HEVEA, un traducteur de LaTeX vers HTML en Caml

Luc Maranget

Inria Rocquencourt, 78153 Le Chesnay Cedex
Luc.Maranget@inria.fr

1. Introduction

Écrire ses papiers en LaTeX est une tradition relativement bien établie de notre domaine dont on peut penser qu'elle se maintiendra. Cela tient sans doute d'une part à l'excellente qualité finale des impressions papier obtenues et d'autre part à des facteurs plus sociologiques, tels la gratuité de LaTeX, le poids des habitudes et un certain consensus entre auteurs et éditeurs.

Toutefois, nous disposons aujourd'hui, avec le web, d'un nouveau moyen de diffusion de nos œuvres. La technique la plus simple de publication électronique d'un article est la mise à disposition d'une versions formatée par LaTeX (`.dvi` ou PostScript). Cette démarche simplifie le travail de l'auteur qui réutilise le fichier qu'il envoie à son éditeur. La démarche du lecteur internaute est ensuite finalement la même que celle du lecteur d'un journal scientifique ou des actes d'une conférence : après une lecture rapide du document il imprimera l'article pour le lire tranquillement si il est intéressé.

La publication du document en HTML offre des possibilités supplémentaires : recherche de mots-clés dans le document, lecture non linéaire à partir d'une table de liens, accès par un simple click de souris à d'autres documents, tels des articles référencés ou des documents complémentaires omis dans la version papier faute de place. En outre, le fichier HTML sera plus petit, ce qui minimise les temps de chargement, et sa lecture se fera à travers l'outil universel de l'internaute : le browser web. En effet, tous les systèmes (et en particulier les ordinateurs personnels) ne proposent pas un visualiseur `dvi` ou PostScript. Par ailleurs, la piètre qualité des documents imprimés à partir du HTML interdit l'écriture d'articles directement en HTML.

Les possibilités offertes par la lecture web se justifient particulièrement dans le cas des manuels de logiciels : en effet on lit rarement ces manuels d'une traite, on butine plutôt dedans à la recherche d'une information précise. Par ailleurs, la lecture sur écran est ici un avantage qui offre au lecteur un aller et retour rapide entre une fenêtre où il utilise le logiciel et une autre où il consulte sa documentation. Enfin, les mises à jour des manuels lors des changements de versions des logiciels sont immédiates.

Les manuels étant souvent de taille importante, et leur mise au point demandant de nombreuses itérations, il est particulièrement important de disposer d'un traducteur rapide. L'utilisateur doit également pouvoir facilement paramétrer les traductions ou même d'étendre la fonctionnalité du traducteur. En effet, il est vain de croire que l'auteur d'un traducteur puisse tout prévoir ; d'autre part il est parfois impossible de rendre en HTML certaines constructions LaTeX ou certains symboles. Dans ce dernier cas, il faudra rendre le symbole par un équivalent, dont le choix dépend du goût de l'utilisateur et de la nature de son document.

Les lecteurs intéressés par par un manuel de référence d'HEVEA consulteront [7]. En effet, cet article décrit plutôt le fonctionnement interne d'HEVEA. Il commence établir le principe d'une traduction lexicale de LaTeX vers HTML (sections 2). De ce principe résulte l'architecture d'HEVEA qui distingue nettement l'émission de HTML par le gestionnaire de sortie (section 3) et la reconnaissance de l'entrée par l'analyseur lexical principal (section 4). La section 4 explique également comment reconnaître et traduire les formules mathématiques en HTML. La réalisation du mécanisme de commande et d'appel par nom de LaTeX et son utilisation pour paramétrer HEVEA sont ensuite décrites en détail (sections 5 et 6). Enfin, je conclue par une rapide comparaison d'HEVEA avec d'autres traducteurs et par quelques perspectives d'avenir.

2. Principes

2.1. TeX ou LaTeX ?

Une première réponse est brutale, après tout nous écrivons plutôt en LaTeX. Néanmoins la question mérite d'être un peu approfondie, car elle permet de mieux définir les objectifs d'un traducteur.

Rappelons que LaTeX est en fait un ensemble de macros TeX. LaTeX est généralement préféré car il se charge de maints détails de mise en page et de composition, ce qui doit permettre à l'auteur de se concentrer sur l'écriture de son texte et non pas sur sa composition.

De fait, le source TeX apparaît comme celui d'un langage très influencé par son implantation. Autrement dit, il faut comprendre comment fonctionne TeX pour pouvoir s'en servir. Une telle démarche est raisonnable pour l'auteur qui compose finement son texte, elle est dissuasive pour l'auteur qui veut simplement écrire. LaTeX propose un langage source qui met plus en avant les concepts que leur réalisation. Typiquement, pour comprendre, par exemple, un changement de style des polices de caractères, on doit en TEX se référer à un ≪ état courant ≫ du moteur interne, tandis que LATEX propose des concepts de plus haut niveaux, comme une commande qui compose son argument avec un attribut de police déterminé. De même un des concepts centraux de TEX est la boite, qui de plus peut être horizontale (un mot) ou verticale (un paragraphe), ce concept provient du monde de l'imprimerie et intéresse finalement peu l'auteur du document.

Une conséquence intéressante est qu'un document écrit en LATEX ressemble beaucoup plus à un programme écrit dans un langage moderne (donc réputé de haut-niveau), qu'un document écrit en TEX qui lui ressemble à de l'assembleur. On peut même penser qu'il est impossible d'écrire un traducteur complet de TEX vers HTML, car HTML n'offre pas énormément de contrôle sur l'aspect final du document, qui d'ailleurs dépend fortement du dispositif utilisé pour le visualiser (cependant, un traducteur commercial TEXpider [9] suit cette démarche, mais il y a fort à parier que ce programme est bien plus complexe qu'HEVEA).

En résumé, traduire LATEX vers HTML est possible et raisonnable, parce que ces deux langages sont tous deux des langages qui spécifient une certaine structure du document, plus qu'ils ne décrivent complètement quel sera l'aspect du produit final. En outre, le langage LATEX est relativement uniforme, toutes les constructions ayant sensiblement la même syntaxe, ce qui facilite grandement sa compréhension par un programme.

Classiquement, un traducteur d'un langage vers un autre possède une phase initiale d'analyse syntaxique qui, à partir du source construit un arbre de syntaxe abstraite. Cet arbre est en fait le support de la sémantique du langage, qui très souvent procède par assemblage récursif de sous-eléments. La traduction finale se fait ensuite par un parcours de cet arbre ou d'un arbre transformé. Les compilateurs rapides et peu optimisants font l'économie de la construction de l'arbre de syntaxe et génèrent directement leur produit dans la phase d'analyse syntaxique. Dans notre cas, c'est cette démarche qui est adoptée, tant pour des raisons d'efficacité que parce que, comme on va le voir, les langages LATEX et HTML ont des structures syntaxiques proches. En outre il n'est pas immédiate de définir complètement LATEX sous forme d'une grammaire BNF, un préalable indispensable si l'on veut effectivement construire un arbre de syntaxe abstraite d'un document LATEX. Ici, on se contente d'admettre que les documents ont des structures comparables dans les deux langages, sans expliciter exactement ce qu'est cette structure.

2.2. Structuration

Donc, il est frappant de constater que les documents LaTeX et HTML ont nombre de structures communes. Les environnements LaTeX (ouverts par "`\begin{`*env*`}`" et refermés par "`\end{`*env*`}`" correspondent aux éléments de niveau bloc HTML (≪ *block-level elements* ≫, ouverts par la balise "<*bloc*>" et refermés par la balise "</*bloc*>"). Les documents complets sont structurés selon le même modèle de sections et sous-sections (les ≪ sectional units ≫ de LaTeX, introduites par les commandes "`\section{...}`", "`\subsection{...}`", etc et les ≪ *headers elements* ≫ "`<H1>...</H1>`", "`<H2>...</H2>`", etc. de HTML).

Le principe général de la traduction est donc celui d'une traduction lexicale : reconnaître les entités lexicales et les traduire une à une la structuration du document restant inchangée. Par entité lexicale, j'entends ici un groupe de caractères significatif et de taille réduite. Par exemple :

```
\begin{quote}
\begin{flushright}
Ceci est une citation de  deux lignes,\\
avec fer \begin{em}à droite\end{em}.
\end{flushright}
\end{quote}
```

Se formate comme suit :

Ceci est une citation de deux lignes,
avec fer à droite.

