Overlen Stéphane 10 janvier 2017

TECHNIQUES DE COMPILATION
Analyseur SGML

INTRODUCTION

La compilation consiste à produire du code exécutable par un processeur à partir d'un programme écrit dans un langage source. Le but de ce TP est de se familiariser avec l'analyseur syntaxique CUP, utilisé avec l'analyseur lexical Jflex en réalisant les programmes nécessaires avec le couple JFlex/CUP pour vérifier qu'un fichier SGML est bien structuré (ouverture/fermeture des balises, identifiants entre guillemets).

ETAPE 1

La première étape consiste à vérifier qu'un fichier SGML est bien structuré : balises correctement ouvertes/fermées et identifiants entre guillemets. Ainsi, le code suivant (dans ***file.txt***) :

```
<C/>
<A>
        <D b="1,23">
                texte
        </D>
</B>
```

Figure 1

doit retourner :

C +A +D -D ERROR

Figure 2

I.1) PARTIE FLEX

Le fichier ***FLEX*** (dans ***monflex.flex***) contient le code qui envoie les balises constituées au code ***CUP*** . Dans ce fichier, nous trouvons les définitions suivantes :

```
11 value = ["\""]((-|\+)?[0-9]+([\.|,][0-9]+)?)["\""]
12 param = [a-z]+[ ]*[=][ ]*{value}
13 open = <[A-Z]+([ ]+{param})*>
14 close = (<[/][A-Z]+>)
15 selfclose = (<[A-Z]+[ ]*[/]>)
16 text = [^"\t\r\n <"]+
17 text2 = <[^"\t\r\n <>"]+
```

Figure 3

- ***value*** : récupère les nombres. Ils peuvent être positifs, négatifs, entiers ou flottants, entre guillemets, donc sous la forme "***-23,45"*** ou "***67.890"*** ou encore "***+123"***
- ***param*** : les paramètres. Ils devront être de la forme ***variable={value}*** .
- ***open*** : les balises ouvrantes, avec ou sans paramètre : <***DIV***> ou <***P var="12"***> ou encore <***HTML var="12" var2="13"***> etc.
- ***close*** : les balises fermantes de la forme <***/A***>

- ***selfclose*** : récupère les balises auto-fermantes de la forme <***C/***>
- ***text*** et ***text2***: contiennent à eux deux tous les caractères que l'utilisateur peut être amené à écrire dans le code, même les signes < et >. Nous y reviendrons plus tard.

Dans un soucis de flexibilité, j'ai aussi ajouté la possibilité de mettre des espaces un peu partout, par exemple avant et après les signes =. Ceci afin d'éviter la frustration des erreurs causées par l'ajout d'un simple espace à des endroits "non-clés".

La suite du code permet de retourner au code ***CUP*** les séquences détectées dans le fichier ***file.txt*** par le code ***FLEX*** .

```
21 {open}          { return new Symbol(sym.OPEN,       yytext()); }
22 {close}         { return new Symbol(sym.CLOSE,      yytext()); }
23 {selfclose}     { return new Symbol(sym.SELFCLOSE,  yytext()); }
24 {text}          { return new Symbol(sym.TEXT,       yytext()); }
25 {text2}         { return new Symbol(sym.TEXT,       yytext()); }
```

Figure 4

En prenant l'exemple de code en **Figure 1** , le code ***FLEX*** va détecter <***C/***> et déterminera alors que c'est une ***balise auto-fermante*** (***selfclose***) car elle contient la séquence suivante :

selfclose = ***(<[A-Z]+[ ]*[/]>)*** = ***<[A-Z][/]>*** = <***C/***>

Il retournera alors au code ***CUP*** le symbole :

return new Symbol(sym.SELFCLOSE, yytext());

avec ***yytext()*** =***"<C/>"*** . De la même manière, quelques lignes plus loin, il détectera la séquence ***open avec paramètres <D b="1,23">*** :

open = ***<[A-Z]+([ ]+[a-z]+[ ]*[=][ ]*["\""]((-|\+)?[0-9]+([\.|,][0-9]+)?)["\""])*>***
= ***<[A-Z][ ][a-z][=]["\""][0-9][,][0-9]+["\""]>***
= ***<D b="1,23">***

Et ainsi de suite.

