

# Devoir Surveillé 2

18 novembre 2023

## INFORMATIQUE MP2I

DURÉE DE L'ÉPREUVE : 4 heures

**L'usage de la calculatrice et de tout dispositif électronique est interdit.**

*Ce sujet comporte quatre pages numérotées de 1/19 à 19/19*

*Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.*

**Tournez la page S.V.P.**

## Vue d'ensemble du sujet

Ce sujet est composé de 7 parties indépendantes, utilisant les langage de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I s'intéresse aux chaînes de caractères et à la création et l'affichage de motifs en C et en OCaml.
- La partie II s'intéresse aux nombres parfaits, puis au nombres premiers, en C.
- La partie III s'intéresse à l'analyse d'une solution d'un problème classique appelé 'Dutch Flag' en C, via des invariants.
- La partie IV s'intéresse à la modification de chaînes de caractères, en C, et aux choix d'un jeu de tests.
- La partie V s'intéresse à un problème de dénombrement d'objets satisfaisant une définition par récurrence, en OCaml.
- La partie VI s'intéresse à différents systèmes de numérotations, autre que la base  $b$  classique, en OCaml.
- La partie VII s'intéresse à l'analyse d'un programme permettant d'énumérer les anagrammes d'une chaîne de caractère, en OCaml.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

## Partie I. Chaînes de caractères et affichage

### Pyramide (C – 5 pts)

On souhaite créer une fonction en C qui prend en entrée un entier positif  $n$  et qui affiche un motif pyramidal sur  $n$  lignes, comme dans l'exemple ci-dessous, à droite. On propose la fonction ci-dessous, à gauche, où on suppose que la fonction `void print_char(char c)` affiche un caractère.

<pre style="margin: 0;">affichage de pyramide <span style="float: right;">C</span>  void print_pyramide(int n) {     for (int i = 0; i &lt; n; i = i + 1) {         for (int j = 0; j &lt;= i; j = j + 1) {             print_char('#');         }         print_char('\n');     } }</pre>	<pre style="margin: 0;"># ## ### #### ##### ##### #####</pre>
--	---

□ 1 – Combien de caractères sont affichés par cette fonction ?

**Réponse 1.** Il suffit de compter le nombre de fois que la fonction `print_char` est appelée.

Pour chaque itération de la boucle extérieure, la boucle intérieure effectue  $i + 1$  itérations. On a donc le nombre de caractères affichés égal à :

$$\sum_{i=0}^{n-1} i + 2 = \sum_{i=2}^{n+1} i = \frac{n(n+3)}{2}$$

On souhaite désormais écrire cette chaîne de caractère dans une variable `char pyramide[?]` plutôt que de l'afficher sur la sortie standard.

□ 2 – Par quel nombre doit-on remplacer ? entre les crochets si on veut une pyramide de taille 7 (celle représentée en exemple) ?

**Réponse 2.** Il y a  $\frac{n(n+3)}{2} = \frac{7 \times 10}{2} = 35$  caractères, il faut donc un tableau de taille 36 caractères car il faut une case pour stocker le caractère sentinelle `'\0'`.

□ 3 – Écrire la fonction `void pyramide(int n, char pyramide[])` qui prend en entrée un entier positif  $n$  et un tableau de taille suffisante, et qui le remplit avec la chaîne de caractères souhaitée.

**Réponse 3.**

```
void pyramide(int n, char pyramide[]) {
    int pos = 0;
    for (int i = 0; i < n; i = i + 1) {
        for (int j = 0; j <= i; j = j + 1) {
            pyramide[pos] = '#';
            pos = pos + 1;
        }
        pyramide[pos] = '\n';
        pos = pos + 1;
    }
    pyramide[pos] = '\0';
}
```

**Motifs (OCaml – 5 pts)**

□ 4 – Écrire une fonction `mult : string -> int -> string` qui prend en entrée une chaîne de caractère  $s$  et un entier  $n$  et qui renvoie la chaîne contenant  $n$  fois  $s$  d'affilé.

