**Licence 3ème année**

# Traitement syntaxique & Sémantique — P-F. Bonnefoi

*Version du 10 janvier 2024*

Table des matières


1 Contenu et objectifs . . . . . . . . 6

2 Langage interprété *vs* compilé : la compilation . . . . . . . . 7

Langage interprété *vs* compilé : l'interprétation . . . . . . . . 8

Langage interprété *vs* compilé . . . . . . . . 9

3 L'analyse lexical : les automates à nombre fini d'états . . . . . . . . 10

Expressions régulières ou *expressions rationnelles* . . . . . . . . 11

Analyseur lexical . . . . . . . . 13

L'outil Lex . . . . . . . . 14

Analyseur lexical . . . . . . . . 15

Analyseur lexical : un exemple . . . . . . . . 16

Analyseur lexical : un autre exemple . . . . . . . . 17

Analyseur lexical : exemple d'un format de fichier . . . . . . . . 18

Analyseur lexical : exemples . . . . . . . . 20

Analyseur lexical : utilisation . . . . . . . . 21

Lex : la gestion des accents . . . . . . . . 23

4 Rappels sur les grammaires . . . . . . . . 25

Rappels sur les grammaires . . . . . . . . 26

Les «*parsers*» LR . . . . . . . . 31

Les différentes catégories d'analyseurs LR . . . . . . . . 32

Principe de fonctionnement d'un analyseur LR . . . . . . . . 33

Exemple de table . . . . . 34
Interprétation intuitive d'une table . . . . . 40
Conflits . . . . . 41
Exemple de conflit . . . . . 42
5 Analyseur lexical : un avant goût de grammaire ? . . . . . 44
6 Liens entre Lex et un analyseur syntaxique . . . . . 48
7 L'analyseur syntaxique YACC . . . . . 56
Lex & YACC : un exemple de calculette . . . . . 59
YACC : gestion des erreurs . . . . . 63
8 XML : les différents outils . . . . . 65
XML vs HTML . . . . . 68
Afficher un document XML : XSL . . . . . 72
Format XML : DTD, «*Document Type Description*» . . . . . 73
Utilisation de XSL et DTD . . . . . 74
DTD : notion de classe de document . . . . . 75
Le format XML . . . . . 76
XML : les atouts . . . . . 77
Exemple de document XML . . . . . 78
XML : Utilisation des balises spécifiques . . . . . 79
XML : Syntaxe des éléments constitutifs . . . . . 80
9 DTD : Définition d'une classe de documents . . . . . 81
DTD : La notion d'entités . . . . . 83
DTD : La notion d'élément . . . . . 84

DTD : les attributs d'éléments . . . 86
DTD : la notion d'espace de nom . . . 88
10 CSS : présentation rapide . . . 90
XML & CSS . . . 92
11 XSLT : aller plus loin que les CSS... . . . 93
XSLT : accès aux données . . . 94
XSLT : opérations avancées . . . 95
XSLT : un exemple de liste de livres . . . 96
XSLT : suite de l'exemple de liste de livres . . . 97
XPath . . . 98
DTD : les limitations . . . 105
12 XSD, «*XML Schema Definition*» . . . 106
XSD : exemple de carte de visite . . . 107
XSD, un autre exemple de liste de livres . . . 109
XSD : la définition des «éléments» constitutifs du format . . . 110
XSD : les types de données pour un élément . . . 111
XSD : la définition d'attributs . . . 114
XSD : exemples de déclaration d'élément et d'attribut . . . 116
XSD : dérivation de type . . . 117
13 XML et Java . . . 118
SAX : Utilisation . . . 119
SAX : Utilisation . . . 120
SAX : un exemple . . . 121

14 XML DOM & XSLT : utilisation de JavaScript . . . . . 123
15 XML DOM : JavaScript & AJAX . . . . . 126
XML AJAX : un exemple . . . . . 128
XML AJAX : un exemple – suite . . . . . 129
JQuery : la bibliothèque qui vient au secours d'Ajax . . . . . 133
16 XML & SOAP . . . . . 138
17 JQuery, AJAX et XML . . . . . 140

# 1 Contenu et objectifs

**Objectifs** :

▷ retour sur les analyseur lexicaux et syntaxiques : Expression régulières & Grammaires ;

▷ réalisation d'un «parser».
  - ◇ générateur automatique d'analyseurs lexicaux : utilisation de Lex ;
  - ◇ générateur automatique d'analyseurs sémantiques basés sur les grammaires LR(1) : utilisation de Yacc ;

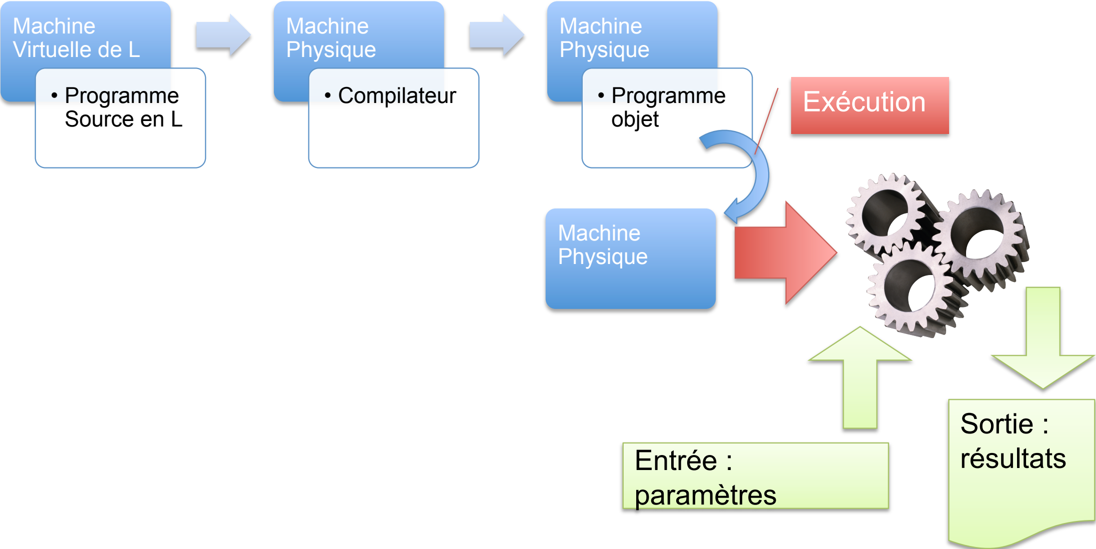
▷ apprentissage et manipulation d'XML :
  - ◇ définition d'un format ;
  - ◇ utilisation d'un parser :

▷ réalisation d'un projet : Utilisation d'XML pour... ?

## 2 Langage interprété *vs* compilé : la compilation

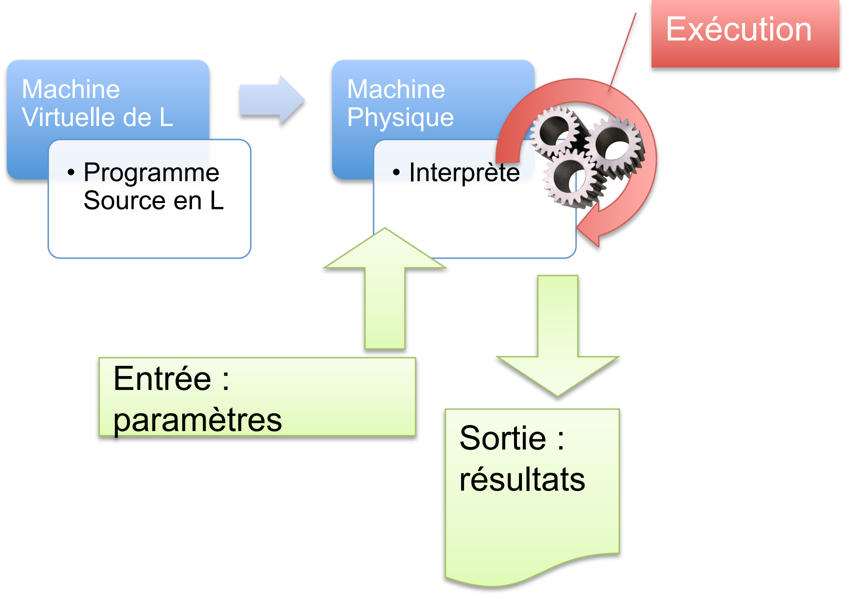
On écrit un programme en **langage L**, par exemple en C++ : ce langage est destiné à une machine «idéalisée», ou virtuelle, qui est plus ou moins proche de la vraie mais où le langage est très bien adapté (en réalité la machine ne gère pas d'objets par exemple).

Enfin, grâce au compilateur, on **traduit** le programme en vraies instructions de la machine physique.



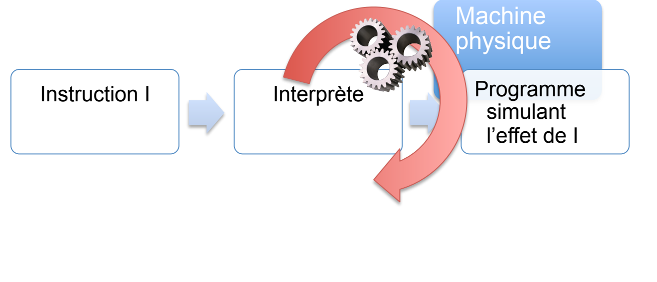

On peut alors **exécuter** le programme sur des entrées et il donne des sorties.

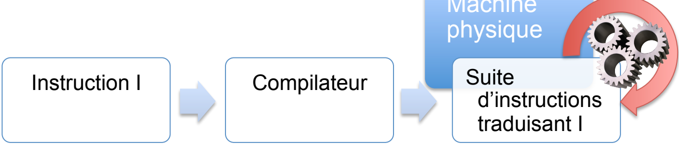
On écrit un programme en langage L, par exemple en Perl : comme dans le cas d'un langage compilé, ce langage est destiné à une machine «idéalisée», ou virtuelle, plus ou moins proche de la vraie, mais où le langage est très bien adapté (en réalité la machine ne gère pas d'objets par ex.).
Enfin, grâce à l'interpréteur, on traduit chaque instruction du programme en une suite d'instructions de la machine physique, au fur et à mesure de l'exécution du programme (sur des entrées et il donne des sorties).



# Langage interprété *vs* compilé

Lors de l'**interprétation** d'une instruction, on fait appel à des morceaux de programme réalisant les effets de cette instruction sur la machine :



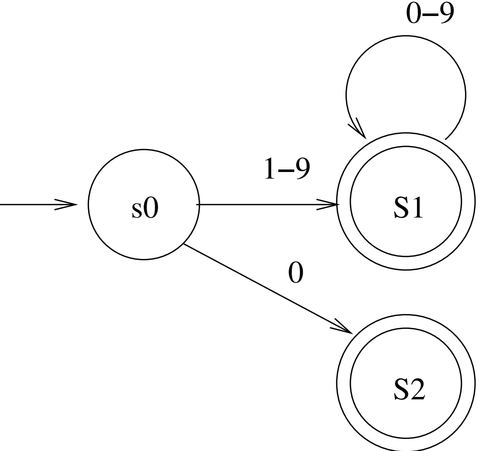




Lors de l'**exécution** du programme **compilé**, chaque instruction du programme correspond à des instructions de la machine physique.

# 3 L'analyse lexical : les automates à nombre fini d'états

Il est possible de reconnaître un texte à l'aide d'un **automate à nombre fini d'états**.

Par exemple, voici l'automate permettant de reconnaître des nombres :




*Un nombre ne peut commencer par le chiffre zéro que s'il vaut zéro.*

Un automate à nombre fini d'états permet de reconnaître des langages dits rationnels.
À un automate, on peut faire correspondre une expression rationnelle (ou régulière, comme en anglais «*regular expression*»).

# Expressions régulières ou *expressions rationnelles*

Une ER permet de faire de l'appariement de motif, *pattern matching* : il est possible de savoir si un motif est **présent** dans une chaîne, mais également **comment** il est présent dans la chaine (en mémorisant la séquence correspondante).

Une expression régulière est exprimée par une suite de meta-caractères, exprimant :

⋆ une *position* pour le motif
- `^` : début de chaîne
- `$` : fin de chaîne

⋆ un caractère pour lui-même ;
- `.` : n'importe quel caractère
- `[ ]` : un caractère parmi une liste
- `[^ ]` : tous les caractères sauf…

⋆ une alternative
- `|` : *ceci* ou *cela*

⋆ des quantificateurs, qui permettent de répéter le caractère qui les précédent :
- `*` : zéro, une ou plusieurs fois
- `+` : **une** ou plusieurs fois
- `?` : zéro ou une fois
- `{ n }` : *n* fois
- `{ n, m }` : entre *n* et *m* fois

⋆ des caractères spéciaux : `\n` : retour à la ligne, `\t` : tabulation

**Un exemple sur la reconnaissance des nombres**

Group #1
0
One of:
One of:
Begin!
1-9
0-9
End!
0 or more times

http://jex.im/regulex/

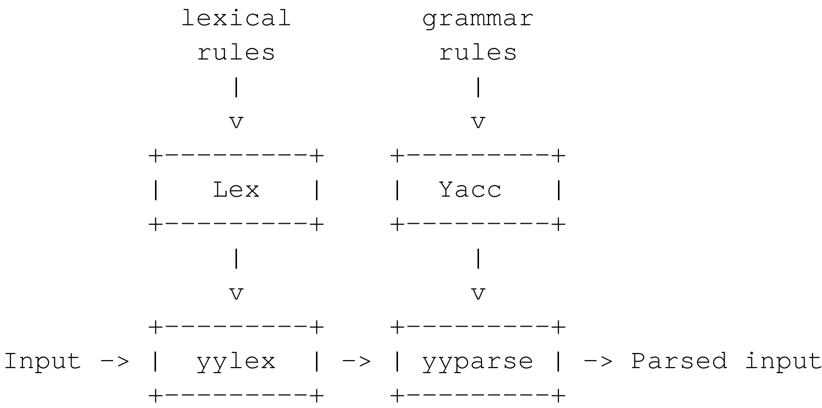
# Analyseur lexical

Un analyseur lexical ou «lexer» :

- ▷ reçoit un flux de caractères en entrée ;
- ▷ lorsqu'il rencontre une suite de caractères qui correspond à un mot clé défini, une unité lexicale ou lexème, un «token», il exécute un ensemble d'actions prédéfinies.

La construction d'un analyseur lexical peut être automatisée à l'aide de l'outil «Lex».

Cet outil peut ensuite être associé à l'outil YACC, «Yet Another Compiler Compiler» ou «bison» :



```
                lexical        grammar
                 rules          rules
                   |              |
                   v              v
              +---------+    +---------+
              |   Lex   |    |  Yacc   |
              +---------+    +---------+
                   |              |
                   v              v
              +---------+    +---------+
    Input ->  |  yylex  | -> | yyparse | -> Parsed input
              +---------+    +---------+
```

Il est possible de disposer de l'outil «flex» qui réalise le même travail et accepte le même format de fichier en entrée.

**Structure d'un fichier Lex :**

```
1  %{
2  Prologue
3  %}
4
5  Déclaration
6
7  %%
8  Expressions régulières de reconnaissance pour chaque token & Actions associées
9  %%
10
11 Epilogue
```

La sortie de l'outil Lex est un programme C :

- ⋆ définissant la fonction `yylex()` permettant l'analyse lexical au sein d'un autre programme ;
- ⋆ soit autonome : si rien n'est précisé à la compilation (un appel à la fonction `yylex` est ajouté automatiquement) ;
- ⋆ soit sous forme d'une bibliothèque à relier, «*linker*» : dans ce cas c'est au programme principal de réaliser l'appel à `yylex`.

**Explications :**

- le **prologue** délimité par «%{» et «%}» contient :
  - les déclarations de types de données
  - les variables globales utilisées par les **actions.**
  - des commandes du pré-processeur, des macros ;
  - des «`#include`» des bibliothèques C.
- la **déclaration** : contient des définitions de sous expressions régulières :

```
D                   [0-9]
E                   [DEde][-+]?{D}+
%%
{D}+                printf("integer");
```

- les **expressions régulières** : regroupent les **règles** de reconnaissance des tokens et associe le code des **actions** à réaliser ;

  Il existe un certain nombre de **variables C** définie par Lex et utilisable dans les actions :
  - `yytext` : qui fournie la valeur reconnue par l'E.R. sous forme de chaîne de caractères ;
  - `yyleng` : qui fournie la longueur de la chaîne précédente.

- l'epilogue : il peut contenir des fonctions supplémentaires (bibliothèques de fonction).

  En particulier, on peut redéfinir les opérations **prédéfinies** de base de l'analyseur lexical, comme les opérations de gestion des caractères de l'entrée (`input()`, `output()`, `yysetstr()`, *etc.*)

*Le «prologue» et «l'épilogue» peuvent être vides, par exemple pour définir une commande de filtre sur du texte en entrée, c-à-d un programme autonome réalisant des traductions du texte en entrée vers le texte en sortie*.

# Analyseur lexical : un exemple

Soit le fichier `exemple_analyseur.lex`:

```
1 %{
2 #include <stdio.h>
3 %}
4
5 %%
6 stop    printf("Machine arretee\n");
7 start   printf("Machine en marche\n");
8 %%
```

Ensuite, on donne ce fichier à la commande flex qui crée un fichier `lex.yy.c` qui pourra ensuite être compilé :

```
xterm
$ flex exemple_analyseur.lex
$ gcc -o mon_parseur lex.yy.c -lfl
```

Lors de la compilation, on ajoute `-lfl` pour faire le lien avec la bibliothèque de flex (contenant une fonction `main` par défaut).

Lors de l'exécution, on obtient :

```
xterm
$ ./mon_parseur
stop
Machine arretee
```

*Un `ctrl-d` permet de mettre fin à la saisie (fermeture de l'entrée standard).*

# Analyseur lexical : un autre exemple

```
1 %{
2 #include <stdio.h>
3 %}
4
5 %%
6 [0-9]+            printf("NOMBRE\n") ;
7 [a-zA-Z][a-zA-Z0-9] printf("MOT\n");
8 [ \t]+              /* ignorer les espaces et les tabulations*/
9 %%
```

| En entrée : | En sortie : |
|---|---|
| `$ ./mon_parseur` | `NOMBRE` |
| `1` | `MOT` |
| `toto` | `MOT` |
| | `MOT` |
| `t0` | `NOMBRE` |
| `1t0` | `MOT` |
| | `NOMBRE` |
| `12t` | `t` |

*Question : pourquoi un* `t` *se retrouve en sortie ?*

# Analyseur lexical : exemple d'un format de fichier

```
 1 infos {
 2   categorie sport { basket };
 3   categorie technologie  { smartphone };
 4 };
 5
 6 publication "a_la_une" {
 7   type nouvelles, depeche;
 8   fichier "/donnees/nouvelles/lundi.txt";
 9   fichier "/donnees/depeche/national.log"
10 };
```

En analysant le fichier ci-dessus, on peut définir les catégories de mots-clés (tokens) suivants :

- ⋆ `MOT` : 'infos', 'categorie', 'publication', 'type', *etc*
- ⋆ `NOM_FICHER` : '/donnees/depeche/national.log'
- ⋆ `CARACTERE_SPECIAL` : '/' et '.'
- ⋆ `ACCOLADE_GCHE` : {
- ⋆ `ACCOLADE_DTE` : }
- ⋆ `POINT_VIR` : ';'
- ⋆ `VIRGULE` : ','
- ⋆ `DOUBLEQUOTE` : '"'

**Attention**

*Il faut faire attention à ce que les différents tokens soient suffisamment différentiables.*

# Analyseur lexical : exemple d'un format de fichier

```
 1 %{
 2 #include <stdio.h>
 3 %}
 4
 5 %%
 6 [a-zA-Z][a-zA-Z0-9]*    printf("MOT\n");
 7 [a-zA-Z0-9/-\.]+        printf("NOM_FICHIER\n");
 8 \"                      printf("DOUBLEQUOTE\n");
 9 \{                      printf("ACCOLADE_GCHE\n");
10 \}                      printf("ACCOLADE_DTE\n");
11 ;                       printf("POINT_VIR\n");
12 \n                      printf("\n"); /* retour à la ligne*/
13 [ \t]+                  /* ignore espaces et tabulation*/
14 %%
```

**Remarques :**

▷ La différence entre un `MOT` et un `NOM_FICHIER` tient à la présence des caractères '/' dans le texte indiquant un chemin d'accès à un fichier.

