Principes d'Interprétation des Langages

L2 MPI

Université Paris-Sud

CHAPITRE 6 : BASES D'ANALYSE ASCENDANTE ET DE YACC

Analyse ascendante

- On construit l'arbre de dérivation du bas (feuilles) vers le haut (racine)
- On lit le mot en entrée et on s'autorise deux types d'opérations :
  - Décalage (shift) : passer à la lettre suivante.
  - Réduction (reduce) : on "réduit" une chaîne (une suite consécutive de lettres de $A \cup V$ à gauche de la lettre courante et finissant à la lettre courante) en la remplaçant par un non-terminal selon une des règles de la grammaire.

Analyse ascendante : exemple

- Grammaire :

  S → F=E

  F → x

  E → E'+E' | E'*E'

  E' → 1|2|3

- Mot à analyser : x=2+3

Analyse ascendante : exemple

S → F=E
F → x
E → T+T | T*T
T → 1|2|3

x=2+3 r (réduction)
F=2+3

Analyse ascendante : exemple

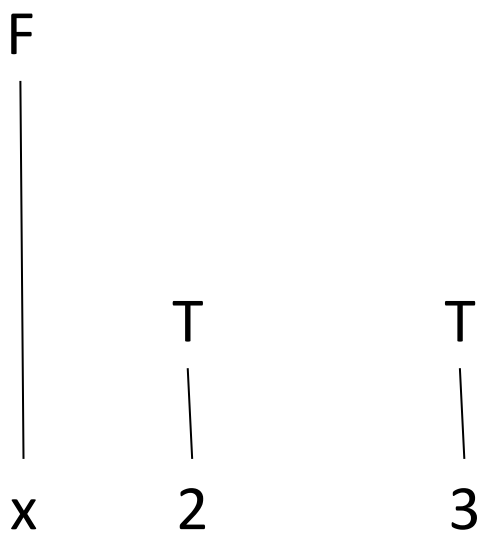
S → F=E
F → x
E → T+T | T*T
T → 1|2|3

x=2+3	r	(réduction)
F=2+3	d	(décalage)
F=2+3	d	
F=2+3	r	
F=T+3		

Analyse ascendante : exemple

S → F=E
F → x
E → T+T | T*T
T → 1|2|3

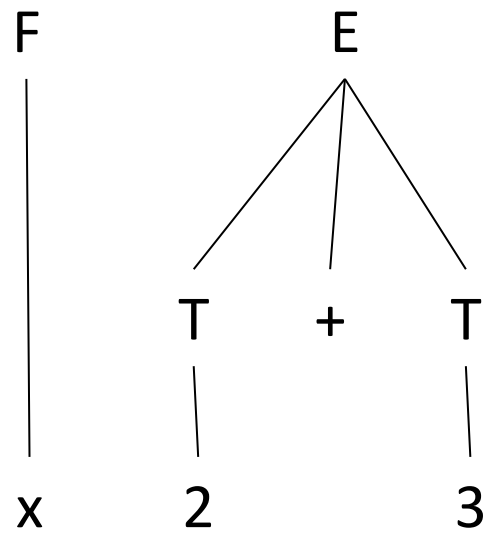
x=2+3	r	(réduction)
F=2+3	d	(décalage)
F=2+3	d	
F=2+3	r	
F=T+3	d	
F=T+3	d	
F=T+3	r	
F=T+T		