Réponse 4. Version naïve :

```
let rec mult str n =
  if n <= 0 then ""
  else str ^ mult str (n-1)
```

Version exponentiation rapide/square and multiply :

```
let rec mult str n =
  if n <= 0 then ""
  else
    let half = mult str (n/2) in
    if n mod 2 = 0 then half ^ half
    else str ^ half ^ half
```

□ 5 – Écrire une fonction `motif : int -> unit` qui prend en entrée un entier  $n$  et qui dessine le motif constitué d'un carré de croisillons # de taille  $n$  privé d'un carré en diagonal qui est constitué d'espaces. Voici les premiers motifs à renvoyer pour  $n$  allant de 1 à 9 :

```
# ## ### #### ##### ##### ##### ##### #####
## # # # # ## ## ## ## ## ## ## ## ##
### # # # # # # ## ## ## ## ## ## ##
#### ## ## # # # # # # ## ##
##### ## ## ## ## # # # #
##### ## ## ## ## ## ##
##### ## ## ## ## ##
##### ## ## ## ## ##
##### ## ## ## ## ##
#####
```

Réponse 5.

```

let motif n =
  let rec ligne i =
    if i = 0 then mult "#" n
    else
      let nb_hash = (n+1)/2-i in
      let hashes = mult "#" nb_hash in
      hashes ^ (mult " " (n - 2 * nb_hash)) ^ hashes
  in
  let rec boucle i =
    let l = ligne i in
    if i < n / 2 then
      begin
        print_endline l;
        boucle (i+1);
        print_endline l;
      end
    else if n mod 2 = 1 then
      print_endline l
  in boucle 0

```

## Partie II. Diviseurs, nombres parfaits et nombres premiers (C – 15 pts)

Un nombre parfait est un nombre dont la somme des diviseurs propre est égale à lui même.

$$n \text{ est parfait} \Leftrightarrow \sum_{d \in \mathcal{D}} d = n, \mathcal{D} = \{d \mid d \text{ divise } n \text{ et } d \neq n\}$$

□ 6 – Écrire une fonction `bool divise(int d, int n)` qui prend en entrée deux entiers positifs  $d$  et  $n$  et qui renvoie un booléen valant `true` si et seulement si  $d$  divise  $n$ .

Réponse 6.

```

bool divise(int d, int n) {
  return (0 == n % d);
}

```

□ 7 – Écrire une fonction `bool est_parfait(int n)` qui prend en entrée un entier positif  $n$  et qui renvoie un booléen valant `true` si et seulement si  $n$  est un nombre parfait.

Réponse 7.

```

bool est_parfait(int n) {
  int acc = 0;
  for (int d = 1; d < n; d = d + 1) {
    if (divise(d,n)) {
      acc = acc + d;
    }
  }
  return (acc == n);
}

```

On veut maintenant implémenter le crible d'Ératosthène pour trouver tous les nombres premiers inférieurs à un certain nombre  $n$ . On rappelle le principe de cet algorithme :

- on commence avec un tableau contenant tous les nombres jusque  $n$ ;
- on barre 0 et 1;
- on parcourt le tableau et chaque fois qu'on tombe sur un nombre qui n'est pas barré : ce nombre est un nombre premier, on barre tous ses multiples plus grand que lui.

On représentera ce tableau par un tableau de booléens `bool crible[]` dont la case `bool[i]` vaut `true` si  $i$  n'est pas barré et `false` si  $i$  est barré.

□ 8 – Écrire une fonction `void init_rible(int n, bool crible[])` qui prend en argument un entier positif  $n$  et un tableau de booléens de taille  $n$  et qui remplit le tableau en accord avec les deux premières étapes du crible d'Ératosthène.

### Réponse 8.

```
void init_rible(int n, bool crible[]) {
    crible[0] = false;
    crible[1] = false;
    for (int i = 2; i < n; i = i + 1) {
        crible[i] = true;
    }
}
```

□ 9 – Étant donné deux entiers positifs  $i$  et  $n$ , combien y-a-t-il de multiples  $m$  de  $i$  tels que  $i < m \leq n$  ?

**Réponse 9.** Si  $n = q \times i + r$ , avec  $0 \leq r < i$ , alors les multiples de  $i$  à prendre en compte sont  $1 \times i, 2 \times i, 3 \times i, \dots, q \times i$ , il y a donc  $q = n/i$  (au sens de la division entière) multiples de  $i$  inférieurs à  $n$ .

