse : il n'y a qu'une table dans l'environnement. Autrement dit, la traduction de `\begin{table}[x]y\end{table}` sera `<Table>y</Table>`. Dans y on s'attent à voir un élément `<table>`, et éventuellement un label et un `<caption>`. Le label donnera un attribut `id` et le caption donnera un `<head>`, qui sera mis dans la `<table>`. On supprime l'attribut `rend='inline'`, et on rend le `<table>`. Tout ce qui reste dans le `<Table>` sera du junk.

La difficulté ici est la traduction des tables (environnements `tabular` et `array`), pas le post-traitement. Commençons par une partie non triviale : la traduction des arguments de positionnement. Il y a une procédure qui traduit r, l et c en `halign='left'`, etc. Il y a une procédure qui traduit r, l et c, avec une barre verticale autour (optionnelle), le résultat est `left-border='true'` ou `right-border='true'`. Le tout sera mis dans une balise `<col>` Par exemple, la traduction de `|c||lrr|`

sera

```
<col left-border='true' right-border='true' halign='center'/>
<col left-border='true' halign='left'/>
<col halign='right'/>
<col right-border='true' halign='right'/>
```

La traduction de `\multicolumn{a}{b}{c}` est la suivante : l'argument b est un argument de positionnement. Il est traité comme indiqué plus haut. Ceci donne un élément `<col>`, dont on récupère les attributs. L'argument a est le nombre de colonnes. Il donnera un attribut `cols='a'`. L'ensemble de ces attributs sera rajouté à l'élément `<cell>` courant (rappel : dans la pile, on

a </cell>, qui est codé comme \837!cell!, l'élément courant a le numéro 37). L'argument c n'est pas traduit : ce sera le contenu de la cellule courante.

La traduction de & est la suivante : si la pile contient <cell>, on dépile et on empile un élément cell. De façon précise, on dépile un </cell>, on construit un nouvel élément, on empile le </cell>, et le résultat est la concaténation de la balise de fin de la cellule précédente et la balise de début de la nouvelle. Dans les autres cas, & est une erreur.

La traduction de \\ est la suivante : on regarde d'abord s'il y a une étoile (ignorée), puis un argument optionnel x. Si la pile contient p ou cp, on fait comme si on avait vu \par (en particulier l'argument optionnel est ignoré). C'est une erreur si la pile ne contient ni p, ni cp, ni cell. Dans le dernier cas, on dépile cell, on dépile row, on empile row, on empile cell. S'il y a un argument optionnel x, il donnera un attribut space-before='x' à la <row>.

La traduction de \begin{tabular} est : on empile <table> (avec l'attribut rend='inline', puis on traite l'argument de position, on empile row et cell. Quand on voit la fin de l'environnement, on dépile ces trois quantités. Il y a un petit problème non résolu : que faire des paramètres de positionnement ? dans la version actuelle, tous les <col> ainsi générés sont regroupés dans un élément <modele/>. Il y a un autre problème : si la table se termine par un \\, on aura une dernière ligne qui contient une cellule vide. Il faut par ailleurs traiter le cas des \hline. Rappelons que les \hline se placent juste derrière les \\. Dans le cas où on voit un élément <row> qui contient un <cell> qui contient un <hline/>, tout ceci à la fin de la table, on supprime le dernier <row>, et on rajoute au <row> précédent l'attribut bottom-border='true'. Pour tous les autres \hline, qui sont dans une <cell> dans une <row>, on rajoute l'attribut top-border='true' à la <row>. Finalement, si la dernière ligne du tableau contient une seule cellule, et si elle est vide, on supprime la ligne. Tout ceci est fait par le post-processeur.

Il reste maintenant à traiter le cas des environnements array et eqnarray. Quand on voit le début d'un tel environnement, on procède comme suit. Dans le cas de eqnarray, on empile <formula type='display'> et <math>. Dans tous les cas, on empile <mtable>. On regarde s'il y a un argument optionnel (il est ignoré). Dans le cas de array, il y a un argument, qui sera ignoré. On empile <mtr> et <mtd>, puis on traite les cellules. Quand on voit \end{array} ou \end{eqnarray}, on dépile le </mtd> et le </mtd> (s'il y a de tels objets dans la pile). On dépile le </mtable>. Dans le cas de eqnarray, on dépile aussi le </math> et le </formula>.

Le traitement des cellules dans une table se fait ainsi : tant qu'on n'a pas fini, on appelle TE_translate_math avec 3 comme argument. Rappel : cette procédure scanne le texte, jusqu'à voir un \\, un &, un dollar ou un \end, et le traduit. C'est un erreur si on voit un dollar. On s'arrête si on voit un \end (le traducteur du \end vérifie que c'est bien le bon). Si on voit un &, on dépile un <mtd>, et on en empile un, comme dans le cas de tabular. Si on voit un \\, le résultat est le même, à un détail près : on ignore l'argument optionnel du \\. Par ailleurs, si le \\ est suivi d'un \end, on n'empile pas de <mtr>, <mtd>. Ceci a comme avantage de ne pas être obligé de tester s'il y a une une ligne vide à la fin. En particulier, le post processeur n'a pas besoin de traiter les <mtable>.