Analyse ascendante : exemple

S → F=E
F → x
E → T+T | T*T
T → 1|2|3

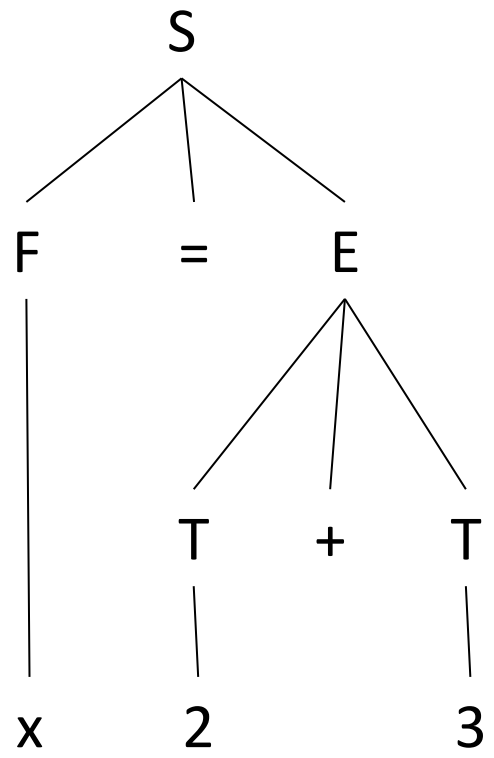
x=2+3	r	(réduction)
F=2+3	d	(décalage)
F=2+3	d	
F=2+3	r	
F=T+3	d	
F=T+3	d	
F=T+3	r	
F=T+T	r	
F=E		



Analyse ascendante : exemple

S → F=E
F → x
E → T+T | T*T
T → 1|2|3

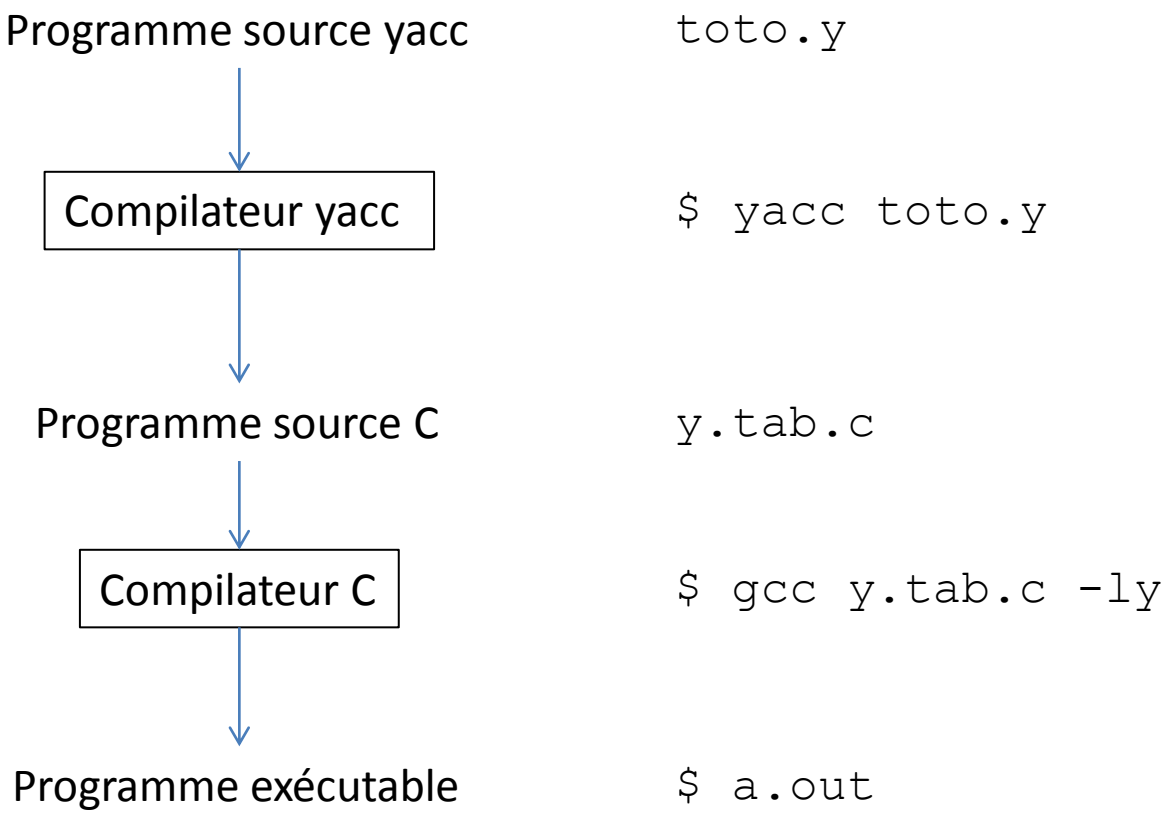
x=2+3	r	(réduction)
F=2+3	d	(décalage)
F=2+3	d	
F=2+3	r	
F=T+3	d	
F=T+3	d	
F=T+3	r	
F=T+T	r	
F=E	r	
S		gagné !