▷ Pour utiliser un caractère réservé comme « { » on le précède du caractère d'échappement « `\backslash` ».

```
1 /* -- inversecasse.lex --
2 Ce programme inverse la casse de toutes les lettres */
3
4 #include <ctype.h>
5 %%
6 [a-z]   printf("%c", toupper(yytext[0]));
7 [A-Z]   printf("%c", tolower(yytext[0]));
```

**Un second exemple**

```
 1 /* -- par.lex --
 2  Ce programme compte le nb de parentheses ouvrantes et
 3  verifie si le nb de parenthese fermante correspond. */
 4
 5  int nbParOuv=0; /* Variable globale du nb de par ouvrantes */
 6 %%
 7 "("      {nbParOuv++; ECHO;}
 8 \)       { if (nbParOuv>0)
 9             {
10               nbParOuv--;
11               ECHO; /* affiche yytext */
12             }
13          else
14             printf ("Votre parenthesage est incorrect\n");
15     }
```

# Analyseur lexical : utilisation

⋆ Lancement de l'analyseur lexical :

```
1 yylex(); /* retourne 0 si plus de lexème */
```

⋆ Redéfinition des fichiers d'entrée et de sortie :

- `yyin` : désigne le descripteur du fichier d'entrée, par défaut `stdin` ;
- `yyout` : désigne le descripteur du fichier de sortie, par défaut `stdout`.

Il est possible de les redéfinir :

```
1 yyin = fopen("mon_entree", "r");
2 yyout = fopen("ma_sortie", "w");
```

⋆ Recul dans le flux d'entrée :

```
1 yyless(3)
```

*Supprime les 3 derniers caractères de* `yytext`*. Les caractères supprimées seront fournis pour la reconnaissance du lexème suivant.*

⋆ Fusion avec l'unité lexicale suivante :

```
1 yymore()
```

*Permet de fusionner* `yytext` *avec l'unité lexicale reconnue précédemment.*

⋆ Action exécutée lors de l'arrivée à la fin de l'entrée courante :

```
1 yywrap()
```

*Cette fonction doit renvoyer 1 si la fin est effective, ou bien zéro dans le cas contraire (ouverture d'un nouveau fichier par exemple pour redéfinir l'entrée courante).*

**La gestion du «\n»**

Lex gère le «retour à la ligne» d'une manière spéciale :

- le caractère «\n» ne s'associe pas avec le « . » :
  - il sert, par défaut, de **terminaison** à l'appariement de motif («pattern matching») ;
- il peut être intégré dans une expression régulière uniquement par son expression explicite :

```
1 \n  /* ignorer le retour à la ligne */
```

- il faut faire attention à son intégration pour ne pas «traiter» le fichier d'entrée d'un seul coup **avec une seule expression régulière !**

**La gestion de la fin de fichier «End-Of-File»**

Il existe un symbole spécial : `<<EOF>>`

```
1 <<EOF>> { printf("Fin du fichier\n");
2       yyterminate();
3      }
```

*Ce symbole ne peut pas être utilisé au sein d'une expression régulière : il doit être utilisé seul pour définir une règle associé à son traitement.*

*Si l'on veut relancer le travail de l'analyseur sur un contenu différent (par exemple, à partir d'un nouveau fichier d'entrée), il faut regarder le travail de la fonction `yywrap`.*

**Attention**

La gestion explicite de l'EOF **ne permet plus de quitter l'analyseur** : il faut utiliser la fonction `yyterminate` pour le terminer.

**Gestion des caractères accentués sous Unix : utilisation du codage UTF-8**

Le codage UTF-8, «*Universal Coded Character Set + Transformation Format – 8-bit*» permet de conserver un codage sur un octet pour les caractères latins non accentués et de passer sur un codage sur deux, voire trois ou quatre octets pour des caractères non latins ou des idéogrammes.

```
xterm
$ echo -n "e" | xxd
00000000: 65                                      e
```

*Le caractère «e» est codé avec la valeur de l'octet 65.*

```
xterm
$ echo -n "é" | xxd
00000000: c3a9                                    ..
```

*Le caractère «é» est codé sur deux octets.*

```
xterm
echo -n "電腦" | xxd
00000000: e99b bbe8 85a6                          ......
```

*L'idéogramme « 電 » est codé sur 3 octets. Les deux symboles signifient «ordinateur».*

Le symbole «€», euro, est également sur 3 octets :

```
xterm
$ echo -n "€" | xxd
00000000: e282 ac                                 ...
```

**Solution ancienne : passage à un codage étendu sur 8 bits au lieu de 7bits**

▷ codage «ISO-8859-1» ou «LATIN1» : permet de coder les caractères accentués des pays d'Europe de l'ouest.
*Il ne permet pas de coder le symbole de l'euro «€», ni le «œ».*

▷ le codage «Windows-1252» : permet de coder l'ensemble des symboles du français ainsi que le symbole de l'euro.

**Solution actuelle : utilisation du codage UTF8 et Consommation mémoire**

□ un **caractère non accentué** dont la valeur associée suivant le code ASCII ou ANSI est entre 32 et 127 (soient 7 bits) : codé sur 1 octet ;

□ un **caractère accentué** : codé sur 2 octets ;

□ un **caractère non latin** ou un **idéogramme** : 2,3 ou 4 octets (6 au maximum).

⟹*L'utilisation de caractères accentués change la taille des données et complique la tâche de l'analyseur lexical.*

Il est possible de convertir le texte d'UTF-8 vers le codage 1252 à l'aide de la commande `iconv` :

```
xterm
$ echo -n "é" | iconv -f UTF8 -t WINDOWS-1252 | xxd
00000000: e9                                          .
```

*La conversion inverse est également possible (elle est obligatoire pour un affichage correct dans un environnement Unix configuré pour traiter de l'UTF8)*

## 4 Rappels sur les grammaires



Pour décrire la syntaxe d'un langage de programmation, on utilise une grammaire.

Une grammaire, *hors contexte*, $G$ définie par $(V_T \cup \{\$\}, V_N, S, P)$, où :

- $V_T$ est l'ensemble des symboles terminaux ;
- $V_N$ est l'ensemble des symboles non terminaux ;
- $S$ est l'axiome, c-à-d l'élément de départ de la grammaire ;
- $P$ est l'ensemble des règles de production (hors contexte → un seul non terminal à gauche pour chaque règle) ;
- $\$$ désigne la fin de la chaîne à analyser ;
- $V = V_T \cup V_N$ est le «vocabulaire» de la grammaire.

Exemple : Soit $G$ définie par :

- $V_T = \{id, +, *, ), (\}$ ;
- $V_N = \{E, T, F\}$ ;
- $S = E$ ;
- $P = \{E \to E + T, E \to T, T \to T * F, T \to F, F \to id, F \to (E)\}$

Soit $p$, $id * id + id$, une séquence de terminaux.

*Comment déterminer si $p$ est correcte, c-à-d si $p$ est acceptée par $G$ ?*

On peut utiliser la méthode suivante (analyse ascendante) :

1) on lit les symboles terminaux de $p$ les uns après les autres ;

2) lorsqu'on a lu une séquence $\alpha$ de terminaux qui peut constituer la partie droite d'une règle $A \rightarrow \alpha$ alors il est possible de remplacer $\alpha$ par $A$ avant de continuer à lire la suite des terminaux de $p$. Ce remplacement s'appelle une **réduction** ;

3) durant la lecture de $p$, après avoir fait d'éventuelles **réductions**, la séquence de terminaux qui a été lue depuis le début a été transformée en une séquence $\gamma$ de terminaux et de non-terminaux.

4) lorsqu'une partie $\alpha$ de $\gamma$ peut être réduite par un non-terminal $A$ à l'aide d'une règle $A \rightarrow \alpha$, alors il est possible de faire une **réduction** avant de continuer à lire les terminaux de $p$.

5) $p$ est correcte si après sa lecture complète, il est possible de la réduire en S.

COMPLÉMENT


Sur l'exemple, avec $P = \{$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow id$
6. $F \rightarrow (E)\}$

Ce qui donne :

par (5) : $id * id + id \leftarrow F * id + id$
par (4) $F * id + id \leftarrow T * id + id$
par (5) $T * id + id \leftarrow T * F + id$
par (3) $T * F + id \leftarrow T + id$
par (2) $T + id \leftarrow E + id$
par (5) $E + id \leftarrow E + F$
par (4) $E + F \leftarrow E + T$
par (1) $E + T \leftarrow E$

On a procédé à des réductions à droite, c-à-d que les règles utilisées correspondent à des dérivations à droite.

On peut représenter les différentes réductions par un tableau.

**Explications :**

- *Colonne de gauche* : représente le contenu courant de $\gamma$ stocké dans une pile ;
- *Colonne du milieu* : représente la partie de $p$ qui n'a pas été encore traitée ;
- *Colonne de droite* : indique l'opération réalisée pour passer à la ligne suivante :
  - réduction d'une partie droite de $\gamma$, dans ce cas on précise la règle utilisée ;
  - lecture du prochain terminal.

| Pile | Partie de $p$ non traitée | Règle appliquée (si réduction) |
|---|---|---|
| | $id * id + id$ | *lecture* |
| $id$ | $*id + id$ | $F \rightarrow id$ |
| $F$ | $*id + id$ | $T \rightarrow F$ |
| $T$ | $*id + id$ | *lecture* |
| $T*$ | $id + id$ | *lecture* |
| $T * id$ | $+id$ | $F \rightarrow id$ |
| $T * F$ | $+id$ | $T \rightarrow T * F$ |
| $T$ | $+id$ | $E \rightarrow T$ |
| $E$ | $+id$ | *lecture* |
| $E+$ | $id$ | *lecture* |
| $E + id$ | | $F \rightarrow id$ |
| $E + F$ | | $T \rightarrow F$ |
| $E + T$ | | $E \rightarrow E + T$ |
| $E$ | | |

**Utilisation de la pile**

Sur le tableau précédent, la chaîne courante $\gamma$ se trouve dans une pile :

- ⋆ l'extrémité gauche de $\gamma$ est en bas de la pile ;
- ⋆ l'extrémité droite de $\gamma$ est en haut de la pile.

Cette approche est pratique :

- ▷ une réduction s'effectue sur une partie droite de $\gamma$, ce qui se traduit par une modification d'une partie du sommet de la pile ;
- ▷ une lecture ajoute un symbole terminal à droite de $\gamma$, ce qui se traduit par l'ajout d'un symbole sur le sommet de la pile.

**Problème :**

- □ Lorsqu'il est possible de réaliser une réduction, il faut décider s'il faut effectuer cette réduction avant de continuer la lecture.
- □ Lorsque plusieurs réductions sont possibles et que l'on décide de réaliser une réduction, il faut choisir la règle de production à utiliser.

*Si on fait un mauvais choix, on risque de ne pas aboutir au symbole de départ, même si la chaîne de départ est correcte !*

**Exemple de problème :**

Sur l'exemple précédent :

| Pile | Partie de $p$ non traitée | Règle appliquée (si réduction) |
|---|---|---|
| | $id * id + id$ | *lecture* |
| $id$ | $*id + id$ | $F \rightarrow id$ |
| $F$ | $*id + id$ | $T \rightarrow F$ |
| $T$ | $*id + id$ | *lecture* |

La pile contient le symbole non terminal T, on peut :

- choisir de continuer la lecture (*lecture* de *), ce qui a été fait ;
- choisir de réduire par l'utilisation de la règle $E \rightarrow T$, ce qui aurait donné :

| | | |
|---|---|---|
| $F$ | $*id + id$ | $T \rightarrow F$ |
| $T$ | $*id + id$ | $E \rightarrow T$ |
| $E$ | $*id + id$ | *lecture* |
| $E*$ | $id + id$ | *lecture* |
| $E * id$ | $+id$ | $F \rightarrow id$ |
| $E * F$ | $+id$ | $T \rightarrow T * F$ |

*Bloqué !*

## Les «*parsers*» LR



Ce sont des **analyseurs syntaxiques** fonctionnant de manière **ascendante** :

- ▷ L signifie que le texte analysé est lu de gauche, «*left*», à droite ;
- ▷ R signifie qu'on effectue une séquence de réductions à droite, «*right*».

*Une séquence de réductions est dite «à droite» si elle consiste à parcourir une séquence de dérivations à droite en sens inverse*.

**Exemple** :

⋆ Une séquence de **dérivations à droite** :
$E \rightarrow E + T \rightarrow E + F \rightarrow E + id \rightarrow T + id \rightarrow F + id \rightarrow id + id$

⋆ La séquence de **réductions à droite** correspondante :
$id + id \leftarrow F + id \leftarrow T + id \leftarrow E + id \leftarrow E + F \leftarrow E + T \leftarrow E$

## Les différentes catégories d'analyseurs LR



Il existe plusieurs catégories d'analyseurs LR, les plus connus sont :

- SLR(1), LR(1), LALR(1) ;
  - LR veut dire «*Left to right, Rightmost derivation*» ;
  - SLR veut dire «*Simple LR*»
  - LALR veut dire «*Look-Ahead LR*» (avec anticipation) ;
- le paramètre 1 indique qu'à tout moment le parser connaît le premier symbole de la partie non encore traitée du texte à analyser ;
- les analyseurs SLR(1) sont les plus restrictifs ;
- un analyseur SLR(1) est un analyseur LALR(1) particulier ;
- un analyseur LALR(1) est un analyseur LR(1) particulier (produit par YACC).

Ce qui donne : $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1)$

Les analyseurs LR(1) sont les **plus généraux** et peuvent s'appliquer à un **plus grand nombre** de grammaires.

## Principe de fonctionnement d'un analyseur LR



Le fonctionnement des **différents analyseurs** repose sur l'utilisation d'une **table** composée de deux parties :

- ⋆ une partie «action» ;
- ⋆ une partie «branchement» (à la manière d'un `goto`).

*Cette table de transitions d'états correspond à une écriture particulière d'un automate à nombre fini d'états.*

La **table** contient aussi des informations supplémentaires indiquant à l'analyseur quand il faut effectuer :

- ◦ une **réduction**, «*Reduce*» ;
- ◦ une **lecture**, «*Shift*».

Les trois types d'analyseurs, SLR(1), LR(1) et LALR(1), sont :

- ▷ **différents** dans leur méthode de **construction** de la table ;
- ▷ **identiques** dans leur méthode **d'utilisation** de la table.

# Exemple de table

Soit $G$ définie par :

- $V_T = \{id, +, *, ), (\}$ ;
- $V_N = \{E, T, F\}$ ;
- $S = E$ ;
- $P = \{$
  1. $E \to E + T$
  2. $E \to T$
  3. $T \to T * F$
  4. $T \to F$
  5. $F \to id$
  6. $F \to (E)\}$

La table pour un analyseur SLR(1) :

| État | $+$ | $*$ | $id$ | ( | ) | \$ | $E$ | $T$ | $F$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | | s5 | s4 | | | 1 | 2 | 3 |
| 1 | s6 | | | | | OK | | | |
| 2 | r2 | s7 | | | r2 | r2 | | | |
| 3 | r4 | r4 | | | r4 | r4 | | | |
| 4 | | | s5 | s4 | | | 8 | 2 | 3 |
| 5 | r5 | r5 | | | r5 | r5 | | | |
| 6 | | | s5 | s4 | | | | 9 | 3 |
| 7 | | | s5 | s4 | | | | | 10 |
| 8 | s6 | | | | s11 | | | | |
| 9 | r1 | s7 | | | r1 | r1 | | | |
| 10 | r3 | r3 | | | r3 | r3 | | | |
| 11 | r6 | r6 | | | r6 | r6 | | | |

## Exemple de table



**Explications :**

- colonne de gauche : différents états dans lesquels l'analyseur peut se trouver ;

  *Dans l'exemple, il y a 12 états (0 à 11)*

- ligne de haut : symboles terminaux et non-terminaux de la grammaire

  *Le symbole (terminal)* $\$$ *est ajouté à la fin du texte à analyser* ;

- partie centrale, *sous les symboles terminaux* : **Partie action**
  - lettre s signifie qu'il faut effectuer une opération «*shift*» (lecture)
  - lettre r signifie qu'il faut effectuer une opération «*reduce*» (réduction)

- Partie de droite *sous les symboles non-terminaux* : **Partie branchement (goto)**

## Exemple de table

Algorithme LR :

▷ Initialisation :
- ◇ symbole $\$$ mis à la fin du texte à analyser;
- ◇ état 0 (de départ) est empilé;
- ◇ prochain_symbole_terminal_non_traité := premier_symbole_du_texte_à_analyser;

▷ **Répéter**
- ◇ $j$ := état_au_sommet_de_la_pile; */*état courant de l'analyseur */*
- ◇ $a$ := prochain_symbole_terminal_non_traité;
- ◇ si $(table[j,a] = sm)$ alors
  - ⋆ symbole $a$ et état $m$ sont empilés;
  - ⋆ prochain_symbole_terminal_non_traité := symbole_qui_suit($a$);
- ◇ si $(table[j,a] = rm)$ alors
  - ⋆ B := partie_gauche_de_la_règle_de_production(m); */* B = symbole non-terminal */*
  - ⋆ L := longueur_de_la_partie_droite_de_la_règle_de_production(m)
  - ⋆ on dépile 2L éléments de la pile; */* L paires (symbole,état) */*
  - ⋆ k := état_au_sommet_de_la_pile; */* après avoir dépilé */*
  - ⋆ symbole B et état table[k,B] sont empilés;
- ◇ si $(table[j,a] = OK)$
  - ⋆ alors texte_à_analyser est acceptée
  - ⋆ sinon erreur_détectée

▷ **Jusqu'à ce que** texte_à_analyser est acceptée ou erreur_détectée.