3.2.21 Post traitement des mathématiques

Si une formule de math est suivie par un \label, (traduit en <mylabel id=x/>), on rajoute le id à la formule de maths. Note : nous avons vu que le post-traitement des figures et tableaux fait également quelque chose avec les mylabel. Tous les mylabel qui restent seront convertis

comme suit : s'ils sont dans un <head>, on va les mettre juste derrière le <head>. Ensuite, si un <mylabel> suit un <head>, on va rajouter son id comme attribut au <head>. Notons qu'un élément peut avoir un seul attribut id. Ainsi, tous les <mylabel/> qui restent seront renommés en <anchor/>.

Si on ne sait pas traduire une formule de math, on demande à Ω de le faire. Dans le fichier, il faut bien entendu remplacer les \1foo! par \foo, et autre caractères spéciaux. Par contre, les \\9N! sont laissés tels quels (la valeur est un bout de code XML). On définit la macro suivante :

```
\def\9#1!{{{\cal ABCDEFGHIJKLMONPQ}#1{\cal ABCDEFGHIJKLMONPQ}}}
```

Supposons que le \cal soit traduit par Ω en X (c'est une expression horrible). La traduction de \917! aura donc la forme : X, suivi de la traduction de 17, suivi de X, dans un <mrow>. On remplace tous les X par ::sp_pattern (qui n'apparait pas dans le résultat MathML). Ensuite, on prend les <mrow> avec 2 ::sp_pattern dedans. On supprime les <mn> et </mn>, les <mrow> et </mrow>. Ceci donnera 17, que l'on va remplacer par \917!. Après cette manipulation un peu compliquée, les \9N! initiaux seront remplacés par la même valeur.

On fait également d'autres substitutions, du type, remplacer &infty; par ∞, ϱ par ϱ, etc, et d'autres qui servent à rectifier de petits bugs de Ω. Ceci dit, on a des formules F_1, F_2, etc. On remplace les <nosmath id=N/> par la N-ième formule. On fait de même pour les nomath, et les nodmath, mais en ajoutant les balises <math> et <formula>. On remplace également les \9N! par leur valeur ; a priori, les substitutions doit être faites en parallèle, ce qu'on fait, c'est de substituer les nomath puis les \9N!, tant qu'il reste un nomath ou un \\9N!.

Le post traitement fait d'autres choses : par exemple, un \vspace{3cm} va ajouter comme attribut space-before='3cm' au <p> qui suit (et s'il n'y en a pas ?). Nous avons dit plus haut ce qu'il en est des notes de bas de page, des figures, des tableaux, des labels. Il y a deux points importants : la section composition de l'équipe contient un module, par construction. On l'enlève, et on ne garde que le contenu du module. Finalement, on supprime tous les paragraphes vides.

3.2.22 Le fichier de configuration

Le fichier raweb-cfg.sty contient un certain nombre de commandes utiles lors de la génération du Pdf à partir d'un document XML. On commencera par discuter la page de titre. Celle-ci contient, de haut en bas : le logotype Inria, le nom développé de l'Inria (avec un lien hypertexte), le nom court du projet, le nom développé, l'unité de recherche, le thème, et le logotype rapport d'activité. Le document XML/FO contient une balise <INRIA>, qui va appeler la commande \foinria, suivie du nom du projet et de l'unité de recherche, et une balise <RATHEME> qui va appeler la commande \foratheme. On expliquera ici les deux commandes TEX ; celles-ci utilisent des dimensions données a priori (taille des logotypes par exemple), et d'autres obtenues par approximation successive (on imprime le document, et on mesure à la règle de combien il faut changer les valeurs).

On peut y voir le code suivant :

```
\newbox\ra@atxybox
\long\def\ra@atxy(#1,#2)#3{\global\setbox\ra@atxybox=\hbox
 {\unhbox\ra@atxybox
  \vtop to 0pt{\kern #2\hbox to 0pt{\kern #1\relax #3\hss}\vss}}}
```

La commande \ra@atxy prend trois arguments x, y, et z. Elle crée un boite horizontale, de largeur 0, qui contient un espace de longueur x, le texte z, et un \hss, un espace négatif, qui

s'ajuste pour que la largeur soit effectivement zéro. Le tout est mis dans une boite verticale, de hauteur 0, précédé d'un espace vertical, de longueur y, suivi d'un `\vss`, un espace vertical négatif. Le résultat est donc une boite de dimension 0 par 0, qui contient le texte z en position (x, y). Cette boite est rajoutée à la boite `\ra@atxybox`.