Analyse ascendante et Yacc

- C'était un exemple extrêmement simple d'analyse syntaxique LR. [L : Left-to-right scanning » ; R : Rightmost derivation in reverse.]
- Dans le cas général, l'analyse LR requiert la construction de tables, comme pour l'analyse LL.
- Toutes les grammaires ne sont pas LR. La classe LR (comme la classe LL) sont des sous-classes de la classe des grammaires non contextelles.
- Yacc permet de construire automatiquement des analyseurs LR.

Yacc : schéma d'utilisation



Structure d’un programme yacc

```
Déclarations

%%

Règles de traduction

%%

Fonctions auxiliaires
```

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F              {$$ = $1 * $3;}
       | F ;
F : '(' E ')'             {$$ = $2;}
       | CHIFFRE ;
```

```
%%
       yylex() {
         int c;
         c = getchar();
         if (isdigit(c)) {
           yylval = c-'0';
           return CHIFFRE;
         }
         return c;
       }
```

Partie 1 : Déclarations

- Cette partie peut comprendre
  - des déclarations de variables et de constantes en C, délimitées par « %{« et « %} » (les délimitateurs sont toujours en début de ligne).
  - des déclarations d'unités lexicales (tokens) de la grammaire.

- (Cette partie peut aussi être vide.)

Partie 2 :Règles de traduction

- Chaque règle est formée :
  - D'une règle de la grammaire,
  - De l'action sémantique associée, c'est-à-dire des actions à exécuter chaque fois qu'une réduction est faite par la règle de la grammaire.
- Chaque symbole de la grammaire peut avoir une valeur, cette valeur peut être utilisée dans l'action sémantique :
  - $$ désigne la valeur associée au symbole non terminal de la partie gauche de la règle.
  - $i désigne la valeur associée au ième symbole (terminal ou non terminal) de la partie droite.

Règles de traduction : exemple

- E : E '+' T {$$ = $1 + $3;}

  Lorsqu'on fait une réduction de E+T en E, la valeur du E de gauche devient la somme des valeurs du E et du T de droite.

- E : T

  Si on n'écrit pas de règle sémantique, par défaut l'action {$$ = $1} est exécutée.

Partie 3 : Fonctions auxiliaires

- Cette partie doit contenir au moins une fonction d'analyse lexicale, nommée yylex().
- Son travail est de renvoyer des tokens.

- Cette fonction peut être fournie par lex, lorsque le programme yacc fait appel à un programme lex (ce qui est souvent le cas).

Partie 3 : Fonctions auxiliaires

- La fonction `yylex()` est la fonction « standard » d'analyse lexicale. Son rôle est de lire le texte en entrée standard lettre par lettre et d'appliquer les règles définies dans la partie 2 du programme lex.

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F              {$$ = $1 * $3;}
       | F ;
F : '(' E ')'             {$$ = $2;}
       | CHIFFRE ;
```

```
%%
       yylex() {
         int c;
         c = getchar();
         if (isdigit(c)) {
           yylval = c-'0';
           return CHIFFRE;
         }
         return c;
       }
```

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F              {$$ = $1 * $3;}
       | F ;
F : '(' E ')'             {$$ = $2;}
       | CHIFFRE ;
```

```
%%
       yylex() {
         int c;
         c = getchar();
         if (isdigit(c)) {
           yylval = c-'0';
           return CHIFFRE;
         }
         return c;
       }
```

E → E+T | T
T → T*F | F
F → (E) |1|2|3

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F             {$$ = $1 * $3;}
       | F ;
F : '(' E ')'            {$$ = $2;}
       | CHIFFRE ;
```

```
%%
      yylex() {
        int c;
        c = getchar();
        if (isdigit(c)) {
          yylval = c-'0';
          return CHIFFRE;
        }
        return c;
      }
```

