LES OUTILS LEX & YACC
PRINCIPES ET APPLICATIONS

I. Introduction

LEX (*LEXical parser*) & **YACC** (*Yet Another Compiler Compiler*) sont des *outils qui engendrent des programmes d'analyse de texte*. Les programmes générés offrent des fonctionnalités de reconnaissance, de structuration, de traduction d'un texte écrit dans un langage donné. Par exemple un programme élémentaire de reconnaissance engendré consiste à répondre à la question : "un texte donné est-il écrit dans un langage donné ?"

Afin de générer un programme de reconnaissance d'un langage, une *description formelle du langage doit être fournie au générateur d'analyseur* (Lex ou Yacc). Le schéma d'utilisation de ces outils est montré à la figure 1.

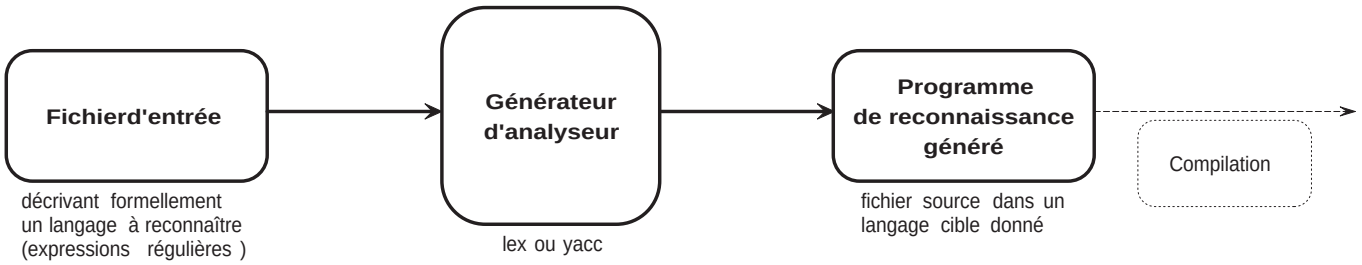

FIGURE 1 – Principe d'utilisation des outils Lex & Yacc.

Les deux outils sont complémentaires :

☞ **LEX** engendre des programmes de reconnaissance pour des langages relativement simples : les **Expressions Régulières**.

Exemple 1 : l'entrée "1.2345E-10" est une constante réelle syntaxiquement conforme à l'expression régulière : `{+|-}?{0..9}*.{0..9}*E{+|-}?{0..9}*`
(le `?` signifie "0 ou 1 fois", le `*` "0 ou `n` fois").

Exemple 2 : l'entrée "123 EST 1 chiffre" est analysée comme suit :
✏ "123" est un nombre,
✏ "EST" est un mot en majuscules,
✏ "1" est un chiffre,
✏ "chiffre" est un mot en minuscules,
où un nombre est désigné par l'expression régulière `{0..9}*`, un mot en majuscules par `{A..Z}*`, etc.

Exemple 3 : l'entrée "x2 : integer := 56 ;" (syntaxe du langage **Ada**) est analysée comme suit :

✏ "x2" est un identificateur,

✏ ":" est un symbole terminal pour la déclaration,

✏ "integer" est un symbole terminal associé au type entier,

✏ " :=" est un symbole terminal pour l'affectation,

✏ "56" est un entier.

☞ **YACC** est plus puissant que **LEX** dans la mesure où il permet de générer des analyseurs de langages descriptibles par des grammaires libres de contexte (*context-free* ou algébriques). Ces grammaires sont constituées de **règles** de la forme `A (non terminal)` $\rightarrow$ `w`.

Exemple 1 : l'entrée "((a+b)+c)" est un mot du langage :

S	$\rightarrow$	Expr
Expr	$\rightarrow$	(Expr)
		| Expr op Expr
		| Ident
Op	$\rightarrow$	+ | -
Ident	$\rightarrow$	a | b | ... | z

☞ **LEX** & **YACC** ne sont cependant pas sans limite : par exemple il n'est pas possible de reconnaître des mots de la forme "AwA" où w est un mot quelconque.

Au-delà de la simple utilisation des outils réclamant déjà une bonne expertise d'informaticien, se pose la question de comment sont conçus LEX & YACC ? Le cours d'informatique théorique donne la réponse :

☞ On sait dériver une expression régulière, c'est-à-dire calculer l'automate fini déterministe qui reconnaît un langage régulier. C'est ce que fait LEX.