Une traduction simple et logique en HTML est :

```
<BLOCKQUOTE><DIV ALIGN=right>
Ceci est une citation de deux lignes,<BR>
avec fer <EM>à droite</EM>.
</DIV>
</BLOCKQUOTE>
```

On a donc traduit `\begin{quote}` en `<BLOCKQUOTE>`, la construction `\begin{flushright}` en `<DIV ALIGN=right>`, `\\` en `<BR>`, etc. L'imbrication des environnement et des éléments est la même dans les deux langages.

De même la traduction d'une liste est particulièrement directe. En LaTeX, on a :

```
\begin{itemize}
\item Premier item.
\item Deuxième item.
\end{itemize}
```

En HTML, il vient :

```
<UL>
<LI>Premier item.
```

```
<LI>Deuxième item.
</UL>
```

Évidemment, les structures des documents LaTeX et HTML ne sont pas identiques en tout point. Par exemple, l'effet des déclarations de changement de style LaTeX court de leur occurrence à la fin de l'environnement ou du groupe courant, tandis que les éléments de niveau texte HTML (≪ *text-level elements* ≫) ont une structure bien parenthésée, leur effet courant entre une balise d'ouverture et une balise de fermeture. Ainsi, le texte ≪ normal, *en italique* normal ≫ est obtenu à partir du source LaTeX suivant :

```
normal, {\itshape en italique} normal
```

Une traduction possible en HTML est "

```
normal, <I>en italique</I> normal
```

En outre, la définition de HTML [8] interdit la présence d'éléments de niveau bloc à l'intérieur des éléments de niveau texte. Nous verrons à la section 3, comment HEVEA garantit le bon parenthésage de toutes les balises d'ouverture et de fermeture, ainsi que cette contrainte.

Pour traiter cette différence de structure entre les deux langages, ainsi que d'autres comme les commandes définies par l'utilisateur, HEVEA s'écarte de la pure traduction lexicale pour revenir à un modèle plus proche de TeX, où les diverses commandes sont interprétées comme des ordres de modification de l'état courant du traducteur.

2.3. Organisation d'HEVEA

La traduction d'HEVEA se divise en deux sous-tâches, d'une part reconnaître du source LaTeX que l'on supposera correct, et d'autre part produire du HTML correct. Le programme HEVEA s'articule donc en deux sous-unités, d'une part un *analyseur lexical principal* et d'autre part un *gestionnaire de sortie*. L'interaction entre les deux sous-unités est simple : l'analyseur réalise des appels aux fonctions du gestionnaire de sortie.

L'analyseur se garde bien d'échouer sur une entrée incorrecte, lorsque l'incorrection n'est pas trop grave (par exemple, une commande inconnue). En effet, ces erreurs peuvent résulter d'une insuffisance d'HEVEA et elle ne présagent pas d'une traduction inacceptable. Néanmoins, un avertissement est donné à l'utilisateur, afin de lui permettre de contrôler visuellement le résultat. En revanche le gestionnaire de sortie échoue lorqu'il a pu détecter qu'il est en train d'émettre du HTML incorrect.

HEVEA lit des documents composé de plusieurs fichiers sources, car il interprète correctement la construction `\input` de LaTeX, mais sa sortie est un fichier HTML unique. Le résultat d'HEVEA peut ensuite être

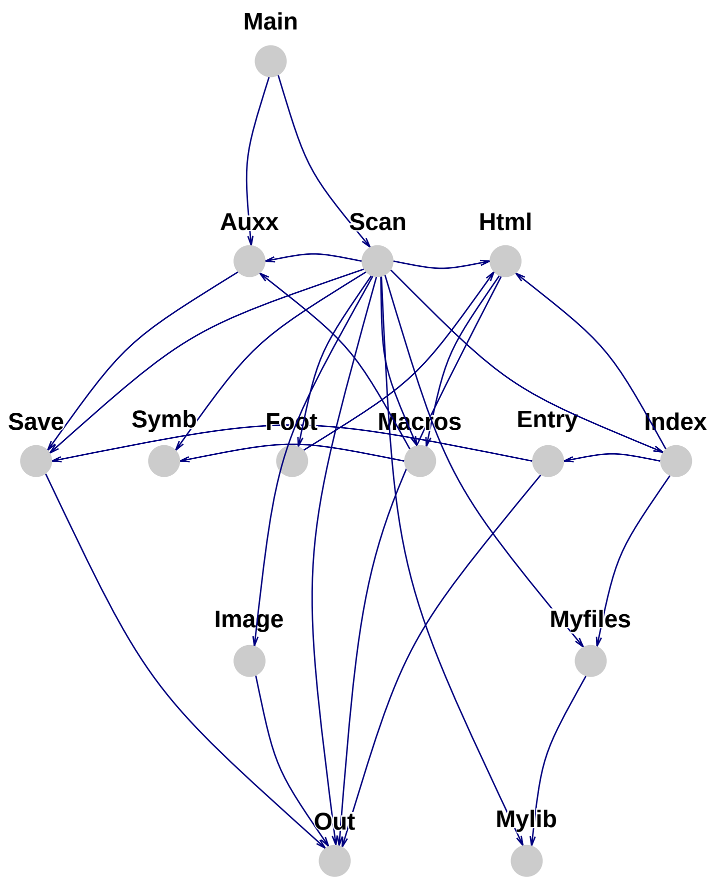


FIG. 1: Les dépendances entre modules d'HEVEA

segmenté en plusieurs fichiers par un autre programme HACHA, qui est inclus dans la distribution d'HEVEA. HEVEA s'efforce de réaliser toutes les constructions LaTeX décrites dans le ≪ *Blue Book* ≫ [4], à l'exception de certains packages.

L'analyseur lexical principal est compilé par `ocamllex`, le générateur d'analyseurs lexicaux inclus dans la distribution d'Objective Caml [5], il est réalisé par le module `Scan`. Le gestionnaire de sortie est réalisé directement en Objective Caml par le module `Html`. HEVEA comprend de nombreux autres modules, Le total du source d'HEVEA et de HACHA s'élevant à un peu moins de 7000 lignes de code. Enfin le développement d'HEVEA, à partir du prototype `htmlgen` [6] de X. Leroy et la rédaction de la documentation [7] m'ont pris de l'ordre de quatre ou cinq mois à mi-temps.

Les dépendances entre les modules sont données à la figure 1, on notera l'analyseur principal `Scan` et le gestionnaire de sortie `Html` et leur dépendance. HEVEA comprend d'autres analyseurs lexicaux, ce sont les modules :

`Auxx` Lecture des fichier auxiliaires `.aux` produits par LaTeX. Les informations lues permettent à `Scan` d'introduire dans sa sortie les étiquettes des références croisées et des citations bibliographiques.

`Save` Lecture des argument des commandes (voir section 5).

`Entry` Lecture du fichier d'index `.idx`. Cette lecture est optionnelle, elle s'impose lorsque l'utilisateur modifie ce fichier après sa production par LaTeX.

Trois modules réalisent la gestion des entrées et des sorties :

`Mylib` Localisation du répertoire de la librairie d'HEVEA et gestion du chemin de recherche des fichiers.

`Myfiles` Localisation et ouverture des fichiers.

`Out` Gestion de toutes les sorties.

Le module `Out` présente une abstraction d'un flot de sortie. Ce flot peut être un fichier ou un tampon mémoire. Tous les analyseurs lexicaux produisent et rendent leur résultat par des appels aux fonctions de ce module.

Les autres modules réalisent certaines des fonctionnalités d'HEVEA :

`Macros` Gestion de l'environnement de commandes LaTeX.

`Index` Production du ou des index.

`Foot` Gestion des notes de bas de page.

`Symb` Définition des symboles mathématiques.

`Image` Production d'un fichier LaTeX qui regroupe les sous-parties du source traduit à transformer en images. Le document final contient ensuite des liens vers les images en lieu du source transformé. Cette procédure est toujours demandée explicitement par l'utilisateur.

3. Le gestionnaire de sortie

3.1. Les blocs

Comme on l'a déjà abondamment évoqué, un document HTML est structuré selon des éléments de niveau bloc imbriqués. Cette structure est naturellement reprise par le module `Html`.