## Exemple de table



Si on utilise l'algorithme sur le texte à analyser $id * id + id$ :

| Pile | texte restant à traiter | Commentaires |
|---|---|---|
| *vide* | $id * id + id$ | – symbole \$ rajouté à la fin du texte ;<br>– 0 est empilé ; |
| 0 | $id * id + id\$$ | $table[0, id] = s5$ donc :<br>– on empile $id$ et 5 ;<br>– on avance au symbole suivant (shift) ; |
| 0 $id$ 5 | $*id + id\$$ | $table[5, *] = r5$ et règle(5) $F \rightarrow id$ donc :<br>– on dépile 2 symboles (5 et $id$) ;<br>– on empile $F$ et 3 (car $table[0, F] = 3$) ; |
| 0 $F$ 3 | $*id + id\$$ | $table[3, *] = r4$ et règle(4) $T \rightarrow F$ donc :<br>– on dépile 2 symboles (3 et $F$) ;<br>– on empile $T$ et 2 (car $table[0, T] = 2$) ; |
| 0 $T$ 2 | $*id + id\$$ | $table[2, *] = s7$ donc :<br>– on empile $*$ et 7 ;<br>– on avance au symbole suivant (shift) ; |
| 0 $T$ 2 $*$ 7 | $id + id\$$ | $table[7, id] = s5$ donc :<br>– on empile $id$ et 5 ;<br>– on avance au symbole suivant (shift) ; |

## Exemple de table



| Pile | texte restant à traiter | Commentaires |
|---|---|---|
| 0 $T$ 2 $*$ 7 $id$ 5 | $+id\$$ | $table[5,+] = r5$ et règle(5) $F \to id$ donc :<br>– on dépile 2 symboles (5 et $id$) ;<br>– on empile $F$ et 10 (car $table[7,F] = 10$) ; |
| 0 $T$ 2 $*$ 7 $F$ 10 | $+id\$$ | $table[10,+] = r3$ et règle(3) $T \to T * F$ donc :<br>– on dépile 6 symboles (10, $F$, 7, $*$, 2 et $T$) ;<br>– on empile $T$ et 2 (car $table[0,T] = 2$) ; |
| 0 $T$ 2 | $+id\$$ | $table[2,+] = r2$ et règle(2) $E \to T$ donc :<br>– on dépile 2 symboles (2 et $T$) ;<br>– on empile $E$ et 1 (car $table[0,E] = 1$) ; |
| 0 $E$ 1 | $+id\$$ | $table[1,+] = s6$ donc :<br>– on empile $+$ et 6 ;<br>– on avance au symbole suivant (shift) ; |
| 0 $E$ 1 $+$ 6 | $id\$$ | $table[6,id] = s5$ donc :<br>– on empile $id$ et 5 ;<br>– on avance au symbole suivant (shift) ; |
| 0 $E$ 1 $+$ 6 $id$ 5 | $\$$ | $table[5,\$] = r5$ et règle(5) $F \to id$ donc :<br>– on dépile 2 symboles (5 et $id$) ;<br>– on empile $F$ et 3 (car $table[6,F] = 3$) ; |

## Exemple de table



| Pile | texte restant à traiter | Commentaires |
|---|---|---|
| 0 $E$ 1 + 6 $F$ 3 | $ | $table[3, \$] = r4$ et règle(4) $T \to F$ donc :<br>– on dépile 2 symboles (3 et $F$) ;<br>– on empile $T$ et 9 (car $table[6, T] = 9$) ; |
| 0 $E$ 1 + 6 $T$ 9 | $ | $table[9, \$] = r1$ et règle(1) $E \to E + T$ donc :<br>– on dépile 6 symboles (9, $T$, 6, +, 1 et $E$) ;<br>– on empile $E$ et 1 (car $table[0, E] = 1$) ; |
| 0 $E$ 1 | $ | $table[1, \$] = Ok$ donc texte correct ! |

Et on a fini !

## Interprétation intuitive d'une table



Construire une **mémoire** de l'analyseur :

- ▷ **Passé courant** d'un analyseur : séquence de «shifts» et de «reduces» effectués

  Exemple : texte à analyser $id * id + id$

  Après lecture de id * , on a affectué :

  - ◇ shift (lecture de id)
  - ◇ reduce (règle 5)
  - ◇ reduce (règle 4)
  - ◇ shift (lecture de *)

  } définit la mémoire après lecture de id *

- ▷ **État courant** d'un analyseur :
  - ◇ informe sur le passé courant du parseur ;
  - ◇ n'informe pas forcément sur tout le passé ;

    *Par exemple, l'état peut nous informer sur la séquence de shifts et reduces , sans préciser les numéros des règles utilisés*

  On observe que :
  - ◇ les **états générés** par un analyseur LR(1) sont les **plus précis** et donc les **plus nombreux** ;
  - ◇ les **états générés** par un analyseur SLR(1) sont les **moins précis** et donc les **moins nombreux**.

## Conflits



Lorsqu'un **analyseur** est **utilisé** pour une grammaire pour laquelle il n'est **pas applicable**, alors des **conflits** peuvent avoir lieu.

C'est par exemple le cas si :

- ⋆ un analyseur SLR(1) est utilisé pour une grammaire LR(1) ou LALR(1) ;
- ⋆ un analyseur LALR(1) est utilisé pour une grammaire LR(1) ;
- ⋆ un analyseur LR est utilisé pour une grammaire qui n'est pas LR (par exemple, une grammaire ambiguë n'est pas LR) ;

Un **conflit** est détecté lors de la construction de la table de l'analyseur, à chaque fois que dans une même position de la partie action de la table, on obtient :

- ⋆ deux actions $si$ et $rj$ : on dit qu'il y a **conflit «*shift*»/«*reduce*»**
  ⟹l'analyseur ne sait pas s'il doit effectuer un «*reduce*» à l'aide de la règle ($j$) ou bien un «shift» ;
- ⋆ deux actions $ri$ et $rj$ : on dit qu'il y a **conflit «*reduce*»/«*reduce*»**
  ⟹l'analyseur sait qu'il doit effectuer un «*reduce*», mais il ne sait pas s'il doit appliquer la règle $i$ ou la règle $j$

## Exemple de conflit



**La gestion du «if-then-else»**

Soient les règles :

1. $Si_Inst$ → IF $Expr$ THEN $Si_Inst$
2. $Si_Inst$ → IF $Expr$ THEN $Si_Inst$ ELSE $Si_Inst$

Soit la phrase IF $Exp1$ THEN IF $Exp2$ THEN $Inst1$ ELSE $Inst2$

Lorsque l'analyseur arrive à ELSE, **il ne sait pas** s'il doit effectuer :

▷ une **réduction** de IF Exp2 THEN Inst1 ⟹ *cela aura pour effet d'associer le ELSE au premier IF*

▷ un **décalage** sur ELSE ⟹ *cela aura pour effet d'associer le ELSE au second IF*

Bien que tout le passé soit connu, il n'est **pas possible de décider**.

*Ceci est dû au fait que la grammaire est ambigüe.*

## Résolution de conflit



- **Première approche** :
  - En cas de conflit $si/rj$, on peut par exemple décider de ne garder que $si$
    *C'est l'approche utilisée par* `YACC`.
    Dans le cas du conflit shift/reduce du If-then-else, `YACC` décidera donc d'associer le ELSE au second IF
  - En cas de conflit $ri/rj$, on peut par exemple décider de garder la réduction ayant le plus petit numéro de règle (Exemple : pour r4/r3, on garde r3)
    *C'est l'approche utilisée par* `YACC`.

- **Seconde approche** : elle consiste à modifier les règles de production de la grammaire
  *On reviendra sur ce point lorsqu'on étudiera* `YACC`.

Il est possible de modifier le comportement de l'analyseur lexical en fonction du «contexte» situé à gauche.

**Utilisation de «commutation entre contextes»**

On veut interpréter l'expression régulière `[a-zA-Z]+` de deux manières, selon le **contexte** :

1. si `mot` entre guillemet, alors il représente une **chaîne de caractères** ;
2. si `mot` pas entre guillemet, alors c'est un **identificateur** ;

On définit deux états `CHAINE` et `NORMAL`, pour les deux contextes.

Ces états sont utilisés comme suit :

- au début, l'état est mis à `NORMAL` par l 'instruction `BEGIN NORMAL` ;
- chaque fois qu'on rencontre un guillemet, on *commute* d'un contexte à l'autre (entre `NORMAL` et `CHAINE`).

Dans le source Lex, on **préfixe** les expressions régulières par un **contexte d'activation.**

Ainsi, lorsqu'on reconnaît une expression régulière `[a-zA-Z]+` :

▷ au départ l'état est `INITIAL`, qui correspond à l'état initial ;

▷ si l'état courant est `NORMAL` : alors c'est un identificateur qui est reconnu ;

▷ si l'état courant est `CHAINE` : alors c'est une chaîne de caractères ;

▷ un état sans contexte correspond à l'état `0`.

*Toute règle sans condition est active tout le temps.*

*Une règle préfixée avec le contexte* `<INITIAL>` *est active au début du travail de l'analyseur.*

Soit le programme suivant :

```
 1 %start NORMAL CHAINE
 2 %%
 3 <NORMAL>[a-zA-Z]+   printf("Reconnaissance d'un identificateur: %s\n", yytext);
 4 <NORMAL>\"      BEGIN CHAINE;
 5 <CHAINE>[a-zA-Z]+ printf("Reconnaissance d'une chaine : %s\n", yytext);
 6 <CHAINE>\"      BEGIN NORMAL;
 7 <NORMAL,CHAINE>.  /* aucune action */
 8 <NORMAL,CHAINE>\n /* aucune action */
 9 %%
10 void main()
11 {
12   yyin = fopen("entree.txt", "r");
13   BEGIN NORMAL; /* On definit le mode par defaut */
14   yylex(); /* yylex appelé une seule fois car pas de return */
15   fclose(yyin);
16 }
```

À la ligne 7 et 8, avec `<NORMAL, CHAINE>`, on définit une règle Lex valable dans les deux contextes.

Gestion des règles ambigües dans Lex :

```
1 integer   /* reconnaissance d'un mot clé */
2 [a-z]+    /* reconnaissance d'un identifiant */
```

Ces deux règles sont **ambigües** car elles peuvent s'appliquer **simultanément** sur un même contenu.

Lex applique les règles de sélection suivantes :

1. **sélection** de l'expression régulière qui donne la **plus longue** correspondance ;
2. si **plusieures expressions régulières** donnent la **même taille**, **sélection** de la **première** par ordre de saisie.

Dans le cas où le texte est `integers` alors c'est la ligne 2 du fichier Lex qui est sélectionnée.

**Attention** : la règle de sélection 1 peut entraîner des erreurs involontaires :

```
1 '.*'  /* reconnait une chaine */
```

Si on donne l'entrée `'premiere' chaine d'abord, 'seconde' ensuite` alors Lex va retourner : `premiere' chaine d'abord, 'seconde`.

Il faut alors réécrire en :

```
1 '[^'\n]*'   /* reconnait une chaine */
```

Ce qui évite le problème.

*D'autre part le caractère* `.` *ne peut pas s'accorder au caractère* `\n`.

# Lex : un avant goût de grammaire

Il est possible d'utiliser des règles Lex ambigües, à l'aide de l'opérateur `REJECT`.
*Attention : la commande yyless(0) ne passe pas à la règle suivante comme REJECT et doit s'accompagner d'un changement de contexte.*

Cette opérateur permet d'aller vers la prochaine règle possible :

```
1 anticonstitutionnel occurrence++; REJECT;
2 constitutionnel     occurrence++;
```

Ici, on va compter à la fois le mot `constitutionel` avec la première règle et avec la seconde.

Pour faire de la **cryptanalyse fréquentielle**, on peut chercher le nombre de **digrammes**, groupe de deux lettres, présents dans un texte :

```
1 %%
2 [a-z][a-z]    {
3                   digramme[yytext[0]][yytext[1]]++;
4                   REJECT;
5               }
6 .             ;
7 \n            ;
```

# 6 Liens entre Lex et un analyseur syntaxique

Pour utiliser un **analyseur lexical**, construit par Lex, dans un **analyseur syntaxique**, il faut pouvoir fractionner le travail de Lex à la reconnaissance d'un seul lexème à la fois.

L'**analyseur syntaxique**, ou *parser*, va effectuer un appel à la fonction `yylex()` pour le traitement de chaque lexème.

Dans ce cas d'utilisation, l'action associée à la reconnaissance d'un lexème :
- retourne la nature du lexème reconnu ;
- affecte la valeur du lexème à une variable globale commune avec le *parser*.

Il est nécessaire de :

a. définir une **valeur de retour** pour chaque action définie dans le source de Lex ;

b. **partager** cette définition avec l'outil utilisant Lex.

La solution est d'utiliser un **fichier d'entête commun**, par exemple nommé «`mes_lexemes.h`», à inclure dans le fichier source Lex et dans l'analyseur syntaxique :

```
1 #define ENTIER     1
2 #define QUOTE    2
3 #define MULTIPLIER  3
4 ...
```

*Ici, on associe pour chaque lexème une* **valeur entière** *que l'on retourne dans l'action.*

Le fichier d'entête est ensuite inclus dans le source Lex :

```
1 %{
2   #include "mes_lexemes.h"
3 %}
4 DIGIT ([0-9])
5 %%
6 {DIGIT}+ { mon_lexeme = atoi(yytext);
7        #ifdef DEBUG
8          fprintf(stderr,"Entier :%d\n", mon_lexeme);
9        #endif
10       return ENTIER;
11 }
```

Pour la variable recevant la valeur du lexème on peut utiliser la définition «`symbole mon_lexeme`» avec :

```
1 typedef union
2 {
3   int entier;
4   char *chaine;
5 }  type_lexeme;
```

*L'union permet de considérer la même valeur soit comme un entier* ***ou*** *soit comme un pointeur sur une chaîne*.

```
1 typedef struct
2 {
3   int type;
4   type_lexeme val;
5 } symbole;
```

*Le struct permet d'associer un entier* ***et*** *un pointeur sur une chaîne*.

YACC, «*Yet Another Compiler Compiler*», est un programme permettant :

- ▷ d'interpréter une grammaire de type LALR(1) ;
- ▷ de produire le programme source d'un analyseur syntaxique pour le langage engendré par cette grammaire ;
- ▷ d'effectuer des actions sémantiques liées à cette grammaire.

La syntaxe des fichiers, d'extension « `.y` », qu'il accepte est proche de celle de Lex :

```
1 %}
2   prologue
3 %}
4   Déclaration
5 %%
6   Règles de productions
7 %%
8   Epilogue
```

Le fichier est traité de la façon suivante :

```
$ bison mon_fichier.y
```

Ce qui produit le fichier `y.tab.c` contenant le source C de l'analyseur syntaxique.

Dans ce fichier est définie la fonction `yyparse()` qui réalise l'analyse syntaxique en utilisant la fonction `yylex()` fournie par Lex.

La partie «prologue» :

* des déclarations et définitions C ;
* des «#include» ;
* des définitions de variables qui seront globales à tout l'analyseur syntaxique.

La partie «Déclaration» :

* définie le type de la variable `yylval` qui sera partagée avec Lex pour récupérer la valeur d'un lexème :
  - par défaut il est de type `int` ;
  - peut être redéfini dans la partie «prologue» :

    ```
    #define YYSTYPE nom-de-type
    ```

    *Dans ce cas là, il faut aussi le mettre dans la partie «Épilogue» du fichier Lex*.
  - peut être redéfini sous forme d'`union` pour définir le vocabulaire de la grammaire :

    ```
    %union
      {   int nombre;
        char *chaine; }
    ```
  - ou sous forme de `struct` :

    ```
    %union
        { struct {  int nb;
                char * val; } bloc }
    ```

La partie «Déclaration», *suite* :

Lorsque YACC reconnaît un symbole non terminal il utilise la variable `yyval` pour le communiquer au programme l'utilisant.
Cette variable est définie par la même définition que `yylval` avec le `%union`.

- ⋆ définition des **symboles terminaux** à l'aide de `%token` ;
  *Lorsque le terminal n'a pas de valeur ou possède une valeur entière, il n'est pas nécessaire de spécifier le type, car il est entier par défaut.*

- ⋆ définition des **symboles non terminaux** à l'aide de `%type`.
  *Lorsqu'un non terminal n'a pas de valeur ou une valeur entière, il n'est pas nécessaire de le déclarer.*
  Les types spécifiés par `%token` et `%type` ont été définis par `%union`.

- ⋆ des informations sur **l'associativité** et la **précédence** des opérateurs à l'aide de `%left` `%right` `noassoc` et `prec` :

```
1 %left PLUS MOINS
2 %left MULT DIV
```

  - ◇ une précédence identique pour PLUS et MOINS ;
  - ◇ une précédence identique pour MULT et DIV ;
  - ◇ une précédence de MULT et DIV supérieure à celle de PLUS et MOINS (définition sur une seconde ligne).

- ⋆ la déclaration du **symbole non terminal de départ** avec `%start`.

Sur l'exemple de la grammaire précédente : Soit $G$ définie par :

- $V_T = \{nb, +, *, ), (\}$ ;
- $V_N = \{E, T, F\}$ ;
- $S = E$ ;
- $P = \{$
  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * F$
  4. $T \rightarrow F$
  5. $F \rightarrow nb$
  6. $F \rightarrow (E)\}$
- les terminaux $*, +, (, )$ n'ont pas besoin de valeur associée ;
- le terminal $nb$ correspond à un nombre entier.

On peut utiliser la déclaration suivante pour «yylval» :

```
1 %union
2   { int *valeur; /*pointeur sur la valeur d'un terminal */
3         int type; /* nature d'un non-terminal */
4   }
```

Ce qui va donner :

```
1 %token <valeur> NB MULT ADD PAROUV PARFER
2 %type <type> E T F
3 %left ADD
4 %left MULT
5 %start E
```

Ainsi l'expression $c * d + e + f * g$ est interprétée comme $((c * d) + e) + (f * g)$

**La partie «Règles de productions»**

Soit $n$ règles de production ayant le même non-terminale en partie gauche :

```
1 Non_terminal :    corps_1 { actions_1}
2         | corps_2 { actions_2}
3         | …
4         | corps_n { actions_n}
5       ;
```

*Chaque* $corps_i$ *correspond à la partie droite de la règle* $R_i$*.*

La partie $actions_i$ permet de manipuler les valeurs des terminaux et non-terminaux :

- soit la règle `A : U1 U2 … UN {actions}`, où :
  - `A` est un symbole non terminal ;
  - `Ui` est un symbole terminal ou non-terminal, avec $i = 1..n$
- dans la partie `actions`, on peut utiliser les symboles suivants :
  - `$$` pour se référer à la valeur de `A` ;
  - `$i` pour se référer à la valeur de `Ui`.
- lorsqu’aucune action n’est précisée, YACC génère l’action `$$=$1;`.

Lorsque la **réduction** correspondante est effectuée, alors la variable globale `yyval` reçoit implicitement la valeur `$$`.

# L'analyseur syntaxique YACC : sur un exemple

Sur la grammaire précédente :

```
1 E : E ADD T   { $$ = $1 + $3; }
2   | T        { $$ = $1; }
3   ;
4 T : T MULT F    { $$ = $1 * $3; }
5   | F        { $$ = $1; }
6   ;
7 F : NB       { $$ = *$1; /*Il faut allouer l'entier en mémoire */}
8   | PAROUV E PARFER {$$=$2;}
9   ;
```

Explications :

▷ la valeur d'un symbole terminal est un **pointeur** sur la valeur du nombre ;

▷ la valeur d'un non-terminal `A` est la **valeur** de l'expression qui correspond à `A`.