☞ On sait à partir d'une grammaire trouver une dérivation la plus à droite. L'analyse qui s'en suit est une analyse ascendante. C'est (presque) ce que fait YACC.

Pourquoi utiliser des générateurs ?
Pour se concentrer sur la forme des expressions à analyser sans se soucier des détails

d'implantation. Cela évite l'écriture manuelle d'un analyseur :

- Avec les risques d'erreurs . . .
- Avec le risque d'écrire un analyseur difficile à maintenir, à faire évoluer . . .
- Avec les aspects pénibles propres aux entrées/sorties d'un langage particulier . . .
- Avec les aspects répétitifs incontournables (si je lis un 'a' alors si je lis un 'b' alors etc.).

Les avantages de ces outils automatiques sont :

- Simplicité d'utilisation : le style est déclaratif, c'est-à-dire basé uniquement sur la description formelle du langage à reconnaître.
- Facilité à maintenir : on peut à tout moment enrichir la description du langage.
- Facilité d'introduire des actions/traitements à effectuer pour engendrer un *traducteur* : on sépare facilement les règles d'une grammaire et les actions sémantiques éventuelles associées à chacune des règles.

Lorsqu'on parle de LEX et YACC, il s'agit de générateurs qui produisent du code en langage **C/C++**. Il existe aussi des générateurs produisant d'autres langages (ex. : Aflex et Ayacc génèrent de l'**Ada**), les idées sont strictement les mêmes et les données d'entrée tout-à-fait similaires.

II. LEX

II.1. Principe et Applications

À partir de la description d'un langage (ensemble d'expressions régulières), **LEX** permet de générer un programme fonctionnant selon le principe montré à la figure 2.

Le **langage** LEX est le langage utilisé pour la description des expressions régulières.
Le **compilateur** LEX est l'outil **pour produire le code source de l'analyseur**.
Les applications principales de LEX sont :

- Analyse statique : vérifications diverses.
- Formateur : mise en page de texte.
- Traducteur / compilateur.
- Interprète.

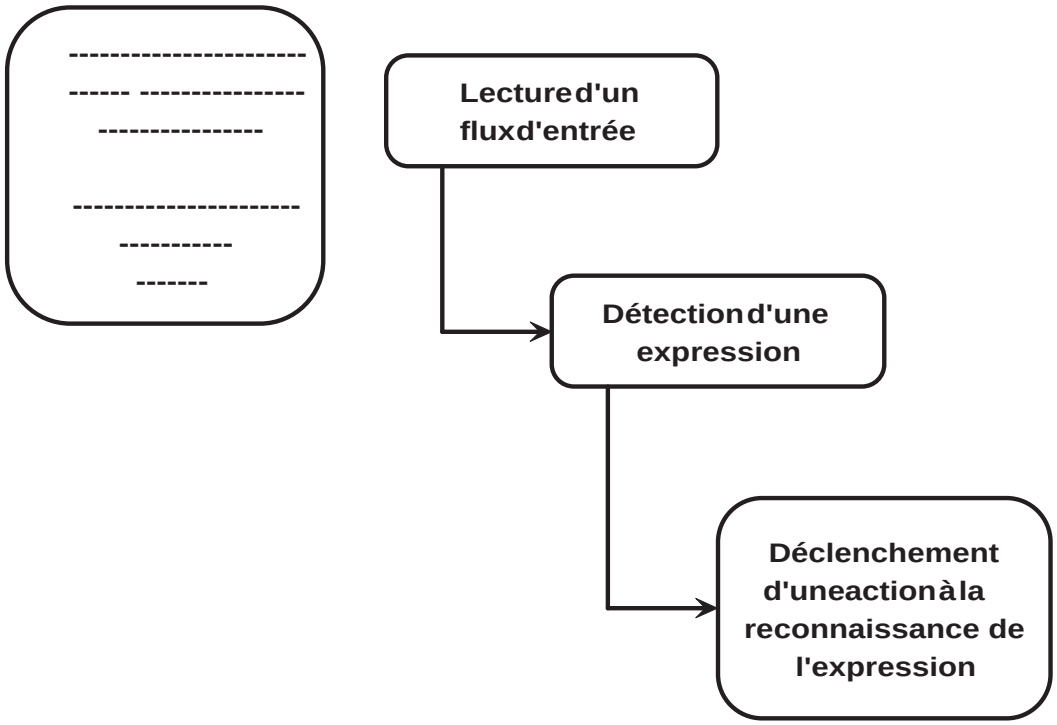

FIGURE 2 – Principe de fonctionnement du générateur d'analyseur **LEX**.

