TD 6 Yacc

1 Musique

On veut développer un analyseur Lex Yacc qui permette de décoder un fichier de notes de musique sous la forme :

`duree note duree note ...`

Il faut au moins une paire `{duree note}` dans le fichier. La durée est décrite par `{R, B, N}`, la note par `{do, re, mi, fa, sol, la, si}`.

1. écrire la grammaire correspondante,
   Correction
   Grammaire :
   $$G1 = (\{S, G\}, \{n, d\}, S, \{S- > ndG, G- > ndG|\lambda\})$$

2. définir les tokens Yacc à utiliser,
   Correction
   Tokens : note, duree

3. donner le code de l'analyseur Lex (avec les en-têtes),
   Correction
   Sources complètes avec l'exercice suivant. Ici les calculs et le switch/case avec le yylval ne sont pas utiles.

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

#include "y.tab.h"

extern int yylval;
%}

NOTE ("do"|"re"|"mi"|"fa"|"sol"|"la"|"si")
DUREE ("R"|"B"|"N")

%%
{NOTE} {
                //printf("lex->note %s\n", yytext);
                return note;
                }
{DUREE} {
                //printf("lex->duree %s\n", yytext);
                switch (yytext[0]) {
                        case 'R':
                                yylval=4;
                                break;
                        case 'B':
                                yylval=2;
                                break;
                        case 'N':
                                yylval=1;
                                break;
                        };
                        return duree;
                        }

[ |\t] ;
```

```
\n return 0;
. return yytext[0];
%%
```

4. donner le code de l'analyseur Yacc, l'action sémantique terminale est seulement un `printf ("ok\n")` pour le moment.
   Correction

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

void yyerror(char * s);
%}

%token note
%token duree

%start P

%%
P : duree note G {$$ = $1 + $3; printf("total %d\n",$$);}
G : duree note G {$$ = $1 + $3;}
        | {$$=0;};

%%

int main()
{
        yyparse();
        return 0;
}

void yyerror(char *s)
{
        fprintf(stderr, "erreur %s\n", s);
}
```

On associe des valeurs numériques aux durées `{4, 2, 1}` pour `{R, B, N}` respectivement. On veut calculer la durée totale des notes.

1. mettre en place cette correspondance durée/valeurs numériques,
   Correction
   Dans la partie Lex, lorsque {DUREE} est reconnue, on évalue la valeur numérique (switch/case sur yytext[0]) pour donner une valeur à yylval
2. comment transporter une valeur de Lex à Yacc ? Faut-il modifier la grammaire ? Y a t'il un encodage particulier à utiliser ?
   Correction
   on utilise yylval. Attention de type entier par défaut (cf #define YYSTYPE). Rien à changer dans la grammaire.
3. modifier les sources Lex et Yacc.
   Correction
   cf. sources précédentes

On veut maintenant ajouter des nombres devant chaque paire `{duree note}` pour tenir compte d'une répétition, par exemple : "2Rdo" correspond à "RdoRdo". Ce nombre n'est pas indispensable (s'il n'y a pas de répétition). Comment faire afficher la liste des notes complètes (étendues) à la fin du décodage ? Faut-il modifier la grammaire ? Donner les sources Lex et Yacc correspondantes.

Correction

Comme on devra faire remonter la note (texte) ou la valeur (répétition ou nombre de temps), on crée une union de type. Il faut également modifier la grammaire puisque la répétition peut être vide. Attention enfin à indicer yyval par le type de la donnée. **Lex** :

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

#include "y.tab.h"

%}

NOTE ("do"|"re"|"mi"|"fa"|"sol"|"la"|"si")
DUREE ("R"|"B"|"N")
ENTIER [1-9][0-9]*

%%
{NOTE} {
                //printf("lex->note %s\n", yytext);
                strcpy(yylval.chaine, yytext);
                return note;
                }
{DUREE} {
                //printf("lex->duree %s\n", yytext);
                switch (yytext[0]) {
                        case 'R':
                                yylval.valeur=4;
                                break;
                        case 'B':
                                yylval.valeur=2;
                                break;
                        case 'N':
                                yylval.valeur=1;
                                break;
                        };
                        return duree;
                        }
{ENTIER} {
                yylval.valeur=atoi(yytext);
                return repet;
                }