□ 10 – Écrire une fonction `void barre_multiples(int n, bool crible[], int i)` qui prend en argument un entier positif  $n$ , un tableau de booléens de taille  $n$ , et un entier positif  $i$  et qui assigne la valeur `false` à tous les multiples de  $i$  compris entre  $i$  exclu et  $n$  inclus.

### Réponse 10.

```
void barre_multiples(int n, bool crible[], int i) {
    for (int j = 2 * i; j < n; j = j + i) {
        crible[j] = false;
    }
}
```

□ 11 – Écrire une fonction `void erathostene(int n, bool crible[])` qui prend en argument un entier positif  $n$  et un tableau de booléens de taille  $n$  et qui assigne la valeur `false` à tous les nombres composites (i.e. qui ne sont pas premiers).

### Réponse 11.

```
void erathostene(int n, bool cribble[]) {
    init_cribble(n, cribble);
    for (int i = 0; i < n; i = i + 1) {
        if (cribble[i]) {
            barre_multiple(in, cribble, i);
        }
    }
}
```

□ 12 – Écrire la fonction `int main(int argc, char *argv[])` d'un programme qui affiche tous les nombres premiers jusque 1 000 en utilisant le crible d'Ératostène.

### Réponse 12.

```
int main(int argc, char *argv[]) {
    int cribble[1000];
    erathostene(1000, cribble);
    for (int i = 0; i < n; i = i + 1) {
        if (cribble[i]) {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

### Partie III. Drapeau de la Pologne (C – 5 pts)

On considère l'algorithme suivant :

```

Polish national flag C

void polish(int taille, int tab[]) {
    // Précondition : tab[] est un tableau d'entiers de taille n
    // contenant uniquement des 0 et des 1
    int i = 0, j = taille-1;
    while (j > i) {
        /*****
         *           i       j           *
         *   +-+---+---+---+---+---+ *
         * tab : |0|0|...|0|?|...|?|1|1|...|1| *
         *   +-+---+---+---+---+---+ *
         *****/
        if (0 == tab[i]) {
            i = i + 1;
        } else {
            tab[i] = tab[j];
            tab[j] = 1;
            j = j - 1;
        }
    }
}

```

13 – Expliciter l'invariant de boucle expliqué par le commentaire.

**Réponse 13.**  $\forall k < i, \text{tab}[k] = 0, \forall k > j, \text{tab}[k] = 1$

14 – Soit  $n$  le nombre de 0 et  $m$  le nombre de 1 dans le tableau au début de la fonction. Montrer l'invariant de boucle suivant :

Le tableau contient  $n$  fois 0 et  $m$  fois 1.

**Réponse 14.**

- L'invariant est vérifié au début de la première itération par définition de  $n$  et  $m$
- En supposant que l'invariant est vérifié au début d'une itération, on a deux cas à vérifier :
  - Si  $(0 == \text{tab}[i])$ , le tableau est inchangé
  - Sinon,  $\text{tab}[i]$  vaut 1, et le nombre de 0 et de 1 reste inchangé car on a juste échangé les valeurs de  $\text{tab}[i]$  et de  $\text{tab}[j]$ .

15 – Que fait cette fonction ?

**Réponse 15.** Cette fonction trie le tableau en mettant les 0 avant les 1.

### Partie IV. leet speak (C – 10 pts)

Le 1337 ou leet speak consiste à remplacer certaines lettres par des chiffres ou autres caractères ASCII. Nous considérons la variante suivante :

- 'l' et 'L' sont transformés en '1';
- 'e' et 'E' sont transformés en '3';
- 't' et 'T' sont transformés en '7'.

16 – Écrire une fonction `void leet(char mot[])` qui prend en entrée une chaîne de caractères et qui lui applique le code 1337.



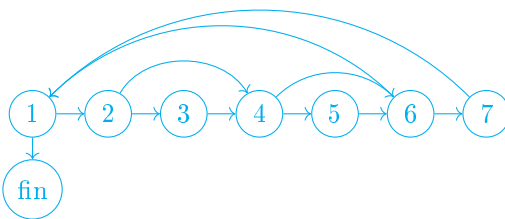
## Réponse 16.