Un bloc du gestionnaire de sortie est caractérisé d'abord par un canal de sortie. Le canal de sortie du premier bloc ouvert est un fichier, tandis que la sortie de tous les autres blocs ouverts ultérieurement sont des tampons mémoire. Les deux types de canaux présentent une interface uniforme et sont définis dans un autre module, le module `Out`. En plus du canal de sortie, un bloc comprend une étiquette (qui est le nom de l'élément HTML représenté par le bloc, un argument de bloc (certaines balise d'ouverture ont des arguments comme "`<DIV ALIGN=right>`" par exemple), ainsi que deux listes de changements de style : les changements de style effectués et les changements de style en attente. Par définition, les changements de style actifs sont la concaténation des changements de style effectués et des changements de style en attente.

Les changements de styles sont rendus par des éléments de niveau texte. Du point de vue d'HEVEA, ces éléments sont de trois sortes, purs changements de style (par exemple, "`<I>`"), changement de taille (par exemple "`<FONT SIZE=4>`") et changement de couleur (par exemple "`<FONT COLOR=blue>`"). Ces styles sont rendus par le type `style` du module `Macros` :

```
type style =
  Style of string
| Size of int
| Color of string
```

Voici ensuite une interface simplifiée pour le module `Html`, qui va permettre une première description du gestionnaire de sortie.

```
val open_block : string -> string -> unit
val close_block : string -> unit
val open_mod : Macros.style list -> unit
val put : string -> unit
```

Le gestionnaire de sortie maintient une pile de blocs ouverts et l'appel de `open_block` déclenche l'empilage du bloc courant. Un nouveau bloc est ensuite ouvert, son canal de sortie est un nouveau tampon, sa liste de changements de style effectués est vide et sa liste de changements de style en attente est constituée des changements de style actifs du bloc précédent.

L'appel à `put` effectue la copie de son argument dans le canal de sortie courant. En outre, cette fonction effectue les changements de style en attente. C'est à dire qu'elle émet les balises d'ouverture appropriées et transfère les styles correspondants dans la liste des changements de styles effectués.

La reconnaissance d'une déclaration LaTeX de changement de style (par exemple "`\itshape`") entraîne un ou plusieurs appels à `open_mod` qui prennent chacun un style en argument (par exemple "`Style "I"`"). La fonction `open_mod` se contente de placer le style donné en argument en tête des changements en attente.

Enfin, la dernière fonction `close_block` ferme le bloc courant. Dans un premier temps, l'argument de `close_block` est comparé à l'étiquette courante, en cas de différence une erreur fatale est signalée. Puis, dans un premier temps, elle émet les balises de fermeture correspondant aux changements de styles *effectués*, qui sont donc écrits en fin du canal à fermer. Ensuite, le bloc en sommet de pile est dépilé, son canal de sortie devient donc le canal courant et les listes de changement de style retrouvent leurs anciennes valeurs. Puis, le gestionnaire de sortie émet la liste des balises de fermeture des changements de style effectués (qui ont été potentiellement refermées à l'ouverture du sous-bloc qui vient d'être fermé), transfère ces changement de style de la liste des changements effectués vers la liste des changements en attente, émet la balise d'ouverture du bloc fermé augmentée de son éventuel argument, copie le contenu du bloc fermé et émet sa balise de fermeture.

Le retardement de l'émission des balises permet d'annuler un bloc facilement, dans ce cas on se contente de dépiler le bloc en sommet de pile après vérification de l'étiquette donnée en argument à `close_block` et rien n'est émis. Cette procédure s'applique en particulier lorsque le tampon mémoire du bloc refermé est vide, une situation provoquée entre autres par les groupes LaTeX vides "`{}`", qui souvent utilisés pour indiquer une séparation de lexème comme dans "`groupes \LaTeX{} vides`".

L'architecture du gestionnaire de sortie impose le bon parenthésage des balises et un respect de la bonne imbrication des deux types d'éléments HTML, à condition bien entendu d'ouvrir et de fermer tous les éléments de niveau bloc par `open_block` et `close_block`, et d'amorcer tous les changements de style par `open_mod`. C'est évident pour les éléments de niveau bloc, par l'existence de la pile des blocs ouverts et le contrôle à la fermeture des blocs. Le cas des changements de style est à peine plus compliqué, toutes les balises ouvertes lors de la réalisation effective des changements de style sont fermées avant d'émettre la balise de fermeture du bloc courant. Enfin, la fermeture des changements de style effectués avant toute ouverture de bloc garantit que l'on ouvrira jamais d'élément de niveau bloc à l'intérieur d'un élément de niveau texte.

3.2. Réalisation des changements de style

Considérons à nouveau l'exemple suivant :

```
\begin{quote}
normal, {\itshape en italique} normal
\end{quote}
```

à traduire en :

```
normal, <I>en italique</I> normal
```

Le mécanisme des blocs réalisé par le module `Html` de la section précédente produit effectivement cette traduction, si, anticipant sur la reconnaissance de LaTeX, on admet que l'environnement `quote` se traduit par un élément de niveau bloc `BLOCKQUOTE` (ouvert par "`Html.open_block "BLOCKQUOTE" ""`" et refermé par "`Html.close_block "BLOCKQUOTE"`"), le groupe `{... }` par un bloc sans argument et de nom vide `""`, "`\itshape`" par un appel "`Html.open_mod (Style "I")`" et que tout autre caractère c donne lieu à un appel "`Html.put` c".

LaTeX et en HTML n'offrent pas exactement les même possibilités de changer les attributs des polices de caractères. En effet, d'une part, ces attributs sont plus nombreux en LaTeX ; d'autre part les documents HTML peuvent facilement inclure des changements de couleur. Une traduction possible des nombreux attributs de style LaTeX vers HTML est donc une combinaison de changements d'attributs et de couleurs. La table 1 décrit quelques-uns des choix par défaut d'HEVEA pour rendre les différentes variations de forme des police (déclarations "`\...shape`"). Ainsi, la reconnaissance de "`\slshape`", par exemple, entraîne deux appels à "`open_mod`", avec les arguments successifs "`Style "I"`" et "`Color "maroon"`".

italique	`\itshape` ou `\it`	`<I>`
penchée	`\slshape` ou `\sl`	`<I> <FONT COLOR=maroon>`
petites capitales	`\scshape` ou `\sc`	`<FONT COLOR=blue>`
normal	`\upshape`	annule les autres déclarations

TAB. 1: Traduction des changements de forme

Le cas de la déclaration `\upshape` est le plus délicat. Par exemple, le source

```
{\itshape en italique {\upshape normal} en italique aussi.}
```

est formaté comme suit :

> *en italique* normal *en italique aussi.*

Le point de vue de LaTeX est que les formes italiques et droites sont simplement deux valeurs distinctes des formes de polices. Mais HTML ne possède pas d'élément équivalent à la déclaration "`\upshape`", pour lui une lettre de forme droite est simplement une lettre de forme normale. On ne pourra donc traduire la déclaration "`\upshape`" dans un contexte où "`\itshape`" est actif qu'en refermant l'élément "`I`" actif. Dans le cas de l'exemple, on souhaite donc la traduction suivante :

```
<I>en italique </I>normal<I> en italique aussi.</I>
```

Plus généralement, le sens de "`\upshape`" ne peut être que de fermer toutes les balises de style actives qui traduisent les autres changements de forme, c'est à dire ici "`<I>`", "`<FONT COLOR=maroon>`" et "`<FONT COLOR=blue>`".

Le module `Html` fournit donc une fonction supplémentaire

```
val erase_mods : Macro.style list -> unit
```

Cette nouvelle fonction annule la liste des styles passée en argument de la façon suivante : les changements de style à annuler et encore non-effectués sont simplement retirés de la liste des changements de style en attente, tandis que les changements de style effectués qui précèdent le dernier changement à annuler sont effectivement refermés. Les changements refermés et qui ne doivent pas être annulés sont ensuite rouverts en attente.

Ainsi, le source :

```
\slshape
{\bfseries penché et gras
\upshape droit et gras}
penché et fin
```

qui se formate ainsi :

> ***penché et gras*** **droit et gras** *penché et fin*

donnera :

```
<I><FONT COLOR=maroon><B>penché et gras
</B></FONT></I><B>droit et gras</B>
<I><FONT COLOR=maroon>penché et fin</FONT></I>
```

(Les sauts de ligne produits ont été changés). Remarquons que toutes les balises actives avant le texte "`droit et gras `" sont fermées puis que la balise "`<B>`" est rouverte.