[ |\t] ;
\n return 0;
. return yytext[0];
%%
```

Yacc :

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(char * s);

char listeNotes[100];
%}

%union {
        int valeur;
        char chaine[3];
```

```
        }

%token <chaine> note
%token <valeur> duree
%token <valeur> repet

%type <valeur>R
%type <valeur> G
%type <valeur> P

%start P

%%
P : R duree note G {$$ = $2*$1 + $4; for (int i=0; i<$1; ++i) strcat(listeNotes, $3); printf("
    total %d, liste %s\n",$$, listeNotes);}
G : R duree note G {$$ = $2*$1 + $4; for (int i=0; i<$1; ++i) strcat(listeNotes, $3); }
        | {$$ = 0;};
R : repet {$$ = yylval.valeur;}
        | {$$ = 1;};

%%

int main()
{
        yyparse();
        return 0;
}

void yyerror(char *s)
{
        fprintf(stderr, "erreur %s\n", s);
}
```

2 Calculatrice

On veut construire une calculatrice sur des nombres entiers, avec les opérations `{+, -, *}` et les parenthèses `{(, )}` qui permettent des sous-calculs. L'analyseur doit fournir les résultats de ces opérations numériques.

1. la grammaire proposée est :

   `G = ({E}, {ENTIER}, {E}, {E -> E + E | E * E | E - E | (E) | ENTIER})`

   Cette grammaire est-elle correcte ? Si non, pourquoi et que faudrait-il changer ?

   Correction

   La grammaire est ambigüe, plusieurs dérivations existent (pour chacune des règles). Note : la sortie de `yacc -Wcounterexamples calculatrice.yacc` donne plus d'informations (limité ici à une règle) :

```
badCalculatrice.yacc: avertissement: 9 conflits par décalage/réduction [-Wconflicts-sr]
badCalculatrice.yacc: avertissement: conflit par décalage/réduction sur le jeton plus [-
    Wcounterexamples]
  Exemple: E plus E ◦plus E
  Dérivation par décalage
    E
    ↪1: E plus E
              ↪1: E ◦plus E
  Exemple: E plus E ◦plus E
  Dérivation par réduction
    E
```

```
    ↪1: E plus E
        ↪1: E plus E ◦
```

2. définir une grammaire correcte,
   Correction

```
E -> E + U | U
U -> U - T | T
T -> T * F | F
F -> (E) | entier
```

   où entier désigne un symbole terminal.

3. écrire les sources Lex et Yacc correspondantes
   Correction
   Attention, la soustraction peut créer des nombres négatifs, qui ne seraient pas gérés par le type courant (int).

Lex :

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

#include "y.tab.h"

extern int yylval;
%}

ENTIER [1-9][0-9]*
PLUS "+"
MOINS "-"
MULT "*"
OPAR "("
FPAR ")"

%%
{ENTIER} {yylval=atoi(yytext); return entier;}
{PLUS} {return plus;}
{MOINS} {return moins;}
{MULT} {return mult;}
{OPAR} {return opar;}
{FPAR} {return fpar;}

[ \t] ;
\n return 0;
. return yytext[0];
%%
```

Yacc :

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

void yyerror(char* s);
%}

%token entier plus moins mult opar fpar
%start EXPR

%%
EXPR : EXPR plus U {$$ = $1 + $3; printf("résultat %d\n", $$);}
```

```
        | U {printf("résultat %d\n", $$);}
        ;

U : U moins T {$$ = $1 - $3; }
        | T
        ;

T : T mult F {$$ = $1 * $3; }
        | F
        ;

F : opar EXPR fpar {$$ = $2;}
        | entier {$$ = yylval;}
        ;

%%

int main()
{
        yyparse();
return 0;
}

void yyerror(char * s)
{
        fprintf(stdout, "erreur -> %s\n", s);
}
```