```

void leet(char mot[]) {
  for (int i = 0; mot[i] != '\0'; i = i + 1) { // 1
    if ('l' == mot[i] || 'L' == mot[i]) { // 2
      mot[i] = '1'; // 3
    }
    if ('e' == mot[i] || 'E' == mot[i]) { // 4
      mot[i] = '3'; // 5
    }
    if ('t' == mot[i] || 'T' == mot[i]) { // 6
      mot[i] = '7'; // 7
    }
  }
  // fin
}

```

□ 17 – Déterminer le graphe de flot de contrôle de cette fonction.



## Réponse 17.

□ 18 – Donner le chemin dans le graphe correspondant à l'exécution de la fonction sur l'entrée "Hello".

**Réponse 18.** (1) – (2) – (4) – (6) – (1) – (2) – (4) – (5) – (6) – (1) – (2) – (3) – (4) – (6) – (1) – (2) – (3) – (4) – (6) – (1) – (2) – (4) – (6) – (1) – (fin)

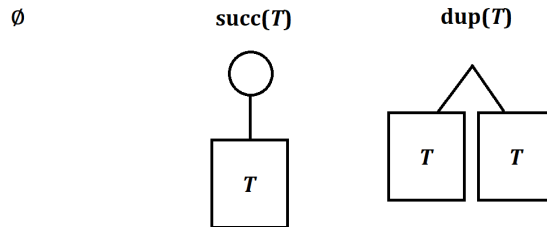
□ 19 – Proposer un jeu de tests réalisant une couverture des branches.

**Réponse 19.** Avec le test "Hello" qui devient "H311o", il manque les branches (6) – (7) et (7) – (1) à couvrir. On peut utiliser à la place par exemple le test "LETleto" qui devient "137137o" pour tester à la fois toutes les branches et toutes les façons possibles de vérifier les conditions.

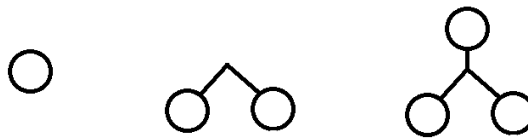
## Partie V. Dénombrement : arbres binaires symétriques (OCaml – 10 pts)

On se propose de compter le nombre d'arbres binaires symétriques que l'on peut construire de la manière suivante. Un arbre binaire symétrique peut être soit :

- Un arbre vide  $\emptyset$ , ne contenant aucun élément ;
- Un arbre **succ**( $T$ ) contenant un élément, puis un sous-arbre binaire symétrique  $T$  ;
- Un arbre **dup**( $T$ ) contenant deux copies d'un sous-arbre binaire symétrique  $T$  non vide.



Par exemple, on a représenté ci-dessous les arbres **succ**( $\emptyset$ ), **dup**(**succ**( $\emptyset$ )), et **succ**(**dup**(**succ**( $\emptyset$ ))) qui possèdent respectivement 1, 2, et 3 éléments :

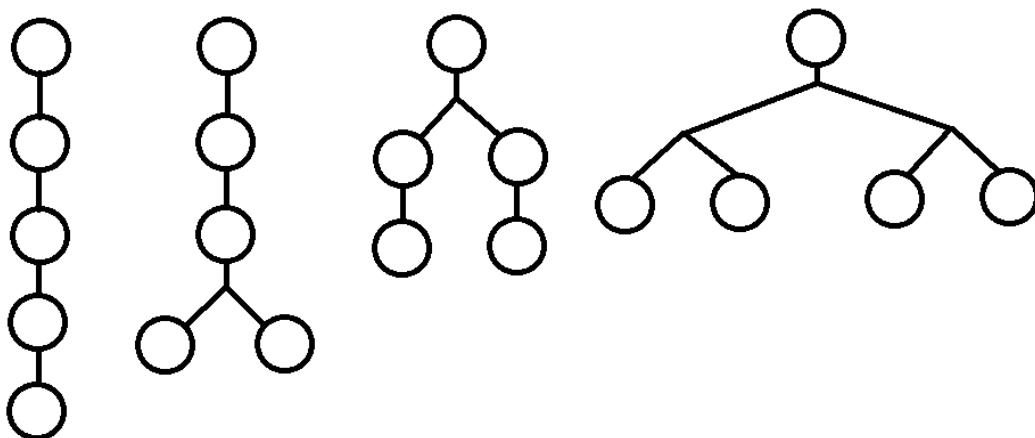


□ 20 – Si un arbre  $T$  contient  $k$  éléments, combien d'éléments comptent **succ**( $T$ ) et **dup**( $T$ ) ?

**Réponse 20.** **succ**( $T$ ) contient  $T$  et un élément donc  $k + 1$  éléments.  
**dup**( $T$ ) contient 2 copies de  $T$  donc  $2k$  éléments.

□ 21 – Dessiner tous les arbres symétriques contenant 5 éléments.

**Réponse 21.** Il y a 5 arbres symétriques contenant 5 éléments :



□ 22 – Écrire une fonction récursive `compte_arbres : int -> int` qui prend en entrée un nombre  $n$  et qui calcule le nombre d'arbres symétriques différents de taille  $n$ .

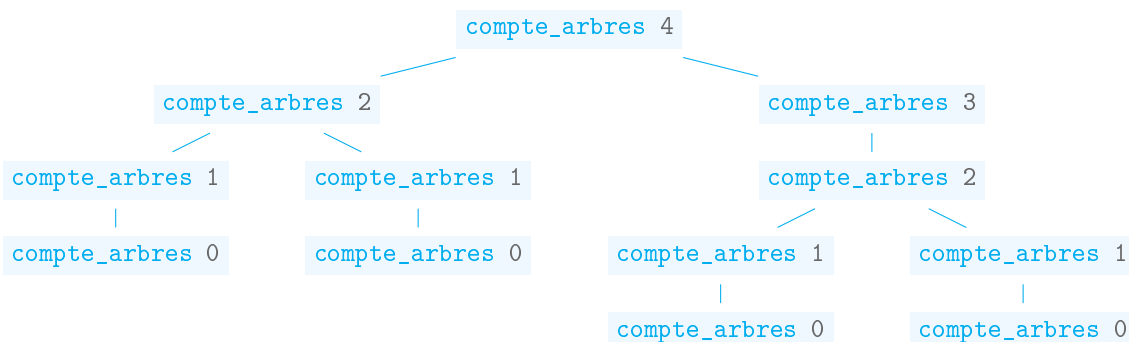
Réponse 22.