L'utilisation de tampons mémoire et le retardement des changements de style permet de limiter l'émission des balises de style inutiles. Par exemple, étant donné le source suivant :

```
\itshape{\bfseries italiques grasses} italiques
```

On obtient en sortie :

```
<I><B>italiques grasses</B></I><I> italiques</I>
```

Cette sortie appelle deux remarques :

- L'élément "I" n'est pas encore ouvert au moment de la traduction du groupe "{\bfseries..." car le changement de style "Style "I"" est encore en attente.
  On comparera avec :
  ```
  \itshape italiques {\bfseries italiques grasses} italiques
  ```
  qui se traduit en
  ```
  <I>italiques </I><I><B>italiques grasses</B></I><I> italiques</I>
  ```
- La sortie n'est pas la meilleure possible Une sortie optimisée éviterait la fermeture suivie d'une ouverture "</I><I>" à la traduction de "}". Plus généralement, on pourrait éviter de fermer puis de rouvrir les styles déjà effectués au moment de l'ouverture d'un sous-bloc à l'intérieur des quels ces styles ne sont pas refermés. J'ai préféré éviter de compliquer encore le gestionnaire de sortie.

4. Reconnaissance de LaTeX

Cette section détaille l'analyseur principal de HEVEA.

4.1. Unité lexicales élémentaires

Chaque unité lexicale reconnue entraînera une action, émission de HTML ou modification de l'état interne du traducteur. La reconnaissance des unité lexicales n'offre aucune difficulté particulière si on utilise `ocamllex` : il suffit d'identifier les expressions régulières qui les décrivent. La lecture répétitive du flot d'entrée `lexbuf` est assurée par des appels récursifs à l'entrée principale `main` de l'analyseur lexical ou à d'autres entrées.

Voici par exemple la reconnaissance des commentaires.

```
rule main = parse
  '%' {comment lexbuf}
      ⋮
and rule comment = parse
  '\n' {main lexbuf}
| _    {comment lexbuf}
```

L'action réalisée ici est extrêmement simple : le contenu d'un commentaire est ignoré. On remarquera la présence du flot d'entrée `lexbuf` comme argument explicite des entrées de l'analyseur.

Considérons ensuite l'analyse de quelques éléments lexicaux particuliers. On se limitera dans un premier temps, aux éléments lexicaux constitués d'un seul caractère, par souci de simplicité. Un certain nombre de caractères sont actifs en LaTeX, ce sont par exemple les caractères de début et de fin de groupe “{” et “}” :

```
| '{'  {Html.open_block "" "" ; main lexbuf}
| '}'  {Html.close_block "" ; main lexbuf}
```

L'action réalisée est l'ouverture et la fermeture d'un bloc par le gestionnaire de sortie. Il s'agit d'une simplification comme on le verra à la section suivante. Notons que, pour vérifier le bon parenthésage des ouvertures et fermetures de groupes, on peut, pour le moment, s'en remettre à la pile des blocs du gestionnaire de sortie.

Voici ensuite quelques exemple de sortie de caractères, les caractères qui “<” et “>” sont traduits, ainsi que l'espace insécable. Les autres caractères sont simplement recopiés dans le flot de sortie courant.

```
| '<'  {Html.put "&lt;" ; main lexbuf}
| '>'  {Html.put "&gt;" ; main lexbuf}
| '~'  {Html.put " " ; main lexbuf}
| _    {Html.put (lexeme lexbuf) ; main lexbuf}
                     (* (lexeme lexbuf) est le lexème~lu *)
```

Ce source illustre une première utilisation de la fonction “lexeme” qui, lorsqu'elle est appelée dans une action de l'analyseur renvoie une chaîne contenant les caractères reconnus.

4.2. Mathématiques

Le principal défi de la traduction de LaTeX vers HTML est posé par les formules mathématique. Il faut d'abord traduire les nombreux symboles mathématiques de LaTeX, comme par exemple “\in” à rendre par “$\in$”. Pour ce faire, HEVEA utilise une extension non-standard de HTML 3.2 implantée par certains browsers (et en particulier les browsers Netscape récents) : tout texte *text* compris dans l'élément “<FONT FACE=symbol>*text*</FONT>” est affiché à l'aide d'une police de symboles qui comprend les lettres grecques et les symboles mathématiques les plus courants. Ainsi, le signe d'appartenance “$\in$” sera traduit en “<FONT FACE=symbol>Î</FONT>”, le code du caractère “Î” correspondant au signe “$\in$” dans la police de symboles. (Le module Symb contient les définitions des symboles rendus selon ce schéma, ces définitions sont mises à disposition de l'analyseur principal par le module Macros.)

Ensuite, il faut pouvoir formater les formules mathématiques dont les règles de composition s'éloignent sensiblement du modèle standard de texte composé en paragraphes commun au mode paragraphe de LaTeX et à HTML. Par exemple la formule “`$$\int_1^2 xdx = \frac{3}{2}$$`”, à formater en mode ≪ *display math* ≫ est composée ainsi par LaTeX :

$$\int_1^2 xdx = \frac{3}{2}$$

Les tables de HTML (c'est à dire, les éléments de niveau bloc “<TABLE>...<\TABLE>”) offrent une possibilité de contrôler le placement relatif par le browser. HEVEA réalise le mode ≪ *display math* ≫ de LaTeX en ouvrant une telle table, les sous-formules étant ensuite arrangées en lignes et colonnes. Ainsi, la formule ci-dessus sera rendue par HEVEA, puis par *Netscape communicator* comme montré à la figure 2 (les bordures des tables qui servent au formatage sont rendues visibles). Les tables

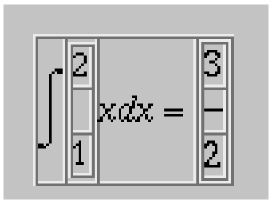


FIG. 2: Composition d'une formule en mode *display*

introduites par HEVEA pour la composition des formules sont appelées ≪ *displays* ≫ Il existe deux sortes de displays, les displays horizontaux qui sont des tables d'une ligne et les displays verticaux qui sont des tables d'une colonne. Les cases des displays (colonnes dans le premier cas, lignes dans le second) peuvent elles même contenir des displays. Ainsi la formule de la figure 2 est constituée d'un display horizontal qui à quatre cases dont la deuxième et la quatrième sont des displays verticaux. Remarquons que les colonnes des displays horizontaux sont centrées verticalement et les lignes des displays verticaux sont centrées horizontalement. Le centrage des cases des displays peut être mis en défaut par certaines formules déséquilibrées (par exemple une fraction avec un numérateur et un dénominateur de tailles verticales très différentes), toutefois la formule reste généralement compréhensible.

L'analyseur principal d'HEVEA utilise deux références booléenes “`in_math`” et “`display`” qui indiquent les modes courants. Commençons par examiner un effet du mode mathématique : les caractères alphabétiques sont systématiquement mis en italiques lorsque le mode mathématique est actif. Pour ce faire, l'analyseur principal comprend la clause suivante :

```
| ['a'-'z' 'A'-'Z']+
   {let lxm = lexeme lexbuf in
```

```
 if !in_math then begin
   Html.put "<I>"; Html.put lxm; Html.put "</I>"
  end else
   Html.put lxm ;
  main lexbuf}
```

Notons que les balises “`<I>`” et “`</I>`” sont ici émises directement car on peut être sûr que “`lxm`” ne contient pas de changement de style. Remarquons aussi que cette clause est plus prioritaire que la clause par défaut “`| _ {...}`”, de part la règle classiques de reconnaissance des expressions régulières les plus longues (≪ *longuest match* ≫) et parce que la clause “`['a'-'z''A'-'Z']+`” précède la clause par défaut dans le source de l'analyseur.