E → E+T | T
T → T*F | F
F → (E) |1|2|3

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F             {$$ = $1 * $3;}
       | F ;
F : '(' E ')'            {$$ = $2;}
       | CHIFFRE ;
```

```
%%
       yylex() {
         int c;
         c = getchar();
         if (isdigit(c)) {
           yylval = c-'0';
           return CHIFFRE;
         }
         return c;
       }
```

E → E+T | T
T → T*F | F
F → (E) |1|2|3

Une règle supplémentaire pour indiquer que l'expression s'arrête à la fin de la ligne (donc expression sur une ligne seulement)

Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F             {$$ = $1 * $3;}
       | F ;
F : '(' E ')'            {$$ = $2;}
       | CHIFFRE ;
```

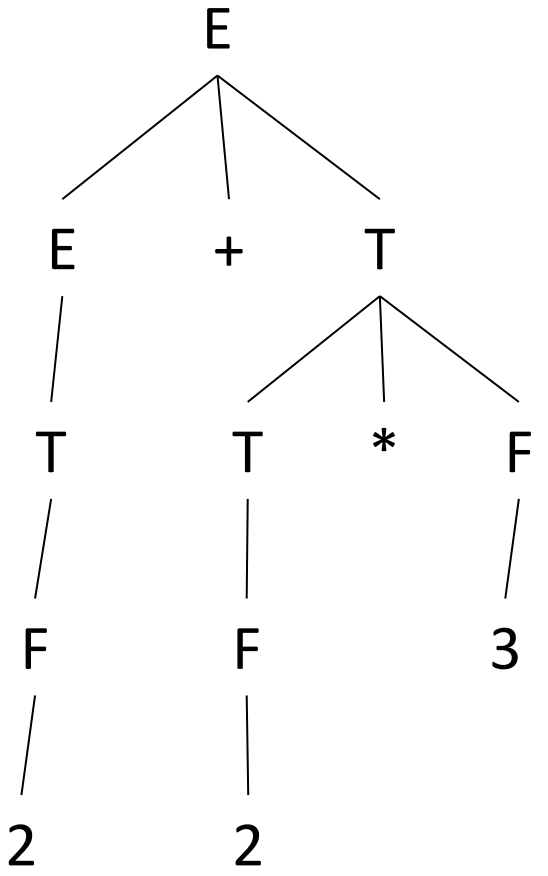
```
%%
       yylex() {
         int c;
         c = getchar();
         if (isdigit(c)) {
           yylval = c-'0';
           return CHIFFRE;
         }
         return c;
       }
```

E → E+T | T
T → T*F | F
F → (E) |1|2|3

Le programme calculera la valeur numérique de l'expression arithmétique contenue dans le fichier passé en entrée.

