Mini-projet d'analyse lexicale et syntaxique

David Albert, Valentine Fleuré, Marion Schaeffer

Mai 2019

Table des matières

Introduction

Tout au long de ce semestre, nous avons dû travailler autour d'un mini-projet pour le cours d'Analyse Lexicale et Syntaxique. Nous avons choisi de travailler sur un convertisseur de code Latex en code HTML. Dans ce rapport, nous commencerons par vous présenter le sujet que nous avons choisi, nous continuerons par détailler les outils que nous avons utilisé et les méthodes que nous avons développé avant de finir par vous donner les résultats que nous avons obtenus.

1 Présentation du sujet

Pour notre mini-projet, nous avons choisi de travailler sur un convertisseur de code Latex en code HTML. Nous avons décider d'étudier ce sujet car il présentait un véritable défi : de nombreux exemples existent sur internet traitant des calculatrices ou du pseudo-code, mais très peu traitent des chaines de caractères à convertir. D'ailleurs, nous en reparlerons lorsque nous aborderons les difficultés rencontrées. Latex est un code très utilisé pour mettre en forme des rapports comportant de la syntaxe mathématiques. Nous l'utilisons très souvent durant notre scolarité, c'est donc un outil qui nous est familié. Il existe de très nombreuses commandes, autant pour écrire des mathématiques à proprement parler que pour la mise en page du document lui-même. Nous nous sommes plutôt attardé sur la seconde partie pour la mettre en correspondance avec du code HTML, qui s'occupe de la mise en page pour le web. Ces deux langages sont des langages à balises, ce qui simplifie notre tâche. Cependant leur structure est très différente, ce qui nous a posé beaucoup de difficultés. Pour avoir un aperçu de notre travail, voici les principales commandes que nous avons chercher à transférer :

Code Latex	Code HTML	Fonctionnalités
\begin{document} ... \end{document}	<body> ... </body>	Regroupe le contenu de la page
\section{...}	...	Titre de la plus grande section
\subsection{...}	...	Titre de la deuxième section
\subsubsection{...}	...	Titre de la troisième section
\paragraph{}	...	Titre d'un paragraphe
\begin{itemize} ... \end{itemize}	...	Annonce le début et la fin d'une liste
\item	...	Ajoute un item à une liste
\begin{comment} ... \end{comment}	< !-- ... – >	Insère un commentaire dans le code
\includegraphics{...}	<img src= "...">	Insère une image dans le document
\textbf{...}	...	Met le texte en gras
\textit{...}	...	Met le texte en italique
\underline{...}	<u> ... </u>	Souligne le texte
\sout{...}	<s> ... </s>	Raye le texte

Le but était donc de saisir les commandes Latex dans un fichier texte servant d'entrée à notre programme, d'enregistrer la sortie dans un fichier HTML et d'observer le résultat sur un navigateur web (firefox dans notre cas).

2 Outils utilisés

Pour réaliser notre projet, nous avons utilisé les outils **bison** et **flex**. Le premier outil, **bison**, est un compilateur de compilateur, version gnu de la commande yacc. Il construit un compilateur d'un langage décrit par un ensemble de règles et actions d'une grammaire. Le second outil **flex**, qui est la version gnu de la commande lex construit un analyseur lexical à partir d'un ensemble de règles et actions décrites par des expressions régulières. Nous les avons préférés à lex et yacc car il nous permettait de manipuler des chaines de caractères plus simplement. Nous avons également insérer du code C dans notre bison et dans notre flex pour pouvoir ajouter du traitement aux chaines de caractères.

3 Méthodes développées

Pour créer notre grammaire nous avons dans un premier temps créer des variables :
BLANC [⍑]
LETTRE [a-zA-Z]
CHIFFRE [0-9]
SIGNE [+-*/=]
PONCT [. ? ! :' ;)(_]