```

let rec compte_arbres n =
  assert (n >= 0);
  if n = 0 then
    1
  else if n mod 2 = 0 then
    compte_arbres (n/2) + compte_arbres (n-1)
  else
    compte_arbres (n-1);;
    
```

□ 23 – Dessiner l’arbre des appels récursifs pour l’expression `compte_arbres 4;;`.

Réponse 23. Le voici :



## Partie VI. Changeons de base (OCaml)

Rappel : Écrire un nombre  $n$  en base  $b \geq 2$ , c’est l’écrire  $\overline{n_\ell n_{\ell-1} \dots n_1 n_0}^b$ ,

avec  $\forall i \in [0, \ell], n_i \in [0, b-1]$ , tels que  $n = \sum_{i=0}^{\ell} n_i b^i$ . Par exemple,  $13 = \overline{1101}^2$  car  $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$ .

Pour trouver l’écriture d’un nombre dans une base  $b$ , on procède par divisions euclidiennes successive par  $b$  :

$$\begin{aligned}
 13 &= 6 \times 2 + 1 \\
 6 &= 3 \times 2 + 0 \\
 3 &= 1 \times 2 + 1 \\
 1 &= 0 \times 2 + 1
 \end{aligned}$$

On s’arrête Lorsque le quotient vaut 0, et l’écriture du nombre se lit de bas en haut.

### Bases négatives (16 pts)

On peut étendre ce concept pour écrire des nombres en base  $-b$ ,  $b \geq 2$ , toujours avec des chiffres entre 0 et  $b-1$ .

Par exemple, en base  $-10$ , le nombre 123 s’écrit  $\overline{283}^{-10}$  car  $123 = 2 \times 100 + 8 \times (-10) + 3 \times 1$ .

Et le nombre  $-123$  s’écrit  $\overline{1937}^{-10}$  car  $-123 = 1 \times (-1000) + 9 \times 100 + 3 \times (-10) + 7 \times 1$ .

Les conversions peuvent toujours s’effectuer par divisions euclidiennes successives, mais il faut l’adapter pour les diviseurs négatifs.

□ 24 – Convertir 42 en base  $-3$ .

**Réponse 24.**

$$\begin{aligned}
42 &= -14 \times (-3) + 0 \\
-14 &= 5 \times (-3) + 1 \\
5 &= -1 \times (-3) + 2 \\
-1 &= 1 \times (-3) + 2 \\
1 &= 0 \times (-3) + 1
\end{aligned}$$

Donc  $42 = \overline{12210}^{-3}$ .

Vérification :  $81 + 2 \times (-27) + 2 \times 9 - 3 = 81 - 54 + 18 - 3 = 42$ .

□ 25 – Convertir en décimales  $\overline{111111}^{-2}$ .

**Réponse 25.**

$$\begin{aligned}
\overline{111111}^{-2} &= (-2)^5 + (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1 + (-2)^0 \\
&= -32 + 16 - 8 + 4 - 2 + 1 \\
&= -21
\end{aligned}$$

Pour implémenter cette conversion en OCaml, nous avons besoin d'une division euclidienne, `/` et `mod` ne conviennent pas lorsque le dividende est négatif :