Il est moins évident de détailler un exemple d'utilisation du mode display. Disons que de nombreuses constructions sont affectés par ce mode, exposants, indices, grands symboles, fractions, etc. Lorsque le mode display n'est pas actif, HEVEA génère uniquement des éléments de niveau texte et donc ne génère pas d'élément `TABLE`. En effet, les tables sont toujours précédées et suivies d'un retour à la ligne lorsqu'elles sont rendues par les browsers et leur introduction briserait la continuité du texte. Par exemple la formule “`$\int_1^2 xdx = \frac{3}{2}$`” placée dans le cours du texte sera composé à l'aide des seuls éléments de niveau texte, ce qui donne :

```
<FONT FACE=symbol>ò</FONT>
<SUB><FONT SIZE=2>1</FONT></SUB><SUP><FONT SIZE=2>2</FONT></SUP>
<I>xdx</I> = 3/2
```

À comprendre comme suit :

4.3. Réalisation des modes mathématique et *display*

Cette section décrit en détail la réalisation des displays. Dans un souci de simplification, seuls les displays horizontaux seront abordés, on les désignera désormais simplement comme des ≪ *displays* ≫. Le gestionnaire de sortie d'HEVEA fournit les fonction suivantes :

```
val open_display : unit -> unit
val close_display : unit -> unit
val item_display : unit -> unit
```

L'ouverture d'un display est essentiellement l'ouverture d'une table (un bloc d'étiquette `TABLE`), d'une ligne (un bloc `TR`) et d'une première case (un bloc `TD`). La fermeture d'un display est donc la fermeture successive de trois blocs, `TD`, `TR` et `TABLE`. Enfin, la spécification d'une limite de case (par `item_display`) est la fermeture d'un bloc `TD` suivie de l'ouverture d'un nouveau bloc `TD`.

L'introduction des displays complique notablement la gestion des blocs. En effet, l'ouverture (par `Html.open_block`) d'un bloc quelconque en mode display doit entraîner l'ouverture d'un display à l'intérieur du nouveau bloc, display qui devra aussi être fermé avant de refermer ce bloc. En effet, si tel n'était pas le cas, les balises "`</TD>`" et "`<TD>`" émises à l'occasion des limites de cases seraient improprement placées à l'intérieur de blocs quelconques et non pas dans un bloc `TR`. Il faut également *a priori* émettre une limite de case avant l'ouverture de bloc et après la fermeture de bloc, afin d'assurer le centrage vertical d'éventuels displays verticaux présents dans le bloc.

L'ouverture et la fermeture d'un bloc du gestionnaire de sortie ne se feront donc plus directement, mais à l'aide des deux fonction auxiliaires suivantes :

```
let top_open_block label args =
  if !display then begin
    Html.item_display () ;
    Html.open_block label args ;
    Html.open_display ()
  end else
    Html.open_block label args
and top_close_block label =
  if !display then begin
    Html.close_display () ;
    Html.close_block label ;
    Html.item_display ()
  end else
    Html.close_block label
```

Le mode mathématique est ouvert et refermé par le caractère "`$`" tandis que le mode ≪ *display maths* ≫ est ouvert et fermé par le groupe de deux caractères "`$$`". Par ailleurs, la construction "`\mbox{`*text*`}`", formate toujours *text* en mode texte (c'est à dire non mathématique et non display, même lorqu'elle apparaît en mode mathématique ou display). Néanmoins *text* peut contenir des éléments à formater en mode mathématique ou display. C'est à dire que l'on peut supposer que les trois constructions précédentes sont arbitrairement imbricables (ce n'est pas tout à fait le cas en LaTeX, mais qui peut le plus peut le moins).

Les changements des variables d'état "`in_math`" et "`display`" sont donc gérés à l'aide de deux piles "`math_stack`" et "`display_stack`". En outre, il devient nécessaire lorsque l'on rencontre une accolade fermante "`}`" de savoir si elle ferme un "`{`" ou un "`mbox{`" car, dans le second cas, il faudra éventuellement revenir en mode mathématique. Pour ce faire, l'analyseur principal maintient un environnement courant "`cur_env`" et une pile "`stack_env`" des environnements et groupes LaTeX ouverts :

```
let open_env name =
  push stack_env !cur_env ;
  cur_env := name

and close_env name =
  if name <> !cur_env then
    failwith ("Fatal error in latex environments: "^
      name^" closes "^ !cur_env) ;
  cur_env := pop stack_env
```

Par convention, `cur_env` est la chaîne vide “`""`” dans un groupe ordinaire “`{...}`”, “`"*math"`” dans un groupe mathématique “`$...$`” ou “`$$...$$`” et “`"*mbox"`” dans “`\mbox{...}`”.

Examinons maintenant les reconnaissances des débuts et fin de groupes mathématiques :

```
| '$' | "$$"
   {let lxm = lexeme lexbuf in
   if !in_math then begin
      in_math := pop stack_math ; close_env "*math" ;
      if lxm = "$" then
        top_close_block ""
      else begin
        display := pop stack_display ;
        Html.close_display () ; top_close_group "DIV"
      end
   end else begin
      push stack_math !in_math ; in_math := true ;
      if lxm = "$" then
        top_open_block "" ""
      else begin
        top_open_group "DIV" "ALIGN=center" ;
        push display_stack !display ; display := true ;
        Html.open_display ()
      end ;
      open_env "*math"
   end ;
   main lexbuf}
```

On remarquera que “`in_math`” indique si l'on est en présence d'une ouverture ou d'une fermeture et que seule la gestion de “`display`” change entre “`$`” et “`$$`”. En outre, les groupes “`$$...$$`” sont centrés comme en LaTeX.

Le cas de “`mbox{`” est donné ci-dessous, on notera que l'on n'utilise pas `top_open_group` pour ouvrir le bloc, mais directement `Html.open_block` :

```
| "\\mbox{"
   {push math_stack !in_math ; in_math := false ;
```

```
  if !display then Html.item_display () ;
  push display_stack !display ; display := false
  Html.open_block "" "" ;
  open_env "*mbox" ;
  main lexbuf}
```

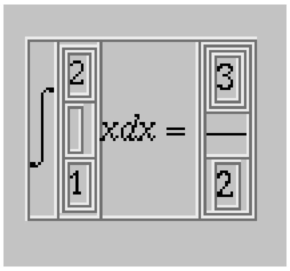
Restent les clauses pour "{" et "}" qui remplacent les clauses simplistes de la section 4.1 :

```
| '{' {top_open_block "" "" ; open_env "" ; main lexbuf}
| '}'
  {if !cur_env = "*mbox" then begin
     Html.close_block "" ;
     in_math := pop stack_math ; display := pop display_stack ;
     if !display then Html.item_display () ;
     close_env "*mbox"
  end else begin
    top_close_block "" ;
    close_env ""
  end ;
  main lexbuf}
```

On peut se convaincre que, sur un source LaTeX correct, les modes sont bien gérés : à chaque empilage d'une variable d'état lors d'une ouverture de construction ("{", "$", etc.) correspond un dépilage de cette même variable d'état lors de la fermeture de cette construction ("}", "$", etc.).

Mieux, les clauses précédentes permettent à d'autres constructions LaTeX bien parenthésées d'initier un passage en mode display, par un code similaire à celui de "$$". HEVEA procède effectivement de la sorte pour les tableaux (environnements `array` et `tabular`) dont les éléments sont systématiquement formatés en mode display, ce qui en améliore sensiblement le rendu.

Dans la réalisation d'HEVEA, le gestionnaire de sortie s'efforce de ne générer les displays et les limites de cases que lorsqu'elle sont nécessaires : si le display fermé ne comprend qu'une case, le contenu de cette case est simplement copié dans le canal de sortie courant, les limites de cases ne sont réellement émises qu'autour des cases contenant un display vertical, etc. Ce genre de décision ne peut être prise qu'à la fermeture des displays et des cases, le gestionnaire de sortie doit donc identifier certains événements, comme l'émission effective d'un display vertical. Ces dispositions limitent grandement le nombre de tables émises dans la sortie d'HEVEA. Par exemple, on comparera la figure 2 avec ce rendu de la même formule sans les astuces :



On notera que le relatif coût de la recopie des tampons de sortie trouve ici une compensation en terme de taille et de complexité du HTML généré.

L'analyseur principal reconnaît d'autres constructions LaTeX, telles les tableaux, les listes, les exposants, les indices, les délimiteurs (`\left`...`\right`) etc. Je ne détaillerai pas la réalisation de ces constructions, qui exploite largement les possibilités de retard, d'annulation et de modification de l'émission du HTML par le gestionnaire de sortie.

5. Les commandes

Les commandes LaTeX forment la vaste majorité des éléments actifs dont le traitement entraîne des actions plus spécifiques que l'émission de caractères.