TEXT (LETTRE—CHIFFRE)(LETTRE—CHIFFRE—BLANC—PONCT)*
CONTENU BLANC(LETTRE—CHIFFRE—BLANC—PONCT)*

Avec ces variables nous avons définis un certains nombre de mots clés par rapports aux fonctions Latex que nous souhaitions traiter, comme par exemple des instructions typographiques ou le type d'une section. Tout détailler serait trop long et plus compliqué, ainsi voici notre code flex :

```
def corpus2list (corpus): # Renvoie une
↪  liste(liste(liste(mot)))
    res =[]
    i = 0
    j = 0
    articles = titres(corpus) # Tout le corpus est séparée en
    ↪  articles/texte
    while i < len(articles): # Pour chaque article
        res_article = []
        if i == len(articles) -1: art =
        ↪  corpus[articles[i][2]+2:]
        else : art =
        ↪  corpus[articles[i][2]+2:articles[i+1][1]-2]
        i =i+1
        m = mot(art) # renvoie une liste (mot)
        res.append(m)
    return res

#---------RECHERCHE-DU-THEME-------------
# chargement des themes
#file='BDDThemes.csv'
themes_l = []
themes = []
i = 0
with open ('BDDThemes.csv', 'r', encoding='utf-8') as csvfile:
        spamreader = csv.reader(csvfile, delimiter=',')
        for row in spamreader:
            for word in row:
                themes_l.append(word)
            themes.append(themes_l)
            themes_l=[]

# comparaison mot cles
res=[]
for cle in mots_cles:
    for ligne in range (len(themes)):
            if cle in themes[ligne]:
                res.append(ligne)

e = leplusfrequent(res)
print (themes[(e)][0])
```

Nous avons ensuite voulu créer une grammaire afin de faire les transformations de Latex à HTML. Notre variable initiale est "Instr". Une fois de plus notre grammaire possède de très nombreuses règles qu'il serait trop long à détailler.

Voici notre code bison :

```
%{
#include <stdlib.h>
```

```
3  #include <string.h>
4  #include <stdio.h>
5
6  %}
7
8  %debug
9
10 %union{
11     char* chaine;
12 }
13
14 %token  <chaine> SECTION SUBSECTION SUBSUBSECTION PARAGRAPH
   ↪  TITLE
15 %token  <chaine> GRAS ITALIQUE SOULIGNE BARRE
16 %token  <chaine> DOCUMENT ITEMIZE COMMENT
17 %token  <chaine> TEXT CONTENU
18 %token  <chaine> ITEM IMAGE
19 %token  <chaine> DEBUT_BALISE FIN_BALISE
20 %token  <chaine> DEBUT_COMMANDE FIN_COMMANDE
21 %type   <chaine> BaliseOuvrante BaliseFermante BaliseSection
   ↪  FinCommande DebutCommande
22 %type   <chaine> Text Contenu TypeItem Liste DebCommentaire
   ↪  FinCommentaire Image
23 %type   <chaine> TypeImage BaliseTypo TypeTypo
24 %type   <chaine> NomBalise TypeSection
25
26 %start Instr
27
28 %error-verbose
29 %%
30
31 Instr:
32     /* Vide */
33 |    Instr BaliseOuvrante Instr
34 |    Instr BaliseFermante Instr
35 |    Instr BaliseSection Instr
36 |    Instr DebCommentaire Instr
37 |    Instr FinCommentaire Instr
38 |    Instr Contenu Instr
39 |    Instr Liste Instr
40 |    Instr Image Instr
41 |    Instr BaliseTypo Instr
42 ;
43
44 BaliseOuvrante :
45     DebutCommande DEBUT_BALISE NomBalise FinCommande
       ↪  {printf("%s%s%s", $1, $3, $4);}
46 ;
47
```

```
48  BaliseFermante :
49      DebutCommande FIN_BALISE NomBalise FinCommande
        ↪  {printf("%s/%s%s", $1, $3, $4);}
50  ;
51
52  BaliseSection :
53      DebutCommande TypeSection Text FinCommande
        ↪  {printf("%s%s%s%s%s/%s%s", $1, $2, $4, $3, $1, $2,
        ↪  $4);}
54  ;
55
56  DebCommentaire :
57      DebutCommande DEBUT_BALISE COMMENT FinCommande
        ↪  {printf("%s!-- ", $1);}
58  ;
59
60  FinCommentaire :
61      DebutCommande FIN_BALISE COMMENT FinCommande
        ↪  {printf("--%s", $4);}
62  ;
63
64  Liste :
65      TypeItem Text {printf("<%s> %s </%s>", $1, $2, $1);}
66  ;
67
68  Image :
69      DebutCommande TypeImage Text FinCommande {printf("%s%s
        ↪  src=\"%s\"%s",$1, $2,$3,$4);}
70  ;
71
72  TypeImage :
73      IMAGE {$$="img";}
74  ;
75
76  BaliseTypo :
77      DebutCommande TypeTypo Text FinCommande
        ↪  {printf("%s%s%s%s%s/%s%s", $1, $2, $4, $3, $1, $2,
        ↪  $4);}
78  ;
79
80  FinCommande :
81      FIN_COMMANDE      {$$=">";}
82  ;
83
84  DebutCommande :
85      DEBUT_COMMANDE      {$$="<";}
86  ;
87
```

```
88  Text :
89      TEXT          {$$=yylval.chaine;}
90  ;
91
92  Contenu :
93      CONTENU      {$$=yylval.chaine; printf("%s", $$);}
94  ;
95
96  NomBalise :
97      DOCUMENT      {$$="body";}
98  |    ITEMIZE      {$$="ul";}
99  ;
100
101 TypeSection :
102     SECTION      {$$="h1";}
103 |    SUBSECTION      {$$="h2";}
104 |    SUBSUBSECTION      {$$="h3";}
105 |    PARAGRAPH      {$$="h4";}
106 ;
107
108 TypeItem :
109     ITEM         {$$="li";}
110 ;
111
112 TypeTypo :
113     GRAS         {$$="b";}
114 |    ITALIQUE     {$$="i";}
115 |    SOULIGNE     {$$="u";}
116 |    BARRE         {$$="s";}
117 ;
118
119 %%
120
121 void yyerror(char *s) {
122   printf("%s\n",s);
123 }
124
125 int main(void) {
126   yyparse();
127   return 0;
128 }
```