Il y a la définition suivante :

```
\def\ra@useatxy{{%
  \let\@themargin\oddsidemargin
  \vtop to 0pt{\kern-\headsep \kern-\topmargin \kern-\headheight
               \kern-1in \kern-\voffset
    \hbox to 0pt{\kern-\@themargin \kern-1in \kern-\hoffset
    \unhbox\ra@atxybox \hss}\vss}}%
    \global\let\@texttop\relax}
```

Lorsque LaTeX construit une page, il exécute la macro `\@texttop`. Supposons que la position absolue dans la page soit (X, Y). Par exemple X est la somme de trois quantités, la marge courante, la valeur de `\hoffset`, et un pouce. On redéfinit `\@texttop` comme étant un alias pour `\ra@atxy`.

Le code ressemble à `\ra@atxy` : essentiellement, on construit une boite de taille 0 par 0, qui contient un texte z, en position (x, y). Le texte z est le contenu de la boite `\ra@atxybox`, et la position (x, y), est la position $(0, 0)$, en coordonnées absolues dans la page, donc l'opposé de (X, Y). Cette boite est imprimée, et `\@texttop` est remise à son ancienne valeur (qui ne fait rien). Ce mécanisme permet donc de placer des images ou du texte, en coordonnées absolues sur une page (la page de titre).

Le fichier contient le code suivant :

```
\def\foinria{%
  \ra@atxy(7.8cm,2.5cm){\includegraphics[width=5.7cm]{Logo-INRIA-couleur}}%
  \ra@atxy(55mm,173mm){\includegraphics{LogoRA\ra@year}}%
 \setbox0\hbox to 14cm{%
     \noindent\hskip3cm\hfill
     {\fontencoding{T1}\fontfamily{ptm}\fontseries{m}%
   \fontshape{n}\fontsize{10pt}{12pt}\selectfont
        \href{http://www.inria.fr}
         {INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE}}%
      \hskip-5cm\hfill}%
  \null\vskip0.7cm \leavevmode\hskip-3.5cm\box0\null\vskip2cm\vfil}
```

Le code est formé de deux parties : d'abord, on demande l'inclusion du logotype rapport d'activité de l'année courante, et du logotype Inria, en position absolue sur la page. On construit ensuite une boite, `\box0`, qui contient essentiellement le nom Inria développé dans la bonne police de caractère, avec un lien HTML. Notons que, pour centrer horizontalement ce nom, on utilise une façon peu catholique de procéder. La boite est précédée de 7mm d'espace vertical (elle va se retrouver sous le logotype), suivie de 2 centimètres d'espace vertical, et d'un `\vfil`. Ceci fait que le texte qui suit est centré verticalement, entre ce `\vfil` et le suivant, voir le bout de code qui suit, et qui contient un `\vskip8cm`.

Le fichier contient la définition suivante :

```
\def\foratheme#1{\vskip8cm \vfil
  \ra@atxy(70mm,174mm) {\hbox to 72mm{%
       \hrulefill\hspace{8mm}
         \def\firstchar##1##2\relax{##1}
```

```
\href{http://www.inria.fr/recherche/equipes/projets_theme\firstchar#1\relax
     .fr.html}{TH\'EME \uppercase{#1}}%
  \hspace{8mm}\hrulefill}}
```

On suppose que l'argument de la commande est de la forme `1A` ou `3b`, donc un chiffre et une lettre. La commande `\firstchar`, définie localement va récupérer le chiffre, tandis que le `\uppercase` va convertir `3b` en `3B`. Le premier argument de `\href` se termine donc par `projets_theme3.fr.html`, est le deuxième argument sera « THÈME 3 ». On met 8 millimètres d'espace de part et d'autre du texte, et un filet de la taille adéquate (le tout dans une boite de largeur 72 millimètres). Cet objet est positionné sur la page en position absolue (70mm,174mm).

Bibliographie

[1] David Carlisle, Michel Goossens, and Sebastian Rahtz. De XML à PDf avec `xmltex` et Passive TEX. In *Cahiers Gutenberg*, number 35-36, pages 79–114, 2000.

[2] M. Goossens, F. Mittelbach, and A. Samarin. *The LaTeX companion*. Addison Wesley, 1993.

[3] Michel Goossens and Jean-Yves Le Meur. Afficher les documents scientifiques sur le Web. In *Cahiers Gutenberg*, number 28-29, pages 181–195, 1998.

[4] D. E. Knuth. *The TeXbook*. Addison Wesley, 1984.

[5] The Unicode Consortium. *The Unicode Standard, version 3.0*. Addison Wesley, 2000.

Table des matières

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803