Analyse Syntaxique et Compilation, TP n° 1 : OcamlLex

Quelques pointeurs vers des docs sur `ocaml` et en particulier `ocamllex` et `ocamlyacc` :
– `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, ou en Français :
– `http://www.univ-valenciennes.fr/ROI/poirriez/ensuvhc/ocamluvhc/DA-OCAML/`.

Exercice 1 : Échauffement

Le fichier `/ens/habermeh/Compil/tp2/exemple.mll` contient le programme d'un analyseur lexical ("lexeur") qui ne fait que recopier l'entrée standard vers la sortie standard.

1. Récupérer ce fichier, le lire, et le comprendre (à l'aide notamment de la référence fournie au verso).
2. Compiler le fichier grâce à la commande `ocamllex exemple.mll`. Parcourir (par exemple avec `less`) le fichier `exemple.ml` produit.
3. Finir la compilation du lexeur. Exécuter le programme en lui fournissant sur l'entrée standard un fichier quelconque : `./exemple < fichier`

Exercice 2 :

Écrire des fichiers sources OcamlLex pour :

1. Changer toutes les occurrences du mot "avant" dans un fichier en "apres". Que se passe-t'il si le fichier contient "avantage" ? Comment faire pour que ce mot ne soit pas modifié ?
2. Enjoliver un programme Pascal, en passant en majuscule ses mot-clés `begin` et `end`. Ainsi, `BegIN` devra devenir `BEGIN`. Attention : un nom de variable tel que `endive` ne devra pas être modifié.
3. Compter le nombre de lignes, de mots (séquences de caractères séparées par des espaces, tabulations ou fins de ligne) et de caractères d'un fichier.
4. Compter le nombre d'identificateurs, de nombres, et de chaînes de caractères (d'une ligne au plus et délimitées par des guillemets) apparaissant dans un fichier source Caml.

Exercice 3 : cpp en Ocamllex

Écrire un fichier source Ocamllex qui implémente une version rudimentaire du pré-processeur `cpp` : le pré-processeur copie le fichier d'entrée et remplace les directives de la forme `#include` *nom_de_fichier* par le contenu du fichier *nom_de_fichier* (qui peut lui même contenir d'autres `#include`, et ainsi de suite). *nom_de_fichier* est le chemin d'accès au fichier à inclure par rapport au répertoire où le pré-processeur est exécuté.
Indication : pour créer un `lexbuf` à partir d'une chaîne de caractères, utiliser les fonctions :

```
> open_in;;
- : string -> in_channel = <fun>
> Lexing.from_channel;;
- : in_channel -> Lexing.lexbuf = <fun>
```

Le module Lexing de la librairie standard

```
(*** 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. *)
```