

Faculté
des Sciences
& Techniques



Université
de Limoges

Licence 3^{ème} année



Compilation 2

—

P-F. Bonnefoi

Version du 26 mai 2021

Table des matières

1	Contenu et objectifs	5
2	Langage interprété vs compilé : la compilation	6
	Langage interprété vs compilé : l'interprétation	7
	Langage interprété vs compilé	8
3	L'analyse lexical : les automates à nombre fini d'états	9
	Expressions régulières ou <i>expressions rationnelles</i>	10
	Analyseur lexical	12
	L'outil Lex	13
	Analyseur lexical	14
	Analyseur lexical : un exemple	15
	Analyseur lexical : un autre exemple	16
	Analyseur lexical : exemple d'un format de fichier	17
	Analyseur lexical : exemple	19
	Analyseur lexical : utilisation	20
	Lex : la gestion des accents	22
4	Rappels sur les grammaires	24
	Rappels sur les grammaires	25
	Les «parsers» LR	30
	Les différentes catégories d'analyseurs LR	31
	Principe de fonctionnement d'un analyseur LR	32

Exemple de table	33
Interprétation intuitive d'une table	39
Conflits	40
Exemple de conflit	41
5 Analyseur lexical : un avant goût de grammaire ?	43
6 Liens entre Lex et un analyseur syntaxique	47
7 L'analyseur syntaxique YACC	55
Lex & YACC : un exemple	58
YACC : gestion des erreurs	62
8 XML : les différents outils	64
XML vs HTML	67
Afficher un document XML : XSL	71
Format XML : DTD, « <i>Document Type Description</i> »	72
Utilisation de XSL et DTD	73
DTD : notion de classe de document	74
Le format XML	75
XML : les atouts	76
Exemple de document XML	77
XML : Utilisation des balises spécifiques	78
XML : Syntaxe des éléments constitutifs	79
9 DTD : Définition d'une classe de documents	80
DTD : La notion d'entités	82
DTD : La notion d'élément	83

DTD : les attributs d'éléments	85
DTD : la notion d'espace de nom	87
10 CSS : présentation rapide	89
XML & CSS	91
11 XSLT : aller plus loin que les CSS...	92
XSLT : accès aux données	93
XSLT : opérations avancées	94



Objectifs :

- ▷ retour sur les analyseurs 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... ?

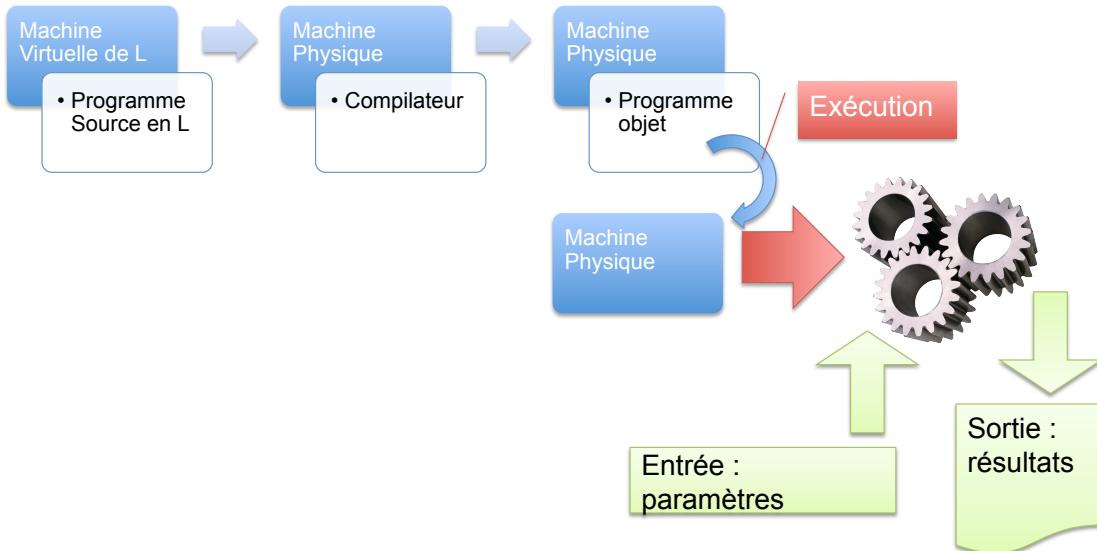


2 Langage interprété vs compilé : la compilation

6

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.

