

# Corrigé TD 4

## Yacc / Bison

### Utilisation de Yacc sur des grammaires simples

## 1 Exercice 1: Une grammaire simple

---

Pour les questions 1 à 6, chaque solution tient dans un fichier unique.

### 1.1 Version de base

Dans cette version, on a ajouté des actions sur chacune de règles pour voir les réduction opérées (ce n'était pas demandé). Par ailleurs, on a rajouté un non terminal, pour pouvoir afficher un message lorsque l'analyse est correcte.

Le fichier yacc [exo1\\_1.y](#) est donné ci-dessous:

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

void yyerror(const char* msg);
int yylex(void);
}%

%%
G:      S                { printf("Analysis OK\n"); }
      ;

S:      S 'a'            { printf("Règle S -> Sa (r1)\n"); }
      | S 'x'            { printf("Règle S -> x  (r2)\n"); }
      | S 'y'            { printf("Règle S -> y  (r3)\n"); }
      ;

%%
int main() { return yyparse(); }

void yyerror(const char* msg) { fprintf(stderr, "Error: %s\n", msg); }

int yylex(void) {
    int c;

    do c = getchar(); while (isspace(c));
    return c;
}
```

Avec cette version, on affichera:

- le message "Analysis OK" en fin de fichier si on entre un mot conforme à la grammaire (i.e. un mot de la forme  $(x|y)a^n$ ).
- le message d'erreur "Error: Syntax error" si le mot entré n'est pas correct.

## 1.2 Etats de l'automate

Pour cette question, on compile le fichier précédent avec la commande `bison` et les options

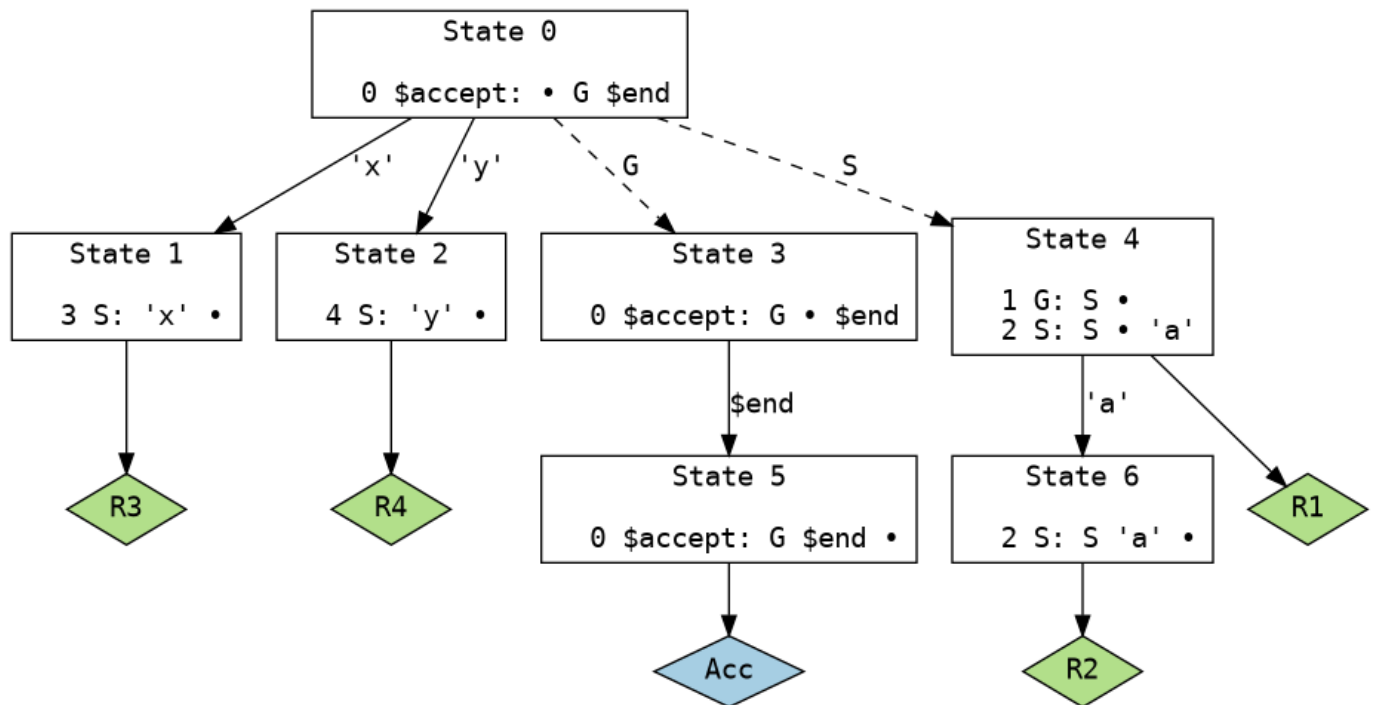
- `-g` pour avoir un graphe de l'automate
- `-v` (équivalent à `--report=state`) pour avoir un rapport sur les états de l'automate.

Ainsi, pour la grammaire **G**, on obtient un automate où on retrouve ce que l'on avait vu dans le TD précédent:

- la notation «point»
- les décalages représentés par une arête dont la valeur est la valeur du token en fenêtre.
- les réductions sont symbolisées par des losanges colorés.

La numérotation des états est un peu différente de ce que l'on avait, car la méthode de construction n'est pas tout à fait identique, mais vous devriez retrouver «vos petits».

Pour la grammaire **G**, on a :



## 1.3 Suite de mots

Il suffit ici de modifier la règle **G** pour que l'on reconnaisse une suite de mots de **G** séparés par des `'\n'`. Cette règle devient:


```
G:      G S '\n'      { printf("Analyse OK\n"); }
      |      /* empty */
      ;
```

Bien sûr, il faut maintenant que le lexical laisse «échapper» le caractère `'\n'` puisque c'est devenu désormais un terminal.


La fonction `yylex` devient donc:

```
int yylex(void) {
    int c;

    do c = getchar(); while (c == ' ' || c == '\t');
    return c;
}
```

Fichier `ex01_3.y`  complet.

## 1.4 Affichage d'un prompt

Pour cette question, on écrit une fonction `prompt` qui se charge de l'affichage du prompt. Cette fonction est appelée quand on a vu une fin de ligne, c'est à dire dans l'action associée à la règle G. Quand au numéro de ligne, il est incrémenté dans le lexical .

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

    void yyerror(const char* msg);
    int yylex(void);
    void prompt(void);

    int lineno = 1;
}%

%%

G:          G S '\n'          { printf("Analysis OK\n"); prompt(); }
    |      /* empty */
    ;

S:          S 'a'             { printf("Règle S -> Sa (r1)\n"); }
    |      'x'                 { printf("Règle S -> x (r2)\n"); }
    |      'y'                 { printf("Règle S -> y (r3)\n"); }
    ;

%%

int main() {
    prompt();
    return yyparse();
}

void yyerror(const char* msg) { fprintf(stderr, "Error: %s\n", msg); }

int yylex(void) {
    int c;

    do
        c = getchar();
    while (c == ' ' || c == '\t');

    if (c == '\n') lineno += 1;
    return c;
}

void prompt(void) {
    printf("[%d] ", lineno);
    fflush(stdout);
}
```

## 1.5 Messages d'erreurs

Dans cet exercice, seule la fonction `yyerror` change. Elle devient:

```
void yyerror(const char* msg) {  
    // Ici le token est toujours un caractère (càd un entier < 256)  
    // puisque nous n'avons pas encore utilisé de directive %token  
    fprintf(stderr, "Error: line %d. Token: %c", lineno, yychar);  
    fprintf(stderr, " %s\n", msg);  
}
```

## 1.6 Récupération d'erreurs

Ici on essaie de se «rattraper» sur une fin de ligne. Il suffit donc de modifier un peu la règle sur `G`. Celle-ci devient:

```
G:      G S '\n'      { printf("Analyse OK\n"); prompt();}  
      | error '\n'    { yyerrok; }  
      | /* empty */  
      ;
```

Ainsi si une erreur se produit, *yacc* sautera tout le texte jusqu'à une fin de ligne et pourra reprendre l'analyse normalement 📄.

