

Devoir Surveillé 4

10 février 2024

INFORMATIQUE MP2I

DURÉE DE L'ÉPREUVE : 4 heures

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

Ce sujet comporte huit pages numérotées de 1/8 à 8/8

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 le langage de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traitées 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 à une fonction mystère en OCaml.
- La partie II s'intéresse au choix de structures de données pour modéliser des situations.
- La partie III s'intéresse à l'implémentation d'ensembles d'entiers par une liste triée simplement chaînée en C.
- La partie IV s'intéresse aux types structurés en OCaml, et aux relations d'ordres.
- La partie V s'intéresse à l'implémentation de files d'attente à double extrémités par des listes doublement chaînées circulaires en C.
- La partie VI s'intéresse au tri par bulle sur des piles en OCaml.
- La partie VII s'intéresse au problème du bon parenthésage avec plusieurs types de parenthèses.

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. En OCaml, les noms des arguments des fonctions sont donnés avec des indications de type. Il n'est pas nécessaire de recopier ces indications de type sur votre copie.

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. Une fonction mystère (OCaml) – 10 pts

La constante entière `max_int` désigne le plus grand entier représentable par OCaml. Nous donnons la fonction `mystere` suivante.

Fonction mystère

```
let rec mystere z = match z with
(* La fonction mystere calcule ... *)
| [] -> max_int
| [a] -> a
| a :: b :: y -> mystere ((if a <= b then a else b) :: y);;
```

- 1 – Donner la signature de la fonction `mystere`. Justifier brièvement.
- 2 – Dire si, quelle que soit l'entrée `z` respectant le typage, le calcul de `mystere z` se termine et le démontrer. Préciser, le cas échéant, le nombre d'appels à la fonction `mystere`.
- 3 – Compléter sommairement le commentaire de la ligne 2. Énoncer une propriété qui caractérise exactement la valeur de retour de la fonction `mystere`. Démontrer cette propriété.

Dans la question suivante, le terme *complexité en espace* désigne un ordre de grandeur asymptotique de l'espace utilisé en mémoire lors de l'exécution d'un algorithme pour stocker tant l'entrée que des résultats intermédiaires et la valeur de retour.

- 4 – Quelle est la complexité en espace de l'appel `mystere z` ? Est-elle optimale ?

Partie II. Analogie avec le supermarché – 5 pts

Toutes les réponses de cette section seront à justifier brièvement.

□ 5 – Quelle structure de donnée abstraite représente le mieux un chariot de supermarché ?

Réponse 5. Une pile car les éléments placés en premier dans le chariot ne peuvent pas être retirés si il reste des éléments placés au dessus.

□ 6 – Quelle structure de donnée abstraite représente le mieux le tapis roulant de la caisse au supermarché ?

Réponse 6. Une file car les éléments placés en premier sur le tapis sont ceux que l'on récupère en premier.

□ 7 – Que peut-on dire de l'ordre des éléments dans le chariot avant et après le passage en caisse ?

Réponse 7. L'ordre est inversé : les éléments du dessus avant la caisse sortent premier du tapis et se retrouvent en dessous après la caisse.

Partie III. Listes triées simplement chaînées (C) – 16 pts

Nous supposons définies deux constantes entières INT_MIN et INT_MAX qui désignent respectivement le plus petit entier et le plus grand entier représentables par le type `int` en machine. Elles sont représentées par $-\infty$ et $+\infty$ dans les schémas.

Indication C : Nous introduisons une structure de maillon constituée de deux champs par la déclaration suivante.

```

1. struct maillon {
2.     int donnee;
3.     struct maillon *suivant;
4. };
5. typedef struct maillon maillon_t;

```

Définition : Dans l'ensemble de cette partie, nous réalisons le type abstrait `ENSEMBLEENTIERS` à l'aide d'une liste de maillons simplement chaînés. Nous supposons que tous les entiers insérés, supprimés ou recherchés sont strictement compris entre INT_MIN et INT_MAX . Nous maintenons les trois invariants suivants :

- 1. Pour tous maillons m et m' consécutifs dans la liste chaînée, de champs `donnee` respectifs u et u' , on a l'inégalité $u < u'$ (autrement dit, la liste est triée et ne contient pas de doublons).
- 2. La liste est encadrée par deux maillons sentinelles ayant INT_MIN comme champ `donnee` en tête de liste et INT_MAX en fin de liste.
- 3. Pour toute valeur entière u contenue dans l'ensemble, il existe un maillon accessible depuis le maillon sentinelle de tête ayant u comme champ `donnee`.