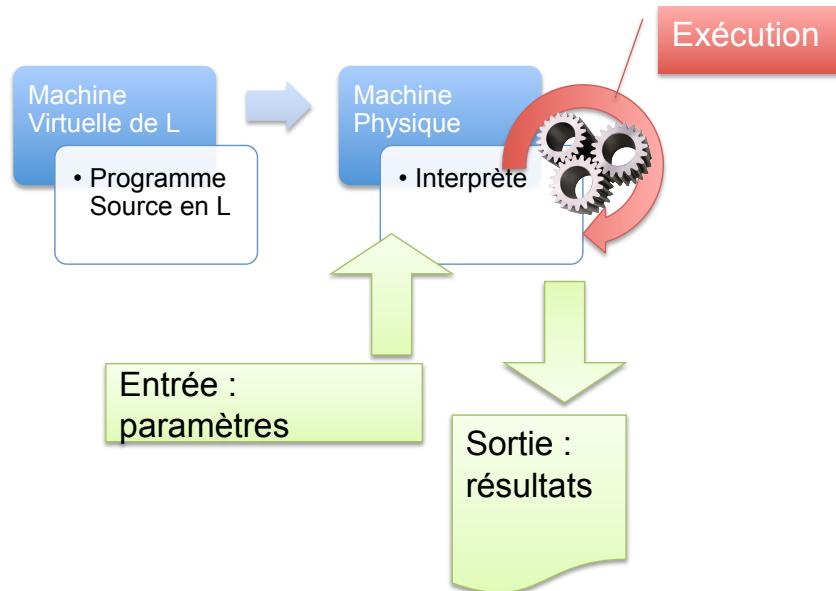


Langage interprétré vs compilé : l'interprétation

7

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.).

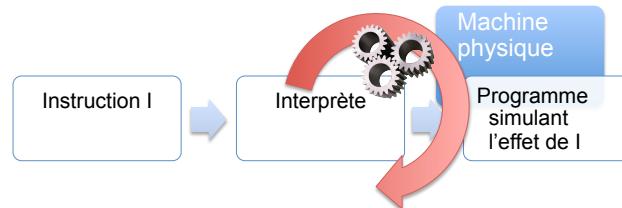
Enfin, grâce à l'interpréteur, on traduit chaque instruction du programme en une suite d'instructions de la machine physique, au fur et à mesure de l'exécution du programme (sur des entrées et il donne des sorties).



Langage interprété vs compilé

8

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



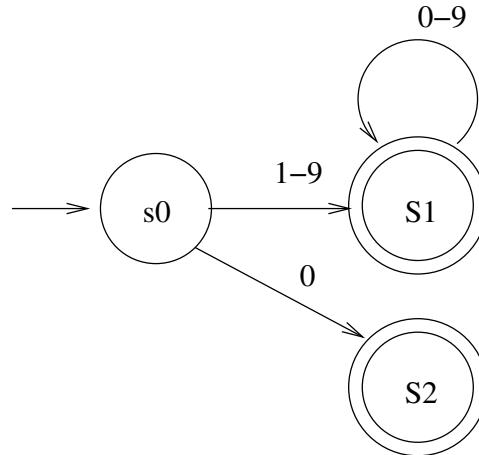
Lors de l'**exécution** du programme **compilé**, chaque instruction du programme correspond à des instructions de la machine physique.

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

9

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*»).



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 chaîne (en mémorisant la séquence correspondante).