La partie «Epilogue» contient du code C :

▷ des fonctions utilisées dans les actions associées aux règles ;

▷ le programme principal qui fait appel à l'analyseur syntaxique (c-à-d à la fonction `yyparse()`).

Exemple :

```
1 void main () {
2   if ( yyparse() == 0 ) printf("résultat = %d \n", yyval.nbre);
3   else printf("Erreur de syntaxe \n")
4 }
```

**Pour réaliser l'association avec Lex**

a. dans la partie «prologue» de lex, on inclus un fichier d'en-tête que va produire YACC pour définir le vocabulaire de la grammaire, les «tokens» (par ex. «`defs_a_inclure.h`») ;

b. on inclus le fichier «`global.h`» dans la partie «prologue» de Lex et de YACC :

```
1 #define YYSTYPE mon_type
2 extern YYSTYPE yylval;
```

c. on demande à YACC (ou ici *bison*) de générer le fichier `defs_a_inclure.h`:

```
xterm
$ bison -d analyseur.y --graph=desc.txt
$ cp analyseur.tab.h defs_a_inclure.h
```

On va créer un **outil de calcul interactif** permettant de faire des calculs d'expression mathématique contenant des réels et utilisant les opérateurs de puissance, multiplication, division, soustraction et addition.

▷ Soient les fichiers `analyse_lexical.l` et `analyse_syntaxique.y` dont le contenu est donné dans les transparents suivants ;

▷ le fichier `global.h` qui va définir le type de `yylval` pour échanger entre Lex et YACC :

```
1 #define YYSTYPE double
2 extern YYSTYPE yylval;
```

▷ la procédure de compilation et un exemple d'utilisation :

```
xterm
$ lex analyseur.l
$ bison -d analyseur.y --graph=desc.txt
analyseur.y: conflits: 10 décalage/réduction
$ cp analyseur.tab.h calc.h
$ gcc -o Mon_analyseur analyseur.tab.c lex.yy.c -lm -lfl
$ ./Mon_analyseur
7*4
Resultat : 28.00000
```

```
 1 %{
 2 #include "global.h"
 3 #include "calc.h"
 4 #include <stdlib.h>
 5 %}
 6 blancs     [ \t]+
 7 chiffre    [0-9]
 8 entier     {chiffre}+
 9 exposant   [eE][+-]?{entier}
10 reel    {entier}("."{entier})?{exposant}?
11 %%
12 {blancs}   /* On ignore */
13 {reel}     { yylval=atof(yytext);
14         return(NOMBRE); }
15 "+"   return(PLUS);
16 "-"   return(MOINS);
17 "*"   return(FOIS);
18 "/"   return(DIVISE);
19 "^"   return(PUISSANCE);
20 "("   return(PARGAUCHE);
21 ")"   return(PARDROITE);
22 "\n"  return(FIN);
```

Les différents «tokens» sont `PLUS`, `MOINS`, `PARGAUCHE`, *etc*

Ces tokens sont définis dans le fichier «`analyse_syntaxique.tab.h`» qui est généré automatiquement par yacc lors du traitement du fichier de l'analyse sémantique et qui est ensuite renommé en «calc.h».

```
 1 %{
 2   #include "global.h"
 3   #include <stdio.h>
 4 %}
 5 %token  NOMBRE
 6 %token  PLUS  MOINS FOIS  DIVISE  PUISS
 7 %token  PARGAUCHE PARDROITE
 8 %token  FIN
 9 %left PLUS  MOINS
10 %left FOIS  DIVISE
11 %left NEG
12 %right  PUISSANCE
13 %start Input
14 %%
15 Input: /* Vide */
16   | Input Ligne
17   ;
18 Ligne:FIN
19   | Expression FIN {printf("Resultat:%f\n",$1);}
20   ;
21 Expression: NOMBRE      {$$=$1;}
22   | Expression PLUS Expression  {$$=$1+$3;}
23   | Expression MOINS Expression {$$=$1-$3;}
24   | Expression FOIS Expression  {$$=$1*$3;}
25   | Expression DIVISE Expression  {$$=$1/$3;}
26   | MOINS Expression %prec NEG  {$$=-$2;}
27   | Expression PUISS Expression {$$=pow($1,$3);}
28   | PARGAUCHE Expression PARDROITE  { $$=$2;}
29   ;
30 %%
31 int yyerror(char *s) { printf("%s\n",s); }
32 int main(void) {  yyparse(); }
```

*Vous noterez que lors de la génération de l'analyseur syntaxique, il y a un conflit «décalage/réduction» sur la règle 10. Cette règle peut être identifiée à l'aide du contenu du fichier* `desc.txt`.

**Attention**

*Ne pas oublier d'appeler la fonction yyparse() pour déclencher l'analyseur syntaxique (dans la fonction* `main`*).*

**Création d'un Makefile pour automatiser déclarativement la construction de l'analyseur :**

```
 1 CC=gcc
 2 LDFLAGS=-lfl -lm
 3 EXEC_NAME=mon_analyseur
 4 OBJETS=syntaxique.o lexical.o
 5
 6 .y.c:
 7     bison -d $<
 8     mv $*.tab.c $*.c
 9     mv $*.tab.h $*.h
10
11 .l.c:
12     flex $<
13     mv lex.yy.c $*.c
14
15 .c.o:
16     $(CC) -c $<
17
18 all: $(EXEC_NAME)
19
20 $(EXEC_NAME): $(OBJETS)
21     $(CC) -o $@ $^ $(LDFLAGS)
22
23 clean:
24     rm $(OBJETS) $(EXEC_NAME)
```

*tabulation*

**Explications :**

- ligne 2 : on indique les bibliothèques à «linker» pour construire l'exécutable ;
- ligne 4 : les objets composant l'exécutable ;
  ***Remarque*** *: l'objet correspondant à l'analyseur sémantique produit par yacc **doit être placé avant** celui produit par lex pour la définition des tokens.*
- ligne 6 : on indique à «make», comment générer un source «C» à partir du fichier yacc «.y» ;
- ligne 11 : pareil pour un fichier lex «.l» ;
- ligne 15 : pour créer un objet, il faut compiler le «.c» associé ;
- ligne 18 : la règle «all» construit l'exécutable ;
- ligne 20 : l'exécutable, «EXEC_NAME» est composé de tous les objets obtenu à partir de leur compilation individuelle.

**Attention**

Respecter les **décalages** dans le Makefile : vous devez utiliser des **tabulations** sinon votre Makefile ne fonctionnera pas.

**Exemple : reconnaissance de la date où le mois est donné sous forme de texte**

```
1  %{
2  #include <stdlib.h>
3  #include "analyseur_syntaxique.tab.h"
4  char *m;
5  %}
6  mois      Janvier|Fevrier|Mars|Avril|Mai|Juin|
7  Juillet|Aout|Septembre|Octobre|Novembre|Decembre
8  %%
9  {mois}      {   m=(char *)calloc(yyleng+1,
10              sizeof(char));
11                 strcpy(m,yytext);
12                 yylval.texte=m;
13                 return(token_MOIS);
14             }
15 [0-9]{1,2} {   yylval.valint=atoi(yytext);
16                 return(token_JOUR);
17             }
18 [0-9]{4}   {   yylval.valint=atoi(yytext);
19                 return(token_ANNEE);
20             }
21 \,          {   return ',';
22             }
```

Dans cet exemple, lex va retourner 3 tokens différents de deux types (entier ou chaîne).

Par défaut :

- la variable partagée entre Lex & Yacc est de type «int» ;
- la numérotation des tokens commence à 258 pour permettre à Lex de retourner directement la valeur d'un caractère/octet lu (valeur entre 0 et 255).

*On ne va pas utiliser de fichier «global.h» pour définir le type de ces tokens mais directement yacc.*

La déclaration associée dans le fichier yacc «analyseur_syntaxique.y» :

```
1 %union { int valint ;char *texte ;}
2 %token <valint> token_JOUR
3 %token <valint> token_ANNEE
4 %token <texte> token_MOIS
```

Dans le cas où un «terminal» peut prendre plusieurs types, il faudra également définir le type des pour les «non terminaux» de la grammaire :

```
1 %type <texte> date
```

Dans cet exemple, YACC défini automatiquement le fichier «analyseur_syntaxique.tab.h» qui contient :

- la liste des tokens et les valeurs numérique associées que devra retourner Lex ;
- la déclaration et le type de yylval.

Le fichier «syntaxique.y»:

```
 1 %{
 2    #include "syntaxique.h"
 3    #include <stdio.h>
 4    #include <math.h>
 5 %}
 6 %union{  float valeur; }
 7 %token    <valeur> NOMBRE
 8 %type     <valeur> Expression
 9 %token  PLUS  MOINS FOIS  DIVISE  PUISS
10 %token  PARGAUCHE PARDROITE
11 %token  FIN
12 %left PLUS  MOINS
13 %left FOIS  DIVISE
14 %left NEG
15 %right  PUISSANCE
16 %start Input
17 %%
18 Input: /* Vide */
19    | Input Ligne
20    ;
21 Ligne:FIN
22    | Expression FIN {printf("Resultat:%f\n",$1);}
23    ;
24 Expression: NOMBRE      {$$=$1;}
25    | Expression PLUS Expression  {$$=$1+$3;}
26    | Expression MOINS Expression {$$=$1-$3;}
27    | Expression FOIS Expression  {$$=$1*$3;}
28    | Expression DIVISE Expression  {$$=$1/$3;}
29    | MOINS Expression %prec NEG  {$$=-$2;}
30    | Expression PUISS Expression {$$=pow($1,$3);}
31    | PARGAUCHE Expression PARDROITE  { $$=$2;}
32    ;
33 %%
34 int yyerror(char *s) { printf("%s\n",s); }
35 int main(void) {  yyparse(); }
```

Le fichier «lexical.l»:

```
 1 %{
 2    #include "syntaxique.h"
 3    #include <stdlib.h>
 4 %}
 5 blancs     [ \t]+
 6 chiffre    [0-9]
 7 entier     {chiffre}+
 8 exposant   [eE][+-]?{entier}
 9 reel     {entier}("."{entier})?{exposant}?
10 %%
11 {blancs}   /* On ignore */
12 {reel}     {  yylval.valeur = atof(yytext);
13          return(NOMBRE); }
14 "+"    return(PLUS);
15 "-"    return(MOINS);
16 "*"    return(FOIS);
17 "/"    return(DIVISE);
18 "^"    return(PUISSANCE);
19 "("    return(PARGAUCHE);
20 ")"    return(PARDROITE);
21 "\n"   return(FIN);
```

❶⇒on inclus le fichier renommé par le «Makefile» vu précédemment ;

❷⇒on définit l'union ;

❸⇒qui contient le type flottant, que l'on associe aux terminaux (token) et non terminaux (type) ;

❹⇒on utilise l'union dans le fichier lex.

Les fichiers sont disponibles à **https://git.p-fb.net/PeFClic/lex_et_yacc.git**

# YACC : gestion des erreurs

Lorsqu'une erreur est rencontrée lors de l'analyse syntaxique, la fonction `yyerror()` est appelée.
*Cette fonction doit être définie dans la partie «épilogue» du fichier YACC.*

Vous pouvez par exemple affichez un message qui spécifie :
- ▷ le numéro de la ligne ;
- ▷ le dernier terminal lu.

*Si la fonction `yyerror()` n'a pas été défini alors l'analyse s'arrête simplement.*

**Poursuite de l'analyse syntaxique au-delà d'une erreur**

Le terminal `error` peut être utilisé dans la grammaire pour **permettre de dépiler** le contenu de la pile de l'analyseur syntaxique jusqu'à ce terminal.

Exemple : $A \rightarrow error\,\alpha$, où :

- □ $A$ est un non terminal qui correspond à une structure formant un «tout» cohérent dans le langage.

  Exemple :
  - ◇ une expression ;
  - ◇ une instruction ;
  - ◇ une définition de procédure ;
  - ◇ une déclaration de variable, *etc.*

  *Il est sûr que si une erreur se produit, cette erreur va entrainer une erreur complète de la structure où elle se produit.*

- □ $\alpha$ est une séquence, éventuellement vide, de symboles terminaux ou non terminaux.

Exemple de gestion d'erreur :

```
1 Programme:  error POINTVIRGULE
2   | Instruction POINTVIRGULE
3   | Programme Instruction POINTVIRGULE
4   …
5   ;
```

*Ici, l'erreur entraine le passage à l'instruction suivante, $\alpha$ est `POINTVIRGULE` ce qui fait que l'on force l'analyse à reprendre après le «;» suivant.*.

Lorsqu'une erreur se produit l'analyseur est dans un mode spécial, dont il faut sortir pour reprendre le mode normal :

```
1 A : error {
2         yyerrok;
3       }
4 ;
```

On utilise la macro spéciale `yyerrok`.

**Il faut également prévoir de défaire ce qui avait été fait pendant l'analyse qui a abouti à une erreur (désallocation de mémoire, ré-initialisation, *etc.*).**

- **SGML**, «*Standard Generalized Markup Language*» : développé dans les années 70, chez IBM qui deviendra un standard ISO 8879 en 1986.
  But : *gestion de documents techniques de plusieurs milliers de pages*.
- **HTML** : une application de XML (la plus populaire).
  But : *spécialisé dans l'écriture de pages Web et uniquement, il n'est pas extensible ou adaptable à d'autres utilisations.*
- **XML** : février 1998, XML v1.0
  But : *Bénéficier des avantages de SGML en le simplifiant et en enlevant ce qui ne marchait pas (pas utilisé).*
- **XSL**, «*eXtensible Stylesheet Language*» : une application d'XML.
  But : *Permettre la visualisation d'un document XML dans un navigateur.*
  - **XSLT**, «*XSL Transformation*» : *permet de transformer un document XML pour la représentation en Web ou bien dans d'autres contextes.*
  - XSL-FO, «*XSL Formatting Object*» : *permet de décrire la composition des pages pour l'affichage des pages en Web ou à l'impression.*
- **CSS**, «*Cascading Style Sheet*» : utilisé pour la représentation des documents HTML
  But :*Permettre la représentation de documents XML comme HTML à partir de la v2.*

- XLL, «*eXtensible Link Language*» :
  But : *permettre de définir des modèles de liaisons pour relier des documents XML dans un réseau HyperTexte*.
  - XLink : *pour décrire la relation*
  - XPointer : *pour identifier une partie du document XML*

  Mais…
  - **XPath** : *normaliser les définitions de XPointer et celles utilisées dans XSLT pour identifier une partie du document XML.*
  - XInclude : *évolution de XLink pour la définition de liens entre documents et fragments de documents.*
- **DOM**, «*Document Object Model*» : arborescence objet
  But : *Définir une interface standardisée pour l'accès à un contenu XML depuis un environnement de programmation (Java, JavaScript, C++).*
- SAX, «*Simple API for XML*» :
  But : disposer d'une API commune pour la commande de parseur XML.
- **XML Schema** : Le **DTD** n'exprime pas de typage de données ce qui est un inconvénient pour la gestion de données structurées.
  But : *permettre de décrire un modèle de document XML en XML, très complexe d'utilisation*

- XML Encryption : l'échange d'un document XML peut être sécurisé au travers du protocole d'échange utilisé.
  But : *Sécuriser le contenu du document : confidentialité & signature numérique. Le format XML Canonical permet de diminuer les différences entre documents (suppression des espaces inutiles, normalisation des guillemets).*
- XML 1.1 : XML 1.0 est déjà basé sur unicode 2.0
  But : *Tenir compte des ajouts dans Unicode pour le support des langages : mongol, birman, cambodgien*
- des formats XML dédiés : SOAP, SVG, XHTML, MathML, XForms, etc.

Dans un document HTML, on trouve mélangés le contenu et sa présentation :

- polices de caractères ;
- tableaux ;
- titres ;
- paragraphes ;
- images, liens hypertextes, etc.

L'ensemble est orienté homme-machine, c-à-d qu'une personne visualise ce document sur un écran d'ordinateur.

Un fichier XML parait identique à un fichier HTML, mais il est plus exigeant car il contient les données et leur structure logique.

```
<html>
<p>
<b>Campus de la Borie</b>
<br /> 123 avenue Albert Thomas
<br /> 87060 Limoges CEDEX
</p>
</html>
```

HTML n'offre qu'un jeu limité de balises (tag) auxquelles il ne sera possible que d'affecter des effets de mise en forme, par exemple grâce à une feuille de style CSS.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Test SYSTEM "Test.dtd"> <Test>
<nom>Campus de la Borie</nom>
<adresse>
<rue>123 avenue Albert Thomas</rue>
<code-postal>87060</code-postal>
<localite>Limoges CEDEX</localite> </adresse>
</Test>
```

XML permet de définir ses propres balises (en gras) et donc de leur donner du sens. Il sera ainsi possible de leur affecter non seulement des effets de mise en forme, mais aussi de leur appliquer des traitements logiques complexes.

**Structure d'un document XML**

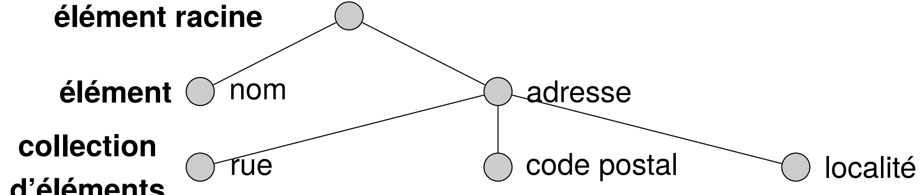
Un document XML est représenté comme un **arbre d'éléments**.

Il intègre la notion de «lien» entre documents.

Les éléments XML peuvent posséder des **attributs** :

▷ **l'attribut** est un couple (nom, valeur) associé à un **élément** et précisant ses caractéristiques.

**Structure logique d'un document XML**



**Intérêt des langages utilisant des balises**

L'introduction du format HTML et sa très grande diffusion ont relancé l'intérêt pour les documents structurés à l'aide de balises :

- indiquées de manière textuelle;
- intégrées au document lui-même.

*Elles visent à séparer la structure du contenu du document.*

**Constat sur HTML**

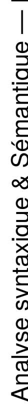
- ⋆ très grand succès ;
- ⋆ bien adapté à la diffusion d'informations ;

*mais*

- ▷ pas extensible ;
- ▷ peu structuré ;
- ▷ peu d'outils de validation des données d'un document.

Les limitations d'HTML se traduisent par :

- les utilisateurs ne peuvent pas définir leurs propres balises pour enrichir leurs documents ;
- les documents sont «plats», les enrichissements typographiques sont restreints ;
- il existe des problèmes lors d'échanges d'informations devant respecter une certaine organisation, c-à-d que l'information transmise ne peut être suffisamment structurée.



**Buts de XML**

XML permet de définir à la fois la structure logique d'un document et la façon dont il va être affiché :

- gérer des documents **mieux et plus** structurés qu'avec HTLM ;
- permettre à ces documents d'être **traités**, **indexés**, **fragmentés**, **manipulés** plus facilement qu'avec HTML

XML est une «*application*» de SGML, «*Standard Generalized Markup Language*», plus simple, qui peut être mise en œuvre pour le Web et dans des applications utilisant peu de ressources.

**Présentation d’un document XML**

Pour présenter un document XML sur un support quelconque, il doit être associé à une feuille de style XSL, par type de format en sortie (XML, HTML, WML, etc) : offrir à l’utilisateur plusieurs types d’accès: papier, écran, CDROM, SmartPhone ou même en braille, alors que la présentation d’un document HTML ne peut être visualisé que par un navigateur Web.

**Exemple : la feuille de style «test.xsl»** :