J'ai déclaré ***text*** et ***text2*** car toutes les autres combinaisons que j'ai testées ne permettaient pas d'écrire une balise au milieu du texte. Ainsi, elles sont déclarées comme suit :

- ***text*** = tout ce qui ne contient pas de tabulation, de retour chariot, de retour à la ligne, d'espace et de signe <
- ***text2*** = tout ce qui commence par le signe < et qui ne contient pas de tabulation, de retour chariot, de retour à la ligne, d'espace et de signe < ou >

*Il est important de laisser **text** et **text2** en dernier car l'ordre permet d'éviter que le compilateur prenne les balises pour du texte. En effet, dans **text** et **text2**, nous cherchons également les signes tels que <, >, /, ", etc.*

I.2) PARTIE CUP

Le code ***CUP*** (dans ***moncup.cup***) permet par la suite de tester l'ordre dans lequel le code ***FLEX*** lui envoie les informations, et effectuer un traitement selon les informations reçues.

```
terminal TEXT, OPEN, CLOSE, SELFCLOSE;

non terminal A, B, C, D, O, T;
```

Figure 5

Pour commencer, il faut déclarer les symboles terminaux et non terminaux.

```
B ::= A | B A | O B C | B O B C
;
A ::= T | O C | D
;
D ::= SELFCLOSE:d   {:  System.out.print(d.toString().split("<|/")[1] + " ");
                    :}
;
C ::= CLOSE:c       {:  c = c.toString().split("/|>")[1];
                        if(letter.peek().equals(c)) {
                            letter.pop();
                            System.out.print("-" + c + " ");
                        }
                        else{
                            System.out.print("**ERROR: opening " + letter.peek()
                                             + ", closing " + c + "**\n");
                            System.exit(0);
                        }
                    :}
;
O ::= OPEN:o        {:  o = o.toString().split("<| |>")[1];
                        letter.push(o);
                        System.out.print("+" + o + " ");
                    :}
;
T ::= TEXT
;
```

Figure 6

Cela nous permet ensuite de définir la grammaire utilisable dans le fichier ***file.txt*** . On commence par associer chaque symbole terminal à un symbole non terminal :

T ::= TEXT
;

Puis on définit l'ordre. Par exemple, on ne peut pas trouver une balise ouvrante qui n'a pas de balise fermante : ***<A>*** est forcément suivi (pas nécessairement immédiatement) par ***</A>*** :

A ::= T | O C | D
;

Signifie que ***A*** peut être :

- du ***texte ou***
- une ***balise ouvrante*** suivie d'une ***balise fermante ou***
- une ***balise auto-fermante***.

B ::= A | B A | O B C | B O B C
;

Signifie que le fichier peut contenir :

- ***A tout seul***
- ***A précédé de B***
- ***une balise ouvrante et une balise fermante avec B entre les 2***
- ***la même chose précédée de B***

Tous les tests que j'ai effectués démontrent que tous les cas de figures qui pourraient être utilisés dans l'étape 1 sont bien acceptés par la grammaire du code ***CUP*** : peu importent le nombre et la position du texte et des balises, tant que chaque balise ouverte est fermée par la suite, le code est bon.

Pour finir, il faut afficher les lettres précédées d'un + lorsque le ***CUP*** reçoit une balise ouvrante, un – lorsqu'il reçoit une balise fermante, et la lettre seule lorsque c'est une balise auto-fermante. Il faut également afficher une erreur lorsque la mauvaise balise est fermée.

Il suffit de récupérer l'objet retourné par le code ***FLEX*** :

O ::= OPEN:o

o est alors un objet contenant la balise retournée. Afin de récupérer la lettre de la balise, il faut alors le convertir en chaîne de caractères avec la méthode ***toString()*** , puis isoler la lettre en utilisant la méthode ***split()*** de ***java.lang.String*** .

```
parser code {:
    Stack letter = new Stack();
:}
```

Figure 7

En ce qui concerne les fermetures de balises erronées, j'ai créé une pile ***Stack letter = new Stack()*** . Pour insérer, obtenir et supprimer les lettres de la pile, il faut utiliser respectivement les méthodes ***push()***, ***peek()*** et ***pop()*** de ***java.util.Stack*** . À chaque balise ouvrante, la lettre de la balise est stockée dans la pile. À chaque fermeture de balise, la lettre de la balise est comparée à la lettre présente en haut de la pile grâce à la fonction ***equals()*** de ***java.lang.Object*** . Si elles ne correspondent pas, on affiche une erreur et on quitte le programme. Si elles correspondent, on enlève la lettre en haut de la pile et on continue.

