# Notes de cours lex et yacc

## Introduction

### Concept d'un parser : objectifs de lex et de yacc

- Un parser lit un fichier texte respectant une syntaxe et produit une structure de donnée en mémoire.
- Éventuellement un parser peut définir un interpréteur de commande. Dans ce cas le fichier d'entrée est le clavier et quand il reconnaît une commande, il ne construit pas de structure de données mais il appelle une commande.
- Le travail du parser est décomposé en 2 parties :
    1. reconnaître les mots (token) -> lex / flex
    2. reconnaître les phrases (règles) -> yacc / bison
- si on n'est pas contraint par le format de fichier on peut utiliser xml

### Liens pour une documentation externe

- Lex & Yacc Essentiel L'essentiel des outils est rappelé ici, à lire absolument.
- Lex & Yacc Compact Un peu de théorie, et un exemple classique. Document à lire pour comprendre un peu comment ces outils fonctionnent.
- Lex Official LE manuel officiel de flex.
- Yacc Official LE manuel officiel de Yacc.

## lex

### Qu'est-ce un token ?

- un type de mot défini par une expression régulière
    - entier : -?[0-9]+
    - identificateur : [a-zA-Z][a-zA-Z0-9_]*
    - mot clé : BEGIN, END, [eE][nN][dD]
    - ponctuation : +

### lex est un générateur de code C

- fichier .l -> (flex) -> fichier .c
- fonction int yylex() lit le fichier FILE *yyin rend un numéro représentant le type (un int) du token reconnu. Par exemple pour entier on rend 300, pour un identificateur on rend 263, pour le mot clé BEGIN on rend 402, etc...
- par convention pour les tokens monocaractère ('+', ':', ...) yylex rend le code ascii.
  Pour les autres on rend un nombre à partir de 257.
  256 est utilisé comme code d'erreur.
- La valeur du token est rendu dans la variable yylval dont le type est une union de types.
  L'appelant de yylex() sait quel type a la variable yylval en fonction du type rendu par yylex().

### Syntaxe d'un fichier lex

```
%{
head user code          #include, global variables
%}
definitions              ENTIER [0-9]+
%%
rules                     regex action
%%
tail user code          main, encapsulation de yylex(), yywrap()
```

- l'ordre des règles est important car en cas ambiguïté la première est choisie.

### Actions associés aux règles

- quand lex reconnaît un token il exécute l'action correspondante
- le buffer yytext[] contient les caractères composant le token.
- on affecte yylval.type avec quelque chose qui dépend de yytext[]
- on rend un numéro

### Compilation d'un fichier lex

- si on l'utilise seul:

```
$ flex -t vst.l > vst.yy.c
$ gcc -Wall -Werror -o scanner vst.yy.c -lfl
```

- libfl.a yywrap()

## yacc

### Définition

- grammaire LALR : Look Ahead Left Recursive
- *Look Ahead Left to right scanning of the input constructing a Rightmost derivation in reverse with 1 look ahead token.*
- variante de Backup Naur Form (John Backup et Peter Naur Algol60)

### qu'est-ce qu'une règle ?

- nom : listes de TOKENs et de noms de règles
- le nom est  "un élément non terminal de la grammaire"
- les TOKEN sont des éléments terminaux
- règles linéaires : définies en fonction d'autres règles/tokens avec plusieurs alternatives possibles
- regles récursives: définies en fonction d'elle-même, il y a nécessairement des alternatives.
  La récursion peut être directe ou à travers d'autres règles.

## yacc est aussi un générateur de code C

- fichier .y -> (bison) -> fichier .c + fichier.h
- yyparse() qui utilise yylex() pour vider le fichier yyin et qui produit comme effet de bord la construction d'une structure de donnée représentant le contenu du fichier.
- yyparse rend 0 si ok, 1 en cas d'erreur de syntaxe.
- le fichier .h contient des déclarations de numéro de token et le type union de la variable yylval que doit utiliser lex.
- en cas d'erreur yyparse() appelle la fonction yyerror(char *)

