Devoir surveillé nº 01 — 4h

LA CALCULATRICE N'EST PAS AUTORISÉE.

On veillera à présenter *très clairement* sa copie, on attachera un soin particulier à la rédaction et on encadrera les réponses. Jusqu'à deux points sur vingt pourront être consacrés à la présentation, à la lisibilité et au soin accordé à la rédaction. Il est impératif d'utiliser un brouillon.

Les programmes seront rédigés clairement et proprement, en mettant en valeur l'indentation et en choisissant des noms de variables explicites. Il est conseillé d'utiliser de la **couleur**, par exemple pour les **mots-clés** du langage CAML ou C. Les commentaires des programmes doivent dans tous les cas être dans une autre couleur que le programme lui-même. Il est impératif de respecter l'indentation usuelle des fonctions CAML et C.

L'introduction et l'utilisation de fonctions auxiliaires est autorisée, et même encouragée, d'autant plus si cela améliore la lisibilité et la compréhension. Les fonctions auxiliaires **doivent être précédées de leur type (en CAML) et commentées**.

Lorsque l'on pose des hypothèses sur les arguments, il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée.

De nombreuses questions comportent une sous-question sur la complexité. On prendra garde à ne pas lire trop vite l'énoncé et à ne pas ainsi oublier de répondre à une partie de la question.

Le sujet comporte un unique problème découpé en plusieurs parties relativement indépendantes. Cependant, une bonne compréhension des parties précédentes est souvent nécessaire pour aborder la partie suivante. Le sujet se veut progressif et est conçu pour être traité linéairement. Certaines parties sont à rédiger en CAML et d'autres en C.

Introduction

Les listes et les arbres CAML sont des structures persistantes qui se prêtent bien à des parcours récursifs. Cependant elle ne permettent pas des modifications locales, ajout ou suppression d'un élément au milieu de la structure par exemple, sans devoir la reconstruire en partie, ce qui n'est pas toujours efficace. Dans ce problème, on s'intéresse à une technique connue sous le nom de *zipper* [1] qui permet de naviguer et de modifier localement une telle structure de données, à la manière d'un *curseur* dans un texte. En parallèle, on présente l'utilisation de structures impératives en C qui permettent directement ces modifications locales.

1. Gérard HUET. The Zipper. *Journal of Functional Programing*, 7(5) : 549-554, Sempember 1997.

I Questions de cours (CAML)

Dans les deux questions suivantes, et uniquement dans celles-ci, il est interdit d'utiliser des fonctions de la bibliothèque `List` ou encore l'opérateur `@`. Par la suite, on pourra utiliser cette bibliothèque et en particulier les fonctions `List.rev_append` et `List.rev`, que l'on demande justement de réimplémenter ci-dessous.

Q 1) Écrire une fonction `rev_append : 'a list -> 'a list -> 'a list` tel que l'appel `rev_append l1 l2` renvoie une liste comportant les éléments de la liste `l1` dans l'ordre inverse suivis des éléments de la liste `l2` dans le même ordre. Par exemple `rev_append [1; 2; 3] [4; 5]` doit s'évaluer en la liste `[3; 2; 1; 4; 5]`. Cette fonction devra être récursive terminale. Quelle est sa complexité ?

Q 2) Écrire une fonction `rev : 'a list -> 'a list` qui renvoie le miroir d'une liste CAML. Cette fonction devra être récursive terminale ou faire appel à des fonctions récursives terminales et devra avoir une complexité linéaire en la taille de la liste, ce que l'on justifiera rapidement.

II Un petit éditeur de texte

Supposons que l'on veuille modéliser un petit éditeur de texte par une liste de caractères. On veut alors pouvoir disposer d'un *curseur* que l'on veut pouvoir déplacer en avant ou en arrière, et on veut pouvoir ajouter ou de supprimer des caractères à la position du curseur, de manière analogue à ce qui se passe dans un éditeur de texte comme EMACS.

Par exemple, supposons que le texte soit composé du mot « prodige » comportant sept lettres et que le curseur soit placé sur la quatrième lettre, comme illustré par la figure 1 :

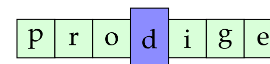

FIGURE 1 – Curseur placé sur la quatrième lettre, à l'indice 3.

