TP 4

The syntax analyzer generator — Yacc

Exercise 1.

Write and compile the following two specification files: **lex** and **yacc**:

```
%{
#include "y.tab.h"
%}
chiffre [0-9]
%%
[" "\t\n] { }
{chiffre}+  { yylval = atoi(yytext);
return nombre; }
.  { return yytext[0]; }
%%
int yywrap(void)
{ return 1; }
```

```
%{
void yyerror(char *);
%}
%token nombre
%%
session : session expression '=' { printf("result : %d\n", $2); }
        |
        ;
expression : expression '+' terme { $$ = $1 + $3; }
        | expression '-' terme { $$ = $1 - $3; }
        | terme
        ;
terme  : terme '*' facteur { $$ = $1 * $3; }
        | terme '/' facteur { $$ = $1 / $3; }
        | facteur
        ;
facteur  : nombre
        | '(' expression ')' { $$ = $2; }
        ;
%%
void yyerror(char *s) {
   printf("<<< \n%s\n", s);
}
main() {
   yyparse();
   printf("Au revoir!\n");
}
```

To compile a **Lex/Yacc** program, follow these steps :

1. **Save the files**:
   - Save the Lex file with the **.l** extension (example: **lexique.l**)
   - Save the Yacc file with the **.y** extension (example: **syntaxe.y**)
2. **Compile the Yacc file** using the **Yacc** compiler with the command:

   yacc -d syntaxe.y

The result of this compilation is a **C specification file** named **y.tab.c** and a **C header file** named **y.tab.h**.

3. **Compile the Lex file** using the **Lex** compiler with the command:

   lex lexique.l

   The result of this compilation is a **C specification file** named **lex.yy.c**.

4. **Compile the C files** generated by **Lex** and **Yacc** using the **GCC** compiler with the command:

   gcc lex.yy.c y.tab.c -o tplexyacc.exe -lfl

   The result of this compilation is an executable file named **tplexyacc.exe**.

5. **Run the executable** with the command:

   ./tplexyacc.exe

Exercise 2.

Replace the **"production rules and semantic actions"** section of the Yacc file from the first exercise with**:**

```
%%
session     : session expression '=' { printf("result: %d\n", $2); }
            |
            ;
expression  : expression '+' expression { $$ = $1 + $3; }
            | expression '-' expression { $$ = $1 - $3; }
            | expression '*' expression { $$ = $1 * $3; }
            | expression '/' expression { $$ = $1 / $3; }
            | '(' expression ')' { $$ = $2; }
            | nombre
            ;
%%
```

a) **Compile and run** this program. What do you observe?

b) **Resolve the shift/reduce conflicts** by using **left associativity** of the operators with the directive **%left**.