## 1.7 Version finale: lex + yacc

Dans cette version, on utilise deux fichiers:

- un fichier de définitions pour *lex/flex* ( `exo1_7.1` ) et
- un fichier de définitions pour la grammaire du langage ( `exo1_7.y` ).

Pour compiler le tout, il faut faire:

```
$ bison -g -v -d -o exo1_7.c exo1_7.y  
$ flex -o exo1_7.lex.c exo1_7.1  
$ gcc -o exo1_7 exo1_7.c exo1_7.lex.c
```

La première ligne permet de construire l'analyseur syntaxique dans `exo1_7.c`, mais aussi, grâce à l'option `-d`, le fichier `exo1_7.h`.

On peut ensuite (seconde ligne) construire l'analyseur lexical (qui importe bien sûr le fichier `exo1_7.h`).

Enfin, la dernière ligne construit l'exécutable `exo1_7` à partir des deux fichiers C précédents.

**Note:** On pourra voir ici une version assez complète de la fonction `yyerror`.

[Source lex](#) 📄:

```

%{
    #include <stdio.h>
    #include "exo1_7.h"
%}
%option noyywrap
%option yylineno

%%

[ \t]          { /* skip */ }
"\n"           { return '\n'; }
[aA]           { return TOK_A; }
[xX]           { return TOK_X; }
[yY]           { return TOK_Y; }
.              { return yytext[0]; } /* cela provoquera une erreur syntaxique */
%%

```

Source yacc: 

```

%{
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

void yyerror(const char* msg);
int yylex(void);
void prompt(void);

#define YYERROR_VERBOSE 1
extern int yylineno, yylval;
}%

%token TOK_A TOK_X TOK_Y

%%
gram:      gram s '\n'      { printf("Analysis OK\n"); prompt(); }
        |   error '\n'      { yyerrok; }
        |   /* empty */
        ;

s:         s TOK_A          { printf("Règle S -> Sa (r1)\n"); }
        |   TOK_X          { printf("Règle S -> x  (r2)\n"); }
        |   TOK_Y          { printf("Règle S -> y  (r3)\n"); }
        ;

%%
int main() {
    prompt();
    return yyparse();
}

void yyerror(const char* msg) {
    fprintf(stderr, "Error: line %d. Token: ", yylineno);
    // afficher la valeur de yychar. L'affichage se fait sous forme de
    // caractère si yychar est < 256, sous la forme d'une chaîne
    // (après conversion) sinon
    if (yychar < 256)
        fprintf(stderr, "'%c' (as int: 0x%x)", yychar, yychar);
    else
        switch(yychar) {
            case TOK_A: fprintf(stderr, "'A' ou 'a'"); break;
            case TOK_X: fprintf(stderr, "'X' ou 'x'"); break;
            case TOK_Y: fprintf(stderr, "'Y' ou 'y'"); break;
        }
    fprintf(stderr, "\nmsg='%s'\n", msg);
    prompt();
}

void prompt(void) {
    printf("[%d] ", yylineno);
    fflush(stdout);
}

```

Un exemple d'utilisation de notre programme est donné ci-dessous;

```
[1] xaAaA
Règle S -> x (r2)
Règle S -> Sa (r1)
Règle S -> Sa (r1)
Règle S -> Sa (r1)
Règle S -> Sa (r1)
Analyse OK
[2] A
Error: line 2. Token: 'A' ou 'a'
msg='syntax error, unexpected TOK_A, expecting $end or TOK_X or TOK_Y'
[2]
```

## 2 Exercice 2: Expressions

### 2.1 Une première version

Cette version est proche de celle du cours. Elle est constituée de deux fichiers (un fichier *lex* et un fichier *yacc*).