On pourrait vouloir avancer le curseur de trois positions (figure 2).

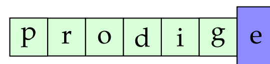

FIGURE 2 – Curseur placé sur la dernière lettre, à l'indice 6.

Et insérer un nouvel élément, juste avant la position du curseur (figure 3).

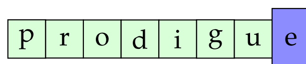

FIGURE 3 – Curseur placé sur la dernière lettre, à l'indice 7.

Puis avancer jusqu'à la fin pour être prêt à continuer la saisie de la suite (figure 4).

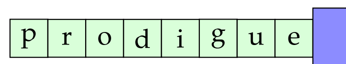

FIGURE 4 – Curseur placé à la toute fin, à l'indice 8.

II.1 Modélisation naïve (CAML)

Pour modéliser un curseur, on peut simplement utiliser un indice de position dans la liste. On propose donc le type CAML suivant :

OCAML
```
type 'a editor = int * 'a list
```

L'état de l'éditeur de la figure 1, où le curseur est placé sur la lettre d d'indice 3, est représenté par le couple `(3, ['p'; 'r'; 'o'; 'd'; 'i'; 'g'; 'e'])`.

L'éditeur de la figure 4 où le curseur est placé à la toute fin, *après* le dernier élément, est représenté par `(8, ['p'; 'r'; 'o'; 'd'; 'i'; 'g'; 'u'; 'e'])`.

☞ Remarque 1

Dans la question suivante, comme dans toutes les questions en CAML, on ne modifie pas les structures passées en paramètres, puisque l'on travaille avec des structures persistantes. Il faut donc comprendre que l'on demande de renvoyer une nouvelle structure identique à celle passée en paramètre à ceci près que le curseur est placé sur la position précédente.

Q 3) Écrire une fonction `back : 'a editor -> 'a editor` qui fait reculer le curseur d'une position. Le curseur ne doit pas bouger s'il était déjà en première position. Quelle est sa complexité ?

Q 4) Écrire de même une fonction `next : 'a editor -> 'a editor` qui fait avancer le curseur d'une position. Le curseur ne doit pas bouger s'il était déjà placé après le tout dernier élément. Quelle est sa complexité ? Commenter.

Q 5) Écrire une fonction `insert : 'a -> 'a editor -> 'a editor` qui insère un caractère juste avant la position du curseur, position que l'on suppose valide. Expliquer et commenter votre fonction. Quelle est sa complexité ?

On pourrait écrire de même une fonction `backspace` de type `'a editor -> 'a editor` qui supprime l'élément présent juste avant la position du curseur s'il existe et renvoie l'éditeur à l'identique sinon.

Q 6) Écrire une fonction `delete_all : 'a editor -> 'a editor` qui « supprime » tout le contenu d'un éditeur. Comme indiqué dans la remarque ci-dessus, en réalité cette fonction ne « supprime » rien, mais s'évalue simplement en un nouvel éditeur vide où le curseur est placé à la seule position possible. La complexité de cette fonction doit être en $\Theta(1)$.

II.2 Le zipper sur les listes (CAML)

Plutôt que d'utiliser la représentation précédente qui n'est pas vraiment efficace on se propose ici d'utiliser deux listes : l'une contenant les éléments avant le curseur (exclu) et l'autre contenant les éléments qui se trouvent après le curseur (inclus). On peut voir la première liste comme un chemin que l'on a suivi à partir du début du texte pour arriver juste avant le curseur. Pour des raisons d'efficacité, on représente plutôt ce chemin et donc cette liste « à l'envers », c'est-à-dire comme le chemin à suivre depuis le curseur jusqu'au début du texte (« $o \to r \to p$ » sur la figure 5).

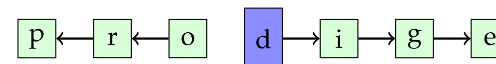

FIGURE 5 – Représentation du zipper contenant le mot « prodige » avec le curseur sur la quatrième lettre d en position 3.

C'est cette structure formée de deux listes que l'on appelle le *zipper*. L'image est celle d'une fermeture éclair (*zipper*) que l'on ouvre ou ferme lorsque l'on navigue dans un sens ou dans l'autre.