## Comment exprimer une règle récursive ?

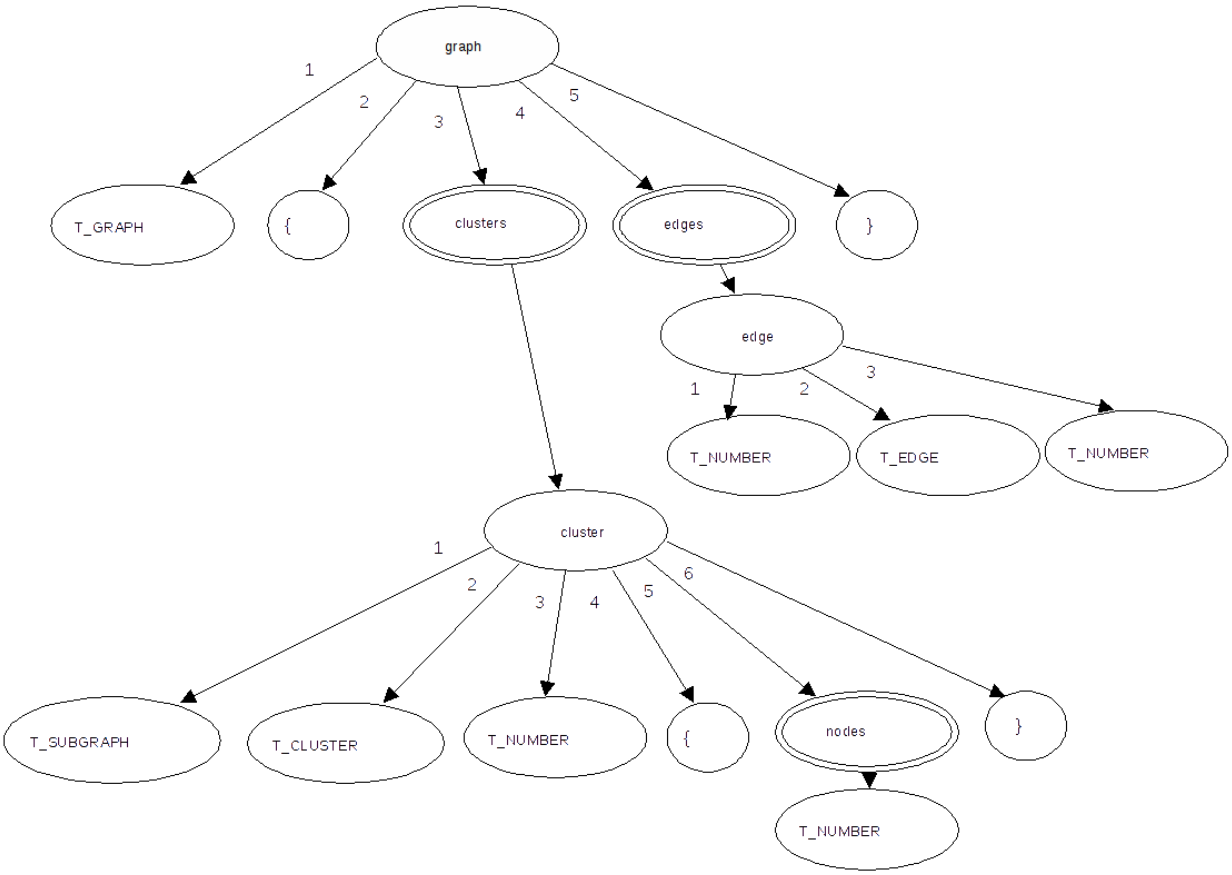
- récursion à gauche (la récursion à droite est moins efficace)

```
/* liste = ele1 ele2 ... eleN avec N > 0 */
liste    : ele
         | liste ele

/* liste = ele1 ele2 ... eleN avec N >= 0 */
liste    : /* rien */
         | liste ele

/* liste = ele1, ele2, ... eleN avec N > 0 */
liste    : ele
         | liste ',' ele

/* liste = ele1; ele2; ... eleN; avec N >= 0 */
liste    : ele ';'
         | liste ele ';'
```




- shift : on avance dans le fichier
- reduce on reconnait une règle
- shif/reduce conflit c'est quand on ne sait pas s'il faut avancer ou reconnaitre : grammaire ambigue
- reduce/reduce c'est quand on peut reconnaitre deux règles : grammaire fausse

## Syntaxe d'un fichier yacc

```
%{
head user code          #include, global variables
%}
definitions             %token, %union, %left, %start, %type
%%
rules                   règles : action
%%
tail user code          main, encapsulation de yyparse()
```

## Actions associées aux règles.

- La valeur associées aux éléments (terminaux/token ou non terminaux) est accessibles par des variables spéciales $<num> où <num> est le numéro d'ordre dans la règle.
- On peut associer une valeur à une règle par la variable spéciale $$

### Appel de yacc et compilation du code.

```
bison -t -d vst.y
gcc -Wall -Werror -o parser vst.tab.c vst.yy.c -lfl
```

- l'option -v permet de voir l'automate pour le debug