```
1 <xsl:stylesheet version="1.0">
2 <xsl:output method="xml">
3 <xsl:template match="/Test">
4 <font face="times" size="12"> <xsl:apply-templates/> </font> </xsl:template>
5 <xsl:template match="nom"> <font size="+2" style="bold"> </font> </xsl:template>
6 </xsl:stylesheet>
```

*Dans cette feuille de style, on associe un style de caractère à chaque balise définissant un type de données :*

– *la police de caractères par défaut est "Times 12"*
– *le nom doit apparaître en "Times 14" et en gras.*

L'interêt de XML, par rapport à HTML, est la séparation du contenu de la structure des données.
La description des données peut être placée dans un fichier séparé qui donne le DTD, «*Document Type Description*».

Le DTD sert :
- ⋆ à décrire toutes les balises du document XML,
- ⋆ à définir les relations entre les éléments,
- ⋆ à valider un document XML car il contient les règles à suivre pour respecter sa structure.

Le DTD sert à définir des **classes de documents**.

Exemple «test.dtd»

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!ELEMENT nom (#PCDATA)>
3 <!ELEMENT adresse (rue, code-postal, localite)>
4 <!ELEMENT rue (#PCDATA)>
5 <!ELEMENT code-postal (#PCDATA)>
6 <!ELEMENT localite (#PCDATA)>
```

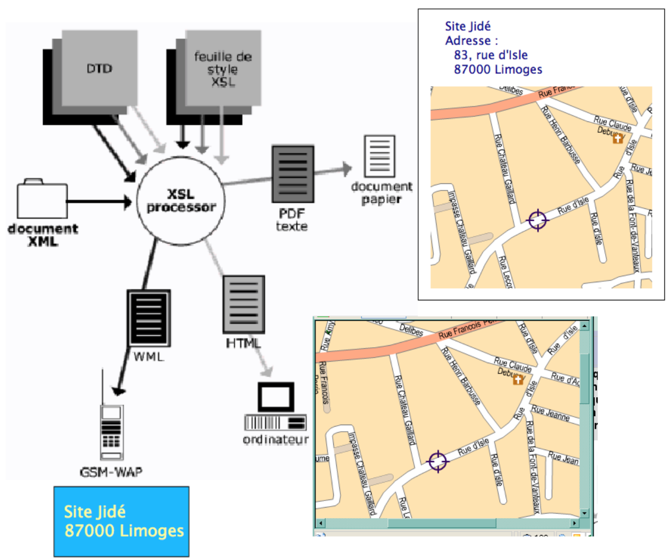
L'élément adresse comprend trois éléments distincts: la rue, le code postal et la localité.
La syntaxe utilisée pour la DTD n'est pas la même que celle utilisée pour les données et les feuilles de style.

**Génération des documents cibles**

On utilise :

– une feuille de style spécifique pour chaque support (à chaque type de transformation effectuée à partir du document source XML)

– éventuellement, en tenant compte d'une DTD spécifique à chaque support.



# DTD : notion de classe de document

XML utilise la notion de classes de documents ou de type de document au travers d'un DTD.

Ce DTD est un fichier qui décrit une structure :

- description des différentes balises utilisables ;
- description des imbrications possibles de ces balises.

Ainsi, un document :

- doit être construit suivant les directives de XML ;
- peut suivre le format décrit dans un DTD particulier.

Un document est :

▷ «bien formé», c-à-d syntaxiquement correct s'il suit les règles de XML

▷ «valide», s'il se conforme à la DTD associée.

Il est plus proche de SGML que de HTML :

- HTML est une application de SGML;
- XML est un sous-ensemble des fonctionalités de SGML.

*La puissance de XML réside dans sa capacité à pouvoir décrire n'importe quel domaine de données grâce à son extensibilité.*

*Il suffit dans ce cas là de définir :*

- *le vocabulaire (les balises et leurs attributs) ;*
- *la structure des imbrications de ce vocabulaire ;*
- *la syntaxe de ces données ;*
- *un DTD particulier pour le domaine de données voulu.*

**Les règles pour obtenir un document valide**

Si le document suit ces règles on dit qu'il est «bien formé» :

⋆ respecter la casse des balises `<societe>` ≠ `<Societe>`

⋆ toujours mettre une balise de fin `<p>` … `</p>`

⋆ mettre les valeurs des attributs entre guillemets `<balise attr="valeur">`

⋆ ne pas entrelacer les ouvertures et fins de balises différentes

  ◇ `<B>` … `<I>` … `</B>` … `</I>` est interdit

  ◇ `<B>` … `<I>` … `</I>` … `</B>` est autorisé

  Il doit correspondre à un arbre correct.

Il est possible de vérifier le caractère bien formé avec la commande shell `xmllint` :

```
$ xmllint document_annuaire.xml
document_annuaire.xml:5: parser error : Opening and ending tag mismatch: companyname line
4 and response
</response></companyname>
           ^
```

et la validité d'un document XML :

```
$ xmllint --valid --noout mon_fichier.xml --dtdvalid ma_dtd.dtd
```

- **acceptation** en tant que standard pour la description de données par les principaux acteurs du marché (Adobe, Microsoft, Apple…).
- **format utilisé** aussi bien dans des logiciels de mise en page comme «Quark Xpress» ou «Indesign» d'Adobe, et qui permet, par exemple, dans ces logiciels de mettre en rapport un document avec le contenu d'une base de donnée :
  - les données sont décrites au format XML ;
  - la mise en page de ces données s'appuie sur XML.

  *On obtient alors un document mis en page dont le contenu peut être dynamique !*
- **lisible** : aucune connaissance ne doit théoriquement être nécessaire pour comprendre un contenu d'un document XML ;
- structure **arborescente** : ce qui permet de modéliser une majorité de problèmes informatiques ;
- **universel** et **portable** : les différents jeux de caractères sont pris en compte ;
- **facilement échangé** : facilement distribué par n'importe quel protocole à même de transporter du texte, comme HTTP ;
- **facilement utilisable** dans une application : un document XML est utilisable par toute application pourvue d'un analyseur syntaxique, un «*parser*» de code XML ;
- **extensible** : un document XML doit pouvoir être conçu pour organiser les données de tous domaines d'applications

Le document XML contient différentes parties :

- □ déclaration de la version XML utilisée ;
- □ type du document utilisé ;
- □ corps du document :

```
1 <?xml version="1.0" ?>
2 <!DOCTYPE individu SYSTEM "individu.dtd">
3 <body>
4   <individu>
5   <h1><nom>Binks</nom></h1>
6     <prenom>JarJar</prenom>
7     <img src="photo.gif"></img>
8   </individu>
9 </body>
```

Sur l'exemple, on voit qu'il y a des **balises** :

- ▷ habituelles dans HTML (`<h1>`, `<img>`, ...)
- ▷ spécifiques à l'application (ici `<individu>`, `<nom>`, `<prenom>`)

*Ces balises spécifiques fournissent la structure d'une donnée où, chaque individu est composé d'un nom et d'un prénom.*

Les balises peuvent avoir des **attributs** : `<individu id="125">`

Ces balises permettent de stocker les informations comme dans une table de BD :

```
1 <individu>
2     <nom>Padme</nom>
3     <prenom>Amidala</prenom>
4 </individu>
5 <individu>
6     <nom>Obi Wan</nom>
7     <prenom>Kenobi</prenom>
8 </individu>
```

est équivalent à la table suivante :

| Nom | Prénom |
|---|---|
| Padme | Amidala |
| Obi Wan | Kenobi |

Les données en XML peuvent également se représenter sous la forme d’attributs :

```
<individu nom="Padme" prenom="Amidala"></individu>
```

*En général, les attributs sont utilisés pour les informations non affichables dans les navigateurs (exemple : la valeur* `id` *qui est la clé d’enregistrement de l’individu dans la base de donnée).*

# XML : Syntaxe des éléments constitutifs

Pour les identifiants des balises :

- les noms peuvent contenir des lettres, des chiffres ;
- ils peuvent contenir des caractères accentués ;
- ils peuvent contenir les caractères «_», «.», «-» ;
- ils ne doivent pas contenir : «?», «'», «$», «^», «;», «%» ;
- ils peuvent contenir le «:» pour l'utilisation d'espace de nom ;
- les noms ne peuvent débuter par un nombre ou un signe de ponctuation ;
- les noms ne peuvent commencer par les lettres xml (ou XML ou Xml...) ;
- les noms ne peuvent contenir des espaces ;
- la longueur des noms est libre mais on conseille de rester raisonnable ;

Pour les attributs :

▷ ils sont entourés de guillemets ou d'apostrophes ;

▷ il n'existe qu'un seul attribut avec un nom donné (si il y a un besoin, il faut utiliser des éléments et non des attributs).

Pour l'«encoding» :

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Il doit se trouver en première ligne du document.

*Actuellement, on préfère utiliser l'unicode, ce qui permet d'être dans l'encodage naturel d'XML.*

# 9 DTD : Définition d'une classe de documents

Les définitions de classes de document peuvent être :

- **internes**, c-à-d intégrées au document lui-même ;
- **externes**, c-à-d chargées à partir d'un fichier auquel on fait référence à l'aide d'une URL.

**Définition interne**

La balise `<!DOCTYPE >` fournit la structure du document :

```
<!DOCTYPE nom [ … ] >
```

Elle crée un type de document et se compose de deux parties :

1. une déclaration de constantes (ou entités) ;
2. une déclaration des éléments du document.

Exemple :

```
1 <?xml version="1.0" ?>
2   <!DOCTYPE simple  [
3     <!ELEMENT mondocument #PCDATA>
4         ] >
5 <mondocument>
6 Cette partie apparaît dans le navigateur
7 </mondocument>
```

**Définition externe**

Elle se fait à l'aide de la balise : `<!DOCTYPE nom statut url >` où :

- ▷ nom : du type de document
- ▷ statut :
  - ◇ `PUBLIC` correspond à un type de document public ;
  - ◇ `SYSTEM` correspond à un type de document local à une organisation.

Exemple :

```
<!DOCTYPE manual PUBLIC "-//DTD manual//EN" "http://www.unilim.fr/manual.dtd">

<!DOCTYPE individu SYSTEM "individu.dtd">
```

# DTD : La notion d'entités



La balise <!ENTITY nom valeur > définit une constante utilisable dans le document.
Lorsque l'analyseur du document XML lit une entité, il la remplace par sa substitution.

Exemple : <!ENTITY euro "6.55957 F">, où :

- nom_de_l'entité = euro
- substitution_de_l'entité = 6.55957 F

Une entité est utilisée sous la forme «&nom_de_l'entité;».
Exemple : la valeur d'un euro est €.

**Les entités paramétrées**

Il est possible de définir également des entités «paramétrées» :

```
<!ENTITY % nom url >
```

Exemple : <! ENTITY %compagnies "http://localhost/compagnies.den">
L'emploi de %compagnies; avec dans le fichier compagnies.den les définitions suivantes :

```
<!ENTITY IBM "International Business Machines" >
<!ENTITY ATT "American Telephone and Telegraph" >
```

permet d'utiliser les constantes &IBM; et &ATT; dans un document
*Avantage : on regroupe toutes les entités «similaires» dans un même fichier.*

`<!ELEMENT nom (type) >`

Exemple : `<!ELEMENT mondocument (#PCDATA)>`, où :

- nom : définition d'un élément, c-à-d le nom de la balise (ici `mondocument`) ;
- type : type de l'élément.

**Types de données disponibles dans XML**

- `#PCDATA`: «Parsed Character DATA», ce sont des données interprétées qui peuvent être tout caractère sauf `< > &` encodés respectivement `<` `>` et `&` ;
  Il est possible d'utilisé dans ces caractères des entités.
- `ANY` : tout ce que l'on veut comme élément déjà déclarés ;
- `EMPTY` : rien !

Exemple : `<!ELEMENT BR EMPTY >` veut dire que la balise <BR> doit s'utiliser de la façon suivante : `<BR></BR>` ou `<BR/>`

**Les types de données définis par l'utilisateur**

Exemple : `<!ELEMENT individu nom >`

Cela veut dire que la balise `<individu>` ne peut contenir qu'une définition de `nom`.

L'utilisation :

- `<individu><nom>Padme</nom></individu>` est correcte ;
- `<individu>hello</individu>` est mauvaise.

Il est possible d'utiliser plusieurs types à la suite les uns des autres en utilisant des opérateurs pour spécifier des «enchaînements» de type :

```
|   alternative
,   séquence
?   optionnel
*   0 ou n
+   1 ou n
( ) regroupement
```

Exemple :

```
<!ELEMENT nom (#PCDATA) >
<!ELEMENT prenom (#PCDATA) >
<!ELEMENT individu (nom,prenom*) >
<!ELEMENT livre (titre,auteur,remerciements?,chapitre+) >
```

**Type mixte**

Il est possible également de définir un type mixte :

```
<!ELEMENT définition (#PCDATA | terme)*>
```

Ce qui permet d'avoir du texte, et dans ce texte une ou plusieurs balises `<terme>`.

# DTD : les attributs d'éléments

```
<!ATTLIST nom_de_l'element
nom_de_l'attribut type propriété
       ...     >
```

Le lien se fait par le nom de l'élément avec la liste d'attributs.

Exemple :

```
<!ELEMENT individu (nom,prenom+) >
<!ATTLIST individu
        noss ID #REQUIRED
        adresse CDATA #IMPLIED
        situation (celibataire|marie|divorce) "celibataire"
        pays CDATA #FIXED "France">
```

**Types reconnus**

- ▷ `CDATA` : chaîne de caractères quelconques non interprétée ;
- ▷ `NMTOKEN` : chaîne de caractères composée de lettres, chiffres, et de « `. _ - :` » ;
- ▷ `NMTOKENS` : liste de `NMTOKEN` séparée par des espaces ;

```
<! ATTLIST concert dates NMTOKENS #REQUIRED>
<concert dates="08-27-2010 09-15-2010">
```

;

**Types reconnus – Suite**

▷ `ID` : la valeur de l'attribut doit être unique au sein du document, sinon l'analyseur syntaxique XML signale une erreur ;

▷ `IDREF` : la valeur de l'attribut doit faire référence à une valeur d'ID existante dans le document, sinon l'analyseur syntaxique XML signale une erreur ;

▷ `IDEREFS` : une liste de `ID` séparés par des espaces ;

▷ énumération : la valeur de l'attribut doit correspondre à l'une des valeurs énumérées.

```
<!ELEMENT date EMPTY>
<!ATTLIST date mois (Janvier | Février … | Décembre) #REQUIRED>
<date mois="Janvier"/>
```

**Propriétés reconnues**

▷ `#REQUIRED` : attribut obligatoire, sinon l'analyseur syntaxique XML signale une erreur ;

▷ `#IMPLIED` : attribut facultatif ;

▷ `#FIXED` : l'attribut a une valeur fixe (donnée par défaut) ;

```
<!ATTLIST projet version FIXED "1.0">
```

▷ valeur par défaut donnée entre guillemets

```
<!ATTLIST page_web protocole NMTOKEN "http">
```

# DTD : la notion d'espace de nom

Elle permet de grouper des éléments et des attributs ensembles.

Exemple :

```
1 <?xml version='1.0' encoding='UTF-8' ?>
2 <document>
3 <nom>Rapport d'activité</nom>
4 <graph:graphique xmlns:graph= 'http://maths.com/dtds/graphml'>
5 <graph:nom>Courbe d'amortissement</graph:nom>
6 ...
7 </graph:graphique>
8 </document>
```

Un espace de nom est défini par `xmlns:préfixe=URL`, où `URL` est une valeur unique garantissant l'unicité de l'espace de nom.

*Ici, le préfixe `graph` a été défini comme accès à l'espace de nom défini dans le DTD `graphml`.*

L'intérêt des espaces de noms est de faire cohabiter dans un même document des parties différentes pouvant être traitées séparèments de manière automatique.

**Les commentaires**

Ils sont compris entre les balises `<!--` et `-->`.

Exemple : `<!-- Ceci est un commentaire XML -->`

**Les sections CDATA**

Permettent de définir des zones de textes non interprétées par les navigateurs XML

Comprises entre les balises `<![CDATA[` et `]]>`

Exemple : `<![CDATA[ ici <h1> n'est pas une balise mais du texte ]]>`

**Instructions de traitement**

`<?Programme` et `?>` désigne le programme chargé d'interprété les instructions.

Exemple : `<?php … ?>`

**Résumé XML**

Il y a seulement 4 balises principales :

- `<?xml >` : version XML utilisée ;
- `<!DOCTYPE >` : classe de documents ;
- `<!ELEMENT >` : les balises du document ;
- `<!ATTLIST >` : leurs attributs.

Ces 4 balises sont suffisantes pour construire «*n'importe quelle*» classe de documents !

# 10 CSS : présentation rapide

Les «*Cascading Style Sheets*» sépare la structure de sa mise en page.

Il existe deux concepts essentiels : les **sélecteurs** et les **propriétés**.

Les propriétés possèdent différentes valeurs :

| | |
|---|---|
| `color` | `red yellow rgb(212 120 20)` |
| `font-style` | `normal italics oblique` |
| `font-size` | `12pt larger 150% 1.5em` |
| `text-align` | `left right center justify` |
| `line-height` | `normal 1.2em 120%` |
| `display` | `block inline list-item none` |

Un **sélecteur** est une liste d'étiquette.

Pour chaque sélecteur, on associe des valeurs à certaines propriétés :

```
b {color: red; font-size: 12pt}
i {color: green}
```

Des sélecteurs plus long permettent de s'adapter au contexte :

```
table b {color: red; font-size: 12pt}
form b {color: yellow; font-size: 12pt}
i {color: green}
```

*C'est le sélecteur le plus «spécifique» dont la valeur s'applique*.

Exemples :

```
<head>
<style type="text/css">
b {color: red;}
b b {color: blue;}
b.foo {color: green;}
b b.foo {color: yellow;}
b.bar {color: maroon;}
</style>
<title>CSS Test</title> </head>
```

```
<body>
<b class=foo>Hey!</b>
<b>Wow!
<b>Amazing!</b>
<b class="foo">Impressive!</b>
<b class="bar">k00l!</b>
<i>Fantastic!</i>
</b>
</body>
```

**Hey! Wow! Amazing! Impressive! k00l!** ***Fantastic!***

```
h1 { color: #888; font: 50px/50px "Impact"; text-align: center; }
ul { list-style-type: square; } em { font-style: italic; font-weight: bold; }
```

```
<html> <head><title>Phone Numbers</title>
<link href="style.css" rel="stylesheet" type="text/css"> </head>
<body> … </body>
</html>
```

Le fichier «annuaire.xml»

```
1 <?xml version="1.0" encoding="UTF-8"
2   standalone="yes"?>
3 <?xml-stylesheet href="annuaire.css"
4   type="text/css"?>
5 <response>
6 <companyname>Ma compagnie à moi</companyname>
7 <telephone>05 55 43 69 83</telephone>
8 </response>
```

Le résultat dans un navigateur :

**Ma compagnie à moi**

*05 55 43 69 83*

Le fichier «annuaire.css»

```
1  <style type="text/css">
2  response{}
3  telephone {
4  display: block;
5  font-size: 11pt ;
6  font-style: italic;
7  font-family: arial ;
8  padding-left: 10px;
9  color: red;
10 }
11 companyname {
12 display: block;
13 width: 250px;
14 font-size: 16pt ;
15 font-family: arial ;
16 font-weight: bold;
17 background-color: teal;
18 color: white;
19 padding-left: 10px;
20 }
21 </style>
```

# 11 XSLT : aller plus loin que les CSS…

Ce **langage** permet de définir des «transformations» à appliquer sur un document XML auquel il est appliqué.