On considère donc le type CAML suivant :

OCAML
```ocaml
type 'a zipper = {path : 'a list; next : 'a list}
```

L' état de l'éditeur de la figure 1 pourrait donc être représenté par l'enregistrement `{path = ['o'; 'r'; 'p']; next = ['d'; 'i'; 'g'; 'e']}`.

Q 7) On avance le curseur d'une position à partir de l'état représenté ci-dessus. Dessiner le zipper obtenu et donner l'enregistrement CAML correspondant.

Q 8) Écrire une fonction `next : 'a zipper -> 'a zipper` qui fait avancer d'un cran la position du curseur. Le curseur ne doit pas bouger s'il était déjà placé après le tout dernier élément. Quelle est sa complexité ? Commenter.

On écrirait de même une fonction `back : 'a zipper -> 'a zipper` qui fait reculer le curseur si possible.

Q 9) Écrire une fonction `insert : 'a -> 'a zipper -> 'a zipper` qui insère un élément juste avant la position du curseur. Remarquons que la position du curseur est nécessairement valide avec cette représentation. Quelle est sa complexité ?

Q 10) Écrire une fonction `backspace : 'a zipper -> 'a zipper` qui supprime l'élément juste avant la position du curseur. Si le curseur est en toute première position cette fonction renvoie le zipper à l'identique.

Q 11) Écrire une fonction `of_list : 'a list -> 'a zipper` qui initialise un zipper à partir d'une liste passée en argument où le curseur est placé tout à gauche, c'est-à-dire au tout début de la liste. Quelle est sa complexité ?

Q 12) Écrire une fonction `to_list : 'a zipper -> 'a list` qui converti un zipper en liste. Cette fonction doit être correcte quelle que soit la position du curseur. Quelle est sa complexité ?

II.3 Listes doublement chaînées (C)

On se propose d'utiliser en C une liste doublement chaînée (mutable) pour implémenter un éditeur. On considère la structure C ci-contre pour représenter un maillon (`cell`) d'une liste doublement chaînée. Un maillon comporte un champ `data` contenant un caractère ainsi qu'un pointeur vers le maillon précédent et le maillon suivant. Un texte est représenté par une liste doublement chaînée de maillons, terminée par une sentinelle qui sera un maillon fictif, dont le champ `data` sera le caractère nul `'\0'` et le pointeur `next` sera toujours `NULL`.

C
```c
struct cell {
  char data;
  struct cell* prev;
  struct cell* next;
};

typedef struct cell cell;
```

Remarquons qu'une liste doublement chaînée ne sera donc jamais vide puisqu'elle comportera toujours au moins la sentinelle : le dernier maillon fictif. Un éditeur sera donc simplement un pointeur (jamais nul) `cursor` de type `cell*` vers le maillon qui est sous la position du curseur (figure 6), la sentinelle jouant le rôle du maillon représentant la toute fin du texte (figure 7).

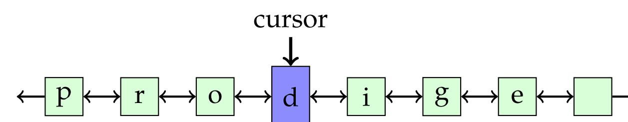

FIGURE 6 – Représentation de la liste doublement chaînée contenant le mot « prodige » avec le curseur sur la quatrième lettre *d* en position 3.

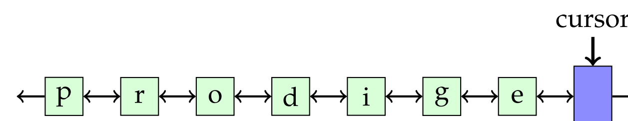

FIGURE 7 – Représentation de la liste doublement chaînée contenant le mot « prodige » avec le curseur à la fin en position 7.

Q 13) Écrire une fonction de prototype `cell* empty_editor(void)` qui créé et initialise un editeur sans texte, avec un curseur au seul endroit possible et qui renvoie ce curseur sur cet éditeur « vide ».

Q 14) Écrire une fonction de prototype `cell* next(cell* cursor)` qui « fait avancer » le curseur d'un cran si possible et le laisse sur place sinon. Quelle est la complexité de cette fonction ? *Indication : vu le prototype de cette fonction, il s'agit de renvoyer un pointeur vers le maillon suivant s'il y en a un. Cette fonction ne « modifie » rien.*

On suppose écrite de même une fonction `cell* back(cell* cursor)` qui fait reculer le curseur d'un cran si possible.