I.3) COMPILATION, EXECUTION ET RÉSULTATS

Le code est accompagné d'un fichier ***makefile*** qui exécute les lignes suivantes :
Compilation :

- ***jflex monflex.flex***
- ***java -jar java-cup-11b.jar.jar moncup.cup***
- ***javac -classpath java-cup-11b.jar:. sym.java***
- ***javac -classpath java-cup-11b.jar:. parser.java***
- ***javac -classpath java-cup-11b.jar:. Lexer.java***
- ***javac -classpath java-cup-11b.jar:. Main.java***

Exécution :

- ***java -classpath java-cup-11b.jar:. Main file.txt***

Code testé :

```
<HTML>
    <C/>
    <D t="-23,45" fds="+28,45">
        test
    </D>
    <E>
        <F>
            test2
            <K/>
        </F>
    </E>
</HTML>
<G>
</H>
```

Figure 8

Résultat :

```
steph@stephane:~/Documents/TCP/labo2etape1$ make clean
rm -rf sym.class parser.class Lexer.class sym.java Lexer.java parser.java *.class
steph@stephane:~/Documents/TCP/labo2etape1$ make
java -jar java-cup-11b.jar moncup.cup
------- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -------
  0 errors and 0 warnings
  6 terminals, 6 non-terminals, and 12 productions declared,
  producing 18 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
---------------------------------------------------- (CUP v0.11b 20160615 (GIT 4ac7450))
javac -classpath  java-cup-11b.jar:. sym.java
javac -classpath  java-cup-11b.jar:. parser.java
jflex monflex.flex
Reading "monflex.flex"
Constructing NFA : 82 states in NFA
Converting NFA to DFA :
.......................
27 states before minimization, 24 states in minimized DFA
Writing code to "Lexer.java"
javac -classpath  java-cup-11b.jar:. Lexer.java
javac -classpath  java-cup-11b.jar:. Main.java
Note: Main.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
rm parser.java
steph@stephane:~/Documents/TCP/labo2etape1$ java -classpath java-cup-11b.jar:. Main file.txt
+HTML C +D -D +E +F K -F -E -HTML +G **ERROR: opening G, closing H**
```

Figure 9

On constate que toutes les balises ouvrantes, fermantes et auto-fermantes sont affichées et que l'erreur à la fin est bien détectée.

I.4) DIFFICULTÉS RENCONTRÉES

- **Où s'arrête FLEX, où commence CUP ?**

La plus grosse difficulté de cette étape a été de savoir comment répartir les tâches entre les codes ***FLEX*** et ***CUP***. Ma première approche était de constituer les balises dans le fichier ***CUP*** :

```
D ::= A B C D          //  <G/> = INF LETTRE SLASH SUP
;
A ::= INF              // <
;
B ::= LETTRE           // G
;
C ::= SLASH            // /
;
D ::= SUP              // >
;
```

Seuls ***numbers*** , ***text*** et chaque symbole étaient déclarés dans le code ***FLEX*** . Cette approche était prometteuse, mais rendait toute tentative de code compliquée et ambiguë. Cela m'a fait perdre pas mal de temps avant de comprendre que cette étape pouvait être réalisée dans le code ***FLEX*** .

ETAPE 2

Le but est maintenant d'attribuer des valeurs à des variables pour pouvoir tester si des conditions sont vraies. Par exemple :

```
<#set id="12"/>
<C>
        <#if id=="13">
                texte1
        <#if/>
        <#if id=="12">
                texte2
        <#if/>
</C>
```

Figure 10

Le code précédent doit afficher le code suivant :

```
<C>
        texte2
</C>
```

Figure 11

Si la condition est vraie, le code est exécuté.