Le fichier *lex* :

```
%{
    #include <stdio.h>
    #include "exo2_etf.h"
}%

%option noyywrap
%option yylineno

%%
[ \t]      { /* skip */ }
[0-9]+     { yylval = atoi(yytext); return NUMBER; }
.|\n      { return yytext[0]; }
%%
```

La grammaire utilisée ici est basée sur la grammaire vue en cours.

Le fichier *yacc*  est donc:

```

%{
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

void yyerror(const char* msg);
int yylex(void);
void prompt(void);

#define YYERROR_VERBOSE 1
extern int yylineno, yylval;
}%

%token NUMBER

%%

lines:      lines expr '\n'      { printf("==> %d\n", $2); prompt(); }
          | lines '\n'          // pour permettre des lignes vides
          | error '\n'          { yyerrok; }
          | /* empty */
          ;

expr:       expr '+' term        { $$ = $1 + $3; }
          | expr '-' term        { $$ = $1 - $3; }
          | term
          ;

term:       term '*' sfactor      { $$ = $1 * $3; }
          | term '/' sfactor      { if ($3) $$ = $1 / $3;
                                   else fprintf(stderr, "division by 0\n"); }
          | sfactor
          ;

sfactor :   '-' factor           { $$ = -$2; }
          | factor               { $$ = $1; }
          ;

factor:     '(' expr ')'         { $$ = $2; }
          | NUMBER
          ;

%%

void yyerror(const char* msg) {
    fprintf(stderr, " %s\n", msg);
    prompt();
}

void prompt(void) {
    printf("[%d] ", yylineno);
    fflush(stdout);
}

int main() {
    prompt();
    return yyparse();
}

```

#### Remarque:

Noter l'utilisation du non terminal **sfactor** pour implémenter le **moins unaire**.



## 2.2 Utilisation des priorités dans yacc

La seconde version de notre calculatrice utilise une grammaire où les priorités sont exprimées avec les directives `%left` ou `%right`. Une telle grammaire est aussi présentée dans le cours.

La version suivante ne manipule que des nombres entiers. Le fichier de spécifications pour *lex* est identique au précédent (si ce n'est le nom du fichier `.h` qui est inclus). Quand au fichier *yacc*, il est défini ci-dessous:

```
%{
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

void yyerror(const char* msg);
int yylex(void);
void prompt(void);

#define YYERROR_VERBOSE 1
extern int yylineno, yylval;
}%

%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS

%%

lines:      lines expr '\n'      { printf("==> %d\n", $2); prompt(); }
          | lines '\n'          // pour permettre des lignes vides
          | error '\n'          { yyerrok; }
          | /* empty */
          ;



expr:       expr '+' expr        { $$ = $1 + $3; }
          | expr '-' expr        { $$ = $1 - $3; }
          | expr '*' expr        { $$ = $1 * $3; }
          | expr '/' expr        { if ($3) $$ = $1 / $3;
                                else fprintf(stderr, "division by 0\n"); }
          | '(' expr ')'         { $$ = $2; }
          | '-' expr %prec UMINUS { $$ = -$2; }
          | NUMBER
          ;

%%

void yyerror(const char* msg) {
    fprintf(stderr, " %s\n", msg);
    prompt();
}

void prompt(void) {
    printf("[%d] ", yylineno);
    fflush(stdout);
}



int main() {
    prompt();
    return yyparse();
}
```

- Source *lex* 
- Source *yacc* 

## 2.3 Version avec des nombres réels

Cette dernière version est identique à la précédente, si ce n'est que les nombres manipulés sont des réels en double précision.

Au niveau du code, la seule différence est l'utilisation de la macro `YYSTYPE` qui permet de changer le type de la variable `yylval` (et bien sûr le passage de `%d` à `%g` pour l'affichage du résultat).