On cherche à écrire une fonction `cell* insert(char data, cell* cursor)` qui insère un caractère juste avant le curseur. On remarque qu'il n'est en réalité pas nécessaire de renvoyer le curseur puisque celui-ci pointe nécessairement toujours vers le même maillon après l'insertion, mais nous allons conserver des signatures homogènes.

Un élève propose la fonction suivante :

C

```c
cell insert(char data, cell* cursor) {
  cell* c = malloc(sizeof(cell*);
  c->data = data;
  c->prev = cursor->prev;
  c->next = cursor;
  if (cursor->prev <> NULL) {
    cursor->prev->next = c
  }
  cursor->prev = c;
  return cursor;
}
```

Q 15) Cette fonction comporte exactement 5 erreurs différentes. Les identifier explicitement en étant très clair sur la nature de ces erreurs. *Indications : il s'agit plutôt de coquilles et non pas d'erreurs d'algorithmique. Ne pas transtyper (cast) le retour de* `malloc` *n'est pas une erreur en* C.

Q 16) Expliquer à l'aide d'un schéma le comportement de cette fonction, que l'on suppose désormais corrigée.

Q 17) Quelle est la complexité de cette fonction ?

Q 18) Écrire une fonction de prototype `cell* backspace(cell* cursor)` qui supprime l'élément juste avant le curseur, si celui-ci existe et ne fait rien sinon. Ici encore, cette fonction va modifier l'éditeur tout en renvoyant le *même* curseur que celui passé en argument. Quelle est la complexité de cette fonction ? *Indication : attention à ne rien oublier !*

Q 19) Écrire une fonction de prototype `void free_editor(cell* cursor)` qui libère entièrement la mémoire allouée pour un éditeur. Quelle est la complexité de cette fonction ? *Attention : le curseur peut a priori être à n'importe quelle position !*

Q 20) Écrire une fonction de prototype `cell* from_string(char* str)` qui créé un éditeur contenant une chaîne de caratères passée en argument et qui renvoie un curseur placé à la toute fin. Quelle est la complexité de cette fonction ?

Q 21) Que pourrait-on faire pour avoir un curseur sur le début du texte plutôt qu'à la fin, comme c'était le cas pour `of_list` dans la partie précédente ? Quelle serait alors la complexité de cette fonction ? *On ne demande pas de le faire, mais simplement d'expliquer en une ou deux phrases.*

Q 22) Écrire une fonction de prototype `char* to_string(cell* cursor)` qui renvoie une chaîne de caractères contenant le texte d'un éditeur étant donné un curseur placé n'importe où. N'oubliez pas d'expliquer votre démarche et de commenter votre fonction. Quelle est la complexité de cette fonction ?

III Le zipper pour les arbres (CAML)

On généralise maintenant cette notion de curseur sur une liste aux arbres binaires. On considère le type suivant pour représenter les arbres en CAML, dans lequel le constructeur `E` représente l'arbre vide (*empty*) et le constructeur `N` un nœud (*node*) :

OCAML

```ocaml
type 'a tree =
  | E
  | N of 'a tree * 'a * 'a tree
```

Quel est l'analogue d'un curseur pour un arbre ? C'est la désignation d'un nœud particulier et la possibilité de naviguer[2], c'est-à-dire de remonter, descendre à gauche ou à droite dans l'arbre.

Considérons l'arbre de la figure 8, à la page suivante, dans lequel on place le curseur sur le nœud d. Le zipper contient à la fois l'arbre « avant » le curseur d'un côté (sans couleur de fond) ainsi que l'arbre « après » le curseur de l'autre (sous le triangle coloré). Représenter l'arbre « après » le curseur est facile : il s'agit simplement du sous-arbre enraciné en ce nœud. Pour représenter l'arbre « avant » le curseur, on va généraliser la notion de chemin de la racine à un nœud. Un chemin de la racine à un nœud de l'arbre peut être donné par une séquence de déplacements « gauche » ou « droit ». Par exemple, le chemin à suivre pour arriver au curseur est « gauche-droite ».

