Licence Informatique **2018/2019**

Compilation TP Yacc

Introduction :

'lex' est un analyseur lexical, ce n'est pas un analyseur grammatical. C'est-à-dire que 'lex' ne vous permet pas de saisir une grammaire complexe d'un langage et l'analyser. C'est le rôle de 'yacc'. 'yacc' vous permet de faire ce que l'on appelle un 'parser'. Vous pouvez reconnaître tout type de langage à partir du moment où vous avez sa grammaire, exemple un fichier .ps, .html, .c, .pas, etc...

'yacc' est un analyseur grammatical qui a été spécialement conçu pour utiliser 'lex' comme analyseur lexical, ils marchent ensemble et chacun à son niveau. La partie 'lex' lit le fichier, décode des tokens et les renvoie à la partie 'yacc'. La partie 'yacc' ne lit pas le fichier en entrée mais seulement les tokens renvoyés par 'lex', au travers de la fonction 'yylex()'. La communication entre les deux se passe aux niveaux des tokens et aussi de la variable yylval' que nous verrons plus tard.

Le format :

Le format d'un fichier yacc est identique au format de 'lex', il est composé de trois parties séparées par '%%'.

```
Les définitions
%%
Les règles
%%
Le code utilisateur
```

Le format des règles :

Dans la partie des règles de 'yacc', contrairement à 'lex' où vous donnez un ensemble d'expressions régulières, vous entrez l'ensemble des règles qui définissent votre grammaire.

Exemple :

Soit une grammaire au format BNF :

<EXPR_CALCS> ::= EXPR_CALC
<EXPR_CALCS> ::= EXPR_CALCS EXPRE_CALC

<EXPR_CALC> ::= <EXPR_NUM> =

<EXPR_NUM> ::= <FACTEUR>
<EXPR_NUM> ::= <EXPR_NUM> + <FACTEUR>
<EXPR_NUM> ::= <EXPR_NUM> - <FACTEUR>

<FACTEUR> ::= <TERME>
<FACTEUR> ::= <FACTEUR> * <TERME>
<FACTEUR> ::= <FACTEUR> / <TERME>

<TERME> ::= <NB>
<TERME> ::= (<EXPR_NUM>)

Suivant cette grammaire, vous pouvez reconnaître les expressions de la forme '2+3=', ou encore '2*(5-2)/(8+4)='

Le programme 'yacc' qui reconnait les expressions et les affiche s'écrit :

```
/* -- eval.yac --
Evaluation d'une expression
Partie analyseur grammatical.
*/
%{
#include <stdio.h>
void yyerror(char *s);
extern int yylex();
%}
%token Tnb
%start EXPR_CALCS
%%
EXPR_CALCS : EXPR_CALC {printf ("EXPR_CALCS1\n");}
| EXPR_CALCS EXPR_CALC {printf ("EXPR_CALCS2\n");}
;
EXPR_CALC : EXPR_NUM '=' {printf ("EXPR_CALC ");}
;
EXPR_NUM : FACTEUR {printf ("facteur(expr num) ");}
                    | EXPR_NUM '+' FACTEUR {printf ("ADDITION ");}
                    | EXPR_NUM '-' FACTEUR {printf ("SOUSTRACTION ");}
;
FACTEUR : TERME {printf ("terme(facteur) ");}
| FACTEUR '*' TERME {printf ("PRODUIT ");}
| FACTEUR '/' TERME {printf ("DIVISION ");}
;
TERME : Tnb {printf ("ENTIER ");}
| '(' EXPR_NUM ')' {printf ("EXPR entre (). ");}
;
%%
int main(void)
{
if ( yyparse() != 0 )
{ fprintf(stderr,"Syntaxe incorrecte\n"); return 1; }
else
return 0;
}

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

La communication entre 'lex' et 'yacc' (1) : les 'tokens'

Mais qu'est-ce qu'un 'Tnb' ? Et d'où viennent les caractères des opérations ? Ils sont analysés par 'lex' et fourni à 'yacc'. Chaque élément trouvé est renvoyé tel quel (les opérateurs), soit vous renvoyez sous formes de constantes représentant un type de terminal lexical (Tnb).

Le fichier 'lex' correspondant est très simple car il ne s'occupe que de rechercher les opérations et les nombres :