5.1. Syntaxe

Par souci de simplification et compte tenu de la pratique des auteurs de documents LaTeX (et non de fichiers de style) les noms de commandes reconnus par HEVEA sont décrit par l'expression régulière suivante :

```
let command =
  '\\' (('@' ? ['A'-'Z' 'a'-'z']+ '*'?) | [^ 'A'-'Z' 'a'-'z'])
```

C'est à dire que les noms de commande commencent toujours par "\", suivi d'une suite non-vide de caractères alphabétiques ou d'un unique caractère non-alphabétique (cf. la commande accent aigu "`\'`"), la suite de caractères alphabétiques pouvant être préfixée par "`@`" (cas des commandes internes d'HEVEA), ou suivie de "`*`" (cas des variantes étoilées de LaTeX).

Selon [4], les commandes prennent des arguments et sont invoquées ainsi (où *commande* est un nom de commande) :

$$command\texttt{\{}arg_1\texttt{\}\{}arg_2\texttt{\}}\ldots\texttt{\{}arg_n\texttt{\}}$$

Le premier argument d'une commande peut être optionnel, auquel cas on l'invoque ainsi :

$$command\texttt{[}arg_1\texttt{]\{}arg_2\texttt{\}\{}\ldots\texttt{\{}arg_n\texttt{\}}$$

Il faut noter que les délimiteurs d'arguments "`{`" et "`}`" (ou "`[`" et "`]`") peuvent apparaître à l'intérieur des arguments, à condition d'être bien parenthésés.

La lecture de quelques fichiers source LaTeX montre qu'en pratique, la syntaxe de l'invocation de commande est moins rigide. On rencontre fréquemment les variations suivantes :

- Le caractère "`{`" ou "`[`" qui introduit les arguments peut être précédé d'espaces, de sauts de ligne et même de commentaires...
- Les délimiteurs "`{`" et "`}`" peuvent être omis lorsque l'argument d'une commande à un seul argument est un seul caractère (cf. "`\'el\`eve`"), ou un nom de commande (cf. "`\^\i`").
- Certaines commandes internes prennent plus d'un argument optionnel.

HEVEA traite donc les commandes LaTeX, plus les trois variations ci-dessus, rien n'est fait pour reconnaître la syntaxe la plus générale des macros TeX.

5.2. Commandes internes

L'analyseur lexical d'HEVEA reconnaît les occurrences des noms de commande. À chaque nom de commande est associé le nombre et la valeur par défaut des arguments optionnels et le nombre total d'arguments de la commande. Cette information est suffisante pour récupérer les arguments en attente dans le flot d'entrée courant. Les arguments sont lus l'un après l'autre par un nouvel analyseur lexical défini dans le module `Save` et qui suit les règles de la section précédente. L'analyseur du module `Save` comprend en gros une entrée `arg` qui renvoie une chaîne contenant l'argument en tête du flot d'entrée (sans ses délimiteurs) et une entrée `opt` qui fait de même pour un argument optionnel. Remarquons que l'analyseur principal ignore les subtilités de la reconnaissance des arguments réalisée par le module `Save` et qu'il s'en trouve simplifié d'autant.

L'analyseur principal contient une simple clause pour reconnaître tous les noms de commande, le nom reconnu étant ensuite recherché parmi les commandes directes c'est à dire les commandes directement réalisées par l'analyseur principal :

```
| command
    {let lxm = lexeme lexbuf in
    begin match lxm with
      ⋮
    end}
```

Le choix de reconnaître les commandes directes par un filtrage "`match...`" plutôt que par de nombreuses clauses de l'analyseur limite sérieusement la taille de ce dernier, et ceci sans augmenter sensiblement les temps d'exécution comme je l'ai vérifié.

Dans le cas d'une commande directe, les arguments sont lus explicitement et l'action est réalisée par du code Caml. Cette procédure concerne en premier lieu les commandes internes d'HEVEA. Voici par exemple la clause du `match` ci-dessus qui reconnaît la commande interne `\@print` utilisable dans le source LaTeX pour écrire directement dans la sortie d'HEVEA :

```
| "\\@print" ->
     let arg = Save.arg lexbuf in
     Html.put arg ; main lexbuf
```

Il existe de nombreuses autres commandes internes et en particulier `\@open` et `\@close` qui sont utilisables pour appeler les fonctions cruciales du gestionnaire de sortie `open_block` et `close_block` à partir du source LaTeX.

Certaines des commandes pré-définies de LaTeX sont également reconnues par l'analyseur principal. Voici par exemple la reconnaissance de la commande LaTeX `\typeout`, qui écrit son argument sur la console :

```
| "\\typeout" ->
    let what = Save.arg lexbuf in
    prerr_endline what ;
    main lexbuf
```

(La fonction Caml `prerr_endline` affiche son argument dans la sortie d'erreur du programme.)

5.3. Appel par nom

Cette section décrit la réalisation par HEVEA des commandes définies par l'utilisateur. On se limite à des commandes utilisateur à *nargs* arguments, tous non-optionnels. Une telle commande se définit ainsi :

`\newcommand{`*command*`}[`*nargs*`]{`*body*`}`

(La valeur par défaut de l'argument optionnel *nargs* est zéro.)

Vu de TeX, l'appel de commande donne lieu à une simple réécriture du flot d'entrée selon un principe de macro expansion : le corps de commande *body* est mis en tête du flot d'entrée, certains lexèmes (les paramètres formels `#1`, `#2`,... ,`#9`) étant remplacés par les arguments donnés à l'appel (ou paramètres effectifs). Si l'on fait abstraction de certaines constructions TeX qui modifient ce processus, il s'agit là d'une réalisation de l'appel par nom, selon la règle de copie la plus simple.

La réalisation de cet appel par nom par HEVEA est différente : d'une part, le flot d'entrée d'HEVEA est un flot de caractères (et non de lexèmes comme en TEX) ; d'autre part, compte tenu d'une limitation bien compréhensible des analyseurs lexicaux de Caml, il n'est pas possible d'ajouter des éléments devant le flot d'entrée courant. On va donc, d'une part, retarder la substitution des paramètres formels, afin de préserver les limites de lexèmes ; et, d'autre part, réaliser l'illusion d'un flot d'entrée continu. Pour donner cette illusion, on utilise la fonction de bibliothèque Caml `Lexing.from_string` qui crée un flot d'entrée à partir d'une chaîne passée en argument. On se donne donc la définition auxiliaire suivante, qui applique l'analyseur passé en premier argument sur la chaîne passée en second argument :

```
let scan_this lexfun s =
  lexfun (Lexing.from_string s)
```

Le module `Macros` gère dynamiquement un environnement des commandes. À chaque nom de commande enregistré sont associés, un motif qui résume les informations sur les arguments de la commande (nombre, valeur par défaut des arguments optionnels), ainsi qu'un corps de commande. On enregistre et on récupère une définition de commande par les deux fonctions suivantes :

```
val def_macro: string -> pat -> string -> unit
val find_macro: string -> pat * string
```

Voici ensuite un source simplifié qui schématise la la réalisation de l'appel de commande :

```
  | command
      {let lxm = lexeme lexbuf in
      begin match lxm with
        ⋮
(* Appel de commande, cas général *)
      | _ ->
         let pat,body = Macros.find_macro name in
         push env_stack !env_args ;
         env_args :=  make_env pat lexbuf ;
         scan_this main body ;
         env_args := pop env_stack ;
         main lexbuf
       end}
(* Fin du flot d'entrée *)
  | eof ()
```

Le fragment ci-dessus utilise deux nouvelles variables globales, une référence vers un tableau de chaînes "`env_args`" et une pile de ces tableaux "`env_stack`". Le tableau "`env_args`" incarne la liaison courante entre paramètres formels et paramètres effectifs. Initialement il n'y a ni corps de commande ni arguments et la référence "`env_args`" contient un tableau de taille zéro. L'environnement "`env_args`" est initialisé lors de l'appel de commande, par l'appel de fonction "`make_env pat lexbuf`" dont nous nous contenterons d'admettre qu'il lit les arguments en attente dans le flot courant "`lexbuf`" à l'aide de l'analyseur `Save.arg`. Notez que l'environnement actif à la reconnaissance de l'appel de commande est sauvé (par `push`) et restauré (par `pop`) avant et après l'analyse du corps de cette commande. En effet, le "`main lexbuf`" final doit se faire dans un contexte identique à celui qui prévalait avant la reconnaissance de l'appel.