2. Ainsi que d'insérer ou supprimer localement, ce que l'on pourrait faire mais qui nous mènerait un peu trop loin dans ce sujet.

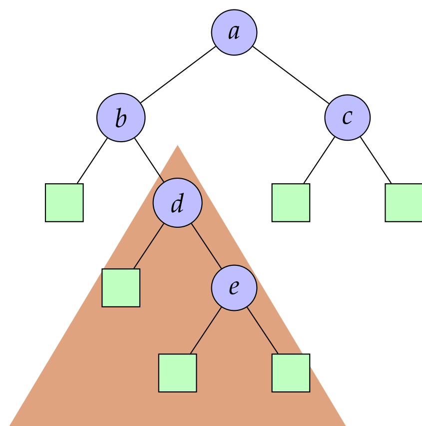

FIGURE 8 – Un curseur dans un arbre.

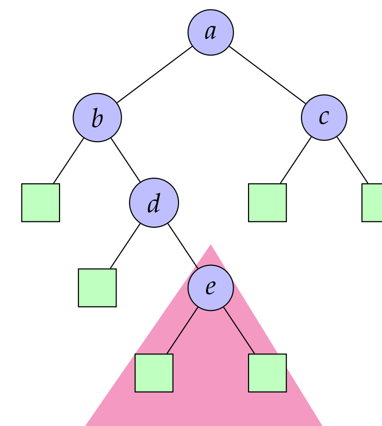

FIGURE 9 – Le curseur obtenu en allant « à droite » à partir du curseur de la figure 8. À partir de ce nouveau curseur, aller « à droite » ou « à gauche » mène sur un arbre vide. « Remonter » nous replace dans la configuration de la figure 8.

Supposons que l'on souhaite déplacer le curseur « à droite ». Cela revient à descendre dans le sous-arbre droit. On obtient alors le curseur de la figure 9 ci-contre. Le chemin depuis la racine est maintenant « gauche-droite-droite ».

On introduit le type suivant pour représenter un chemin tout en conservant le reste de l'arbre.

OCAML
```
type 'a path =
  | Top
  | Left of 'a path * 'a * 'a tree
  | Right of 'a tree * 'a * 'a path
```

Le constructeur `Top` représente le chemin vide, c'est-à-dire la position de la racine de l'arbre. Les constructeurs `Left` et `Right` indiquent respectivement un déplacement vers le sous-arbre gauche ou droit. Comme on veut représenter tout l'arbre « avant » le curseur, on conserve également le sous-arbre dans lequel on n'est pas descendu.

Un zipper pour un arbre binaire est alors donné par le type suivant :

OCAML
```
type 'a zipper = {path : 'a path; next : 'a tree}
```

Le champ `next` contient le sous-arbre sous le curseur. Le champ `path` contient le chemin du nœud vers la racine. **Comme pour les listes, on va stocker le chemin dans l'autre sens, du nœud vers la racine, toujours pour des raisons d'efficacité**. Par exemple le zipper correspondant à la figure 8 ci-dessus est :

OCAML
```
let p = Right (E, 'b', Left (Top, 'a', N (E, 'c', E))) in
let n = N (E, 'd', N (E, 'e', E)) in
{path = p; next = n}
```

Le premier constructeur du chemin est `Right` car le curseur désigne un sous-arbre droit. La suite du chemin est `Left` car le nœud b est la racine d'un sous-arbre gauche. Enfin, le chemin atteint la racine a de l'arbre.

Q 23) Donner, en CAML, le zipper correspondant au deuxième exemple, c'est-à-dire après avoir fait descendre le curseur à droite. Présentez cela lisiblement !

Q 24) Écrire une fonction `of_tree : 'a tree -> 'a zipper` qui initialise un zipper où le curseur est placé à la racine de l'arbre.

Pour déplacer le curseur « à gauche », on renvoie un zipper où le curseur est le sous-arbre gauche du curseur et où on étend le chemin avec le constructeur `Left`

pour signifier que l'on est descendu à gauche. On choisit ici de ne pas bouger s'il est impossible de descendre à gauche.

OCaml

```
let move_left z =
  match z.next with
  | E -> z
  | N (l, x, r) ->
     {path = Left (z.path, x, r); next = l}
```

Le noeud d'étiquette `x` et son sous-arbre droit `r` sont conservés dans le chemin, comme arguments du constructeur `Left`.

Q 25) Écrire de même, *en utilisant les mêmes noms de variables*, la fonction `move_right` de type `'a zipper -> 'a zipper` qui déplace le curseur « à droite ».

Q 26) Écrire de même la fonction `move_up : 'a zipper -> 'a zipper` qui fait remonter le curseur vers la racine si cela est possible.

Q 27) Écrire une fonction `to_tree : 'a zipper -> 'a tree` qui à partir d'un zipper dans une position quelconque reconstruit l'arbre tout entier.

IV Comparaison d'arbres binaires de recherche (Caml)

Nous avons vu en cours que l'on pouvait représenter un ensemble à l'aide d'un arbre binaire de recherche pour peu que cet ensemble soit totalement ordonnée. Si l'on dispose d'une fonction `compare : 'a -> 'a -> int` tel que `compare x y` renvoie un entier strictement négatif si x est considéré comme strictement plus petit que y, l'entier 0 si x et y sont considérés comme égaux et un entier strictement positif si x est considéré comme strictement supérieure à y, alors on peut utiliser un arbre binaire de recherche étiqueté par ce type `'a` en utilisant cette fonction pour comparer les éléments entre eux.

Q 28) Rappeler la définition d'un arbre binaire de recherche. On cherche ici à représenter des ensembles donc on pourra supposer que toutes les étiquettes représentent des éléments distincts deux à deux.

On représente un arbre binaire de recherche avec le type `'a tree` introduit précédemment.

Q 29) Écrire une fonction `get : ('a -> 'a -> int) -> 'a -> 'a tree -> 'a option` tel que l'appel `get cmp x t` recherche un élément `x` dans un ensemble représenté par un arbre binaire de recherche `t` et renvoie une option sur l'élément `y` trouvé (qui est tel que `cmp x y = 0`) s'il y en a un et `None` sinon. On utilisera la fonction `cmp : 'a -> 'a -> int` passée en argument pour comparer les éléments entre eux. Cette fonction peut avoir un coût important et ne doit être appelée qu'au plus une fois par appel récursif. Sans compter les appels à `cmp`, quelle est la complexité de la fonction `get` ?

On suppose pour toute la suite que l'on a écrit de même une fonction `insert` de type `('a -> 'a -> int) -> 'a -> 'a tree -> 'a tree` qui permet d'insérer un élément dans un arbre binaire de recherche. Si l'élément ne s'y trouvait pas il est ajouté, sinon l'élément remplace celui qui s'y trouvait.

On considère la fonction suivante :

OCaml

```
let rec to_set cmp liste =
  let t = E in
  match liste with
  | [] -> t
  | tete :: queue ->
     insert cmp tete t;
     to_set cmp queue
```

Q 30) Déterminer le type et le comportement de cette fonction.

Q 31) Écrire une fonction `delete : ('a -> 'a -> int) -> 'a -> 'a tree -> 'a tree` qui supprime un élément d'un arbre binaire de recherche s'il existe et qui renvoie l'arbre à l'identique sinon. Expliquer votre démarche.

On souhaite représenter un ensemble d'ensembles. Pour cela, il est possible d'utiliser un arbre binaire de recherche dont les éléments sont eux-mêmes des arbres binaires de recherche, à condition de munir les arbres binaires de recherche d'un ordre total. On représente donc un ensemble par un arbre binaire de recherche de type `'a tree` et un ensemble d'ensembles par un arbre binaire de recherche de type `'a tree tree`. Pour cela, il faut disposer d'une fonction `compare_tree : 'a tree -> 'a tree -> int` qui permet de comparer deux arbres binaires de recherche, en supposant connue une fonction `compare_elt : 'a -> 'a -> int` qui permet de comparer les éléments eux-mêmes.

Une solution pour implémenter `compare_tree` consiste à construire la liste des éléments pour chacun des deux arbres, dans l'ordre infixe, pour ensuite comparer ces deux listes pour l'ordre lexicographique en utilisant `compare_elt`.

Q 32) Que peut-on dire de la liste des éléments donnée par un parcours infixe d'un arbre binaire de recherche ?

