TP `ocamllex`

Exercice 1 (Aérorythmique)**.** À l'aide d'`ocamllex`, écrire un programme qui prend en entrée un texte et l'affiche en ne montrant que les ponctuations, chaque lettre étant simplement remplacée par une espace.

Exemple : partant de

```
L'eau coule, en boucle calme. Plus ronde que l'air, une larme s'enroule.
```

il faut afficher

```
'          ,                  .                  '    ,             '        .
```

Bonus : faire en sorte d'afficher également les signes portés par les lettres effacées. Dans l'exemple précédent cela revient à afficher en plus le point du `i`. Seraient aussi concernés les accents.

Indication : en caml, le type `char` est limité aux caractères ASCII, en revanche le type `string` supporte très bien UTF8.

Référence : Alain Damasio, *La horde du contrevent*.

Exercice 2 (Pré-processeur)**.** Écrire un programme qui transforme un fichier source C en appliquant ses directives `#define`.

Rappel : dans un programme C, une ligne

```
#define NOM code
```

a pour effet d'introduire une notation pour un fragment de code (valeur, expression ou instruction). Ainsi, toute occurrence de l'identifiant `NOM` dans la suite du programme représente le fragment `code`. Le pré-processeur C, un programme qui agit juste avant le compilateur lui-même, produit un nouveau fichier C à partir d'un fichier source, dans lequel chaque occurrence de `NOM` a été remplacée par le fragment `code`.

Exemple : un fichier `geom.c` dont le contenu est

```
#define LARGEUR  800
#define HAUTEUR  600

int main(int argc, char *argv[])
{
  printf("Hauteur : %i, largeur : %i, surface : %i",
         HAUTEUR, LARGEUR, LARGEUR * HAUTEUR);
}
```

doit être transformé en un fichier `geom.c.pp` dont le contenu est

```
int main(int argc, char *argv[])
{
  printf("Hauteur : %i, largeur : %i, surface : %i",
         600, 800, 800 * 600);
}
```

Attention : le remplacement doit être appliqué y compris dans le code des autres directives `#define`. Ainsi, si l'exemple précédent comprend une troisième ligne

```
#define SURFACE  (LARGEUR * HAUTEUR)
```

alors chaque occurrence de `SURFACE` devra être remplacée par `(800 * 600)`.

Bonus : la directive `#undef NOM` supprime une notation pour la suite du fichier.

Bonus : la directive `#ifdef NOM / #endif` encadre un morceau de code qui ne doit être conservé que si `NOM` est bien une notation définie.

Bonus : les macros introduites par `#define` peuvent être paramétrées. Vous pouvez vous intéresser à la situation où la macro attend exactement un paramètre.

Exercice 3 (Indentation automatique)**.** Écrire un programme qui prend en entrée un fichier source C et affiche son contenu en prenant intégralement en charge la bonne indentation du code. Votre programme doit donc ignorer tous les retraits déjà présents, pour afficher à la place des retraits calculés sur la base du critère suivant : toute accolade ouvrante augmente le niveau de retrait, et toute accolade fermante ramène le retrait au niveau précédent.

Attention : quand une accolade est en début de ligne, le retrait doit diminuer *avant* l'affichage de cette accolade.

Attention : les modifications de retrait ne s'appliquent qu'avec les accolades participant réellement au code, et pas avec celles qui apparaissent dans une chaîne de caractères ou un commentaire.

Bonus : gérer également le niveau de retrait des expressions entre parenthèses. Si une expression ou une séquence d'expressions entre parenthèses s'étale sur plusieurs lignes, il faut :

- que chaque ligne à partir de la deuxième soit alignée sur la colonne suivant celle de la parenthèse ouvrante,
- que la parenthèse fermante, si elle est en début de ligne, soit alignée sur la parenthèse ouvrante.

Indication : il faut pour ce bonus introduire une règle auxiliaire.

Exercice 4 (Analyse lexicale)**.** Écrire un analyseur lexical pour le fragment de caml composé :

- des identifiants
- des nombres entiers
- du mot clé `fun`
- des symboles `->`, `+`, `(`, `)`
- des commentaires délimités par `(*` et `*)`, éventuellement imbriqués

L'analyseur doit définir un type pour les lexèmes, et une règle `token` qui renvoie le prochain lexème reconnu.

Consigne supplémentaire : l'analyseur doit fournir la localisation d'une éventuelle anomalie rencontrée.

- En cas de caractère illégal : donner sa ligne et sa colonne.
- En cas de commentaire qui n'est pas fermé avant la fin du fichier : donner la ligne et la colonne de début de ce commentaire.

Indication : une règle `ocamllex` peut être paramétrée. Ainsi une règle introduite par

```
rule scan n = parse
```

avec `n` un paramètre entier, génère une fonction `scan: int -> Lexing.lexbuf -> ...`

Recommandation : écrire également une boucle de test analysant et affichant tous les lexèmes du fichier donné en entrée.