II.1) PARTIE FLEX

```
11 value = ["\""](-?[0-9]+([\.|,][0-9]+)?)["\""]
12 cond = [a-z]+[ ]*[=][=][ ]*{value}
13 param = [a-z]+[ ]*[=][ ]*{value}
14 myIf = <#[i][f][ ]+{cond}>
15 myEndif = <#[i][f][ ]*[/]>
16 set = <#set[ ]+{param}[/]>
17 open = <[A-Z]+([ ]+{param})*>
18 close = (<[/][A-Z]+>)
19 selfclose = (<[A-Z]+[ ]*[/]>)
20 text = [^"\t\r\n <"]+
21 text2 = <[^"\t\r\n <>"]+
```

Figure 12

J'ai repris la base de l'étape 1 et ai ajouté au code ***FLEX*** les déclarations suivantes :

- ***set*** : récupère les déclarations de variables ***<#set id="12"/>***
- ***myIf*** : condition ***<#if id=="12">***
- ***myEndif*** : fermeture de la balise de condition ***<#if/>***

```
25 {set}          { return new Symbol(sym.SET,        yytext()); }
26 {open}         { return new Symbol(sym.OPEN,       yytext()); }
27 {myIf}         { return new Symbol(sym.MYIF,       yytext()); }
28 {myEndif}      { return new Symbol(sym.MYENDIF,    yytext()); }
29 {close}        { return new Symbol(sym.CLOSE,      yytext()); }
30 {selfclose}    { return new Symbol(sym.SELFCLOSE,  yytext()); }
31 {text}         { return new Symbol(sym.TEXT,       yytext()); }
32 {text2}        { return new Symbol(sym.TEXT,       yytext()); }
```

Figure 13

De la même manière que dans l'étape 1, il faut ajouter ***set*** , ***myIf*** et ***myEndif*** aux symboles retournés par ***FLEX*** à ***CUP.***

II.2) PARTIE CUP

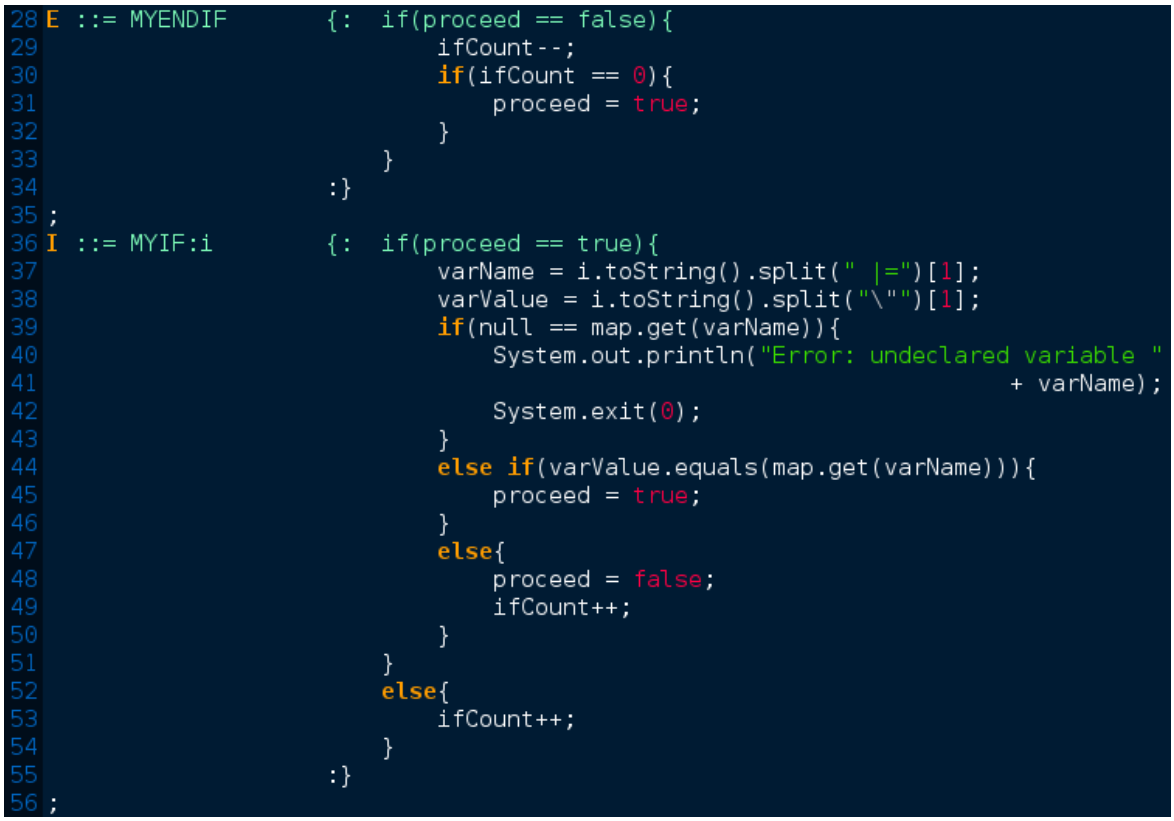
```
17     Map<String, String> map = new HashMap<String, String>();
```

Figure 14

La première chose à faire est de traiter les déclarations de variables grâce à ***set*** . Pour ce faire, j'ai déclaré au début du programme ***CUP*** un dictionnaire ***map***. Ce dictionnaire nous permettra d'enregistrer des variables dans le « tableau » ***map*** de la manière suivante :

<#set a="12"/> ***// map = {a=12}***
<#set b="13"/> ***// map = {a=12 , b=13}***
<#set c="14"/> ***// map = {a=12 , b=13 , c=14}***
<#set a="15"/> ***// map = {a=15 , b=13 , c=14 }***

*À chaque **set**, la variable est ajoutée au dictionnaire.*



```
28 E ::= MYENDIF        {:  if(proceed == false){
29                              ifCount--;
30                              if(ifCount == 0){
31                                  proceed = true;
32                              }
33                          }
34                      :}
35 ;
36 I ::= MYIF:i          {:  if(proceed == true){
37                              varName = i.toString().split(" |=")[1];
38                              varValue = i.toString().split("\"")[1];
39                              if(null == map.get(varName)){
40                                  System.out.println("Error: undeclared variable "
41                                                                          + varName);
42                                  System.exit(0);
43                              }
44                              else if(varValue.equals(map.get(varName))){
45                                  proceed = true;
46                              }
47                              else{
48                                  proceed = false;
49                                  ifCount++;
50                              }
51                          }
52                          else{
53                              ifCount++;
54                          }
55                      :}
56 ;
```

Figure 15

Une fois que les variables sont déclarées et enregistrées, le but est de les tester. C'est ici qu'intervient ***myIf*** (et ***myEndif***) : on utilise la méthode ***split()*** pour récupérer le nom et la valeur de la variable testée. On teste si la variable existe : si ce n'est pas le cas, on affiche une erreur et le programme se termine. Si elle existe et que la condition est remplie, on met ***proceed*** à ***true*** .

La variable ***proceed*** est testée au début de chaque fonction exécutée dans le code ***CUP*** . Elle est initialisée à ***true*** afin que tout le code se déroule correctement. Lorsqu'une condition n'est pas remplie, ***proceed*** est mise à ***false*** et les fonctions présentes dans le contexte de la condition fausse ne s'exécutent plus. Dès que l'on sort du contexte, la variable est remise à ***true*** afin d'exécuter le code correctement par la suite.

```
D ::= SELFCLOSE:d   {:  if(proceed == true){
                            for(i=0; i<indent; i++){
                                System.out.print("\t");
                            }
                            System.out.print(d + "\n");
                        }
                    :}
;
C ::= CLOSE:c       {:  if(proceed == true){
                            if(letter.peek().equals(c.toString().split("/|>")[1])){
                                indent--;
                                for(i=0; i<indent; i++){
                                    System.out.print("\t");
                                }
                                System.out.println(c);
                                letter.pop();
                            }
                            else{
                                for(i=1; i<indent; i++){
                                    System.out.print("\t");
                                }
                                System.out.print("ERROR\n");
                                System.exit(0);
                            }
                        }
                    :}
;
O ::= OPEN:o        {:  if(proceed == true){
                            for(i=0; i<indent; i++){
                                System.out.print("\t");
                            }
                            indent++;
                            letter.push(o.toString().split("<| |>")[1]);
                            System.out.println(o);
                        }
                    :}
;
S ::= SET:s         {:  if(proceed == true){
                            map.put(s.toString().split(" |=")[1],
                            s.toString().split("\"")[1]);
                        }
                    :}
;
T ::= TEXT:t        {:  if(proceed == true){
                            for(i=0; i<indent; i++){
                                System.out.print("\t");
                            }
                            System.out.println(t);
                        }
                    :}
```

Figure 16

Comme on peut le voir dans ce code, si la condition n'est pas remplie :

- le texte ne s'affiche pas
- les déclarations de variables ne se font plus
- les balises ouvrantes et fermantes n'impliquent plus d'actions sur la pile

C'est la variable ***ifCount*** qui, tant qu'elle est différente de ***0***, garde ***proceed*** à ***false***. Elle s'incrémente à chaque nouvelle condition ***if*** et se décrémente à chaque ***endif***.