Par exemple, l'ensemble $\{2, -7, 9\}$ est représenté par la liste chaînée dessinée en figure 1 (où la croix représente le pointeur nul).

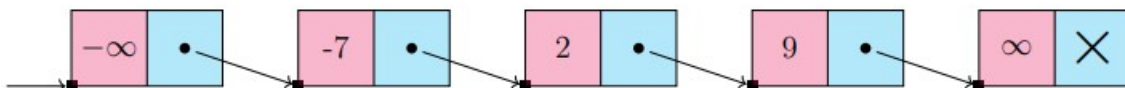


FIGURE 1 – Exemple d'ensemble d'entier représenté par une liste chaînée

- 8 – Écrire en C une fonction `maillon_t *init(void)` dont la spécification suit :
- Effet* : crée une copie de l'ensemble vide par l'instanciation de deux nouveaux maillons sentinelles chaînés entre eux.
- Valeur de retour* : pointeur vers le maillon de tête.

Réponse 8.

```

maillon_t *init(void) {
    maillon_t *p2 = malloc(sizeof(maillon_t));
    p2->suivant = NULL;
    p2->donnee = INT_MAX;

    maillon_t *p1 = malloc(sizeof(maillon_t));
    p1->donnee = INT_MIN;
    p1->suivant = p2;

    return p1;
}

```

- 9 – Écrire en C une fonction `maillon_t *localise(maillon_t *t, int v)` dont la spécification suit :
- Précondition* : Le pointeur t désigne le maillon sentinelle de tête d'une liste chaînée.
- Postcondition* : En notant u le champ `donnee` du maillon désigné par la valeur de retour et u' celui du maillon successeur, on a les inégalités $u < v \leq u'$.

Réponse 9.

```

maillon_t *localise(maillon_t *t, int v) {
    maillon_t *next;
    next = t->suivant;

    while (next->donnee < v) {
        t = next;
        next = t->suivant;
    }

    return t;
}

```

Nous souhaitons écrire une fonction `bool insere(maillon_t *t, int v)` ainsi spécifiée :

Précondition : Le pointeur t désigne le maillon sentinelle de tête d'une liste chaînée.

Postcondition : La liste désignée par le pointeur t contient la valeur entière v ainsi que les autres valeurs précédemment contenues.

Valeur de retour : Booléen `true` si la liste contient un élément de plus et `false` sinon.

- 10 – Présenter sous forme de croquis un jeu de données de test de la fonction `insere`, qui couvre notamment l'ensemble des valeurs de retour possibles. Dans chaque cas, on dessinera les états initial et final de la liste à la manière de la figure 1 et on donnera la valeur de retour.

Réponse 10.

```

f := {}
insere f 1 -> true
f == {1}
insere f 1 -> false
f == {1}

```

Nous proposons le code erroné suivant.

```

6.  bool insere_errone(maillon_t *t, int v) {
7.      maillon_t *p = localise(&t, v);
8.      maillon_t *n = malloc(sizeof(maillon_t));
9.      n->suivant = p->suivant;
10.     n->donnee = v;
11.     p->suivant = n;
12.     return true;
13. }

```

□ 11 – Le compilateur produit le message d’erreur incompatible pointer types passing ‘maillon_t **’ to parameter of type ‘maillon_t *’. Expliquer ce message et proposer une première correction.

Réponse 11. Ligne 7, `t` est de type `maillon_t *` (cf. ligne 6), donc `&t` est de type `maillon_t **`, mais `localise` attend en entrée un argument de type `maillon_t *`. Une correction est de supprimer le `&`.

```

bool insere_errone(maillon_t *t, int v) {
    maillon_t *p = localise(t, v);
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    p->suivant = n;
    return true;
}

```

□ 12 – Discerner le ou les tests de la question 10 manqués par la fonction `insere_errone`. Corriger en conséquence la fonction `insere_errone`.