II.2. Les expressions régulières pour LEX

Pour écrire des expressions régulières, on distingue les caractères de simple texte des caractères (ou opérateurs) génériques permettant de définir des répétitions, des présences optionnelles, etc.

Les opérateurs sont :
" \ { } [ ] ∧ $ < > ? . * + | ( ) /

La signification des opérateurs est la suivante :

x	Un caractère x sauf un opérateur
"x"	Un caractère x même si c'est un opérateur
\x	Un caractère x même si c'est un opérateur
^x	Un caractère x en début de ligne
x$	Un caractère x en fin de ligne
x+	1 ou plus occurrences de x
x∗	0 ou n occurrences de x
x?	Un x optionnel (0 ou 1 occurrence)
(x)	Un x
.	Tout caractère sauf newline (\n)
x | y	x ou y
[xy]	Le caractère x ou le caractère y
[x-z]	Un caractère pris dans la suite de caractères de x à z
[^x]	Tout caractère sauf x
{macro}	Substitution d'une macro-définition (voir plus loin)

On peut aussi utiliser des caractères spéciaux dans les expressions régulières :

\n Newline

\b Backspace

\t Tabulation

II.3. La structure du fichier d'entrée pour LEX

Le nom du fichier LEX est suffixé par `.l` : `exemple.l`.
On trouve trois sections dans un fichier LEX :

```
Section des définitions
%%
Section des règles
%%
Section définie par l'utilisateur
```

☞ Dans la section des **règles** apparaissent des expressions régulières ainsi que des **actions C++ associées**.
Exemple :

```
[a-zA-Z_]([a-zA-Z0-9_])*    { cout << "C'est un identificateur\n"; }
[0-9]+                      { cout << "C'est un entier\n"; }
```

☞ Dans la section des **définitions** peuvent apparaître des macro-définitions (on dit "macros") qui servent à nommer des expressions pour simplifier les définitions des Expressions Régulières.

Exemple : les expressions ci-dessus peuvent être simplifiées en utilisant les macros suivantes :

```
LETTRE      [a-zA-Z_]
CHIFFRE     [0-9]
%%
{LETTRE}({LETTRE}|{CHIFFRE})*     { cout << "Identificateur\n"; }
{CHIFFRE}+                        { cout << "Entier\n"; }
```

Vocabulaire : on dit que l'expression régulière **filtre** (***matche***) un mot ou une chaîne de caractères.

Il peut y avoir des ambiguïtés (deux filtrages possibles à partir des expressions régulières définies). Dans ce cas deux règles s'appliquent pour décider :

➪ La chaîne la plus longue est filtrée.

➪ Si les deux chaînes filtrées sont de même longueur, la première règle définie (dans l'ordre du fichier LEX) est appliquée.

☞ Dans la section définie par l'utilisateur, on place des variables, fonctions, ... nécessaires au fonctionnement du programme.

Voici un exemple complet de fichier LEX nommé `cplusplus.l` et destiné à reconnaître les différentes *unités lexicales* qu'on peut trouver dans un code source en C++ (simplifié) :

```
%{
  #include <iostream>
  #include <cstdlib>
  #include <cmath>
  using namespace std;
%}
CHIFFRE [0-9]
IDENT [_a-zA-Z][_a-zA-Z0-9]*
TYPE "char"|"short"|"int"|"float"|"double"
MOT_CLE "if"|"else"|"while"|"for"|"do"|"cout"|"cin"|"main"|"return"|"endl"
%%
{CHIFFRE}+              { cout<<"Un entier : "<<yytext
                              <<"(" << atoi(yytext)<<")\n";
                        }
```

```
{CHIFFRE}+"."{CHIFFRE}* { cout<<"Un flottant : "<<yytext
                               <<"(" << atof(yytext)<<")\n";
                        }

{MOT_CLE}               { cout<<"Un mot-clé : "<<yytext<<'\n'; }

{TYPE}                  { cout<<"Un type : "<<yytext<<'\n'; }

{IDENT}                 { cout<<"Un identificateur : " <<yytext<<'\n'; }

"+"|"-"|"*"|"/"|"%"|"="|"<"|"<="|">"|">=" {
                          cout<<"Un opérateur : "<<yytext<<'\n'; }

"<<"|">>"               { cout<<"un opérateur de flux : "<<yytext<<'\n'; }

"{"|"}"|"("|")"|"]"|"[" { cout<<"Un caractère encadrant : "
                               <<yytext<<'\n'; }

";"                     { cout<<"Un caractère fin d'instruction : "
                               <<yytext<<'\n'; }

"{"[^}\n]*"}"           { /* eat up one-line comments */ }

[ \t\n]+                { /* eat up white space */ }

.                       { cout<<"Caractère inconnu : "<<yytext<<'\n'; }
%%
int main(int argc, char *argv[])
{
  ++argv, --argc;  /* skip over program name */
  if (argc > 0)
    yyin = fopen(argv[0], "r");
  else
    yyin = stdin;

  yylex(); // Analyse lexicale
} // main()
```