```
(* On souhaiterait -5 = 2 * (-3) + 1 *)
# (-5) / (-3);;
- : int = 1
# (-5) mod -3;;
- : int = -2
```

□ 26 – Écrire une fonction `div_e : int -> int -> int * int` qui prend en entrée deux entiers  $a$  et  $b$  et qui renvoie un couple d'entiers  $(q, r)$  tels que :

$$a = q \times b + r, \quad 0 \leq r < |b|$$

**Réponse 26.**

```
let div_e a b =
  assert (b <> 0);
  let (q,r) = (a/b, a mod b) in
  if r >= 0 then
    (q,r)
  else if b > 0 then
    (q-1, b+r)
  else
    (q+1, r-b);;
```

□ 27 – Proposer un jeu de tests (entrées et sorties correspondantes attendues) que vous utiliseriez pour vérifier votre fonction.

**Réponse 27.** On cherche à couvrir les différents cas positif/négatif, ainsi que la division par 0.

## Jeu de tests pour div\_e

REPL

```
# div_e 11 3;;
- : int * int = (3, 2)
# div_e (-11) 3;;
- : int * int = (-4, 1)
# div_e (-11) (-3);;
- : int * int = (4, 1)
# div_e 11 (-3);;
- : int * int = (-3, 2)
# div_e 11 0;;
Exception: Assert_failure ("//toplevel//", 2, 2).
# div_e 0 0;;
Exception: Assert_failure ("//toplevel//", 2, 2).
# div_e 0 3;;
- : int * int = (0, 0)
```

Pour les questions suivantes, les représentations des nombres dans des bases autres que 10 seront de type `string`. Nous utiliserons les opérateurs `^+` et `+^` définis ci-dessous pour ajouter un caractère à droite et à gauche de chaînes de caractères.

## Concaténation d'un char et une string

REPL

```
# let (^+) str c = str ^ String.make 1 c;;
val ( ^+ ) : string -> char -> string = <fun>
# let (+^) c str = String.make 1 c ^ str;;
val ( +^ ) : char -> string -> string = <fun>
# ("Bonjour" ^+ 'n') ^+ 'o';;
- : string = "Bonjourno"
# 'a' +^ ('b' +^ "cd");;
- : string = "abcd"
```

□ 28 – Écrire une fonction récursive `convertis : int -> int -> string` qui prend en entrée un nombre entier  $n$  et un nombre entier  $b \in \{-10, \dots, -2, 2, \dots, 10\}$ , et qui renvoie la représentation de  $n$  en base  $b$  sous forme d'une string.

On n'oubliera pas de vérifier la précondition sur  $b$  à l'aide d'une assertion.

## Réponse 28.

```
let to_digit d = char_of_int (d + int_of_char '0');;

let rec convertis n b =
  if n = 0 then "0" else
    let (q,r) = div_e n b in
    if q = 0 then
      string_of_int r
    else
      convertis q b ^+ to_digit r;;
```

Intéressons nous à l'addition.

□ 29 – Calculer  $\overline{27}^{-10} + \overline{13}^{-10}$ .

Réponse 29.

$$\begin{aligned}\overline{27}^{-10} + \overline{13}^{-10} &= 2 \times (-10) + 7 + 1 \times (-10) + 3 \\ &= 3 \times (-10) + 10 \\ &= 2 \times (-10) \\ &= \overline{20}^{-10}\end{aligned}$$

□ 30 – Calculer  $\overline{7}^{-10} + \overline{3}^{-10}$ .

Réponse 30.

$$\begin{aligned}\overline{7}^{-10} + \overline{3}^{-10} &= 7 + 3 \\ &= 10 \\ &= 100 + 9 \times (-10) \\ &= \overline{190}^{-10}\end{aligned}$$

□ 31 – Proposer un algorithme permettant d'effectuer l'addition de deux nombres en base -2.

Réponse 31. On pourra s'aider d'une fonction récursive auxiliaire de type