- Source *lex* 
- Source *yacc* 

## 3 Exercice 3: ETF → XML

La difficulté dans cet exercice tient principalement à la construction de chaînes de caractères formatées. Par exemple,

- pour le nombre `2` il faut construire la chaîne `<number>2</number>`
- pour le produit `2*3` la chaîne à construire est `<mul op='*'> <number>2</number> <number>3</number> </mul>`

### 3.1 Première implémentation (allocation statique)

Le plus simple (mais pas le plus joli) est de déclarer l'attribut d'une expression comme un chaîne de taille fixe et de construire le résultat avec la primitive `snprintf`. On obtient donc des règles qui ressemblent à:

```
expr:  expr '+' term    { snprintf($$, MAXBUF, "<add op='+'>%s</add>", $1, $3) ; }
      |  expr '-' term    { snprintf($$, MAXBUF, "<add op='- '>%s</add>", $1, $3) ; }
      |      term        { $$ = $1; }
      ;
```

Ici, on va avoir des objets qui ont des attributs de type différents. En effet, quand on rencontre un entier l'attribut qui nous intéresse est de type entier, alors qu'il est de type chaîne quand on a une `expr`. On va donc passer par un `%union` et on devra typer les différents objets que l'on manipule (puisque'ils ne sont plus tous du même type). Pour le type `%union`, nous avons

```
%union {
    char attr[MAXBUF];
    int val;
}
```

Pour le typage des objets, nous avons

```
%token <val>  NUMBER
%type  <attr> etf expr term factor
```

Ainsi, si on a une action sémantique comme

```
factor: ...
      NUMBER { snprintf($$, MAXBUF, "<number>%d</number>", $1); }
```

*bison* sait que quand on parle de `$1`, il faut aller chercher sa valeur dans le champ `val` (et donc un `int`), et que quand on parle de `$$` on s'adresse au champ `attr` de l'attribut que l'on construit (une chaîne).

#### Remarque:

Les versions récentes de `gcc` sont capables d'inférer la taille maximale de la chaîne qui peut être construite par un `snprintf` et la compilation de la règle précédente provoque pas mal de warnings. On pourra ici éviter la production de ces warnings (pourtant légitimes) avec l'option `-Wno-format-truncation` de `gcc`.

Une solution complète avec cette technique:

- [Source lex](#) 📄
- [Source yacc](#) 📄

## 3.2 Seconde implémentation (allocation dynamique)

L'inconvénient de notre première implémentation est que chaque nœud de l'arbre occupe beaucoup de place et que le résultat final ne peut pas faire plus de `MAXBUF` caractères. L'idéal serait d'avoir un résultat alloué dynamiquement qui fasse juste la bonne taille (l'attribut serait alors non plus un tableau de caractères de taille fixe, mais un pointeur sur un caractère).

Pour cela nous allons définir la fonction `make_string` suivante (une telle fonction apparaît dans la page de manuel de `vsnprintf`, que vous pouvez consulter pour plus de détails):

```
char *make_string(const char *fmt, ...) {
    int size = 0;
    char *p = NULL;
    va_list ap;

    /* Determine required size */
    va_start(ap, fmt);
    size = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (size < 0)
        return NULL;

    size++;          /* For '\0' */
    p = malloc(size);
    if (p == NULL)
        return NULL;

    va_start(ap, fmt);
    size = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (size < 0) {
        free(p);
        return NULL;
    }

    return p;
}
```

Cette fonction nous permet donc d'avoir une fonction de construction de chaînes de caractères dont le résultat est construit dynamiquement.

Si on reprend la règle précédente, on obtient:

```
expr:  expr '+' term    { $$= make_string("<add op='+'>%s%s</add>", $1, $3);
                          free($1); free($3); }
      |  expr '-' term    { $$ = make_string("<add op='-'>%s%s</add>", $1, $3) ;
                          free($1); free($3); }
      |  term             { $$ = $1; }
      ;
```

**Note:** Lorsque l'attribut résultat est constitué, nous n'avons plus besoin des attributs qui ont été synthétisés par les non terminaux en partie droite de règle. La mémoire qu'ils utilisent peut donc être libérée.

Les sources de notre implémentation:

- Source *lex* (identique au précédent)
- Source *yacc*

Pour information, le graphe des états pour analyser cette grammaire est donné ci dessous:

