4 *Guide de lecture de DTD*

Les éléments sont déclarés à l'aide de lignes du type :

```
<!ELEMENT toto ... >
```

Un élément peut être déclaré de deux manières : par un *contenu mixte* ou une *expression régulière*.

Le contenu mixte est écrit de la manière suivante :

```
<!ELEMENT toto (#PCDATA|titi|tata|...|tutu)* >
```

et cela signifie que `toto` peut contenir *du texte* et des éléments `titi`, ... `tutu`, dans n'importe quel ordre.

L'expression régulière sera écrite en utilisant des noms d'élément, des *séparateurs*, des *quantificateurs* et des *parenthèses*. Les séparateurs sont : la virgule (pour indiquer que deux éléments se suivent) et la barre verticale (le booléen «ou»). Ainsi les deux lignes suivantes :

```
<!ELEMENT toto (tata,titi)>
<!ELEMENT fofo (fafa|fifi)>
```

signifient : (a) `toto` contient forcément un `tata` suivi d'un `titi`, et (b) `fofo` contient un `fafa` ou un `fifi`.

Les quantificateurs sont : ? (0 ou 1 fois), * (0, 1 ou plusieurs fois), + (1 ou plusieurs fois). Ainsi les deux lignes suivantes :

```
<!ELEMENT toto (tata?,titi*)>
<!ELEMENT fofo (fafa|fifi+)>
```

signifient : (a) `toto` contient un (ou aucun) `tata` suivi de zéro, un, ou plusieurs `titi`, et (b) `fofo` contient un et un seul `fafa` ou alors un ou plusieurs `fifi`.

Les parenthèses servent à regrouper les expressions. Par exemple :

```
<!ELEMENT toto ((tata|titi),(tete|toto))>
<!ELEMENT fofo ((fafa,fifi)|(fefe,fofo))>
```

Les attributs se déclarent par des lignes du type :

```
<!ATTLIST toto numero CDATA "1">
```

La ligne ci-dessus signifie que l'élément `toto` dispose d'un attribut optionnel appelé `numero` dont la valeur est une chaîne de caractères quelconque et que si cet attribut n'est pas présent sa valeur par défaut est `1`. Plus de détails en cours !

Malheureusement la syntaxe ne s'arrête pas là. Pour simplifier (???) l'écriture des DTD on dispose d'une astuce d'écriture qui s'appelle «entité paramètre». Cela ressemble un peu aux variables de préprocesseur C : on prend une partie du code de la DTD, et on la stocke dans une «variable» qui a un nom. Ensuite on utilise ce nom dans le code.

Ainsi, en écrivant

```
<!ENTITY % ringard "(tata|titi),(tete|toto)">
```

j'ai stocké le contenu de la chaîne ci-dessus dans l'«entité paramètre» `%ringard;`. En écrivant

```
<!ELEMENT toto (%ringard;)>
```

j'obtiens la même chose que si j'avais écrit :

```
<!ELEMENT toto ((tata|titi),(tete|toto))>
```

4.1 Exemple

Prenons un exemple (réel) tiré de la DTD XHTML Strict (celle que nous allons utiliser pour écrire notre page Web) :

```
<!ELEMENT table
     (caption?, (col*|colgroup*), thead?, tfoot?, (tbody+|tr+))>
```

On déclare donc l'élément `table`. Il peut contenir, dans cet ordre : (éventuellement) un `caption`, un groupe contenant des `col` (éventuels) ou des `colgroup` (éventuels), (éventuellement) un `thead`, (éventuellement) un `tfoot`, et finalement un ou plusieurs `tbody` ou alors un ou plusieurs `tr`. Autrement dit : `<table> <thead> ... </thead> <tr> ... </tr> </table>` est valide, `<table> <tr> ... </tr> <tbody> ... </tbody> </table>` ne l'est pas (pourquoi ?)

Autre exemple :

```
<!ELEMENT body %Block;>
<!ATTLIST body
  %attrs;
  onload          %Script;   #IMPLIED
  onunload        %Script;   #IMPLIED
  >
```

C'est la déclaration de l'élément `body`. À première vue elle est bien simple : `%Block;`. Il faut alors consulter la déclaration de l'entité paramètre `Block` :

```
<!ENTITY % Block "(%block; | form | %misc;)*">
```

Autrement dit : `body` contient un mélange de `%block;`, des formulaires `form` ou des `%misc;`. Que contiennent ces deux nouvelles entités paramètre ? Les voilà :

```
<!ENTITY % block
     "p | %heading; | div | %lists; | %blocktext; | fieldset | table">

<!ENTITY % misc "noscript | %misc.inline;">
```

On s'aperçoit que, un peu comme dans le cas des sites Web X, chaque entité paramètre nous envoie vers d'autres entités paramètre, qui nous envoient vers d'autres entités paramètre, et ainsi de suite (à la différence près que... l'on ne nous demande pas de numéro de carte bleue à la fin).

On ne lit pas une DTD comme on lit un roman. Quand on tombe sur un os (un fichier qui n'est pas valide), il faut suivre la déclaration d'un élément pour trouver ce qui ne va pas. Un processus qui s'apparente au débogage d'un programme informatique, et qui est tout aussi charmant.

Exemple : qu'est-ce qui ne va pas avec le code suivant ?

```
<body>
Bonjour le monde
</body>
```

Réponse : en suivant la piste des entités paramètre qui constituent la déclaration de `body` on s'aperçoit rapidement qu'il s'agit d'une déclaration par expression régulière et non pas par contenu mixte (sinon il aurait fallu une déclaration du type `<!ELEMENT body (#PCDATA|...)*>`. Donc il ne peut pas contenir du texte directement... il faut que ce texte soit inclus dans un élément, ainsi par exemple :

```
<body>
<p>Bonjour le monde</p>
</body>
```

est correct.