Réponse 12. La fonction `insere_errone` ne renvoie jamais `false`, donc le test suivant est erroné :

```

f := {1}
insere f 1 -> false
f == {1}

```

```

bool insere(maillon_t *t, int v) {
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee == v) {
        return false;
    }
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    p->suivant = n;
    return true;
}

```

□ 13 – Écrire en C une fonction `bool supprime(maillon_t *t, int v)` dont la spécification suit :

Précondition : Le pointeur `t` désigne le maillon sentinelle de tête d’une liste chaînée.

Postcondition : La liste désignée par le pointeur `t` ne contient pas la valeur entière `v` mais contient les autres valeurs précédemment contenues.

Valeur de retour : Booléen `true` si la liste contient un élément de moins et `false` sinon.

Réponse 13.

```
bool supprime(maillon_t *t, int v) {
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee != v) {
        return false;
    }
    maillon_t *tmp = p->suivant->suivant;
    free(p->suivant);
    p->suivant = tmp;
    /* Variante :
       tmp = p->suivant;
       p->suivant = p->suivant->suivant;
       free(tmp);
    */

    return true;
}
```

14 – Calculer la complexité en temps des fonctions `insere` et `supprime`.

Réponse 14. Dans le pire des cas, il faut parcourir entièrement la liste, donc la complexité est linéaire en la taille de la liste. $O(n)$ si la liste contient n éléments.

15 – Un programme C peut stocker ses données dans différentes régions de la mémoire. Citer ces régions. Dire dans laquelle ou dans lesquelles de ces régions la valeur entière 717 est inscrite lorsque nous exécutons le programme suivant.

```
14.  int v = 717;
15.  int main(void) {
16.      maillon_t *t = init();
17.      insere(t, v);
18.      return 0;
19.  }
```

Réponse 15. Les deux zones de mémoire sont la pile et le tas. La variable v est allouée sur la pile, donc le nombre 717 est stocké dans la pile, cependant, le maillon contenant la valeur 717 qui est inséré dans la liste est alloué sur le tas, donc ce nombre est présent dans les deux zones mémoires. On peut aussi noter qu'il est alloué une seconde fois sur la pile lors de l'appel `insere` mais est libéré à la fin de son exécution.

Partie IV. Diverses représentations de nombres (OCaml) – 19 pts

On souhaite en OCaml regrouper dans un même type différentes manières de représenter des nombres, afin de faciliter leur utilisation :

- les nombres entiers, représentés par le type `int` ;
- les nombres rationnels (non nuls), représentés par deux entiers premiers entre eux ;
- les nombres réels, représentés par le type `float` .

On a ainsi les définitions de type suivantes :

```
type frac = {num : int; denom : int}
type num = Integer of int | Fraction of frac | Real of float
```

On note E l'ensemble des éléments de type `num`. On utilise pour comparer deux éléments de E l'ordre usuel \leq sur les réels. Dans la suite, on omet la valeur particulière `Real nan` qui n'est comparable à aucun élément, afin de garantir la réflexivité de \leq .

□ 16 – Montrer que (E, \leq) n'est pas un ensemble ordonné.

Réponse 16. `Int 0` et `Real 0` sont chacun plus petit que l'autre.

□ 17 – On considère la relation \equiv suivante :

$$\forall x, y \in E^2, x \equiv y \Leftrightarrow x \leq y \wedge y \leq x,$$

où \Leftrightarrow désigne l'équivalence (si et seulement si), et \wedge désigne la conjonction (et). Montrer que \equiv est une relation d'équivalence.

Réponse 17. Réflexivité : $x \leq x$ donc $x \equiv x$ (réflexivité de \leq)

Symétrie : Si $x \leq y$ et $y \leq x$ alors $y \leq x$ et $x \leq y$

Transitivité : Si $x \leq y$ et $y \leq x$ et $y \leq z$ et $z \leq y$, alors $x \leq z$ et $z \leq x$ (transitivité de \leq)

On considère maintenant l'ensemble \bar{E} des classes d'équivalences de \equiv .

□ 18 – Soit $x, y \in E^2$ tels que $x \leq y$ et $\bar{x}, \bar{y} \in \bar{E}^2$ leurs classes d'équivalences. Montrer

$$\forall x' \in \bar{x}. \forall y' \in \bar{y}. x' \leq y'$$

Réponse 18. $x' \leq x \leq y \leq y'$ par définition des classes d'équivalences puis transitivité de \leq .

On définit donc intuitivement la relation d'ordre \leq sur \bar{E} comme $\bar{x} \leq \bar{y}$ s'il existe $x \in \bar{x}, y \in \bar{y}$ tels que $x \leq y$.

□ 19 – Montrer que (\bar{E}, \leq) est un ensemble ordonné.

Réponse 19. La réflexivité et la transitivité découlent de ces propriétés de \leq sur E .

Antisymétrie : soit \bar{x} et \bar{y} tels que $\bar{x} \leq \bar{y}$ et $\bar{y} \leq \bar{x}$. Alors, il existe $x', x'' \in \bar{x}$ et $y', y'' \in \bar{y}$ tels que $x' \leq y'$ et $y'' \leq x''$. Or $x'' \leq x'$ donc $y'' \leq x'$ et $y' \leq y''$ donc $x' \leq y''$, donc $x' \equiv y''$ et $\bar{x} = \bar{y}$.

Cet ensemble est trivialement bien fondé car il ne possède qu'un nombre fini de valeur. Cependant, on peut s'intéresser à ce qu'il se passerait si notre type `int` n'était pas sujet au dépassement de capacité et pouvait représenter tout élément de \mathbb{N} (en pratique, ceci pourrait être implémenté par des listes d'entiers, ou encore des chaînes de caractères, ou des types plus élaborés, c'est le cas notamment dans le langage Python).

□ 20 – Montrer que l'ensemble ordonné (\mathbb{Q}^*, \leq) n'est pas bien fondé.

Réponse 20. $(\frac{1}{n})_n$ est une suite infinie strictement décroissante.

□ 21 – On définit la relation \leq_f suivante sur les fractions irréductibles :

$$\frac{n}{d} \leq_f \frac{n'}{d'} \text{ si } (n, d) = (n', d') \text{ ou } n + d < n' + d'$$

(\mathbb{Q}^*, \leq_f) est-il un ensemble ordonné? bien fondé? \leq_f est-elle totale?

Réponse 21. Oui, oui, non.

□ 22 – Écrire une fonction `float_of_num : num -> float` qui converti un nombre de type `num` en type `float`.

On pourra utiliser la fonction `float_of_int : int -> float`.

Réponse 22.

```
let float_of_num x = match x with
| Integer n -> float_of_int n
| Fraction f -> float_of_int f.num /. float_of_int f.denom
| Real x -> x
```

□ 23 – Écrire une fonction `add : num -> num -> num` qui calcule l'addition de deux nombres en suivant la convention suivante :

- si un des deux nombres est un réel, le résultat est un réel;
- sinon, si un des deux nombres est une fraction, le résultat est une fraction;
- sinon, le résultat est un entier.

On pourra supposer l'existence d'une fonction `add_frac : frac -> frac -> frac` qui effectue l'addition de deux fractions irréductibles et renvoie la somme sous la forme d'une fraction irréductible. Bonus : la coder.

Réponse 23.