C'est cependant un peu naïf car les deux arbres peuvent contenir beaucoup d'éléments mais différer rapidement. On aurait alors construit les deux listes inutilement.

On préférerait une solution plus efficace, travaillant directement sur les arbres. Ce n'est pas simple cependant car deux arbres binaires de recherche peuvent contenir les mêmes éléments sans pour autant avoir la même structure.

Q 33) Justifier que deux arbres binaires de recherche différents peuvent représenter le même ensemble.

On se propose d'utiliser la structure de zipper pour effectuer un parcours infixe *simultané* des deux arbres.

On propose les deux fonctions suivantes :

OCaml

```
let rec myst1 z =
  match z.next with
  | E -> None
  | N (E, _, _) -> Some z
  | _ -> myst1 (move_left z)

let rec myst2 z =
  match z.path with
  | Top -> None
  | Right _ -> myst2 (move_up z)
  | Left _ -> Some (move_up z)
```

Q 34) Déterminer le type et le comportement de ces deux fonctions. On expliquera simplement et clairement ce qu'elles réalisent et on proposera pour chacune un nom adapté.

Q 35) Écrire une fonction `next_infix : 'a zipper -> 'a zipper option` qui, étant donné un zipper passé en paramètre dont le curseur est placé sur un nœud renvoie une option sur un zipper où le curseur est placé sur le nœud qui serait le successeur direct dans un parcours infixe de l'arbre, si celui-ci existe ou `None` sinon. Si le curseur du zipper est placé sur un arbre vide et non sur un nœud, on renverra également `None`.

On suppose donné une fonction `label : 'a tree -> 'a` qui renvoie l'étiquette d'un arbre binaire supposé non vide. On considère le programme incomplet ci-dessous qui vise à comparer deux arbres binaires de recherche représentant deux ensembles. On suppose connue une fonction `cmp : 'a -> 'a -> int` qui permet de comparer des éléments du type de ceux des ensembles. Si `t1` et `t2` sont deux arbres binaires de recherche, on considère que `t1` et `t2` sont égaux s'ils comportent les mêmes éléments et `compare_tree cmp t1 t2` doit renvoyer `0`. Sinon, soit `x` le plus petit élément pour `cmp` qui est dans un seul des deux ensembles représentés par `t1` et `t2`. Si `x` est dans `t1` alors on considère que `t1` est strictement plus petit que `t2` et `compare_tree cmp t1 t2` doit renvoyer `-1`. Le dernier cas est symétrique. Remarquons que cela revient à considérer l'ordre lexicographique sur la liste triée des éléments de `t1` et `t2`. La fonction `compare_tree` est de type `('a -> 'a -> int) -> 'a tree -> 'a tree -> int`. On se propose de faire un parcours infixe simultané des deux arbres. Pour les deux arbres, on initialise un zipper en plaçant le curseur sur le premier élément à considérer, qui est donc le minimum de l'arbre binaire de recherche, s'il existe. Puis, du moins tant que l'on n'a pas rencontré de différence entre les éléments, on avance simultanément les deux curseurs. Si l'un des deux curseurs se termine avant l'autre, on agit en conséquence.

OCaml

```
let compare_tree cmp t1 t2 =
  let zo1 = (* à compléter *) in
  let zo2 = (* à compléter *) in
  let rec parallel_infix zo1 zo2 =
    match zo1, zo2 with
    | None, None -> (* à compléter *)
    | None, _ -> (* à compléter *)
    | _, None -> (* à compléter *)
    | Some z1, Some z2 ->
       let x1 = label z1.next in
       let x2 = label z2.next in
        (* à compléter *)
  in
  parallel_infix zo1 zo2
```

Q 36) Recopier et compléter le programme ci-dessus en mettant en évidence et en couleur les parties complétées.

V Parcours d'arbres itératifs (C)

Comme précédemment pour les listes, une structure mutable à base de pointeurs permet également de modéliser un curseur dans un arbre. Il suffit de suivre les pointeurs pour se déplacer dans l'arborescence.

