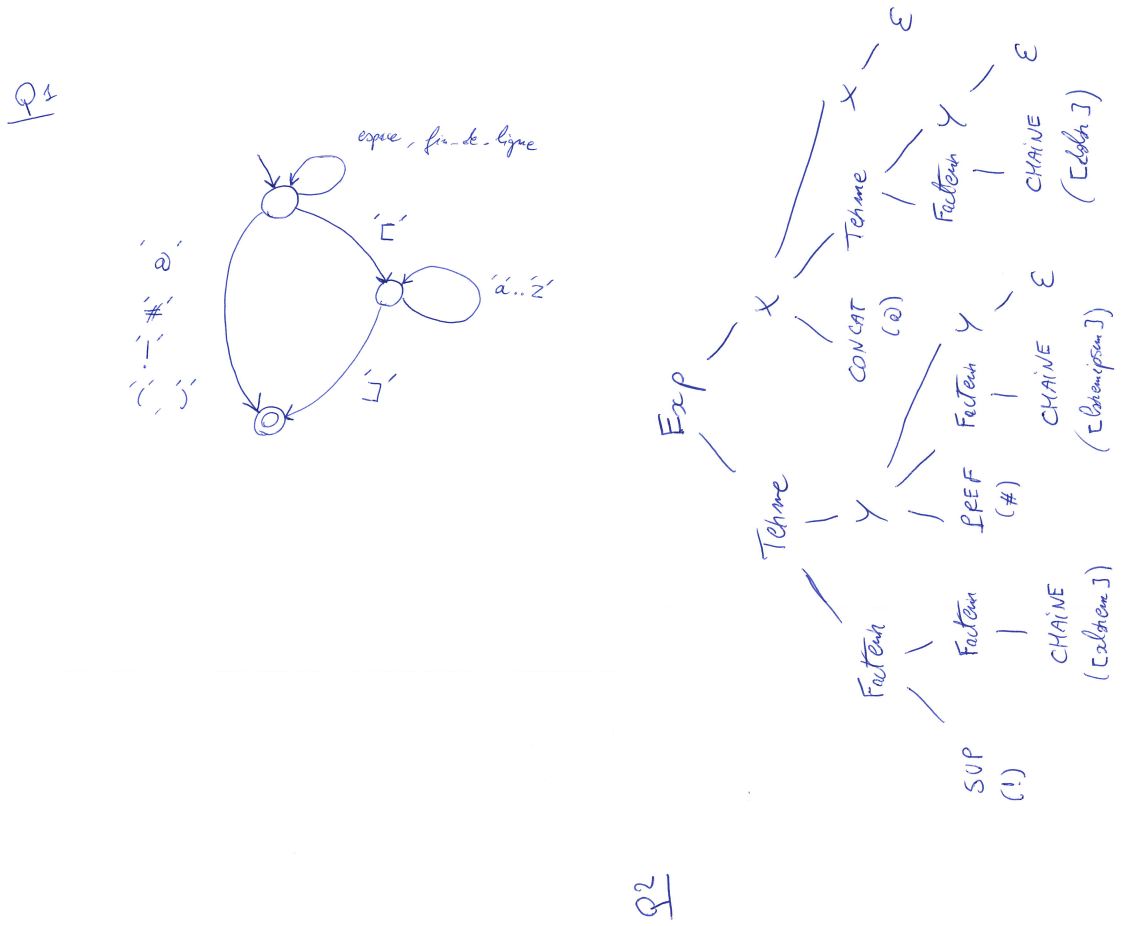
Corrigé Devoir Surveillé du 16 mars 2022



Exemple d'expression comportant une erreur syntaxique mais sans erreurs lexicale :

```
[bon] [jour]
```

Q3.