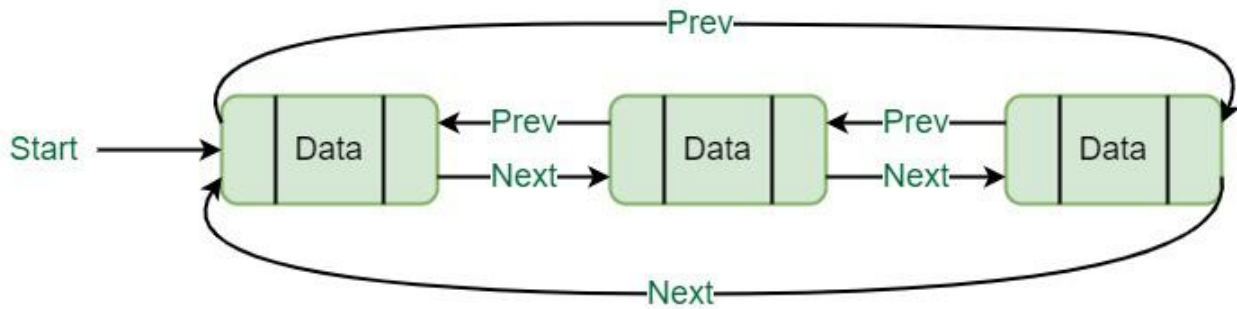
```
let rec pgcd a b = if a = 0 then b else pgcd (b mod a) a

let add_frac f1 f2 =
  let n = f1.num * f2.denom + f2.num * f1.denom in
  let d = f1.denom * f2.denom in
  let k = pgcd n d in
  {num = n/k; denom = d/k}

let add x y = match x, y with
| Real a, b | b, Real a
  -> Real (a +. float_of_num b)
| Fraction f1, Fraction f2
  -> Fraction (add_frac f1 f2)
| Fraction f, Integer n | Integer n, Fraction f
  -> Fraction (add_frac f {num = n; denom = 1})
| Integer n1, Integer n2
  -> Integer (n1 + n2)
```

Partie V. Listes doublement chaînées circulaires (C) – 14 pts

Voici un schéma illustrant le principe d'une liste doublement chaînée circulaire :



Cette structure de données permet d'implémenter facilement une file d'attente à double extrémité (deque), puisque l'on peut accéder facilement à la tête de la file ou à la queue de la file, ce qui permet d'avoir des opérations `enfiler` et `defiler` en $O(1)$ à droite comme à gauche. On propose l'implémentation suivante en C :

```

struct maillon_s {
    int val;
    struct maillon *prev;
    struct maillon *next;
};
typedef struct maillon_s maillon;

struct liste_s {
    maillon *start;
};
typedef struct liste_s liste;

liste *creer_liste(void) {
    liste *l = malloc(sizeof(liste));
    l->start = NULL;
    return l;
}

bool est_vide(liste *l) {
    return l->start == NULL;
}

```

□ 24 – Donner un ou plusieurs invariants qui doivent être vérifiés par tous les maillons pour que la structure reste correcte telle qu'illustrée sur le schéma.

Réponse 24. Pour tout maillon `m`, on a `m->next->prev == m` et `m->prev->next == m`.

□ 25 – Écrire une fonction `maillon *singleton(int x)` qui crée et renvoie l'unique maillon d'une liste qui ne contient que l'élément `x`.

Réponse 25.

```

maillon *singleton(int x){
    maillon *m = malloc(sizeof(maillon));
    m->val = x;
    m->next = m;
    m->prev = m;
    return m;
}

```

- 26 – Écrire une fonction `void pushR(int x, liste *l)` qui permet d'ajouter un élément x à la queue d'une liste circulaire doublement chaînée l . La complexité doit avoir pour $O(1)$.

Réponse 26.

```
void pushR(int x, liste *l) {
    if (est_vide(l)){
        l->start = singleton(x);
        return;
    }
    maillon *m = malloc(sizeof(maillon));
    m->val = x;
    maillon *last = l->start->prev;
    m->prev = last;
    m->next = l->start;
    last->next = m;
    l->start->prev = m;
}
```

- 27 – En déduire la fonction `void pushL(int x, liste *l)` qui ajoute l'élément x en tête de la liste l . La complexité doit être $O(1)$.

Réponse 27.

```
void pushL(int x, liste *l) {
    pushR(x, l);
    l->start = l->start->prev;
}
```

On suppose l'existence de la fonction d'affichage suivante :

```
void print_int(int x) {
    printf("%i\n", x);
}
```

- 28 – Écrire une fonction `void affiche_liste(liste *l)` qui affiche tous les éléments d'une liste circulaire doublement chaînée.