Une **feuille de style** XSLT :

⋆ est définie par :

```
1 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
2 …
3 </xsl:stylesheet>
```

⋆ contient des règles de «templates» ;

⋆ commence le traitement sur le noeud racine du fichier XML.

Une **règle de template** :

◇ est définie par :

```
1 <xsl:template match="…"> …
2 </xsl:template>
```

◇ XLST trouve la règle de template qui correspond au noeud courant en sélectionnant le plus spécifique ;

◇ évalue le corps du template.

On utilise **XPath** pour :

- spécifier les motifs pour la sélection des règles de template ;
- sélectionner les noeuds pour les traiter *syntaxe proche des chemins fichier Unix.* ;
- créer des conditions booléennes ;
- générer du texte pour le document en sortie.

Il est possible de :

- **appliquer** un template : `<xsl:apply-templates/>` (pour le noeud racine ce n'est pas la peine, du moment qu'un template lui correspond indiqué par «/» ou bien simplement son nom) ;
- *à l'intérieur d'un template* :
  - récupérer la valeur d'une **balise** : `<xsl:value-of select="nom_balise"/>`
  - récupérer la valeur d'un **attribut** : `<xsl:value-of select="@nom_attribut"/>`
  - parcourir les **sous-noeuds** : `<xsl:for-each select="nom_sous_noeud">`
  - parcourir les **attributs** : `<xsl:for-each select="@nom_attribut">`

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3  <xsl:template match="/">
4  <html><head><title>Livres</title></head>
5  <body><xsl:for-each select="booklist/book">
6  <p><b><xsl:value-of select="title"/></b>
7  <i><xsl:value-of select="author"/></i>
8  <a href="http://www.amazon.com/exec/obidos/ASIN/{isbn}/">Details</a></p>
9  </xsl:for-each>
10 </body></html>
11 </xsl:template>
12 </xsl:stylesheet>
```

*On peut utiliser la notation «{isbn}» permet de récupérer la valeur d'une balise directement dans une chaîne de caractère (ici, pour définir un lien).*

Toujours à l'intérieur d'un template et dans le cadre d'un parcours avec `<xsl:for-each …>` :

- faire des **tests de conditions** :
  - directement après le `for-each`:
    `<xsl:if test="not(nom='Mon livre secret')">`.
    *Cela permet de traiter ou non un noeud parcouru par le* `for-each`
  - pour tester la présence d'un élément (balise ou attribut) :
    `<xsl:if test="dédicace"><li>dédicace: <xsl:value-of select="dédicace"/> </li></xsl:if>`
- des **tris** :
  `<xsl:sort select="critere" data-type="number"/>`
  *À mettre juste après le* `for-each`
- des **sélections** :

```
1 <xsl:choose>
2 <xsl:when test="@type='essence'"><p>Moteur essence</p></xsl:when>
3 <xsl:when test="@type='eau'"><p>Vous vous moquez !</p></xsl:when>
4 <xsl:otherwise><p>À pédales ?</p></xsl:otherwise>
5 </xsl:choose>
```

# XSLT : un exemple de liste de livres

Exemple de fichier XML :

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="test_xsl.xsl"
3    type="text/xsl" ?>
4  <booklist title="Science Fiction">
5  <book>
6    <title>The Naked Sun</title>
7    <author>Isaac Asimov</author>
8    <isbn>0553293397</isbn>
9  </book>
10 <book>
11   <title>Foundation's Triumph</title>
12   <author>David Brin</author>
13   <isbn>0061056391</isbn>
14 </book>
15 <book>
16   <title>Snow Crash</title>
17   <author>Neal Stephenson</author>
18   <isbn>0553380958</isbn>
19 </book>
20 </booklist>
```

*En ligne 2, on indique dans l'entête du document XML, la feuille de style à appliquer.*

Le résultat obtenu avec la feuille de style du transparent suivant :

**The Naked Sun***Isaac Asimov*Details

**Foundation's Triumph***David Brin*Details

**Snow Crash***Neal Stephenson*Details

Le fichier xslt associé :

```
1 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
2 <xsl:output method="html" encoding="UTF-8"/>
3 <xsl:template match="booklist">
4 <html>
5 <head><title><xsl:value-of select="@title"/></title></head>
6 <body><xsl:apply-templates/></body>
7 </html>
8 </xsl:template>
9
10 <xsl:template match="book">
11 <p>
12 <b><xsl:value-of select="title"/></b>
13 <i><xsl:value-of select="author"/></i>
14 <a href="http://www.amazon.com/exec/obidos/ASIN/{isbn}/">Details</a> </p>
15 </xsl:template>
16 </xsl:stylesheet>
```

**Explications**

▷ ligne 3, on définit le «template» principal sur la racine ;

▷ ligne 5, le «@» désigne une valeur d'attribut ;

▷ ligne 6, on applique les «templates» définis plus bas ;

▷ ligne 10, on définit un «template» pour l'affichage d'un livre ;

▷ ligne 14, on obtient dans la chaine du lien, le contenu de la balise «isbn».

# XPath

XPath permet d'accéder à un élément particulier de l'arborescence d'un document XML, il permet de «naviguer» dans un «chemin» à travers l'arborescence.

XPath reconnait différents types de nœuds :
- le document XML entier ;
- la racine, ou «root» et tous les éléments contenus dedans ;
- les éléments, attributs, et le texte ;

*Il est utilisé en particulier dans XSL.*

```
 1 <root>
 2   <one>
 3     <two/>
 4     <two/>
 5     <three>
 6       <two/>
 7       <lundi meteo="pluie"?/>
 8       <jeudi meteo="neige"/>
 9     </three>
10   </one>
11 </root>
```

- `/root/one` : sélectionner l'élément `<one>`
- `one/two` : saut relatif d'un élement `<one>` à un élément `<two>`
- `/root/one/two` : sélectionner l'élém. *enfant* `<two>`
- `root/one/three/lundi/@meteo` : un attribut est décrit en utilisant le symbole `@`
- `root/one/three/lundi[@meteo='pluie']` : un attribut peut être trouver à l'aide d'une comparaison portant sur sa valeur indiquée entre « `[]` »

Il est possible de sélectionner l'ensemble des éléments dans un document sans tenir compte de leur localisation dans l'arborescence :
- ▷ `//@meteo` : sélectionner tous les attributs «`meteo`» ;
- ▷ `//two` : sélectionner tous les éléments «`<two>`».

Il est possible de sélectionner un ensemble d'éléments en utilisant le *wildcard* « `*` » :

▷ `/root/one/three/*` : sélectionne les éléments à l'intérieur de `<three>` ;

▷ `//*` :sélectionne tous les éléments du document.

Le caractère « `.` » permet de se référer à l'élément courant, les caractères « `..` » permet de se référer à l'élément parent :

▷ `/root/.` : se référe à la racine ;

▷ `/root/one/three/..` : `<one>` est le parent de l'élément «`three`».

Si un document possède une série d'éléments identique dans une liste, on peut utiliser une valeur d'index entre « `[ ]` » pour sélectionner un élément particulier :

▷ `/root/one/two[1]` : le premier élement «`<two>`» ;

▷ `/root/one/two[2]` : le second ;

▷ `/root/one/two[last()]` : le dernier.

**Attention**

La numérotation des éléments commence à 1.

**Exemple**

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
href="xsltext.xsl"?>
<test>
<title size="7">Exemple XSLT</title>
<line>50%</line>
<para>ceci est un paragraphe</para>
<text>Texte 1 </text>
<text>Texte 2 </text>
<text>Texte 3 </text>
<text>Texte 4 </text>
</test>
```

Le résultat :

Exemple XSLT

Texte 1
Texte 2
Texte 3
Texte 4

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html><body><center>
<font color="ff0000">
<xsl:attribute name="size">
<xsl:value-of select="test/title/@size" />
</xsl:attribute>
<xsl:value-of select="test/title"/>
</font> <br/> <font size="2" color="0000ff">
<xsl:value-of select="test/text[1]"/> </font>
<br/> <font size="3" color="008800">
<xsl:value-of select="test/text[2]"/> </font>
<br/> <font size="4" color="ff00ff">
<xsl:value-of select="test/text[3]"/> </font>
<br/> <font size="5" color="0088ff">
<xsl:value-of select="test/text[last()]"/> </font>
<br/> <hr>
<xsl:attribute name="width">
<xsl:value-of select="test/line" /> </xsl:attribute>
</hr> </center> </body> </html>
</xsl:template> </xsl:stylesheet>
```

```
<library>
<book>
    <author-ref>T.Pratchett</author-ref>
    <title>The Colour of Magic</title>
    <year>1983</year>
</book>

<author id="T.Pratchett">
    <last-name>Pratchett</last-name>
    <first-name>Terry</first-name>
</author>
</library>
```

On a définit deux parties distinctes :

- <book>...</book> : où l'on donne le nom du livre et son année d'édition ;
- <author>...</author> : où l'on détaille le nom complet de l'auteur.

*Le fichier XSLT utilise la notion de `key` pour faire un lien entre ces deux parties à la manière d'une base de données relationnelles : c'est la valeur unique de `<author-ref>` qui fait le lien entre le livre et son auteur.*

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:key name="lien" match="author" use="@id" />

<xsl:template match="/library">
<html>
  <head>
  </head>
  <body>
    <h2>Bibliothèque</h2>
    <table>
      <thead>
        <tr>
          <th>Titre</th>
          <th>Année</th>
          <th>Auteur(s)</th>
        </tr>
      </thead>
      <xsl:for-each select="book">
      <tr>
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="year"/></td>
        <td>
          <xsl:value-of select="key('lien', author-
ref)/first-name"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="key('lien', author-
ref)/last-name"/>
        </td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

*Un exemple de guide de voyage : donner son avis sur des villes visitées*

```
 1 <?xml version="1.0"?>
 2 <?xml-stylesheet type="text/xsl" href="uk.xsl"?>
 3 <city> <name location="uk">
 4 <brief>Liverpool is in the North of England.</brief>
 5 <general>Three universities, two cathedrals, one airport </general>
 6 <fun>Bars, clubs, the FACT center</fun>
 7 <places>Albert Dock</places>
 8 <places>Speke Hall</places>
 9 <places>The Cathedrals</places>
10 <comments>An interesting place - Jim</comments>
11 <comments>Scary! - Beaker</comments>
12 <comments>Good fun if you like to party - Sally</comments>
13 </name>
14 </city>
```

On voudrait :

- tenir compte de la localisation du document indiquée par «`location`» ;
- afficher la description «`<brief>`», «`<general>`» et «`<fun>`» ;
- afficher chaque commentaire indiqué par des «`<comments>`» ;
- en **fonction du nombre de commentaires**, afficher si la destination est populaire ou non.

**Début du fichier XSL**

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3  <xsl:template match="/">
4  <html><body>
5  <xsl:if test="/name[@location='uk']">
6    <font size="2"><i>UK version </i></font>
7  </xsl:if>
8  <hr width="60%" align="left"/>
9  <?-- HTML table -->
10 <table border="0">
11 <tr><th>Category</th><th>Info</th></tr>
12 <tr><td><font color="0000ff">Brief Description:</font>
13   </td> <td><xsl:value-of select="/brief"/></td>
14 </tr>
15 <tr><td><font color="0000ff">General info:</font></td>
16   <td><xsl:value-of select="/general"/></td>
17 </tr>
18 <tr><td><font color="0000ff">Fun bits:</font></td>
19   <td><xsl:value-of select="/fun"/></td>
20 </tr>
21 <tr><td><font color="0000ff">Places to Visit:</font>
22 </td>
23 </tr>
```

❶⇒teste si cela s'applique sur la version anglaise «uk»

❷⇒sélection du contenu de la balise brief

❸⇒sélection du contenu de la balise general

❹⇒sélection du contenu de la balise fun

*Le résultat :*

*UK version*

| Category | Info |
|---|---|
| Brief Description: | Liverpool is in the North of England. |
| General info: | Three universities, two cathedrals, one airport |
| Fun bits: | Bars, clubs, the FACT center |
| Places to Visit: | |
| | Albert Dock |
| | Speke Hall |
| | The Cathedrals |
| Your Comments: | |
| | An interesting place - Jim |
| | Scary! - Beaker |
| | Good fun if you like to party - Sally |

Page popularity: High

**La suite du fichier XSL**

```
22 <xsl:for-each select="city/name/places">
23 <xsl:sort select="places"
24       order="descending" data-type="text"/>
25   <tr><td></td>
26     <td><xsl:value-of select="."/> </td>
27   </tr>
28 </xsl:for-each>
29 <tr><td><font color="0000ff">Your Comments:</font></td>
30 </tr>
31 <xsl:for-each select="city/name/comments">
32 <xsl:sort select="comments" order="ascending"/>
33   <tr><td></td> <td> <xsl:value-of select="."/> </td></tr>
34 </xsl:for-each> </table> <hr width="60%" align="left"/>
35 <xsl:choose>
36   <xsl:when test="count(/city/name/comments)>2">
37     <font size="2"> Page popularity: High </font>
38   </xsl:when>
39   <xsl:otherwise> <font size="2"> Page popularity: Average
40 </font>
41   </xsl:otherwise>
42 </xsl:choose>
43 </body></html>
44 </xsl:template>
45 </xsl:stylesheet>
```

❶⇒parcours des différentes «places»
❷⇒récupération de la valeur de la balise courante
❸⇒choix entre différentes conditions
❹⇒permet de compter les éléments et de prendre une décision booléenne

# DTD : les limitations



Soit un DTD pour définir des recettes :

```
 1 <!ELEMENT collection (description,recette*)>
 2 <!ELEMENT description (#PCDATA)>
 3 <!ELEMENT recette (titre,date,ingredient*,preparation,remarque?,
 4 nutrition,relatif*)>
 5 <!ATTLIST recette id ID #IMPLIED>
 6 <!ELEMENT titre (#PCDATA)>
 7 <!ELEMENT date (#PCDATA)>
 8 <!ELEMENT ingredient (ingredient*,preparation)?>
 9 <!ATTLIST ingredient name CDATA #REQUIRED quantité CDATA #IMPLIED
10   unité CDATA #IMPLIED>
11 <!ELEMENT preparation (étape*)>
12 <!ELEMENT étape (#PCDATA)> <!ELEMENT comment (#PCDATA)>
13 <!ELEMENT nutrition EMPTY>
14 <!ATTLIST nutrition calories CDATA #REQUIRED  graisse CDATA #REQUIRED
15   protéine CDATA #REQUIRED>
16 <!ELEMENT relatif EMPTY>
17 <!ATTLIST relatif ref IDREF #REQUIRED>
```

**Problèmes :**

- calories : doit être un nombre positif ;
- protéine : est un pourcentage compris entre 1 et 100 ;
- unité ou quantité : un seul à la fois parmi les deux ;

*Comment faire ? Remplacer les DTDs…*

## 12 XSD, «*XML Schema Definition*»

Buts :

⋆ plus expressif que les DTDs ;
⋆ utilise une notation XML ;
⋆ plus de simplicité ;
⋆ auto-documenté ;

Les contraintes techniques :

⋆ doit gérer les espaces de noms ;
⋆ permet de définir des types de données utilisateurs ;
⋆ supporte de l'héritage à la manière de l'objet ;
⋆ possède une documentation interne ;
⋆ …

Les propositions :

▷ Définition simple de type, *simpleType* : utilisation de chaîne unicode ;
▷ Définition complexe de type, *complexType* : définit un contenu et un modèle d'attribut ;
▷ Déclaration Élement : associer un nom d'élément avec un type simple ou complexe ;
▷ Déclaration Attribut : associer un nom d'attribut avec un type simple ;
▷ Possibilité de définir le nombre d'occurrence d'un élément/attribut.

# XSD : exemple de carte de visite



Exemple : Définition d'une carte de visite

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <b:card xmlns:b="http://businesscard.org"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://businesscard.org
5              business_card.xsd">
6  <b:name>John Doe</b:name>
7  <b:title>CEO, Widget Inc.</b:title>
8  <b:email>john.doe@widget.com</b:email>
9  <b:phone>(202) 555-1414</b:phone>
10 <b:logo b:uri="widget.gif"/>
11 </b:card>
```

**Remarques :**

- *L'espace de nom est «b».*
- *Le `schemaLocation` assure l'unicité du XSD pour éviter les conflits.*
- Il est possible d'utiliser :

```
1   <card xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2     xsi:noNamespaceSchemaLocation="schema.xsd">
```

Pour faire référence à un fichier XSD local, à l'aide du `noNamespaceSchemaLocation`.

# Schémas XML : suite de l'exemple de carte de visite

```
 1 <schema xmlns="http://www.w3.org/2001/XMLSchema"
 2   xmlns:b="http://businesscard.org"
 3   targetNamespace="http://businesscard.org">
 4 <element name="card" type="b:card_type"/>
 5 <element name="name" type="string"/>
 6 <element name="title" type="string"/>
 7 <element name="email" type="string"/>
 8 <element name="phone" type="string"/>
 9 <element name="logo" type="b:logo_type"/>
10 <attribute name="uri" type="anyURI"/>
11 <complexType name="card_type"> <sequence>
12   <element ref="b:name"/>
13   <element ref="b:title"/>
14   <element ref="b:email"/>
15   <element ref="b:phone" minOccurs="0"/>
16   <element ref="b:logo" minOccurs="0"/>
17   </sequence> </complexType>
18 <complexType name="logo_type">
19 <attribute ref="b:uri" use="required"/>
20 </complexType>
21 </schema>
```

xterm

```
pef@darkstar:~/XML$ xmllint bcard.xml --schema bcard.xsl --noout
bcard.xml validates
```

Une description de livre :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <book xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
3 xsi:noNamespaceSchemaLocation="schema.xsd">
4   <title>Cryptography Engineering</title>
5   <author>Bruce Schneier</author>
6   <type>Non-fiction</type>
7 </book>
```

et le fichier `schema.xsd`:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs= http://www.w3.org/2001/XMLSchema
 3     elementFormDefault="qualified">
 4 <xs:element name="book">
 5 <xs:complexType> <xs:sequence>
 6 <xs:element name="title" type="xs:string"/>
 7 <xs:element name="author" type="xs:string"/>
 8 <xs:element name="type" type="xs:string"/>
 9 </xs:sequence> </xs:complexType> </xs:element>
10 <!--end the schema-->
11 </xs:schema>
```

*Ici, l'espace de nom est `xs`, très souvent utilisé par défaut.*

**Définition d'un élément**

Un élément en XSD est soit :

- ⋆ **simple** : il contient du texte mais ne contient pas de sous-éléments ni d'attributs ;
- ⋆ **complexe** : il peut avoir des sous-éléments, des attributs, du texte ou bien être vide.