```
string -> string -> char -> string
```

qui prend en argument deux représentations de nombres en base -2 et une retenue '+', '-' ou '=', et qui ajoute les deux nombres en ajoutant 1 si la retenue est '+', en retirant 1 si la retenue est '-', et sans rien ajouter ni retirer si la retenue est '='.

```

let ajoute a b =
  let rec ajoute_aux a b r =
    if b = "" && r = '=' then
      a
    else if b = "" then
      ajoute_aux a (retenu r) '='
    else if a = "" then
      ajoute_aux b (retenu r) '='
    else
      let n = String.length a in
      let m = String.length b in
      let (an, ap) = (a.[n-1], String.sub a 0 (n-1)) in
      let (bm, bp) = (b.[m-1], String.sub b 0 (m-1)) in
      if an = '1' && bm = '1' then
        if r = '+' then
          ajoute_aux ap bp '-' ^+ '1'
        else if r = '-' then
          ajoute_aux ap bp '=' ^+ '1'
        else
          ajoute_aux ap bp '-' ^+ '0'
      else if an = '0' && bm = '0' then
        if r = '+' then
          ajoute_aux ap bp '=' ^+ '1'
        else if r = '-' then
          ajoute_aux ap bp '+' ^+ '1'
        else
          ajoute_aux ap bp '=' ^+ '0'
      else (* an = 1 et bm = 0 ou l'inverse *)
        if r = '+' then
          ajoute_aux ap bp '-' ^+ '0'
        else if r = '-' then
          ajoute_aux ap bp '=' ^+ '0'
        else
          ajoute_aux ap bp '=' ^+ '1'
      in ajoute_aux a b '=';;

```

Ou en réutilisant la fonction de conversion déjà codée (attention, cette version est sujette au dépassement de capacité des entiers, ce qui n'est pas le cas pour la précédente) :

```
let from_digit c = int_of_char c - int_of_char '0';;

let convertis_inv str b =
  let n = String.length str in
  assert (n > 0);
  let rec aux acc i =
    if i = n then
      acc
    else
      aux (acc * b + from_digit str.[i]) (i+1)
  in aux 0 0;;

let ajoute a b =
  let ap = convertis_inv a (-2) in
  let bp = convertis_inv b (-2) in
  convertis (ap + bp) (-2);;
```



## Numération bijective (12 pts)

Un système de numération bijectif est un système de numération dans lequel chaque nombre (entier positif) possède un unique représentant. Cette propriété n'est pas vérifiée pour la représentation décimale usuelle car le nombre 5 peut aussi être représenté 05 ou 005, etc. On va donc représenter les nombres sans utiliser de chiffre 0. La numération bijective en base  $k$ , ou notation  $k$ -adique, est une représentation en base  $k$ , mais qui utilise comme chiffres les nombres entiers de 1 à  $k$ . Pour  $k = 10$  par exemple, on utilise un chiffre A dont la valeur est 10. Par exemple, le nombre 20 sera représenté  $1A_{10}$  car  $20 = 1 \times 10 + A \times 1$ . La représentation du nombre 179 reste inchangée car elle ne contient pas le chiffre 0.

□ 32 – Compter jusqu'à 10 en notation 2-adique, en utilisant les chiffres 1 et 2.

**Réponse 32.** 1; 2; 11; 12; 21; 22; 111; 112; 121; 122

□ 33 – Écrire le nombre 2010 en notation 10-adique, en utilisant les chiffres 1, 2, 3, 4, 5, 6, 7, 8, 9 et A.

**Réponse 33.** 2010 → 1A10 → 1A0A → 19AA

□ 34 – Proposer un algorithme pour déterminer la notation  $k$ -adique d'un nombre.

**Réponse 34.** On procède par divisions euclidiennes successives, mais en modifiant la division pour garder un reste entre 1 et  $k$  inclus.

Le système bijectif en base 26, aussi appelé base 26 sans zéro, utilise comme chiffres les 26 lettres de l'alphabet en majuscule. C'est le système de numérotation qui est utilisé pour les colonnes des tableaux par exemple.

□ 35 – Combien de colonnes se trouvent à gauche de la colonne "ABC".

**Réponse 35.**

$$\begin{aligned} 1 \times 26^2 + 2 \times 26 + 3 &= 676 + 52 + 3 \\ &= 731 \end{aligned}$$

La colonne "ABC" est la 731-ième colonne, il y en a 730 à sa gauche.

□ 36 – Écrire une fonction récursive `num_col : string -> int` qui prend en entrée une chaîne de caractères ne contenant que des lettres majuscules et qui renvoie le numéro de la colonne correspondante.

Si la chaîne de caractère contient un caractère autre qu'une lettre majuscule, on pourra lever une exception appropriée avec l'expression