Et pour finir, j'ai ajouté les symboles terminaux et non terminaux en ajoutant les nouveaux et en les réorganisant selon la grammaire désirée :

```
terminal SET, TEXT, MYIF, OPEN, MYENDIF, CLOSE, SELFCLOSE;

non terminal A, B, C, D, E, I, O, S, T;

B ::= A | B A | O B C | B O B C | I B E | B I B E
;
A ::= T | S | O C | I E | D
;
```

Figure 17

On voit ici que ***A*** peut également être une balise ***set*** ou ***if*** suivie de ***endif*** et que ***B*** s'est également vu ajouter les balises ***if*** et ***endif*** .

La variable ***indent*** , quant à elle, est juste là pour respecter l'indentation à l'affichage en affichant autant de tabulation que nécessaire.

II.3) COMPILATION, EXECUTION ET RÉSULTATS

(voir compilation et exécution dans l'étape 1)

Code testé :

```
<#set toto="123"/>
<D>
    <#if toto=="123">
        tutu

        <#if toto=="999">
            tata
            <#if toto=="123">
                totor
                <P>
                </P>
                <P/>
            <#if/>
        <#if/>
        titi
    <#if/>
    <#set toto="234"/>
    <U>
        freud
    </U>
    <V>
        <#if toto == "456">
            est un comedien renomme
        <#if/>
        <#if toto == "234">
            est un philosophe
        <#if/>
    </V>
    <P/>
    <tes(t)/\"fdas"'fd<B>hsl'>%+"*ç%&/()=?2345</B>678<#if>90'1^ä£$àè¨-.,;:_{}[]
</D>
```

Figure 18

On peut voir dans ce code la variable ***toto*** à laquelle sont attribuées deux valeurs différentes, plusieurs tests sur cette variable, des balises ouvrantes et fermantes, du texte, et pour finir une suite de caractères en tous genres contenant de fausses et vraies balises.

Résultat obtenu :

```
steph@stephane:~/Documents/TCP/labo2finished$ make clean
rm -rf sym.class parser.class Lexer.class sym.java Lexer.java parser.java *.class
steph@stephane:~/Documents/TCP/labo2finished$ make
java -jar java-cup-11b.jar moncup.cup
------- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -------
  0 errors and 0 warnings
  9 terminals, 9 non-terminals, and 19 productions declared,
  producing 29 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
---------------------------------------------------- (CUP v0.11b 20160615 (GIT 4ac7450))
javac -classpath  java-cup-11b.jar:. sym.java
javac -classpath  java-cup-11b.jar:. parser.java
jflex monflex.flex
Reading "monflex.flex"
Constructing NFA : 192 states in NFA
Converting NFA to DFA :
.............................................................
62 states before minimization, 57 states in minimized DFA
Writing code to "Lexer.java"
javac -classpath  java-cup-11b.jar:. Lexer.java
javac -classpath  java-cup-11b.jar:. Main.java
Note: Main.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
rm parser.java
steph@stephane:~/Documents/TCP/labo2finished$ java -classpath java-cup-11b.jar:. Main file.txt
<D>
        tutu
        titi
        <U>
                freud
        </U>
        <V>
                est
                un
                philosophe
        </V>
        <P/>
        <tes(t)/\"fdas"'fd
        <B>
                hsl'>%+"*ç%&/()=?2345
        </B>
        678
        <#if
        >90'1^ä£$àè¨-.,;:_{}[]
</D>
```

Figure 19

On constate que seules les balises et le texte entre les balises dont la condition n'est pas remplie ne sont pas pris en compte. Ainsi, Freud est un philosophe et non pas un comédien renommé.

III) CONCLUSION

Dans l'étape 1 comme dans l'étape 2, tous les tests que j'ai effectués ont été concluants. Les codes ***FLEX*** et ***CUP*** réalisent ce qui a été demandé, offrant ainsi un début de compilateur fonctionnel. Dès que possible, j'évoluerai le code pour implémenter d'autres fonctions telles que les boucles ***for*** , les conditions ***switch*** , etc.