**Les différences entre «*simple*» et «*complexe*»**

- ▷ `<separateur/>` : pas un type complexe ;
- ▷ `<auteur> Isaac Asimov</auteur>` : pas un type complexe ;
- ▷ `<parking id="456"/>` : type complexe ;
- ▷ `<parking id="456">M. Durand</parking>` : type complexe ;
- ▷ `<individu><nom>James</nom><prenom>Kirk</prenom></individu>` : type complexe.

**Le type «*complexe*»**

```
1 <xs:element name="elementname">
2 <xs:complexType> <xs:sequence>
3 <xs:element name="taga" type="xs:string"/>
4 <xs:element name="tagb" type="xs:string"/>
5 </xs:sequence> </xs:complexType>
6 </xs:element>
```

Il existe différentes possibilitées :

- ∘ `sequence` : indique un ordre fixe sur les sous-élém. ;
- ∘ `all` : n'indique pas d'ordre sur les sous-élém. ;
- ∘ `choice` : indique de choisir les sous-élém. présents ;
- ∘ `maxOccurs` et `minOccurs` : indique combien un sous-élém. peut être présent.

**Exemple de type complexe**

```
1 <xs:element name="optional" type="xs:string" minOccurs="0"/>
2 <xs:element name="nm" type="xs:string" minOccurs="1" maxOccurs="40"/>
3 <xs:element name="maxfive" type="xs:string" maxOccurs="5"/>
4 <xs:element name="infinite" type="xs:string" maxOccurs="unbounded"/>
```

# XSD : les types de données pour un élément

**Les types simples qui s'appliquent aux contenus**

Les **types de données** :

```
1 type="xs:string"
2 type="xs:decimal"
3 type="xs:integer"
4 type="xs:anyURI"
5 type="xs:date"
6 type="xs:time"
7 type="xs:duration"
8 type="xs:byte"
9 type="xs:negativeInteger"
10 type="xs:nonNegativeInteger"
11 type="xs:boolean"
```

Les **valeurs possibles** :

```
1 Chaîne de caractères, ex. "Salut !"
2 Nombre à virgule, ex. 11.6, 5.3
3 Nombre entier, ex. 7, 8, 42
4 Un lien, ex. libpfb.so
5 Année/Mois/Jour, ex. 2003-08-15
6 Heure/Min/Sec, ex. 10:22:00
7 PeriodYrMnthHrMinSec, ex. P4Y3M2D
8 Une valeur sur 8bits de -128 à 127
9 Nombre entier négatif, ex. -256
10 Nombre entier positif, ex. 100, 999
11 TRUE,FALSE, 1 or 0
```

Il est possible de choisir une **valeur par défaut** :

```
1 <xs:element name="month" type="xs:string" default="Jan"/>
2 <xs:element name="pi" type="xs:decimal" fixed="3.14159"/>
```

Pour définir un élément avec un **contenu vide**, on le définit sans type :

```
1 <xs:element name="separateur"/>
```

On définie un nouveau type de donnée qui va s'appliquer sur un «*simpleType*» :

```
1 <xs:simpleType name="restreint">
2 <xs:restriction base="xs:string">
3 <xs:maxLength value="4"/>
4 </xs:restriction>
5 </xs:simpleType>
```

Pour l'utiliser :

```
1 <xs:element name="type" type="restreint"/>
```

**D'autres définitions**

- Pour un type «*string*», il est possible d'utiliser `maxLength` et `minLength`.
- Pour un choix parmi différentes valeurs :

```
1 <xs:enumeration value="News"/>
2 <xs:enumeration value="Sport"/>
3 <xs:enumeration value="Musique"/>
```

- Pour un choix d'une valeur numérique :

```
1 <xs:minInclusive value="50"/>
2 <xs:maxInclusive value="200"/>
```

- Pour une taille exprimée en nombre de chiffres :

```
1 <xs:totalDigits value="7"/> un nombre jusqu'à 7 chiffres
2 <xs:fractionDigits value="2"/>  le nombre 1.55 est valide, 1.552 ne l'est pas.
```

Il est possible de définir des types de données à l'aide :

- d'**expressions régulières** :

```
<xs:pattern value="[a-zA-Z0-9]{4,8}"/>
<xs:pattern value="[a-z]{5}"/>
<xs:pattern value="(\d){4,}"/>
<xs:pattern value="[a-fA-F0-9]{1,4}"/>
```

- **d'union** pour permettre des alternatives :
- de **liste** :

```
<xs:simpleType>
  <xs:list item="xs:int">
</xs:simpleType>
```

le contenu «`10 20 30`» sera accepté.

- **d'énumération** :

```
<xs:simpleType name="couleur">
  <xs:restriction base="xs:string">
    <xs:enumeration value="rouge">
    <xs:enumeration value="vert">
    <xs:enumeration value="bleu">
  </xs:restriction>
<xs:simpleType>
```

On peut aussi traiter les espaces :

```
<xs:whiteSpace value="preserve"/> "Space  text" reste inchangé
<xs:whiteSpace value="collapse"/> "Space  text" se transforme en "Space text"
```

*«collapse» réduit des espaces multiples en un seul espace comme en HTML.*

*«preserve» conserve le formatage.*

*«replace» change les retours à la ligne en espace.*

Pour la **signature électronique**, il est important de gérer les espaces de manière à **supprimer toute ambigüité**.

# XSD : la définition d'attributs

Un attribut est définit avec «`attribute`» au sein d'un **type complexe** uniquement.

Chaque définition d'attribut doit être :

- accompagnée d'un nom et d'un **type simple** uniquement ;
- déclaré comme optionnel ou requis ;
- associé à une valeur par défaut.

```
1 <xs:attribute name="code" type="xs:number" use="optional"/>
2 <xs:attribute name="colour" type="xs:string" use="optional"/>
3 <xs:attribute name="id" type="xs:string" use="required"/>
```

On peut définir des groupes pour lier des attributs et les utiliser dans un élément :

```
1 <xs:attributeGroup name="att">
2 <xs:attribute name="age" type="xs:number"/>
3 <xs:attribute name="titles" type="xs:string"/>
4 </xs:attributeGroup>
5
6 <xs:element name="author">
7 <xs:complexType>
8 <xs:simpleContent>
9 <xs:extension base="xs:string">
10 <xs:attributeGroup ref="att"/>
11 </xs:extension>
12 </xs:simpleContent>
13 </xs:complexType>
14 </xs:element>
```

On fait le lien entre la définition et l'utilisation avec le `ref`.

**La définition d’un élément vide avec attributs**

```
1 <xs:element name="image">
2   <xs:complexType>
3   <xs:attribute name="source"
4     type="xs:string"/>
5   </xs:complexType>
6 </xs:element>
```

*Cette définition s’utilise comme la balise «`img`» du HTML.*

**Un élément avec contenu simple et attributs**

```
1 <xs:element name="individu">
2 <xs:complexType><xs:simpleContent>
3     <xs:extension base="xs:string">
4       <xs:attribut name="nom"
5         type="xs:string"/>
6     </xs:extension>
7 </xs:simpleContent></xs:complexType>
8 </xs:element>
```

**Un élément avec contenu complexe et attributs**

```
1 <xs:element name="individu"> <xs:complexType>
2   <xs:sequence>
3   <xs:element name="nom" type="xs:string"/>
4   <xs:element name="prenom" type="xs:string"/>
5   </xs:sequence>
6   <xs:attribute name="id" type="xs:token"/><xs:complexType>
7 </xs:element>
```

On ne peut inclure dans la balise «`individu`» que les deux balises «`nom`» et «`prenom`».

**Un élément avec contenu mixte**

```
1 <xs:complexType name="personne" mixed="true">
2 <xs:sequence>
3   <xs:element name="nom" type="xs:string"/>
4   <xs:element name="prenom" type="xs:string"/>
5 </xs:sequence>
6 </xs:complexType>
7 <xs:element name="employe" type="personne"/>
```

On peut mélanger texte et balises dans le contenu de la balise «`employe`».

# XSD : exemples de déclaration d'élément et d'attribut

**Exemples** :

- <element name="serialnumber" type="nonNegativeInteger"/>
- <attribute name="alcool" type="pourcentage"/>

**Type simples** :

string, boolean, decimal, float, double, dateTime, time, date, hexBinary, base64Binary, anyURI *etc.*

**Types dérivés contraints**

length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, totalDigits, fractionDigits

**Exemple**

```
1 <simpleType name="score_from_0_to_100">
2 <restriction base="integer">
3 <minInclusive value="0"/> <maxInclusive value="100"/>
4 </restriction>
5 </simpleType>
6
7 <simpleType name="pourcentage"> <restriction base="string">
8 <pattern value="([0-9]|[1-9][0-9]|100)\%"/> </restriction>
9 </simpleType>
```

*À la ligne 6, on utilise une **expression régulière**.*

La «restriction» est une forme de «dérivation» à partir d'un type existant :

```
1 <simpleType name="integerList">
2 <list itemType="integer"/> </simpleType>
3
4 <simpleType name="boolean_or_decimal">
5 <union>
6   <simpleType> <restriction base="boolean"/> </simpleType>
7   <simpleType> <restriction base="decimal"/> </simpleType>
8 </union>
9 </simpleType>
```

*Ici, la dérivation est réalisée sur un type simple. Sur un type complexe, on peut utiliser l'«extension» qui ressemble à de l'héritage…*

**Conclusion**

On appelle «*schéma*», une description formelle de la syntaxe d'un langage basé sur XML :

- ⋆ DTD : language simple de schéma : éléments, attributs, entitées, …
- ⋆ Schéma XML : langage plus avancé :
  - ◇ déclaration d'élément/attribut ;
  - ◇ type simple/complexe/dérivé ;
  - ◇ contrôle de la valeur des contenus ;
  - ◇ *etc.*

Avec SAX, «*Simple API for XML*» :

⋆ qui est une API pour la lecture d'un flux XML :

⋆ interface événementielle : une balise ouvrante est un événement, une balise fermante est un événement, *etc*

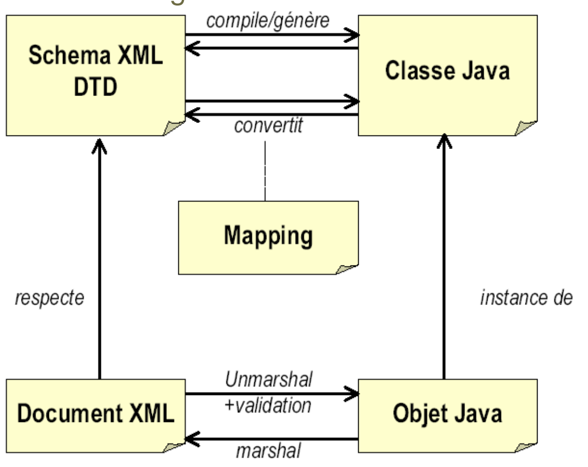
Avec DOM, «*Document Object Model*» :

⋆ génére une arborescence d'objet correspondant à la structure du document XML ;

⋆ possibilité de naviguer dans cette arborescence ;

⋆ lourd !

XML Data Binding :




Permet de faire de la sérialisation d'objet, mais il faut faire attention, l'ordre entre les noeuds peut être perdu et cela peut poser des problèmes lors de la dé-sérialisation : mauvaise valeur d'index, *etc*
Il faut bien définir le format !

## SAX : Utilisation



1. Utilisation du package `org.xml.sax`;
2. Implémentation de l'interface `ContentHandler`;
3. instanciation d'un parser en lui indiquant quel gestionnaire de contenu à utiliser ;
4. fini !

**API du `ContentHandler`**

- `setDocumentLocator` : connaitre la position du curseur de lecture (numéro ligne et de colonne) dans le document ; *utile pour déboger mais à éviter de manipuler.*
- `startDocument` : à appeler une seule fois au démarrage de l'analyse avant toutes les autres (à part setDocumentLocator) et sert à initialiser le parser.
- `endDocument` : à appeler à la fin, permet de nettoyer et de finaliser le parser.
- `processingInstruction` : intervient à chaque instruction de fonctionnement, c-à-d en dehors de ceux concernant la structure du document.
  Exemple : `<?xml version= '1.0'>`
- `startPrefixMapping` : intervient à chaque rencontre de l'indication d'un espace de nom ;
- `endPrefixMapping` : pour la fin de l'utilisation d'un espace de nom ;

## SAX : Utilisation



**API du `ContentHandler` – Suite**

- `startElement` : début de la lecture d'un élément XML (la méthode la plus importante) :
  `startElement (String namespaceUri, String localName, String rawName, Attributs atts)`
  - `nameSpaceUri` : la chaîne de caractères contenant l'URI complète de l'espace de nommage de l'élément ou une chaîne vide si l'élément n'est pas compris dans un espace de nommage ;
  - `localName` : le nom de l'élément sans le préfixe s'il y en avait un ;
  - `rawName` : est le nom de l'élément version xml 1.0 c-à-d `$prefix:$localname` ;
  - `attributs` : la liste des attributs de l'élément que l'on étudiera un peu plus loin.
  - `endElement` : fin de lecture d'un élément.
  - `characters`: correspond au texte entre les balises : `<maBalise>un peu de texte</maBalise>`

## SAX : un exemple



```
 1 // Exemple d'utilisation : récupération de la quantité de farine dans une recette
 2 import java.io.*;
 3 import org.xml.sax.*;
 4 import org.xml.sax.helpers.*;
 5 import org.apache.xerces.parsers.SAXParser;
 6
 7 public class Flour extends DefaultHandler {
 8   float amount = 0;
 9   public void startElement(String namespaceURI, String localName,
10                  String qName, Attributes atts) {
11     if (namespaceURI.equals("http://recipes.org") && localName.equals("ingredient"))
12     {
13        String n = atts.getValue("","name");
14        if (n.equals("flour"))
15        { String a = atts.getValue("","amount"); // assume 'amount' exists
16          amount = amount + Float.valueOf(a).floatValue();}}}
17   public static void main(String[] args) {
18     Flour f = new Flour();
19     SAXParser p = new SAXParser();
20     p.setContentHandler(f);
21     try { p.parse(args[0]); }
22     catch (Exception e) {e.printStackTrace();}
23     System.out.println(f.amount);
24   }
25 }
```

## SAX : suite de l'exemple



```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <?dsd href="recipes.dsd"?>
3  <collection xmlns="http://recipes.org"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://recipes.org recipes.xsd">
6    <description>Some recipes used in the XML tutorial.</description>
7  <recipe>
8    <title>Ricotta Pie</title>
9    <ingredient name="ricotta cheese" amount="3" unit="pound" />
10 …
11   <preparation>
12   <step>Beat the 12 eggs, 2 cups sugar and vanilla extract together.</step>
13   </preparation>
14   </ingredient>
15   <ingredient name="flour" amount="4" unit="cup" />
16 …
17   <ingredient name="vanilla extract" amount="1" unit="teaspoon" />
18   <preparation>
19   <step>Combine the flour, baking powder, and 1 cup of the sugar together.
20   </step>
21 …
22   </recipe>
23 </collection>
```

Dans un fichier XSLT, on peut utiliser du JavaScript, pour manipuler XML & XSL.
On va définir du code JavaScript dans le document HTML produit par le fichier XSLT :

```
1  <xsl:template match="/">
2  <html>
3  <head>
4    <title>Collection de nuages</title>
5    <script type="text/javascript"><![CDATA[
6    var XMLsource = new Object;
7    var XSLsource = new Object;
8    XMLsource = document.XMLDocument;
9    XSLsource = document.XSLDocument;

10     function changecouleur()
11   {   balisefont=XSLsource.documentElement.selectNodes("//font");
12     couleur=balisefont[0].getAttribute('color');
13       …
14     document.body.innerHTML = XMLsource.transformNode(XSLsource); }

15   function affichealtitude(altmax, altmin)
16   { alert("Altitude maximale de ce nuage :"+altmax+"m, minimale :"
17         +altmin+"m."); }
18   ]]></script>
19 </head>
```

Partie «invisible»

```
20 <!--    <body onClick="changecouleur()">-->
21   <body>
22     <h1>Les nuages</h1>
23     <xsl:for-each select="nuages/nuage">
24     <h2 onClick="affichealtitude({altitude/@max}, {altitude/@min})">
25     <xsl:value-of select="nom/text()"/></h2>
26       <p>Ce type de nuage possède les <font color="red">espèces</font>
27       suivantes :</p>
28       <ul>
29         <xsl:for-each select="nom/espece">
30           <li><xsl:value-of select="."/></li>
31         </xsl:for-each>
32       </ul>
33     </xsl:for-each>
34   </body>
35 </html>
36 </xsl:template>
```

**Explications :**

- ligne 5 : on ajoute du code JavaScript dans la partie `<head>` de la page résultat de XSLT, et pour ne pas que ce contenu soit interprété on le met entre `<![CDATA[` et `]]`.
- ligne 8 et 9 : on accède au DOM pour la partie XML et XSL ;
- ligne 15 : la modification d'un élement de XSL est propagé sur la transformation de l'élément XML, ce qui force l'apparence à changer dans le navigateur ;
- ligne 24 : pour chaque «nuage» du document XML, on attache un événement sur le click de la souris qui appelle la fonction en passant en paramètre la valeur des attributs `altitude/@min` et `altitude/@max`.

Le fichier «nuages.xml» associé à l'exemple :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <nuages>
3  <nuage>
4    <nom>altocumulus
5      <espece>lenticularis</espece>
6      <espece>stratiformis</espece>
7      <espece>castellanus</espece>
8      <espece>flocus</espece>
9    </nom>
10   <altitude max="6000" min="2000"/>
11   <hydrometeores>Aucun.</hydrometeores>
12 </nuage>
13 </nuages>
```

Ajax, «*Asynchronous JavaScript + XML*» permet à un document HTML chargé dans un navigateur de faire des requêtes asynchrones, c-à-d sans avoir à recharger le contenu du document, sur un serveur pour récupérer un contenu au format XML.

L'objet JavaScript permettant de faire cette récupération est `XMLHttpRequest` :

- `abort()` : arrête la requête courante ;
- `getAllResponseHeaders()` : retourne la chaîne contenant toutes les en-têtes de la réponse HTTP du serveur ;
- `getResponseHeader("name")` : retourne la valeur sous forme chaîne d'une en-tête donnée par son nom ;
- `open("method", "url", asyncflag, "username", "password")` : réalise la configuration de la future requête où l'on peut donner l'URL, la méthode HTTP employée (en général «GET» ou «POST») ainsi que des options supplémentaires comme un «login/mot de passe» et un drapeau si l'on désire que la requête se fasse de manière asynchrone ;
- `send(content)` : transmet la requête, avec un paramètre optionnel pouvant être une chaîne ou bien des données du DOM ;
- `setRequestHeader("label", "value")` : permet de créer des associations «nom, valeur» qui pourront être transmises avec la requête et suivant la méthode sélectionnée («GET» ou «POST»).

**Les propriétés de l'objet `XMLHttpRequest` :**

- `onreadystatechange` : gestionnaire d'événement, «Event handler», qui est appelé à chaque changement d'état ;
- `readyState` : fournit le statut de l'objet : (0 = unitialized, 1 = loading, 2 = loaded, 3 = interactive, 4 = complete) ;
- `responseText` : une chaîne correspondant au contenu des données retournées par le serveur ;
- `responseXML` : la représentation DOM des données retournées par le serveur ;
- `status` : le code numérique de retour de la requête HTTP au serveur le «HTTP status code» ;
- `statusText` : la chaîne de caractères de description du «code de status».