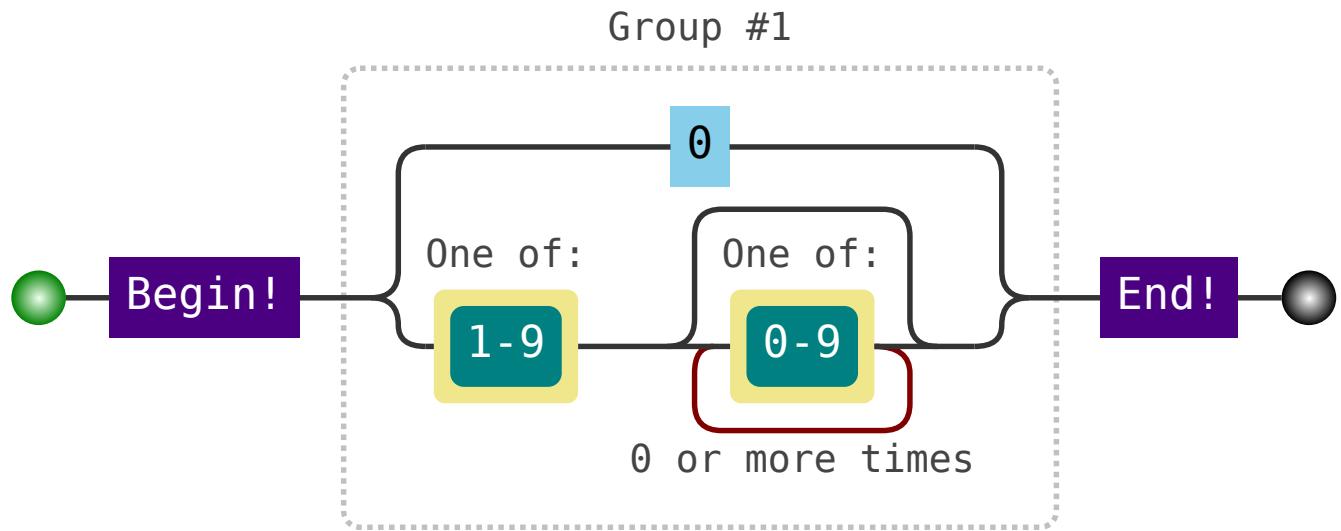
Une expression régulière est exprimée par une suite de meta-caractères, exprimant :

- * une *position* pour le motif
 - ^ : début de chaîne
 - \$: fin de chaîne
- * un caractère pour lui-même ;
 - . : n'importe quel caractère
- * 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

RegExp: `/^(0|[1-9][0-9]*)$/`



<http://jex.im/regulex/>

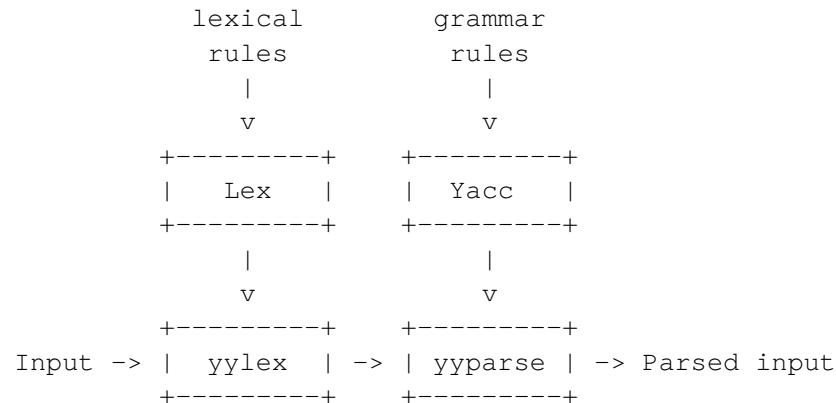


Un analyseur lexical ou «lexer» :

- ▷ 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» :



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'**épilogue** : 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.



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 :

```
$ 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).

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

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



```
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:

```
$ ./mon_parseur
```

```
1
```

```
toto
```

```
t0
```

```
1t0
```

```
12t
```

En sortie:

```
NOMBRE
```

```
MOT
```

```
MOT
```

```
MOT
```

```
NOMBRE
```

```
MOT
```

```
NOMBRE
```

```
t
```

Question : pourquoi un t se retrouve en sortie ?



```
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 différenciables.



```
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 \
9 \
10 \
11 ;
12 \
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 « \backslashbackslash ».



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

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 }
```



- * 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 supprimés 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" | hexdump
0000000 0065
0000001
```

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

```
□ — xterm —
$ echo -n "é" | hexdump
0000000 a9c3
0000002
```

Le caractère «é» est codé sur deux octets.

```
□ — xterm —
echo -n "電腦" | hexdump
0000000 e9 9b bb e8 85 a6
0000006
```

L'idéogramme «» est codé sur 3 octets. Les deux symboles signifiant «ordinateur».



Consommation mémoire et codage UTF8

- 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.*

Passage à un codage étendu sur 8 bits

- ▷ 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.