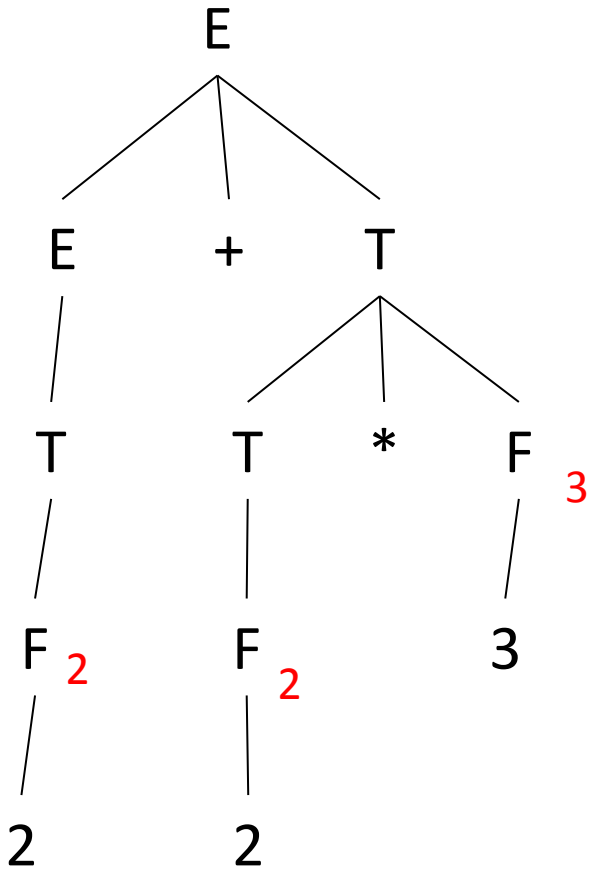
Transmission des valeurs

2+2*3



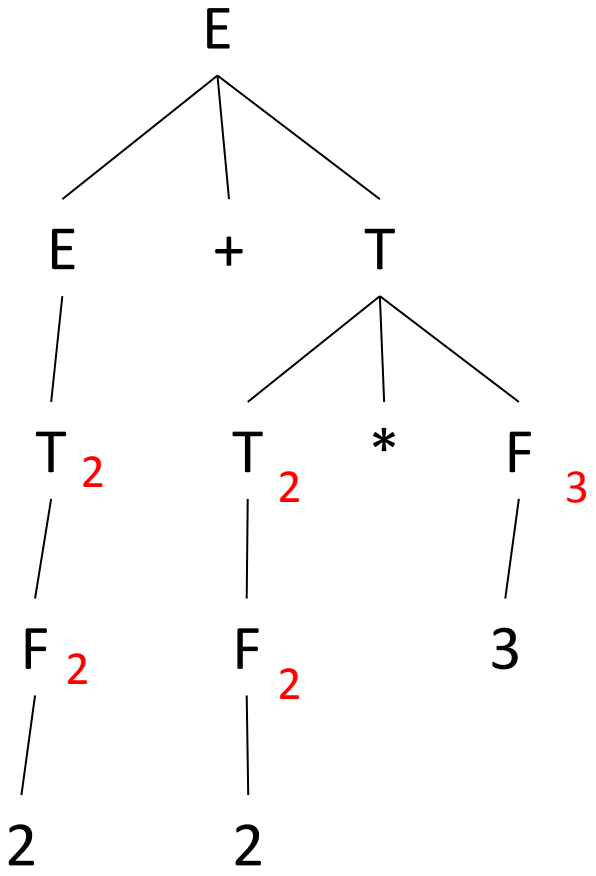
Transmission des valeurs

2+2*3



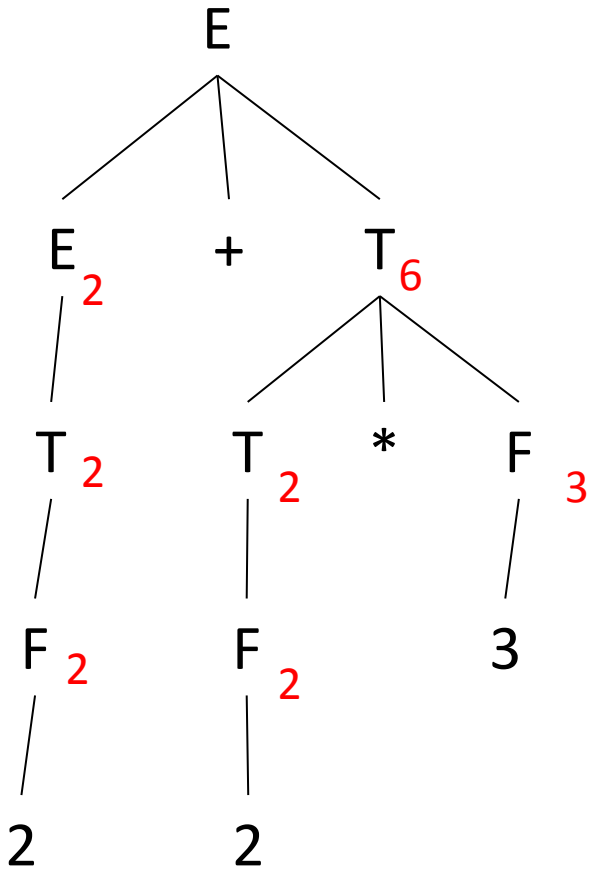
Transmission des valeurs

2+2*3



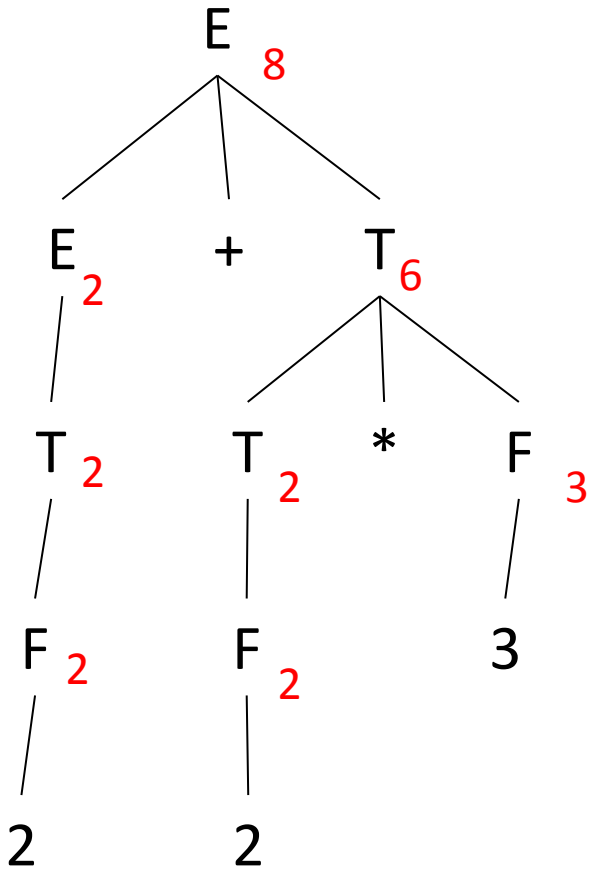
Transmission des valeurs

2+2*3



Transmission des valeurs

2+2*3



Un programme yacc

```
%{
#include <ctype.h>
%}

%token CHIFFRE

%%
ligne  : E '\n'          {printf("%d\n",$1);} ;
E  : E '+' T             {$$ = $1 + $3;}
       | T ;
T  : T '*' F             {$$ = $1 * $3;}
       | F ;
F : '(' E ')'            {$$ = $2;}
       | CHIFFRE ;
```

```
%%
      yylex() {
        int c;
        c = getchar();
        if (isdigit(c)) {
          yylval = c-'0';
          return CHIFFRE;
        }
        return c;
      }
```

E → E+T | T
T → T*F | F
F → (E) |1|2|3

Utilisation de la bibliothèque C contenant la fonction isdigit()

Un programme yacc

```
%%
    yylex() {
      int c;
      c = getchar();
      if (isdigit(c)) {
        yylval = c-'0';
        return CHIFFRE;
      }
      return c;
    }
```

- C'est la fonction d'analyse lexicale. Elle lit le flot d'entrée et retourne, pour chaque unité lexicale, le *token* correspondant, et sa valeur le cas échéant.
- Dans ce cas la fonction est simple car chaque unité lexicale contient un seul caractère.
- la fonction lit le caractère courant.
  - Si c'est un chiffre elle renvoie le *token* CHIFFRE et sa valeur. La variable **yylval**, réservée à yacc, sert à stocker cette valeur pour la transmettre à l'analyseur syntaxique.
  - Si ce n'est pas un chiffre, elle renvoie simplement le caractère lu, il est considéré comme un *token*.

Alliance de yacc et lex

- La fonction yylex() peut être fournie par un programme lex (c'est généralement le cas).
- Si on a un programme lex qui fournit les tokens et leurs valeurs associées, il suffit d'écrire dans la 3ème partie du programme yacc :

```
#include "lex.yy.c"
```

(où lex.yy.c est le nom du programme C obtenu en compilant le programme lex)

Alliance de yacc et lex

- Pour compiler le tout : supposons que le programme lex s'appelle toto.l et le programme yacc titi.y.

```
$ lex toto.l
$ yacc titi.y
$ gcc y.tab.c –ly -ll
```