```
/* -- eval.lex --
Evaluation d'une expression
Partie analyseur lexical.
*/
%{
#include "y.tab.h"
extern void yyerror(char *s);
%}
BLANC [ \n\t]
%%
[0-9]+ return Tnb;
[-+*/()=] return yytext[0]; /* caracteres unites lexicales */
{BLANC}+ ;
. fprintf(stderr, "Caractere (%c) non reconnu\n", yytext[0]);
%%
int yywrap(void) {return 1;}
```

Ce qui donne :

```
$ yacc -d eval.yac
$ lex eval.lex
$ gcc  lex.yy.c y.tab.c -lfl -o eval
$ eval
2 + 3 =
entier terme(facteur) facteur(expr num) entier terme(facteur) addition expr_calc expr_calcs1
```

La communication entre 'lex' et 'yacc' (2) : 'yylval'

Les tokens reconnus par 'lex' peuvent correspondrent à plusieurs chaînes différentes, par exemple dans le programme précédent, le token Tnb représente un nombre entier, mais il peut représenter aussi bien le nombre 2, ou bien le nombre 539... Pour réaliser un programme d'évaluation d'expression de la forme '2 + 539 =', la partie 'yacc' doit connaître la valeur des nombres entiers lus. Pour cela, en plus des tokens, la communication entre 'lex' et 'yacc' est complétée par la variable globale : 'yylval', elle remplace la variable (int value) que nous avions déclarée dans le fichier 'lex' pour l'évaluation d'une expression postfixée. Par défaut 'yylval' est un entier.

Le fichier 'lex' doit être modifié comme suit pour retourner la valeur d'un entier:

```
/* -- eval.lex --
Evaluation d'une expression
Partie analyseur lexical.
*/
%{
#include "y.tab.h"
extern void yyerror(char *s);
%}
BLANC [ \n\t]
%%
[0-9]+ yylval = atoi(yytext); return Tnb;
[-+*/()=] return yytext[0]; /* caracteres unites lexicales */
{BLANC}+ ;
. fprintf(stderr, "Caractere (%c) non reconnu\n", yytext[0]);
%%
int yywrap(void) {return 1;}
```

Dans 'yacc', les tokens, ainsi que chacune des règles, peuvent correspondre à des valeurs. Cette spécificité nous permet de faire le calcul à la volée de façon simple dans l'écriture des actions. La syntaxe est un peu particulière, elle est inspiré des arguments d'un script shell (ou perl). La valeur du premier élément de la règle est représentée par '$1', celle du deuxième est représentée par '$2' et ainsi de suite. La valeur de retour est représentée par '$$'.

Le fichier 'yacc' correspondant est :

```
/* -- eval.yac --
Evaluation d'une expression
Partie analyseur grammatical.
*/
%{
#include <stdio.h>
void yyerror(char *s);
extern int yylex();
%}
%token Tnb
%start EXPR_CALCS
%%
EXPR_CALCS : EXPR_CALC
| EXPR_CALCS EXPR_CALC
;
EXPR_CALC : EXPR_NUM '=' {printf ("%d\n", $1);}
;
EXPR_NUM : FACTEUR
| EXPR_NUM '+' FACTEUR {$$ = $1 + $3;}
| EXPR_NUM '-' FACTEUR {$$ = $1 - $3;}
;
FACTEUR : TERME
| FACTEUR '*' TERME {$$ = $1 * $3;}
| FACTEUR '/' TERME {$$ = $1 / $3;}
;
TERME : Tnb {$$ = $1;}
```

```
| '(' EXPR_NUM ')' {$$ = $2;}
;
%%
int main(void)
{
if ( yyparse() != 0 )
{ fprintf(stderr,"Syntaxe incorrecte\n"); return 1; }
else
return 0;
}