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 | hexdump  
0000000 00e9  
0000001
```

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

24

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 :

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

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

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



Rappels sur les grammaires

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 α de terminaux qui peut constituer la partie droite d'une règle $A \rightarrow \alpha$ alors il est possible de remplacer α 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 γ de terminaux et de non-terminaux.
- 4) lorsqu'une partie α de γ 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 .



Rappels sur les grammaires

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.



Rappels sur les grammaires

Explications :

- *Colonne de gauche* : représente le contenu courant de γ 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 γ , dans ce cas on précise la règle utilisée ;
 - ◊ lecture du prochain terminal.

Pile	Partie de p non traitée	Règle appliquée (si réduction)
	$id * id + id$	<i>lecture</i>
id	$*id + id$	$F \rightarrow id$
F	$*id + id$	$T \rightarrow F$
T	$*id + id$	<i>lecture</i>
$T *$	$id + id$	<i>lecture</i>
$T * id$	$+id$	$F \rightarrow id$
$T * F$	$+id$	$T \rightarrow T * F$
T	$+id$	$E \rightarrow T$
E	$+id$	<i>lecture</i>
$E +$	id	<i>lecture</i>
$E + id$		$F \rightarrow id$
$E + F$		$T \rightarrow F$
$E + T$		$E \rightarrow E + T$
E		



Rappels sur les grammaires

28

Utilisation de la pile

Sur le tableau précédent, la chaîne courante γ se trouve dans une pile :

- * l'extrémité gauche de γ est en bas de la pile ;
- * l'extrémité droite de γ est en haut de la pile.

Cette approche est pratique :

- ▷ une réduction s'effectue sur une partie droite de γ , ce qui se traduit par une modification d'une partie du sommet de la pile ;
- ▷ une lecture ajoute un symbole terminal à droite de γ , 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 !



Rappels sur les grammaires

Exemple de problème :

Sur l'exemple précédent :

Pile	Partie de p non traitée	Règle appliquée (si réduction)
	$id * id + id$	<i>lecture</i>
id	$*id + id$	$F \rightarrow id$
F	$*id + id$	$T \rightarrow F$
T	$*id + id$	<i>lecture</i>

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	$*id + id$	<i>lecture</i>
$E*$	$id + id$	<i>lecture</i>
$E * id$	$+id$	$F \rightarrow id$
$E * F$	$+id$	$T \rightarrow T * F$

Bloqué !



Les «parsers» LR

30

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 dérivation 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

31

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

32

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 :

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

La table pour un analyseur SLR(1) :

État	+	*	<i>id</i>	()	\$	<i>E</i>	<i>T</i>	<i>F</i>
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

34

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

- ◊ $j := \text{état_au_sommet_de_la_pile}$; /*état courant de l'analyseur */
- ◊ $a := \text{prochain_symbole_terminal_non_traité}$;
- ◊ si ($\text{table}[j, a] = sm$) alors
 - * symbole a et état m sont empilés;
 - * prochain_symbole_terminal_non_traité := symbole_qui_suit(a);
- ◊ si ($\text{table}[j, a] = rm$) alors
 - * $B := \text{partie_gauche_de_la_règle_de_production}(m)$; /* $B = \text{symbole non-terminal}$ */
 - * $L := \text{longueur_de_la_partie_droite_de_la_règle_de_production}(m)$
 - * on dépile $2L$ éléments de la pile; /* L paires (symbole, état) */
 - * $k := \text{état_au_sommet_de_la_pile}$; /* après avoir dépiler */
 - * symbole B et état $\text{table}[k, B]$ sont empilés;
- ◊ si ($\text{table}[j, a] = OK$)