La pseudo-expression régulière "`eof`" signale la fin du flot d'entrée. Dans le cas d'un corps de commande, elle indique la fin du corps et l'analyse se termine par l'action vide "`{()}`". Par conséquent, l'analyse d'un corps de commande "`scan_this main body`" termine, la traduction du corps de commande substitué ayant été au préalable générée par effet de bord.

Le remplacement des paramètres formels par leur valeurs est retardé jusqu'à ce que l'analyseur principal découvre leurs occurrences :

```
| '#' ['1'-'9']
    {let lxm = lexeme lexbuf in
    let arg = !env.(Char.code (lxm.[1]) - Char.code '1') in
    let old_env = !env_args in
    env_args := pop env_stack ;
    scan_this main arg ;
    push env_stack !env ;
    env_args := old_env ;
    main lexbuf}
```

L'action de la clause ci-dessus réalise la substitution en récupérant le paramètre effectif "`arg`" à sa place dans l'environnement "`env_args`". La chaîne "`arg`" est ensuite analysée, mais elle peut elle-même contenir des paramètres formels à instancier. Ces paramètres font référence à l'environnement tel qu'il était au moment de l'appel de la commande dont le corps est en cours d'analyse, c'est à dire au moment de la création de "`env_args`". L'environnement correspondant, qui se trouve en sommet de la pile "`env_stack`" est donc restauré par "`env_args := pop env_stack ;`" avant l'analyse de "`arg`".

Il nous reste à examiner la reconnaissance des définitions de commande. On se donne d'abord une fonction `make_pat` qui renvoie un motif à partir d'une liste d'arguments par défaut et d'un nombre total d'argument, ainsi qu'une fonction `save_opt` qui lit un argument optionnel dans un flot d'entrée et renvoie un argument par défaut en cas d'échec. L'analyseur principal comporte ensuite la clause suivante :

```
| "\\newcommand"
   {let name = Save.arg lexbuf in
    let nargs = save_opt "0" lexbuf in
    let body  = Save.arg lexbuf in
    Macro.def_macro (make_pat [] (string_of_int nargs)) body ;
    main lexbuf}
```

Cette présentation simplifiée de la réalisation des commandes est suffisante pour comprendre un des mécanismes centraux d'HEVEA. Les constructions supplémentaires effectivement réalisées sont les commandes utilisateur avec argument optionnel, la redéfinition de commande et une gestion à la LaTeX des espaces qui suivent les commandes sans arguments (ces espaces n'apparaissent pas dans le HTML produit).

Il importe de remarquer qu'HEVEA ne réalise pas toute la fonctionnalité des macros de TeX (et donc des commandes de LaTeX). En particulier, les commandes doivent apparaître suivies de tous leurs arguments.

5.4. Environnements de LaTeX

HEVEA traite les environnements LaTeX tels que décrits dans [4, sections C.8.2 et C.8.3]. Je présente le cas simplifié d'un environnement sans argument.

La définition d'environnement est donc

`\newenvironment{`*env*`}{`$body_1$`}{`$body_2$`}`

Elle provoque la création de deux commandes `\`*env* et `\end`*env* de corps respectifs $body_1$ et $body_2$. L'ouverture d'environnement "`\begin{`*env*`}`" déclenche l'appel de `\env`, tandis que la fermeture "`\end{`*env*`}`" se traduit par un appel à `\end`*env*. Enfin, les environnements définissent la portée des définitions de commande, comme en LaTeX. Ceci est réalisé par l'enregistrement du nom des commandes définies dans une liste à purger par la fonction `close_env` dont je ne détaillerai pas la nouvelle définition (voir la section 4.3 pour l'ancienne définition). Voici des clauses simplifiées pour "`\begin`" et "`\end`" :

```
| "\\begin"
   {let env = Save.arg lexbuf in
   open_env env ;
   scan_this main ("\\"^env) ;
   main lexbuf}
| "\\end"
   {let env = Save.arg lexbuf in
   scan_this main ("\\end"^env) ;
   close_env env ;
   main lexbuf}
```

Le code ci-dessus est subtil. En effet, l'environnement *env* est ouvert *avant* d'exécuter la commande initiale *env*, tandis que *env* est fermé *après* l'exécution de la commande finale `\end`*env*. Cela rend possible les définitions d'environnement qui ouvrent et ferment d'autres environnements, comme par exemple :

```
\newenvironment{monquote}{\begin{quote}\em}{\end{quote}}
```

La reconnaissance de "`\begin{monquote}`" entraîne l'initialisation de l'environnement courant à `"monquote"` (c'est à dire que la référence "`cur_env`" contient "`"mon_quote"`"), puis l'analyse de "`\monquote`" qui à son tour entraîne celle de "`\begin{quote}`". L'environnement courant `"monquote"` est donc immédiatement empilé et remplacé par `"quote"`. Normalement, si les environnements LaTeX sont correctement imbriqués, la reconnaissance de "`\end{monquote}`" se fera alors que l'environnement courant est `"quote"`. Mais, les commandes "`\endmonquote`" puis "`\end{quote}`" seront alors exécutées et l'environnement courant sera revenu à `"monquote"` au moment du contrôle.

6. Configuration d'HEVEA

Un intérêt évident de la reconnaissance de la définition de commande (et d'environnement) est la possibilité de traduire les fichiers source LaTeX qui en contiennent. Un autre intérêt, non moins utile, réside dans la possibilité pour l'utilisateur de paramétrer les traductions ou d'étendre la fonctionnalité d'HEVEA.

6.1. Constructions de base

HEVEA reconnaît finalement assez peu de commandes de façon directe, la majorité des commandes de base de LaTeX sont définies dans un fichier de configuration `hevea.sty` qui est chargé avant de commencer la traduction. Les appels de ces commandes seront ensuite traités par le mécanisme général de l'appel de commande. Par exemple, voici la définition pour HEVEA de quelques commandes et environnements disponibles par défaut en LaTeX :

```
%% Special caracters
\newcommand{\%}{\@print{%}}
\newcommand{\$}{\@print{$}}
\newcommand{\#}{\@print{#}}
  ....

%% Some environnments
\newenvironment{center}{\@open{DIV}{ALIGN=center}}{\@close{DIV}}
\newenvironment{quote}{\@open{BLOCKQUOTE}{}}{\@close{BLOCKQUOTE}}
  ...
```

HEVEA pouvant traiter, les boites, les compteurs LaTeX et les macros conditionnelles de TeX, on dispose de fait d'un petit langage de programmation suffisant pour écrire en LaTeX des définition relativements complexes. Voici par exemple la définition de l'environnement `figure` extraite de `hevea.sty`.

```
%% Box for caption
```

```
\newsavebox{\@caption}
\newcommand{\caption}[1]{\sbox{\@caption}{#1}}
%% Figure
\newenvironment{figure}[1][]{%
  \caption{}\refstepcounter{figure}%
  \@open{BLOCKQUOTE}{}\@print{<HR>}%
}{%
  \par\begin{center}%
  Figure \thefigure\iffrench~\fi: \usebox{\@caption}%
  \end{center}%
  \@print{<HR>}\@close{BLOCKQUOTE}}
```

La définition en LaTeX de la majorité des constructions reconnues permet de simplifier l'analyseur principal. En outre, et c'est peut-être le plus important, l'utilisateur peut redéfinir ces constructions. Supposons par exemple que l'on souhaite mettre en italiques toutes les citations d'un texte. Il suffira alors de redéfinir l'environnemnt `quote` :

```
\renewenvironment{quote}
  {\@open{BLOCKQUOTE}{}\itshape}
  {\close{BLOCKQUOTE}
```

Cette possibilité de changer les définitions par défaut d'HEVEA s'applique à l'ensemble des constructions qui ne sont pas reconnues directement par l'analyseur principal. Un cas particulier intéressant est celui des constructions non-reconnues par HEVEA. Par exemple, le symbole "`\mapsto`" ("$\mapsto$") n'est pas reconnu car il n'existe pas dans la police symbole utilisable par les browsers. Si l'utilisateur fait le choix de rendre ce symbole par une flèche rouge, il écrira :

```
\newcommand{\mapsto}{{\red\rightarrow}}
```

(La commande "`\red`" est définie comme "`\@color{red}`" dans le fichier de configuration `hevea.sty`.)