L'étape de **génération** de l'analyseur se fait en tapant (sous Linux) :

```
$ lex -l cplusplus.l
```

L'analyseur généré s'appelle par défaut `lex.yy.c` (même si c'est du C++).
On le compile en tapant (sous Linux) :

```
$ g++ lex.yy.c -o cplusplus -lfl
```

`cplusplus` est le nom du programme **exécutable** (pas d'extension sous Linux). On s'en sert en lui donnant un nom de fichier texte à analyser, par exemple `cplusplus.txt` qui contient le code suivant :

```
int main()
{ int var = 0;
  while (var <= 10)
  {
    cout << var << endl;
  }
  return 0;
}
```

Pour lancer l'analyse du fichier ci-dessus appelé `cplusplus.txt`, on tape :

```
$ ./cplusplus cplusplus.txt
```

ou par redirection du clavier vers un fichier :

```
$ ./cplusplus < cplusplus.txt
```

III. YACC

Le principe de **YACC** est le même que celui de **LEX** : générer un analyseur de texte. Mais l'outil est plus puissant.

Par exemple **YACC** est capable de reconnaître le *langage des expressions arithmétiques entières bien parenthésées* défini par la grammaire libre de contexte suivante :

Expr	$\rightarrow$	Entier
		| Expr OperateurBinaire Expr
		| '(' Expr ')'
		| OperateurUnaire Expr
OperateurBinaire	$\rightarrow$	'+' | '-' | '*' | '/'
OperateurUnaire	$\rightarrow$	'+' | '-'
Entier	$\rightarrow$	Chiffre | Entier Chiffre
Chiffre	$\rightarrow$	'0' | '1' | ... | '9'

Les symboles **terminaux** s'écrivent entre quotes ('). Les **non-terminaux** commencent par une majuscule. L'**axiome** est le premier non-terminal apparaissant dans la grammaire (ici Expr).

Le format d'un fichier YACC `exemple.y` est le suivant :

```
Section des définitions
%%
Section des règles
%%
Section définie par l'utilisateur
```

☞ La section des **définitions** permet de placer des déclarations utiles au programme ou à la grammaire.

Parmi les non-terminaux, on peut spécifier l'axiome de la grammaire par :

```
%start Expr
```

En l'absence d'une clause `start`, c'est le non-terminal à gauche de la première règle de la grammaire qui est l'axiome.

On peut aussi dans cette section associer un *type de données* aux non-terminaux :

```
#define YYSTYPE int
```

☞ La section des **règles** permet d'écrire les *règles de production* de la grammaire du langage à reconnaître. La forme générale d'une règle est :

```
Non-Terminal : Liste_de_Symboles (Terminaux ou non) ;
```

Exemples et simplifications :

```
A : B C D ;
A : E F ;      peut s'écrire :   A : B C D
A : G ;                            | E F ;
                                   | G
                                   ;
```

À chaque règle on peut ajouter des actions :

```
N : B C D ;     { cout << "action !" << endl; }
A : x y ;       { count++; }
```

Ces actions (en code **C**) peuvent utiliser des attributs affectables (automatiquement) à chaque symbole :

$$ représente l'attribut de l'élément en partie gauche de la règle (non-terminal) ;

$n représente l'attribut du nième symbole de la partie droite.

```
A : B C D ;     { $$ = 1; }
Expr : Expr '+' Expr ; { $$ = $1 + $3; }
```

☞ Dans la section définie par l'utilisateur, il faut fournir le programme principal qui appelle le *parser* c'est-à-dire l'analyseur engendré par YACC ; une procédure `yyparse()` sans argument est générée.