```
1  <html><head><title>Exemple</title>
2
3  <script type="text/javascript" language="JavaScript">
4  var obj;
5  function getData() {
6  document.body.style.cursor='wait';
7  if (window.XMLHttpRequest)
8    { obj = new XMLHttpRequest();}
9  else if (window.ActiveXObject)
10   { obj = new ActiveXObject("Microsoft.XMLHTTP");}
11
12 var goUrl = "http://localhost/document_annuaire.xml";
13 obj.onreadystatechange = xmlReady;
14 obj.open("GET",goUrl, false );
15 obj.send();
16 document.body.style.cursor='auto';
17 }
```

*L'obtention de cet objet dépend du navigateur employé.*

```
18 function xmlReady() {
19   if (obj.readyState == 4) { // On verifie le succes de la requete AJAX
20     if (obj.status == 200) { // On verifie que le document a bien ete trouve
21     var xmldoc = obj.responseXML;
22     var node1 = xmldoc.selectSingleNode('response/telephone');
23     var node2 = xmldoc.selectSingleNode('response/companyname');
24     document.forms[0].telephone.value = node1.firstChild.data;
25     document.forms[0].cname.value = node2.firstChild.data;
26 } }
27 </script></head><body>
28
29 <form id="frmTest">
30 Compagnie:   <input type="text" name="cname" /><br />
31 T&eacute;l&eacute;phone : <input type="text" name="telephone" /><br />
32 <input type="button" value="Click" onclick="getData();" />
33 </form></body></html>
```

**Explications** :

– ligne 19 et 20 : on vérifie que la récupération du document XML s'est bien réalisée sans erreur

Les instructions de manipulation de l'arborescence du DOM :

- `basename` : retourne le nom d'un noeud (mais sans le «namespace») ;
- `childNodes` : retourne la liste de tous les noeuds enfant du noeud sélectionné ;
- `lastChild` : retourne le dernier fils du noeud courant ;
- `nextSibling` : retourne le prochain noeud qui suit immédiatement le noeud courant. Deux noeuds sont «siblings», ou cousin, s'ils ont le même parent ;
- `parentNode` : retourne le noeud parent d'un noeud (pour le noeud racine, renvoie Null) ;
- `text` : retourne le contenu texte d'un noeud et de tous ses enfants
- `xml` : retourne le contenu XML d'un noeud et de tous ses enfants ;
- `hasChildNodes` : retourne vrai ou faux suivant que le noeud possède ou non des enfants ;
- `nextNode` : retourne le prochain noeud d'une collection (cette méthode peut être utilisée avec d'autres pour le parcours du DOM) ;
- `previousNode` : retourne le noeud précédent d'une collection ;
- `selectNodes` : retourne un objet «NodeList» qui contient tous les noeuds correspondant un motif donné, «pattern matching» ;
- `selectSingleNode` : retourne un objet «Node» pour le premier noeud enfant qui correspond à un motif donné.

Exemple en JavaScript :

```
1 for each x in xmlDocument.documentElement.childNodes
2 document.write("<b>" & x.nodename & ":</b> ");
3 document.write(x.text);
4 document.write("<br />");
5 next
```

Sur le document XML suivant :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <response>
3 <companyname>Ma compagnie à moi</companyname>
4 <telephone>05 55 43 69 83</telephone>
5 </response>
```

**Tout dépend du navigateur employé !**

▷ Pour la requête Ajax :
- ◇ la réalisation de la requête AJAX : un objet différent dans Explorer et Firefox ;
- ◇ la gestion des événements asynchrones ou non liès à la requête ;
- ◇ la détection du succès de la requête ;

▷ Pour l'exploitation des données XML reçues :
- ◇ une API de parcours du DOM qui ne fonctionne bien que sous... et mal sous... (mettre le nom de son navigateur favori et celui de son concurrent) ;
- ◇ complexité d'accès aux éléments de l'arborescence XML ;
- ◇ *Que faire ensuite de ces éléments récupérés ? Comment les intégrer dans la page HTML ?*

**La solution**

- ne pas dépendre, et ne pas tenir compte, des spécificités des différents navigateurs ;
- manipuler tous les éléments constitutifs de la page HTML ;
- simplifier la réalisation de requête AJAX et faciliter l'accès aux données reçues (XML, JSON, HTML, text, javascript *etc.*)
- s'intégrer naturellement dans Javascript ;
- que tout soit simple d'utilisation…

$\Longrightarrow$ Utiliser JQuery !

**Qu'est-ce que JQuery ?**

Une bibliothèque Javascript d'environ $19Ko$ compressé ($120ko$ décompressé) à ajouter à une page HTML :

- qui **normalise** l'utilisation des différents navigateurs ;
- qui bénéficie d'une communauté et d'une forte adoption : Amazon, IBM, Nokia, Google, Microsoft, *etc.*
- qui permet de disposer d'une interface évoluée, «JQuery UI», dans le navigateur ;
- qui utilise les sélecteurs de CSS v3 pour intervenir sur les éléments du DOM ;
- qui possède de nombreux *plugins* ;
- qui est simple, élégante, bien documentée : elle*plait* énormément…
- qui permet d'ajouter de nombreux effets dynamiques à une page HTML :
  - des intéractions avec l'utilisateur avec les événements liés à la navigation (c'est du Javascript) ;
  - des effets d'animations graphiques de qualités : *fading*, *Sliding*, *etc.*
  - l'apparition/masquage et l'ajout/retrait d'éléments dans la page HTML ;
  - la manipulation du CSS : ajout de classe à un élément, changement de valeur d'attributs *etc.*
  - la gestion d'Ajax, d'XML.
- qui repose sur les 3 opérations suivantes :
  - la sélection d'éléments du DOM via le CSS ;
  - l'application d'opération sur ces éléments au travers de méthodes proposées par JQuery ;
  - le *chaînage* d'opérations multiples sur ces éléments et l'utilisation *d'itération* automatique sur ces éléments.

| Sélecteur | Action | Paramètres | |
|---|---|---|---|
| `JQuery('p')` | `.css` | `('color','red');` | *changer la couleur des paragraphes* |
| `$('p')` | `.css` | `('color','red');` | *avec le raccourci «$»* |

## L'intégration de JQuery dans la page HTML

On peut inclure JQuery en le téléchargeant depuis le CDN de Google (Content Distribution Network) (avantages : rapidité, et mutualisation avec d'autres pages et ou d'autres sites) :

```
<head>
  <title>Hello jQuery world!</title>
  <script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.0/jquery.min.js">
  </script>
</head>
```

## Sélection des éléments et ajout dynamique de contenu HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<script type="text/JavaScript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
</head>
<body>
<div id="contents"></div>
<script>
var contenu = $('<h1> Un contenu de plus </h1>');
$(contenu).appendTo('#contents');
$('#contents').css({'background-color':'yellow','width':'400px'});
</script>
</body>
</html>
```

- ligne 9 : on crée un contenu HTML à partir de texte.
- ligne 10 : on ajoute ce contenu au `div` qui a pour `id` `contents`
- ligne 11 : on modifie le CSS associé au `div`.

**Un contenu de plus**

# La requête Ajax

En JQuery : `$.ajax(url, [paramètres]);`

| Paramètre | Description |
|---|---|
| url | L'url de la ressource |
| type | La méthode HTTP utilisée pour réaliser la requête (GET ou POST en général) |
| data | Les données à envoyer dans la requête |
| dataType | Le type de données attendues en retour de la requête : XML, HTML, script, JSON, text |
| success(data,textStatus,jqXHR) | Une fonction à appeler en cas de succès de la requête |
| error(jqXHR,textStatus,errorThrown) | en cas d'échec |
| complete(jqXHR,textStatus) | lorsque la requête est finie |
| timeout | délai pour la réalisation de la requête |

```
 1 $.ajax({
 2     type: 'GET',
 3     url: 'books.xml',
 4     dataType: 'xml',
 5     success: function(donnees, textStatus) {
 6           livre = $(donnees).find('author').text(); $('#contents').html(livre); },
 7     error: function(xhr, textStatus, errorThrown) {
 8       alert('An error occurred! ' + ( errorThrown ? errorThrown : xhr.status ));
 9       }
10   });
```

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html>
4  <head><script type="text/JavaScript"
5    src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script></head>
6  <body>
7  <h2>Contenu fourni par Ajax</h2>
8  <div ID="contents"></div>
9  <input type="button" id="requete" value="Ajax !">
10
11 <script>
12 function traiter_livre(index, valeur) {
13   $('#liste_livres').append('<li>'+$(valeur).text()+'</li>'); }
14
15 function traiter_livres(donnees)
16 {   $('<ul id="liste_livres"></ul>').appendTo('#contents');
17     $.each(donnees, traiter_livre); }
18   $('#requete').click(function(){
19     $.ajax({
20       type: 'GET',
21       url: 'books.xml',
22       dataType: 'xml',
23       success: function(donnees, textStatus) {
24             traiter_livres($(donnees).find('author'));
25             },
26       error: function(xhr, textStatus, errorThrown) {
27         alert('Une erreur s'est produite! ' + ( errorThrown ? errorThrown : xhr.status )); }
28     });
29   });
30 </script>
31 </body>
32 </html>
```

Au chargement de la page :

**Contenu fourni par Ajax**

Ajax !

Après appui sur le bouton «`Ajax !`» :

**Contenu fourni par Ajax**

- Isaac Asimov
- David Brin
- Neal Stephenson

Ajax !

**Explications du code proposé sur le transparent précédent**

- ligne 6 : un `div` est défini avec un identifiant unique pour le sélectionner facilement `contents` ;
- ligne 7 : un bouton est ajouté pour déclencher la récupération du document XML ;
- ligne 13 : on définie une fonction pour traiter la liste complète des éléments «`author`» du fichier XML, en créant un contenu HTML dynamiquement basé sur une liste non ordonnée «`ul`» ;
- ligne 10 : on définie une fonction pour traiter un élément `author` : on l'ajoute à la liste comme nouvel élément ;
- ligne 17 : on associe au bouton une fonction qui réalise la récupération du fichier XML :
  - ligne 19 : utilisation de la méthode HTTP «`GET`» ;
  - ligne 20 : url de récupération du fichier XML, en chemin relatif par rapport au serveur d'où on a récupéré le document HTML initial ;
  - ligne 21 : on indique attendre un document au format XML ;
  - ligne 22 : en cas de succès de la requête Ajax, on traite son résultat à l'aide de la fonction définie en ligne 13 ;
  - ligne 25 : en cas d'erreur, on affiche un message d'erreur à l'utilisateur dans une fenêtre modale.

*Attention à ne pas rappuyer plus d'une fois sur le bouton…*

**SOAP, «Simple Object Access Protocol»**

SOAP est un format de message en XML permettant d'échanger des informations à l'aide du protocole de transport HTTP.

Il est utilisé dans le cadre des **applications distibuées** avec des échanges de type client/serveur.

Il est constitué :
- ▷ d'une en-tête contenant des infos à destination du serveur, comme un numéro de message, une date d'expiration ou de requête *etc.*
- ▷ d'un corps où se trouve la requête ou la réponse.

Pour la requête :

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
3  <SOAP-ENV:Header SOAP-ENV:mustUnderstand="1">
4  <requestnum>8312</requestnum>
5  <date>11/05/03</date>
6  </SOAP-ENV:Header>
7
8  <SOAP-ENV:Body>
9  <m:media xmlns: m="http://videoondemandplace.com">
10 <m:name>My documentary</name>
11 <m:length>120m</length>
12 <m:codec>DivX</codec>
13 </m:media>
14 </SOAP-ENV:Body>
15 </SOAP-ENV:Envelope>
```

La réponse associée :

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
3 <SOAP-ENV:Body>
4 <SOAP-ENV:Fault>
5 <faultcode> SOAP-ENV: MustUnderstand</faultcode>
6 <faultstring> Did not understand Header</faultstring>
7 </SOAP-ENV:Fault>
8 </SOAP-ENV:Body>
9 </SOAP-ENV:Envelope>
```

*Par exemple, ici, le serveur renvoie un message d'erreur. Complément*

**Plan**

- Disposer d'un serveur Web léger ;
- Quelques notions de JavaScript.
- JQuery et le DOM :
  - notion de «classe» et d'«id» : sélection d'élément ;
  - modification du DOM : ajouter du contenu HTML, modifier des attributs...
  - interactivité ;
- les éléments de formulaires : cases à cocher, saisie clavier.

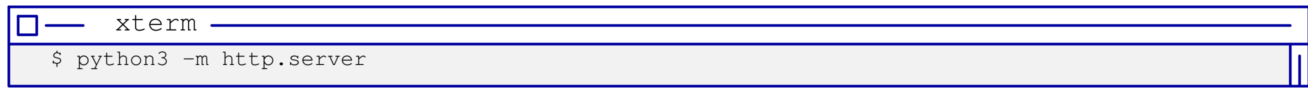
**Accès HTTP**

Il est très important d'accéder au contenu du site web par réseau, service «`http://`», et non par le système de fichier, service «`file://`».

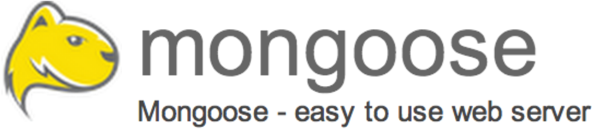
Pour cela, il faut utiliser un serveur Web léger.

**Serveur Web léger**

▷ sous OSX et Linux :




▷ sous Windows : Installation de `https://mongoose.ws/binary/`




*On pourra ensuite s'y connecter dans son navigateur avec l'URL suivante :*

*«`http://localhost:8080/`»*

**Déclaration de variable**

```
1   var ma_variable = "";
```

*La variable n'a pas de* type *préalable, il dépend de ce quelle contient. Ici, une chaîne de caractères vide.*

```
1 var mon_tableau = new Array();
2 var i = 0;
3
4 for(i=0; i<10; i++)
5    {
6       mon_tableau[i] = i;
7   }
```

*Un tableau commence à l'indice zéro.*

**Manipulation de chaîne de caractères**

```
1 var c1 = "un";
2 var c2 = "mot";
3 var c3 = c1 + " " + c2;
4 var taille = c3.length;
```

*La concaténation et l'obtention de la taille avec l'attribut* `length`*.*

```
1 var chaine = "ABCDEFGHIJ";
2 var i = 0;
3 var resultat = false;
4 for(i=0; i < chaine.length; i++)
5 if (chaine[i] == 'E')
6   { resultat = true; }
```

*La chaîne de caractères peut être traitée comme un tableau.*

**Déclaration de fonction**

```
1 function ma_fonction() {
2 //
3 }
```

```
1 var une_fonction = function () {
2 //
3 }
4 une_fonction();
```

*Ici, la fonction est anonyme.*

JQuery se présente comme une bibliothèque à intégrer dans le document HTML :

```
1 <script src="js/jquery-3.6.4.min.js">
2 </script>
3     <script>
4     $(document).ready(function() {
5       // Votre programme est ici
6     });
7 </script>
```

```
1 <script src="js/jquery-3.6.4.min.js">
2 </script>
3 <script>
4 $(function() {
5      // Votre programme est ici
6    });
7 </script>
```

**La méthodologie en deux étapes**

1. Sélectionner un ou des éléments : un «*tag*» en complément d'une «*classe*» ou par un «*id*» :
   `a.menu`
   *renvoie la liste des «tag anchors» de classe «menu»*
   `#CONTENU`
   *renvoie l'élément d'«id» «CONTENU»*

2. Modifier l'élément :
   - changer la valeur d'un attribut ;
   - ajouter un nouveau contenu ;
   - supprimer un élément ;
   - récupérer des informations depuis l'élément ;
   - ajouter/supprimer une classe ;
   - obtenir un «événement» à partir de l'élément.

Il est possible d'enchaîner les opérations :

```
$('#popUp').width(300).height(300);
```

*Le retour d'un appel à JQuery fournit un «objet» capable de supporter des opérations liées à JQuery.*

- Sélectionner un élément :

```
<p class="description">Ceci est une introduction</p>      $('.description')

<p id="contenu">Ceci est une introduction</p>             $('#contenu')
```

- modifier le HTML :

```
<div id="conteneur">
      <div id="erreurs">
        <h2>Erreurs:</h2>
      </div>
</div>
```

```
$('#erreurs').html('<p>2 erreurs dans le formulaire.</p>');
```

*On aurait aussi pu utiliser `.append()` pour ajouter le contenu HTML à celui déjà existant.*

- Supprimer du contenu : `$('#popup').remove()` ou le remplacer avec `replaceWith();`
- Ajouter/supprimer une classe : `$('#alertBox').removeClass('highlight');` ou alterner avec `toggleClass();`
- modifier le CSS avec `.css`: `$('p.highlight').css('border', '1px solid black');`
- modifier un attribut avec `.attr`: `$('#banniere img').attr('src','images/deux.png');`
- pour un champ de formulaire : `<input type="text" id="saisie">`

obtenir la valeur du champ :

```
var ma_valeur = $('#saisie').val();
```

positionner la valeur du champ :

```
$('#saisie').val('toto@unilim.fr');
```

- traiter plusieurs éléments avec `$('selecteur').each();`

```
1 <div id="bibliographie">
2 <h3>Pages référençant cet article</h3>
3 <ul id="bibListe">
4 </ul>
5 </div>
```

```
1 $('a.refbib').each(function() {
2   var extLink = $(this).attr('href');
3   $('#bibListe').append('<li>' +
4     extLink + '</li>');
5 });
```

- gérer des événements :
  - le clic de souris : `$('#menu').click();`
  - le survol par la souris : `$('a').mouseover();`

```
$('#menu').mouseover(function() {
    $('#submenu').show();
});
```

*Gérer un menu et faire apparaitre un sous menu.*

  - pour un champ de formulaire :
    - ▷ `blur()` : lorsque le champ du formulaire est déselectionner ;
    - ▷ `focus()` : lorsque le champ du formulaire est sélectionné ;
    - ▷ `change()` : lorsque son contenu a changé.

- la sélection conditionnelle :

```
<input type="checkbox">
<input type="text">
```

```
$('input[type="text"]').val('votre email');
```

*Ici, on sélectionne l'«input» suivant la condition «est de type `"text"`» et on change sa valeur.*

- La modification de propriétés :

```
<input type="checkbox">
<input type="text">
```

```
$('input[type="checkbox"]').prop('checked', true);
$('input[type="text"]').prop('disabled', true);
```

*Ici, on change l'état coché ou non coché avec le `checked`, la zone d'entrée en désactivée ou non avec `disabled`.*

- l'utilisation des événements clavier :

```
<input type="text" id="chaine">
<button id="mon_bouton" disabled>Envoyer</button>
<script>
$(function() {
function appui_touche_clavier() {
  if ($('#chaine').val().length > 2)
    {   $('#mon_bouton').prop("disabled",false);
    }
  else {  $('#mon_bouton').prop("disabled",true);
    }
}
$('#chaine').keyup(appui_touche_clavier);
});
</script>
```

*Ici, on active le bouton «Envoyer» uniquement lorsque la longueur de la donnée saisie est supérieure à 2.*