6.2. Détail des changements de style

Nous sommes maintenant en mesure de comprendre le mécanisme complet des changements de style et de leur paramétrage par l'utilisateur. Les appels au gestionnaire de sortie sont réalisés par les commandes internes à un argument `\@style`, `\@fontcolor` et `\@fontsize`, dont la reconnaissance provoque un appel à `Html.open_mod`. Par exemple, "`\@style{I}`" entraîne un appel "`open_mod (Style "I")`", "`\@fontcolor{blue}`" un appel "`open_mod (Color "blue")`", etc.

Reprenons l'exemple des changements de forme (cf. la table 1). Les déclarations en vieux style LaTeX sont réalisée par des appels à ces commandes internes, comme montré dans la table **??**.

Les déclarations LaTeX 2_ε doivent annuler tous les autres changements de forme avant de procéder à leur changement. En effet, les formes de

italique	`\newcommand{\it}{\@style{I}}`
penchée	`\newcommand{\sf}{\@style{I}\@color{maroon}}`
petite capitales	`\newcommand{\sf}{\@color{blue}}`

TAB. 2: Déclarations de changement de forme, vieux style

italique	`\newcommand{\itshape}{\@anti{\sc\sf}\it}`
penchée	`\newcommande{\sfshape}{\@anti{\it\sc}\sf}`
petite capitales	`\newcommand{\sfshape}{\@anti{\it\sf}\sc}`
droite	`\newcommand{\upshape}{\@anti{\sc\sf\it}}`

TAB. 3: Déclaration des changements de style, LATEX 2_ε

polices sont mutuellement exclusives. Le fichier `hevea.sty` contient donc les déclaration de la table 3. Où la commande interne `\@anti` annule les changements de style données en argument. Lorsqu'il a reconnu la commande interne `\@anti` et lu son argument dans une chaîne "`arg`", l'analyseur principal procède en deux étapes :

1. Il analyse "`arg`", mais le gestionnaire de sortie est au préalable modifié pour rendre la liste des styles actifs à la fin de l'analyse au lieu de générer une sortie. Ainsi, dans le cas de la déclaration "`\upshape`", la liste "`[Color "blue" ; Style "I" ; ...]`" sera retournée.
2. Il appelle `Html.erase_mods` avec en argument la liste des styles rendus à l'étape précédente.

Un utilisateur qui souhaiterait modifier les déclarations de changement de forme pour par exemple rendre la forme penchée en gras se contentera donc d'écrire :

```
\renewcommand{\sf}{\@style{B}}
```

ou même, s'il ne craint pas une redéfinition de "`\bf`" :

```
\renewcommand{\sf}{\bf}
```

Ainsi la déclaration "`\sfshape`" finira par ouvrir un élément de niveau texte "`B`" et un appel à toute autre déclaration de changement de forme à l'intérieur de l'élément "`B`" le fermera.

7. Conclusion

Je ne me suis pas livré à une comparaison exhaustive des nombreux traducteurs de LATEX vers HTML, dont une liste très à jour peut être trouvée sur le web[1]. HEVEA ne peut de toutes façons être comparé

[1] `http ://www.loria.fr/services/tex/german/outils.html`

qu'à des traducteurs relativement complets. J'ai sélectionné `latex2html`, certainement le traducteur le plus connu et `tth` un traducteur assez semblable à HEVEA mais qui est écrit en langage C. Ces deux traducteurs sont gratuits et à peu près libres de droits. Les performances des traducteurs sur un gros document LaTeX sont données par la table 4. Pour ce qui est de la qualité du HTML produit, disons que les sorties de `tth` et HEVEA sont plus sobres que celles de `latex2html`. En outre, quelques validations de la sortie des trois traducteurs montrent qu'il produisent en général du HTML conforme à la spécification, les erreurs rencontrées étant assez mineures.

On peut argumenter sans fin sur le choix de traduire les formules mathématiques par des images (comme `latex2html`) ou en utilisant une extension non-standard de HTML (comme `tth` et HEVEA). Disons que la qualité finale des images générées par la dernière version disponible de `latex2html` est bonne, mais que la traduction des symboles et autres constructions mathématiques en HTML est plus élégante et économique, tant en terme de taille des fichiers à transmettre que de temps de traduction, la production des images comptant pour une bonne part du temps d'exécution de `latex2html`.

En ce qui concerne les performances pures, `latex2html` est complètement disqualifié. Je pense pouvoir dire comme [6] que `latex2html` est inapte en pratique à la traduction de gros documents. Cet échec est dû à l'écriture de ce traducteur en Perl, l'existence même de `latex2html` et les progrès de sa dernière version par rapport aux précédentes sont plus un prodige de programmation qu'un exemple de développement raisonné de logiciel. Les autres chiffres révèlent qu'HEVEA fait plus que jeu égal avec un programme semblable écrit en C (en effet la version gratuite de `tth` ne découpe pas sa sortie en plusieur fichiers). Il n'y a donc pas lieu de se contraindre à utiliser un langage sans système de modules, typage avancé ou gestion automatique de mémoire, pour un hypothétique avantage en efficacité pure.

Je pense en effet que l'utilisation de Caml a grandement facilité le développement d'HEVEA. Par exemple, le gestionaire de sortie ne se préocupe ni du débordement des tampons, ni de la nature du canal de sortie courant, tampon mémoire ou fichier : ces détails sont réglés une fois pour toute par le module `Out`, dont le code est largement utilisé par toutes les parties d'HEVEA qui produisent une sortie (cf. la figure 1). En outre le code du module `Out` est simple, en raison de la gestion automatique de la mémoire en Caml.

HEVEA est le traducteur rapide, configurable et de large spectre réclamé dans [6] (voir aussi [2]). HEVEA est une amélioration conséquente de l'outil spécialisé `htmlgen` que X. Leroy avait finalement écrit pour traduire le manuel d'Objective Caml. L'extension la plus significative

	temps	taille (`.html` + `.gif`, en ko)
HEVEA (+HACHA)	6,3 s (+ 3,9 s)	1082 + 0
`tth`	10,5 s	956 + 0
`latex2html`	30 min	1854 + 28

TAB. 4: Traduction du manuel Coq (sur un Pentium 200 Mhz)

est sans aucun doute la reconnaissance des définitions de commande qui rend HEVEA configurable et extensible par ses utilisateurs, sans que ceux-ci aient besoin de connaître Caml et de recompiler HEVEA.

Toutefois HEVEA n'est bien entendu pas parfait. Sa configurabilité pourrait être améliorée et l'ensemble des constructions LaTeX ou TeX reconnues pourraient être étendu. Le HTML produit devrait être conforme à la dernière version (4.0) de la spécification HTML. Enfin, la publication récente d'un standard pour écrire les mathématiques en HTML [3] rend caduque l'utilisation de l'extension non-conforme "`<FONT FACE=symbol>`", ainsi que la formatage approximatif des formules à l'aide de tables. Si ce standard devait être compris par les browsers de grande diffusion, alors il importerait qu'HEVEA produise du HTML étendu conforme.

Références

[1] M. Gooseens, F. Mittelbach and A. Samarin. *The LaTeX Companion* Addison-Websley, 1994.

[2] M. Gooseens and Janne Saarela. *From LaTeX to HTML and back* TUGboat 16(3), 1995. Également disponible sur le web en `http://www.loria.fr/services/tex/outils/l2hen.ps.gz`

[3] P. Ion et *al Mathematical Markup Language (MathML) 1.0 Specification.* Disponible sur le web en `http://www.w3.org/TR/REC-MathML`.

[4] L. Lamport. *A Document Preparation System System, LaTeX, User's Guide and Reference Manual.* Addison-Websley, 1994.

[5] X. Leroy *et al. Objective Caml, version 2.00* Logiciel et documentation disponible en `http://caml.inria.fr/`.

[6] X. Leroy, *Lessons learned from the translation of documentation from LaTeX to HTML.* ERCIM/W4G Int. Workshop on WWW Authoring and Integration Tools, 1995. Également disponible sur le web en `http://cristal.inria.fr/~xleroy/w4g.html`.

[7] L. Maranget. *HEVEA, version 1.02.* Logiciel et documentation disponible en `http://para.inria.fr/~maranget/hevea/`.

[8] D. Ragget. *HTML 3.2 Reference Specification*. Disponible sur le web en `http://www.w3.org/TR/REC-html32`, 1997.

[9] *TEXpider, version 1.05.* Version de démonstration du logiciel et documentation disponible en `http://www.micropress-inc.com/webb/wbstart.htm`.