Réponse 28.

```

void affiche_liste(liste *l) {
    if (est_vide(l)) {
        return;
    }
    maillon *m = l->start;
    print_int(m->val);
    m = m->next;
    while (m != l->start) {
        print_int(m->val);
        m = m->next;
    }
}

```

□ 29 – Écrire une fonction `void supprime_maillon(liste *l, maillon *m)` qui supprime le maillon l de la liste m . On suppose que le maillon m appartient bien à la liste l , ce que l'on ne vérifiera pas. Si m est le maillon de tête, on fera attention à modifier le pointeur vers la tête de la liste. La complexité doit être $O(1)$.

Réponse 29.

```

void supprime_maillon(liste *l, maillon *m) {
    if (m->next == m) {
        l->start = NULL;
        free(m);
        return;
    }
    if (l->start == m) {
        l->start = m->next;
    }
    m->next->prev = m->prev;
    m->prev->next = m->next;
    free(m);
}

```

□ 30 – En déduire les fonctions `int popR(liste *l)` et `int popL(liste *l)` qui suppriment de la liste l et renvoie respectivement l'élément de queue de la liste l et l'élément de tête de la liste l . Les complexités doivent être $O(1)$.

Réponse 30.

```

int popR(liste *l){
    int x = l->start->prev->val;
    supprime_maillon(l, l->start->prev);
    return x;
}

int popL(liste *l){
    int x = l->start->val;
    supprime_maillon(l, l->start);
    return x;
}

```

Partie VI. Tri de pile (OCaml) – 18 pts

On rappelle les fonctions suivantes du module `Stack` qui implémente les piles.

- `Stack.create : unit -> 'a Stack.t`
Return a new stack, initially empty.
- `Stack.push : 'a -> 'a Stack.t -> unit`
`Stack.push x s` adds the element `x` at the end of the stack `s`.
- `Stack.pop : 'a Stack.t -> 'a`
`Stack.pop s` removes and returns the first element in stack `s`, or raises `Stack.Empty` if the stack is empty.
- `Stack.top : 'a Stack.t -> 'a`
`Stack.top s` returns the topmost element in stack `s`, or raises `Stack.Empty` if the stack is empty.
- `Stack.is_empty : 'a Stack.t -> bool`
Return `true` if the given stack is empty, `false` otherwise.

□ 31 – Écrire une fonction `retourne : 'a Stack.t -> 'a Stack.t -> unit` qui permet de retourner une pile sur le dessus d'une autre. Après l'appel `retourne s1 s2`, la pile `s2` contient sur le dessus les éléments de la pile `s1` dans l'ordre inverse. Les éléments de `s2` restent au dessous. La pile `s1` est vidée.

Réponse 31.

```
let rec retourne s1 s2 =
  try Stack.push (Stack.pop s1) s2;
    retourne s1 s2
  with Stack.Empty -> ();;
```

Nous souhaitons trier une pile en faisant à chaque étape "couler" le plus grand élément vers le bas de la pile. Voici une fonction qui implémente cette idée en OCaml.

```
let coule s1 =
  let rec coule_aux s1 cur_max s2 =
    if not (Stack.is_empty s1) then
      begin
        let x = Stack.pop s1 in
        let (min,max) =
          if x > cur_max then
            (cur_max,x)
          else
            (x,cur_max)
        in
        Stack.push min s2;
        coule_aux s1 max s2
      end
    else
      Stack.push cur_max s2
  in try
    let s2 = Stack.create () in
    let cur_max = Stack.pop s1 in
    coule_aux s1 cur_max s2; retourne s2 s1
  with Stack.Empty -> ();;
```

□ 32 – Que fait la partie suivante de la fonction ?

```
let (min,max) =
  if x > cur_max then
    (cur_max,x)
  else
    (x,cur_max)
in
```

Réponse 32. Elle définit localement la variable `min` comme étant la plus petite des deux valeurs `x` et `cur_max`, et `max` comme étant la plus grande des deux.

□ 33 – Si on applique cette fonction à une pile contenant les éléments `[3; 2; 1; 5; 4]`, le 3 étant sur le dessus de la pile, qu'obtient-on à la fin de la fonction ?

Réponse 33. Après appel de la fonction la pile contient `[2; 1; 3; 4; 5]`.

□ 34 – Que se passe-t-il si on applique cette fonction à une pile déjà triée (avec le plus petit élément au dessus) ?