4 Résultats obtenus

Pour tester l'ensemble de nos fonctionnalités, voici le fichier texte de test que nous avons rédigé, intitulé *test* :

```
\begin{document}
\section{Essais typographiques}
\subsubsection{Gras}
\textbf{Texte en gras}
\subsubsection{Italique}
\textit{texte en italique}
\subsubsection{Souligne}
\underline{texte souligne}
\subsubsection{Barre}
\sout{Texte barre}
\section{Liste}
\begin{itemize}
\item numero1
\item numero 2
\end{itemize}
\section{Image}
\includegraphics{image.jpg}
\begin{comment}
Ceci est un commentaire
\end{comment}
\end{document}
```

Ensuite, nous avons entré toutes les commandes permettant la compilation de notre programme, intitulé *projet* :

```
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ bison -d projet.y
projet.y: avertissement: 27 conflits par décalage/réduction [-Wconflicts-sr]
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ mv projet.tab.h projet.h
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ mv projet.tab.c projet.y.c
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ flex projet.lex
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ mv lex.yy.c projet.lex.c
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ gcc -c projet.lex.c -o projet.lex.o
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ gcc -c projet.y.c -o projet.y.o
projet.tab.c: In function ‘yyparse’:
projet.tab.c:1184:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
projet.tab.c:1484:9: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
projet.y: At top level:
projet.y:122:6: warning: conflicting types for ‘yyerror’
 void yyerror(char *s) {
      ^~~~~~~
projet.tab.c:1484:9: note: previous implicit declaration of ‘yyerror’ was here
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ gcc -o projet projet.lex.o projet.y.o -ll -lm
```

Nous avons testé l'efficacité de notre programme de deux manières. Nous avons commencé par vérifier que les transcriptions étaient bonnes à l'affichage dans le terminal :

```
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ ./projet <test.txt
<body>
<h1>Essais typographiques</h1>
<h3>Gras</h3>
<b>Texte en gras</b>
<h3>Italique</h3>
<i>texte en italique</i>
<h3>Souligne</h3>
<u>texte souligne</u>
<h3>Barre</h3>
<s>Texte barre</s>
<h1>Liste</h1>
<ul>
<li> numero1
 </li><li> numero 2
 </li></ul>
<h1>Image</h1>
<img src="image.jpg">
<!--
Ceci est un commentaire
-->
</body>
```

Pour finir, nous avons redirigé la sortie vers un fichier html et l'avons afficher à l'aide du navigateur firefox pour vérifier la justesse des balise :

```
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ ./projet <test.txt >test.html
marion@adm1:~/Bureau/INSA/GM4/S8/Analex$ firefox test.html
```

Essais typographiques

Gras

Texte en gras

Italique

texte en italique

Souligne

texte souligne

Barre

~~Texte barre~~

Liste

- numero1
- numero 2

Image

On remarque alors que toutes les typographies sont respectées, les effets de caractères sur la police, les titres plus ou moins importants ainsi que le commentaire qui n'apparait pas, l'image est bien affiché (le logo d'un fichier inexitant est présent car nous n'avons pas téléhargé l'image correspondante)

5 Difficultés rencontrées et conclusion

La plus grosse difficulté que nous avons rencontré durant ce projet était liée au fait que nous traitions des chaines de carctères. En effet, bison et flex offre de nombreuses possibilités mais il est beaucoup plus difficile et contraignant de développer des applications sur des chaines de caractères que sur des simples calculs. De plus, la documentation sur le sujet est très pauvre (même sur internet). On trouve de nombreuses solutions pour les problèmes de type calculatrice mais très peu concernant des transcription de langage, et bien évidement aucun comme le nôtre. Grâce aux bases acquises pendant le cours et à la logique qui s'applique à cet exercice, nous avons réussi à produire le travail que nous souhaitions. Bien sur il pourrait s'étendre à l'infini car les fonctionnalités de Latex et du code HTML sont immenses. Cependant les nombreuses exceptions rendent le travail fastidieux quand il faut traiter les commandes au cas par cas.