```
raise (Invalid_argument "caractère illégal")
```

**Réponse 36.**

```
let letter c =
  if c < 'A' || c > 'Z' then
    raise (Invalid_argument "caractère illégal")
  else
    int_of_char c - int_of_char 'A' + 1;;

let num_col str =
  let n = String.length str in
  if n = 0 then 0 else
    let rec num_col_aux acc i =
      if i = n then
        acc
      else
        num_col_aux (acc * 26 + letter str.[i]) (i+1)
    in
    num_col_aux 0 0;;
```

**Notation ternaire équilibrée (2 pts)**

Le système ternaire équilibré consiste en un système de numération en base 3 dont les chiffres sont +1, 0, et -1, que nous noterons +, 0, et -. Par exemple [+ -0] correspond à  $27 - 9 - 3 = 15$ .

□ 37 – Écrire le nombre 42 en ternaire équilibré.

**Réponse 37.**

$$\begin{aligned}42 &= 14 \times 3 + 0 \\14 &= 5 \times 3 + (-1) \\5 &= 2 \times 3 + (-1) \\2 &= 1 \times 3 + (-1) \\1 &= 0 \times 3 + 1\end{aligned}$$

Donc  $42 = [+--0]$ . Vérification :  $81 - 27 - 9 - 3 = 42$ .

## Partie VII. Anagrammes (OCaml – 5 pts)

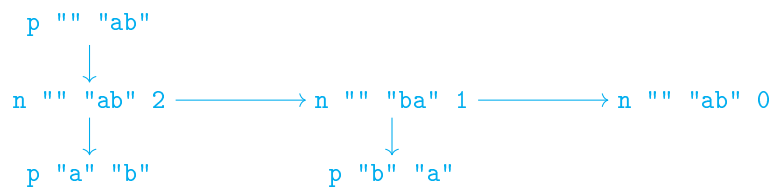
On donne les fonctions suivantes qui permet d'afficher toutes les anagrammes d'une chaîne de caractères :

```
let rec palindrome prefixe str =
  if String.length str <= 1 then
    print_endline (prefixe^str)
  else
    next_letter prefixe str (String.length str)

and next_letter prefixe str n =
  if n = 0 then
    ()
  else
    let premiere_lettre = String.make 1 str.[0] in
    let reste = String.sub str 1 (String.length str - 1) in
    palindrome (prefixe^premiere_lettre) reste;
    next_letter prefixe (reste^premiere_lettre) (n-1);;
```

- 38 – Dessiner l'arbre des appels pour `palindrome "" "ab"`. On pourra noter  $p$  un appel de la fonction `palindrome` et  $n$  un appel de la fonction `next_letter`.

**Réponse 38.**



- 39 – Montrer que les fonctions `palindrome` et `next_letter` terminent.

**Réponse 39.** Chaque chemin dans l'arbre des appels contient des appels  $p$  espacés par des appels  $n$ . Lors des suites d'appel  $n$ , le 3-ème argument est un entier qui décroît strictement, donc ces suites sont nécessairement finies. Si on ignore les appels  $n$  le long d'un chemin, la taille du second argument des appels  $p$  est un entier strictement décroissant, donc ces suites sont nécessairement finies.

Ainsi, toutes les branches de l'arbre des appels récursifs sont finies et les fonctions terminent.

FIN DE L'ÉPREUVE