Réponse 34. À chaque étape, on empile `cur_max` et on fait un appel récursif où `cur_max` prend la valeur suivante de `s1`.

Donc par récurrence, à la fin de l'appel à `coule_aux`, la pile `s2` contient tous les éléments qui étaient dans `s1` dans l'ordre inverse.

Après l'appel à `retourne`, `s1` contient donc la même chose qu'au début.

□ 35 – Que se passe-t-il si on applique cette fonction à une pile triée à l'envers (avec le plus petit élément au dessous) ?

Réponse 35. La valeur de `cur_max` est prise par le premier élément de la pile et ne change jamais. On retrouve donc dans `s2` tous les éléments de `s1` dans l'ordre inverse, sauf le premier qui se retrouve tout en haut.

Après l'appel à `retourne`, l'élément le plus grand se retrouve tout en bas, et l'ordre des autres éléments reste inchangé.

□ 36 – Démontrer que si la pile `s1` contient au moins $k + 1$ éléments dont les k plus grand éléments sont triés sur le dessous de la pile, après l'appel `coule s1`, `s1` contient au moins $k + 1$ éléments triés à la fin.

Réponse 36. Si on note s_1, s_2, \dots, s_{k+1} les $k + 1$ plus grands éléments, au moment de dépiler s_k , `cur_max` a pour valeur s_{k+1} , et l'ordre de ces éléments est conservé comme prouvé à la question 34.

□ 37 – Combien d'appels à la fonction `coule` doit-on effectuer pour être sur que la liste soit triée ?

Réponse 37. Si la pile contient n éléments, elle sera forcément triée au bout de n appels à `coule`, par récurrence en utilisant la question 36.

□ 38 – Proposer une modification (si votre programme est proche, indiquer les lignes qui changent, les ajouts et suppressions uniquement) de la fonction `coule` pour qu'elle renvoie un booléen indiquant si la liste est déjà triée ou non :

```
coule : 'a Stack.t -> bool
```

renvoie `true` si la liste est triée et `false` sinon.

On pourra utiliser une référence de booléen, définie avec le constructeur `ref`, à laquelle on affecte une nouvelle valeur en utilisant l'opérateur `:=`, et dont on lit la valeur avec l'opérateur préfixe `!`.

Réponse 38.

```

let coule s1 =
  let flag = ref true in
  let rec coule_aux s1 cur_max s2 =
    if not (Stack.is_empty s1) then
      begin
        let x = Stack.pop s1 in
        let (min,max) =
          if x >= cur_max then
            (cur_max,x)
          else
            (flag := false;(x,cur_max))
        in
        Stack.push min s2;
        coule_aux s1 max s2
      end
    else
      Stack.push cur_max s2
  in try
    let s2 = Stack.create () in
    let cur_max = Stack.pop s1 in
    coule_aux s1 cur_max s2; retourne s2 s1; !flag
  with Stack.Empty -> (); !flag;;

```

39 – Écrire une fonction `trie : 'a Stack.t -> unit` qui trie une pile.

Réponse 39.

```

let trie s = while (not (coule s)) do () done;;

```

Partie VII. Mots bien parenthésés (OCaml) – 8 pts

On souhaite déterminer si une chaîne de caractère contenant deux types de parenthèses () et [] est bien parenthésée ou non. Par exemple :

"Bonjour (je m'appe[ll]e) [Florian]"

est bien parenthésée, contrairement à

"He[ll]o (Worl]d)"

40 – Écrire une fonction `check : string -> bool` qui teste si une chaîne de caractères est bien parenthésée. Pour cette question, toute trace de recherche, même infructueuse sera prise en compte dans l'évaluation.

Réponse 40.

```
let dyck str =
  let s = Stack.create () in
  let rec parcours i =
    if i = String.length str then
      Stack.is_empty s
    else
      let c = str.[i] in
      if c = '(' || c = '[' then
        (Stack.push c s; parcours (i+1))
      else if c = ')' then
        Stack.pop s = '(' && parcours (i+1)
      else if c = ']' then
        Stack.pop s = '[' && parcours (i+1)
      else
        parcours (i+1)
  in parcours 0;;
```

FIN DE L'ÉPREUVE