Par ordre de priorité décroissante : `!` (le plus prioritaire) ; `#` ; `@` (le moins prioritaire)

Q4.

```
void analyser () {
   Rec_Exp () ;
}

void Rec_Exp () {
Rec_Terme () ; Rec_X ()
}

void Rec_X () {
   if (LC.nature == CONCAT) {
      Avancer ;
      Rec_Terme () ; Rec_X () ;
   }
}

void Rec_Terme () {
    Rec_Facteur () ; Rec_Y ()
}

void Rec_Y ()  {
   if (LC.nature == PREF) {
      Avancer ;
      Rec_Facteur () ;
      Rec_Y () ;
   } ;
}

void Rec_Facteur () {
 switch (LC.nature) {
 case SUP:
           Avancer ;
           Rec_Facteur () ;
           break ;
 case CHAINE:
           Avancer ;
           break ;
 case VIDE:
           Avancer ;
           break ;
 case PARO:
           Avancer ;
           Rec_Exp () ;
           if (LC.nature == PARF) {
               Avancer ;
           } else {
               Erreur_Syntaxique () ;
           } ;
           break ;
 default:
           Erreur_Syntaxique() ;
           break ;
 } ;
}
```

Q5.

la chaîne `! [xlorem] # [loremipsum] @ [dolor]` est égale à la chaîne `[loremdolor]`, sa longueur est donc de 10 caractères.

Q6.

Pour calculer la **longueur** de la chaîne résultat il faut modifier les fonctions `Rec_*` pour qu'elles renvoient explicitement la chaine de caractères résultat ...En effet, si on ne connait pas cette chaîne on ne peut pas interpréter correctement l'opérateur "préfixe".

```
int analyser () {
   // on renvoie la longueur de la chaine retournee par Rec_Exp
   return strlen(Rec_Exp ()) ;
}

char *Rec_Exp () {
    char *s1  ;
    s1 = Rec_Terme () ; // Rec_Terme() renvoie une chaine
    return Rec_X (s1) ; // que l'on propage à Rec_X()
}

char *Rec_X (char *s) {
   char *s1, *s2  ;
   if (LC.nature == CONCAT) {
      Avancer ;
      s1 = Rec_Terme () ;
      s2 = Rec_X (s1) ;
      return strcat(s1, s2) ; // concatenation des chaines s1 et s2
   } else
   return s ; // cas epislon
}

char *Rec_Terme () {
    char * s1  ;
    s1 = Rec_Facteur () ; // Rec_Facteur renvoie une chaine
    return Rec_Y (s1) ;   // que l'on propage a Rec_Terme
}

char *Prefixe (char *s1, char *s2) {
// renvoie le plus long prefixe commun des chaines s1 et s2
     int i=0;
     // si l'une des 2 chaine est vide le plus long prefixe commun est vide
     if (s1 == NULL || s2 == NULL) return NULL ;
     // si les 2 chaines sont non vides on cherche le dernier caractere de
     // leur plus long prefixe commun
     while (s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i])
        i=i+1 ;
     s1[i] = '\0' ; // le prefixe est s1[0..i-1]
     return s1 ;
}
```

```
char *Rec_Y (char *s)  {
   char *s1  ;
   if (LC.nature == PREF) {
      Avancer ;
      s1 = Rec_Facteur () ;
      s2 = Rec_Y (s1) ;
      return Pref (s1, s2)
   } else
      return s ;  // cas epsilon
}

char *Rec_Facteur () {
   char *s1  ;
   switch (LC.nature) {
   case SUP:
           Avancer ;
           s1 = Rec_Facteur () ;
           if (s1 != NULL)
               s1=s1+1 ;   // on incremente s1 pour supprimer le 1er caractere
           else
               Erreur_Semantique() ;
           break ;
      case CHAINE:
           s1 = LC.chaine
           // on enleve les crochets ...
           s1 = s1+1 ; // on supprime le 1er caractere de la chaine (crochet droit)
           // on supprime le dernier caractere de la chaine (crochet gauche)
           s1[strlen(s1)-1]='\0' ;
           Avancer ;
           break ;
      case VIDE:
           s1 = NULL
           Avancer ;
           break ;
      case PARO:
           Avancer ;
           s1 = Rec_Exp () ;
           if (LC.nature == PARF) {
             Avancer ;
           } else {
             Erreur_Syntaxique () ;
           } ;
           break ;
     default:
           Erreur_Syntaxique() ;
           break ;
   } ;
   return s1  ;
 }
```