On considère le type suivant pour représenter un arbre binaire de recherche sur les entiers en C :

```c
struct node {
  int val;
  struct node* left;
  struct node* right;
  struct node* parent;
}
typedef struct node node;
```

Un arbre binaire de recherche `bst` (*Binary Search Tree*) est représenté par un pointeur sur la racine de cet arbre et donc par le type `node*`. Un arbre vide est donc le pointeur nul `NULL`. Un nœud contient une clé, un pointeur vers son fils gauche et son fils droit ainsi qu'un pointeur vers son nœud parent. La racine de l'arbre est le seul nœud dont le parent est `NULL`.

Q 37) Écrire une fonction purement itérative `node* minimum(node* bst)` qui renvoie un pointeur vers un nœud de clé minimale d'un arbre binaire de recherche, s'il en existe un, et un pointeur `NULL` sinon. L'arbre peut être vide.

Q 38) Écrire une fonction `node* next_infix(node* n)` qui renvoie le successeur d'un nœud dans un parcours infixe d'un arbre binaire de recherche s'il en existe un et `NULL` sinon. Le nœud `n` peut-être `NULL`.

On pourrait ensuite, de même que dans la partie précédente, utiliser ces deux fonctions pour réaliser un parcours infixe d'un ou plusieurs arbres en parallèle.

VI Mains de départ dans un jeu de cartes (CAML)

On considère le problème suivant : un joueur passionné de jeux de cartes mais dont la mémoire commence à flancher souhaite établir quelques statistiques sur ses mains de départ à son jeu de cartes préférée (par exemple au bridge, au tarot, aux cartes MAGIC, aux cartes POKEMON, etc.). Il aimerait comptabiliser le nombre de parties qu'il a déjà jouées avec une main de départ donnée.

On se donne un type `carte` qui représente une carte, comme par exemple celui que nous avons rencontré en TP et une fonction `compare_carte : carte -> carte -> int` qui permet de comparer deux cartes.

On représente les statistiques de parties du joueur par un dictionnaire associant à un ensemble de cartes (une main de départ), qui sera la clé, le nombre de parties jouées avec cette main de départ, qui est donc un entier positif correspondant à la valeur associée à la clé. Lorsque le joueur joue une partie, il incrémente la valeur associé à la clé, correspondant à sa main de départ (si elle était présente).

Rappelons que l'on peut utiliser un arbre binaire de recherche de type `('a * 'b) tree` pour modéliser un dictionnaire associant à des clés de type `'a` des valeurs de type `'b`. Rappelons qu'une clé ne peut être présente qu'une seule fois.

Q 39) Définir un type `main` en CAML pour représenter un ensemble de cartes implémenté par un arbre binaire de recherche.

Q 40) Définir un type `stats` en CAML pour représenter un dictionnaire implémenté par un arbre binaire de recherche, où les clés sont des mains et les valeurs associées des entiers.

On peut directement utiliser les fonctions précédentes sur les arbres binaires de recherche lorsque l'on représente des ensembles, mais également lorsque l'on représente des dictionnaires, à condition d'utiliser une fonction de comparaison bien choisie. On considère la fonction suivante :

```ocaml
let compare_key (k1, v1) (k2, v2) =
  compare_tree compare_carte k1 k2
```

Q 41) On considère une main `m : main` qui a déjà été jouée 42 fois et donc associée à la valeur 42 dans un dictionnaire `s : stats`. Que renvoie `get compare_key (m, 42) s` ? Et `get compare_key (m, 0) s` ? Si ce n'est pas déjà le cas, modifier votre fonction `get` pour que le deuxième appel renvoie *la même chose* que le premier. *Indication : il faut que cette fonction renvoie la valeur trouvée sur le nœud et pas celle recherchée. Même si ces deux valeurs sont égales pour la fonction* `compare_key` *ce ne sont pas exactement les mêmes.*

Q 42) Écrire une fonction `nb_parties : main -> stats -> int` qui renvoie le nombre de fois qu'un joueur a commencé avec une main de départ.

Q 43) Écrire une fonction `ajoute_partie : main -> stats -> stats` qui met à jour les statistiques d'un joueur qui vient de jouer une partie avec une main de départ passée en argument.

Suite à un changement dans la réglementation, une main de départ n'est désormais plus possible. Le joueur aimerait donc la supprimer de sa base de données.

Q 44) Écrire une fonction `supprime_main : main -> stats -> stats` qui supprime une main donnée de ses statistiques.

— FIN DU SUJET —