 - * 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	$id * id + id$	<ul style="list-style-type: none"> – symbole \$ rajouté à la fin du texte ; – 0 est empilé ;
0	$id * id + id$$	<p>$table[0, id] = s5$ donc :</p> <ul style="list-style-type: none"> – on empile id et 5 ; – on avance au symbole suivant (shift) ;
0 id 5	$*id + id$$	<p>$table[5, *] = r5$ et règle(5) $F \rightarrow id$ donc :</p> <ul style="list-style-type: none"> – on dépile 2 symboles (5 et id) ; – on empile F et 3 (car $table[0, F] = 3$) ;
0 F 3	$*id + id$$	<p>$table[3, *] = r4$ et règle(4) $T \rightarrow F$ donc :</p> <ul style="list-style-type: none"> – on dépile 2 symboles (3 et F) ; – on empile T et 2 (car $table[0, T] = 2$) ;
0 T 2	$*id + id$$	<p>$table[2, *] = s7$ donc :</p> <ul style="list-style-type: none"> – on empile * et 7 ; – on avance au symbole suivant (shift) ;
0 T 2 * 7	$id + id$$	<p>$table[7, id] = s5$ donc :</p> <ul style="list-style-type: none"> – on empile id et 5 ; – 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) $F \rightarrow id$ donc: – on dépile 2 symboles (5 et id) ; – on empile F et 10 (car $table[7, F] = 10$) ;
0 $T 2 * 7 F 10$	+id\$	$table[10, +] = r3$ et règle(3) $T \rightarrow T * F$ donc: – on dépile 6 symboles ($10, F, 7, *, 2$ et T) ; – on empile T et 2 (car $table[0, T] = 2$) ;
0 $T 2$	+id\$	$table[2, +] = r2$ et règle(2) $E \rightarrow T$ donc: – on dépile 2 symboles (2 et T) ; – on empile E et 1 (car $table[0, E] = 1$) ;
0 $E 1$	+id\$	$table[1, +] = s6$ donc: – on empile + et 6 ; – on avance au symbole suivant (shift) ;
0 $E 1 + 6$	$id\$$	$table[6, id] = s5$ donc: – on empile id et 5 ; – on avance au symbole suivant (shift) ;
0 $E 1 + 6 id 5$	\$	$table[5, \$] = r5$ et règle(5) $F \rightarrow id$ donc: – on dépile 2 symboles (5 et id) ; – on empile F et 3 (car $table[6, F] = 3$) ;



Exemple de table

38

Pile	texte restant à traiter	Commentaires
$0 E 1 + 6 F 3$	\$	$table[3, \$] = r4$ et règle(4) $T \rightarrow F$ donc: – on dépile 2 symboles (3 et F); – on empile T et 9 (car $table[6, T] = 9$);
$0 E 1 + 6 T 9$	\$	$table[9, \$] = r1$ et règle(1) $E \rightarrow E + T$ donc: – on dépile 6 symboles (9, T , 6, +, 1 et E); – 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 utilisées

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

40

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

41

La gestion du «if-then-else» :

Soient les règles :

1. $Si_Inst \rightarrow IF\ Expr\ THEN\ Si_Inst$
2. $Si_Inst \rightarrow IF\ Expr\ THEN\ Si_Inst\ ELSE\ Si_Inst$

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 $IF\ Exp2\ THEN\ Inst1 \Rightarrow$ cela aura pour effet d'associer le ELSE au premier IF
- ▷ un décalage sur ELSE \Rightarrow 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 ambiguë.



- 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.



5 Analyseur lexical : un avant goût de grammaire ?

43

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

Utilisation de «commutation entre conditions»

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 état à l'autre (entre NORMAL et CHAINE).

Dans le source Lex, on préfixe les expressions régulières par une condition 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 condition correspond à l'état 0.

Toute règle sans condition est active tout le temps.

Une règle préfixée avec la condition <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 appele 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 ambiguës dans Lex :

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

Ces deux règles sont ambiguës 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. parmi les expressions régulières donnant 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| '.*' /* reconnaît une chaîne */
```