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

Dans l'exemple suivant, nous voulons étendre notre programme à l'évaluation d'expression comportant des fonctions, par exemple la fonction valeur absolue, ou négation (du signe). Le fichier 'lex' est modifié pour reconnaître un nom de fonction suivant l'expression régulière :

```
[a-zA-Z_]+      return Tid;
```

et le fichier 'yacc' contient les règles supplémentaires suivantes :

```
TERME    : Tnb                  {$$ = $1;}
        | '(' EXPR_NUM ')'       {$$ = $2;}
        | APPEL_FONC             {$$ = $1;}
        ;


APPEL_FONC : Tid '(' EXPR_NUM ')'   {$$ = ??? }
          ;
```

Le problème est de savoir quel est le nom de la fonction reconnue. Le fichier 'lex' reconnaît l'identificateur (qui est contenu dans yytext) mais ne peut renvoyer que le token (Tid). On peut s'inspirer de la méthode utilisée pour les nombres entiers, mais nous avons vu que la variable yylval était un entier. En fait, on peut changer son type et utiliser par exemple le mot clef %union de 'yacc'. Ainsi, yylval devient une union de champ dont vous précisez la liste entre accolades.

```
%union { char chaine[256]; int valeur; }
```

En utilisant le mot clef '%union' de 'yacc', la variable yylval peut prendre plusieurs types (char [], ou bien int). Dans ce cas-là, 'yacc' ne peut déterminer ce que doivent faire vos règles ($$ = $1 + $2, ...). Vous devez alors préciser dans le fichier 'yacc', pour chaque token, le type (le champ utilisé de l'union) entre '<' et '>' et pour chaque règle en utilisant le mot clef '%type'.

Voici le fichier 'lex' complet :

```
   /* -- eval.lex --
Evaluation d'une expression
Partie analyseur lexical.
*/
```

```
%{
#include <string.h>
#include "y.tab.h"
extern void yyerror(char *s);
%}
BLANC [ \n\t]
%%
[a-zA-Z_]+ strcpy(yylval.chaine, yytext); return Tid;
[0-9]+ yylval.valeur = atoi(yytext); return Tnb;
[-+*/()=] return yytext[0]; /* caracteres unites lexicales */
{BLANC}+ ;
. printf("Caractere (%c) non reconnu\n", yytext[0]);
%%
int yywrap(void) {return 1;}
```

Et le fichier 'yacc' complet :

```
/* -- eval.yac --
Evaluation d'une expression
Partie analyseur grammatical.
*/
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int appel_fonc (char *, int, int *);
void yyerror(char *s);
extern int yylex();
%}
%union { char chaine[256]; int valeur; }
%token <chaine> Tid
%token <valeur> Tnb
%type <valeur> EXPR_NUM FACTEUR TERME APPEL_FONC
%start EXPR_CALCS

%%
EXPR_CALCS : EXPR_CALC
| EXPR_CALCS EXPR_CALC
;
EXPR_CALC : EXPR_NUM '=' {printf ("%d\n", $1);}
;
EXPR_NUM : FACTEUR
| EXPR_NUM '+' FACTEUR {$$ = $1 + $3;}
| EXPR_NUM '-' FACTEUR {$$ = $1 - $3;}
;
FACTEUR : TERME
| FACTEUR '*' TERME {$$ = $1 * $3;}
| FACTEUR '/' TERME {$$ = $1 / $3;}
;
TERME : Tnb {$$ = $1;}
```

```
| '(' EXPR_NUM ')' {$$ = $2;}
| APPEL_FONC {$$ = $1;}
;
APPEL_FONC : Tid '(' EXPR_NUM ')' {if (! appel_fonc ($1, $3, &$$)) YYERROR;}
;
%%
int main(void)
{
if ( yyparse() != 0 )
{ fprintf(stderr,"Syntaxe incorrecte\n"); return 1; }
else
return 0;
}

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

int appel_fonc (char *chaine, int valeur, int * val)
{
if (strcmp(chaine, "abs")==0)
*val = abs(valeur);
else if (strcmp(chaine, "neg")==0)
*val = - valeur;
else
{ fprintf(stderr,"Fonction inconnue\n"); return 0; }
return 1;
}
```

A vous d’imaginer des extensions possibles de ce programme….