Pour fonctionner, YACC a besoin des *unités lexicales* ou *tokens* retournés par Lex (grâce à la fonction `yylex()`).

Par exemple dans un langage informatique, les **mots** sont décrits par des expressions régulières : mots-clés, identificateurs, constantes, ... et les **constructions** (blocs d'instructions, Si Alors Sinon, Tant que, Pour, ...) sont décrites par des grammaires.

☞ **LEX** fait l'**analyse lexicale**,

☞ **YACC** fait l'**analyse syntaxique**.

On peut donc sans se soucier du *lexique* reprendre la définition de la grammaire des expressions arithmétiques parenthésées :

Expr	:	Entier
		| Expr OperateurBinaire Expr
		| ParentheseOuvrante Expr ParentheseFermante
		| OperateurUnaire Expr
OperateurBinaire	:	Mult | Div | OperateurUnaire

et spécifier dans la section des définitions du fichier **YACC** que les *léxèmes* analysés par la fonction `yylex()` sont : Entier, ParentheseOuvrante, ParentheseFermante, Mult, Div et OperateurUnaire. Ce qui se note dans la syntaxe Yacc :

```
%token Entier ParentheseOuvrante ParentheseFermante
%token Mult Div OperateurUnaire
```

Voici un exemple complet avec dans l'ordre : le fichier LEX, le fichier YACC puis le fichier du texte à analyser.

Fichier **LEX** (`ab_lex.l`) :

```
%{
/* fichier ab_lex.l */
#include <iostream>
#include "y.tab.h"
%}

CHIFFRE [0-9]
LETTRE  [a-zA-Z_]
%%
("PROCEDURE"|"procedure")       { return (ADAPROC); }
("BEGIN"|"begin")               { return (ADABEGIN); }
("END"|"end")                   { return (ADAEND); }
("IS"|"is")                     { return (IS); }
"("                             { return '('; }
")"                             { return ')'; }
"DECLARATION"                   { return (DECLARATION); }
\n                              { ; }
[\t ]+                          { ; }
.                               { std::cout << " autre ? "; }
%%
```

Fichier **YACC** (`ab.y`) :

```
%token ADAPROC ADABEGIN ADAEND IS '(' ')'
%token DECLARATION

%start axiom

%{ /* fichier ab.y */
#include <iostream>
#include <cstdio>
using namespace std;

#define YYSTYPE int     /* le type des $$ $1 $2 etc. */

extern FILE *yyin;
extern char yytext[];
%}

%%
axiom : pgm             { cout<<"--------------------------------\n";
                          cout<<"Nombre de procédures : "<<$1<<'\n';
                          cout<<"--------------------------------\n"; }
```

```
      ;
pgm   : sspgm pgm       { $$ = $2 + 1; }
      |                 { cout << "Fin de source\n"; $$ = 0; }
      ;

sspgm : entete ADABEGIN ADAEND
      ;

entete: ADAPROC         { cout<<"Lex vient de detecter une UL PROC : ";
                          cout<<yytext << '\n'; }
       '(' DECLARATION ')' IS
      ;
%%
int main(int argc, char *argv[])
{ ++argv, --argc;

  if (argc > 0) yyin = fopen(argv[0], "r");
  else          yyin = stdin;

  yyparse();

  return 0;
} // main()

int yyerror(char *msg) { fprintf(stderr, "%s\n", msg); return 0; }
```

Fichier contenant le texte à analyser (`entree_ab.txt`) :

```
procedure ( DECLARATION ) is
BEGIN
END

PROCEDURE (DECLARATION ) IS
begin
end

PROCEDURE ( DECLARATION)IS BEGIN END
```

La génération de l'analyseur se fait en trois étapes (sous Linux) :

```
$ yacc -d ab.y
$ lex -l ab_lex.l
```

```
$ g++ y.tab.c lex.yy.c -o ab -lfl
```

Le nom de l'exécutable est `ab`. On lance l'analyse du fichier `entree_ab.txt` en tapant :

```
$ ./ab entree_ab.txt
```

ou bien

```
$ ./ab < entree_ab.txt
```