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

Il faut alors réécrire en :

```
1| ' [^\n]*' /* reconnaît une chaîne */
```

Ce qui évite le problème.

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



Il est possible d'utiliser des règles Lex ambiguës, 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 constitutionnel avec la première règle et avec la seconde.

Pour faire de la cryptanalyse fréquentielle, on peut chercher le nombre de digramme présent 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

47

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, «`mes_lexemes.h`», commun 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 définit pour chaque lexème une valeur entière pour la valeur à retourner de chaque 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;
```

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



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 `yyval` 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» :

```
1| #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 :

```
1| %union
2| {   int nombre;
3|     char *chaine; }
```

- ◊ ou sous forme de `struct` :

```
1| %union
2| { struct {   int nb;
3|                 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 `yyval` 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 :

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

On peut utiliser la déclaration suivante pour «`yyval`» :

```
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-terminal 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 $A : U_1 \ U_2 \ \dots \ U_N \ {actions}$, où :
 - ◊ A est un symbole non terminal ;
 - ◊ U_i 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 U_i .
- lorsqu'aucune action n'est précisée, YACC génère l'action $\boxed{\$\$=\$1;}$.

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

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`:

```
$ 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` :

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

- la procédure de compilation suivante :

```
□ — 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}    { yyval=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_synthétique.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(); }
```

Attention

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

Création d'un Makefile

```

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.

```

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(yylen+1,
10                      sizeof(char));
11                      strcpy(m,yytext);
12                      yyval.texte=m;
13                      return(token_MOIS);
14          }
15 [0-9]{1,2} {   yyval.valint=atoi(yytext);
16                      return(token_JOUR);
17          }
18 [0-9]{4}   {   yyval.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
```

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}?{exposant}?
10 %%
11 {blancs} /* On ignore */
12 {reel}    { yyval.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);
```

- ➊ ⇒ on inclus le fichier renommé par le «Makefile» vu précédemment;
- ➋ ⇒ on définit l'union ;
- ➌ ⇒ qui contient le type flottant, que l'on associe aux terminaux (token) et non terminaux (type) ;
- ➍ ⇒ on utilise l'union dans le fichier lex.

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



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 afficher 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 entraîner une erreur complète de la structure où elle se produit.

- α 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 entraîne le passage à l'instruction suivante, α 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 ;
- titres ;
- images, liens hypertextes,
- tableaux ;
- paragraphes ;
- etc.

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

Un fichier XML paraît 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

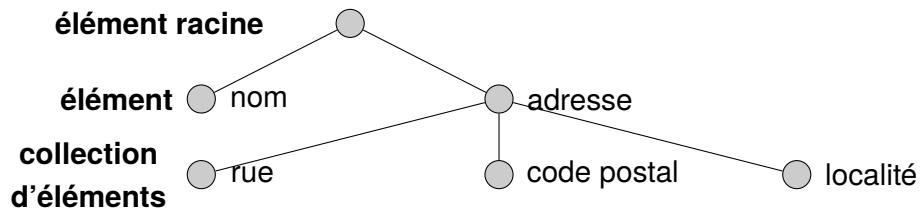
Un document XML est représenté comme un **arbre d'éléments**.

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'intérêt 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)>
```

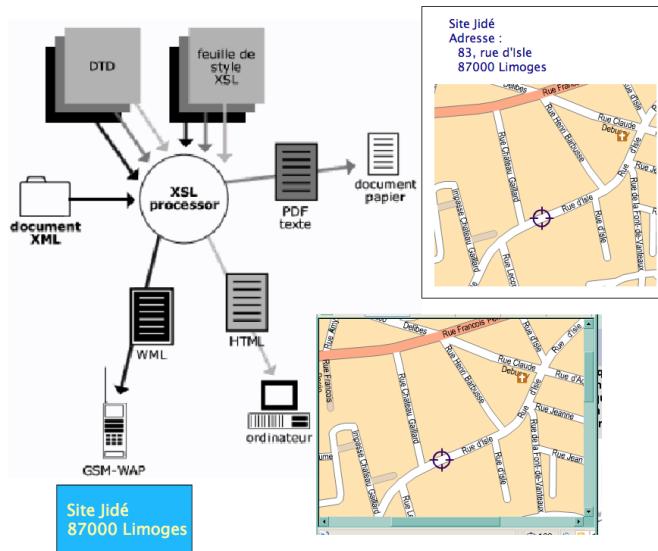
L'élément **adresse** comprend trois éléments distincts: la rue, le code postal et la localité.

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.



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 fonctionnalités 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
 - ◊ ... <I> </I> est interdit
 - ◊ ... <I> ... </I> ... 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     <nom>Binks</nom></h1>
6     <prenom>JarJar</prenom>
7     </img>
8   </individu>
9 </body>
```

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

- ▷ habituelles dans HTML (<h1>, , ...)
- ▷ 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).

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.



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>
6Cette 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">
```



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
 doit s'utiliser de la façon suivante :
</BR> ou

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>.

```
<!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  
    noms 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">
```



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 séparément 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'interpréter les instructions.

Exemple : <?php ... ?>

Résumé XML

Il y a seulement 4 balises principales :

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

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



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 class="foo">Impressive!
<b class="bar">k00l!
<i>Fantastic!
</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>
```



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>
```

- ◊ XSLT 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 `<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>
```



Lien entre éléments XML

95

```
<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>
```

```
<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>
```

