Feuille de TP n° 1 : `ocamllex`

Exemple : un lexer qui recopie l'entrée standard sur la sortie standard :

```
{
exception Eof;;
}

rule token = parse
     |  _    {print_string (Lexing.lexeme lexbuf); token lexbuf}
     | eof   {raise Eof}

{
let lexbuf = Lexing.from_channel stdin in
           try token lexbuf
           with Eof -> exit 0
}
```

Exercice 1 : Écrire un fichier source `ocamllex` pour changer toutes les occurrences du mot “avant” dans un fichier en “apres”, et laisser le reste inchangé.

Exercice 2 : Écrire deux fichiers `code.mll` et `decode.mll` pour coder et decoder un texte : on peut par exemple remplacer chaque caractère du texte par le caractère suivant dans le codage ASCII (utiliser les fonctions `int_of_char :char -> int` et `char_of_int :int -> char`) ou utiliser une fonction de codage plus sophistiquée.

Exercice 3 : Écrire un fichier `entiers.mll` permettant de lire des entiers séparés par des espaces, des sauts de ligne ou des caractères de tabulation. Ce fichier contiendra la définition d'une règle `entier_suivant` produisant un entier.
Écrire une fonction Caml `list_from_string : string -> int list`
renvoyant la liste des entiers contenus dans la chaîne passée en paramètre, en utilisant la fonction `entier_suivant` produite par le fichier `entiers.mll`

Exercice 4 : Écrire un fichier `expr.mll` permettant de lire des expressions comprenant des entiers et les opérateurs `+` et `*`. Ce fichier comprendra la définition d'un type `token` et d'une règle `next_token` produisant un lexème (entier, addition, multiplication).

Le module Lexing de la librairie standard

```
(* Module [Lexing]: the run-time library for lexers generated by [ocamllex] *)
(*** Lexer buffers *)
type lexbuf =
  { refill_buff : lexbuf -> unit;
    mutable lex_buffer : string;
    mutable lex_buffer_len : int;
    mutable lex_abs_pos : int;
    mutable lex_start_pos : int;
    mutable lex_curr_pos : int;
    mutable lex_last_pos : int;
    mutable lex_last_action : int;
    mutable lex_eof_reached : bool }
        (* The type of lexer buffers. A lexer buffer is the argument passed
           to the scanning functions defined by the generated scanners.
           The lexer buffer holds the current state of the scanner, plus
           a function to refill the buffer from the input. *)

val from_channel : in_channel -> lexbuf
        (* Create a lexer buffer on the given input channel.
           [Lexing.from_channel inchan] returns a lexer buffer which reads
           from the input channel [inchan], at the current reading position. *)
val from_string : string -> lexbuf
        (* Create a lexer buffer which reads from
           the given string. Reading starts from the first character in
           the string. An end-of-input condition is generated when the
           end of the string is reached. *)
val from_function : (buf:string -> len:int -> int) -> lexbuf
        (* Create a lexer buffer with the given function as its reading method.
           When the scanner needs more characters, it will call the given
           function, giving it a character string [s] and a character
           count [n]. The function should put [n] characters or less in [s],
           starting at character number 0, and return the number of characters
           provided. A return value of 0 means end of input. *)

(*** Functions for lexer semantic actions *)

val lexeme : lexbuf -> string
        (* [Lexing.lexeme lexbuf] returns the string matched by
           the regular expression. *)
val lexeme_char : lexbuf -> int -> char
        (* [Lexing.lexeme_char lexbuf i] returns character number [i] in
           the matched string. *)
val lexeme_start : lexbuf -> int
        (* [Lexing.lexeme_start lexbuf] returns the position in the
           input stream of the first character of the matched string.
           The first character of the stream has position 0. *)
val lexeme_end : lexbuf -> int
        (* [Lexing.lexeme_end lexbuf] returns the position in the input stream
           of the character following the last character of the matched
           string. The first character of the stream has position 0. *)
```