Devoir surveillé n° 01 — corrigé

I Questions de cours (CAML)

Q 1) C'est une question de cours. Il suffit de transvaser récursivement la première liste sur la deuxième. La fonction est naturellement récursive terminale. La complexité est linéaire en la taille de la première liste.

OCAML
```
let rec rev_append l1 l2 =
  match l1 with
  | [] -> l2
  | t1 :: q1 -> rev_append q1 (t1 :: l2)
```

Q 2) Il suffit de transvaser le contenu de la liste dans une liste vide en utilisant la fonction précédente. La complexité est donc bien linéaire en la taille de la liste.

OCAML
```
let rev lst = rev_append lst []
```

II Un petit éditeur de texte

II.1 Modélisation naïve (CAML)

Q 3) Il faut simplement gérer le cas où le curseur est déjà au tout début. La complexité est constante en $\Theta(1)$.

OCAML
```
let back (cursor, text) =
  if cursor = 0 then
    (0, text)
  else
    (cursor - 1, text)
```

Q 4) On fait de même, en gérant à part le cas où le curseur est à la dernière position possible. Attention, cette position correspond à la longueur de la liste (cas où le curseur est à la toute fin) et pas à la position du dernier élément. L'utilisation de la fonction `List.length` impose une complexité linéaire en la taille de la liste. Ce n'est pas très satisfaisant. On pourrait ajouter cette taille dans notre structure de données pour y avoir accès en temps constant.

OCAML
```
let next (cursor, text) =
  if cursor = List.length text then
    (cursor, text)
  else
    (cursor + 1, text)
```

Q 5) On écrit une fonction auxiliaire récursive `insert_before` de type `int -> 'a list -> 'a list` qui réalise l'insertion juste avant une position donnée. Attention, la position du curseur après insertion est décalée. La complexité est linéaire en la position du curseur puisqu'il faut parcourir (et reconstruire) toute la liste jusqu'à cette position.

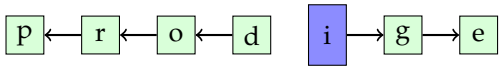
OCAML
```
let insert elt (cursor, text) =
  let rec insert_before i liste =
    match i, liste with
    | 0, _ -> elt :: liste
    | _, [] -> failwith "indice non valable"
    | _, tete :: queue ->
      tete :: (insert_before (i - 1) queue)
  in
  (cursor + 1, insert_before cursor text)
```

Q 6) Comme indiqué, il suffit de renvoyer un éditeur vide. La complexité est en $\Theta(1)$.

OCAML
```
let delete_all (cursor, text) = (0, [])
```

II.2 Le zipper sur les listes (CAML)

Q 7) On obtient le zipper l'enregistrement ci-dessous.



`{path = ['d'; 'o'; 'r'; 'p']; next = ['i'; 'g'; 'e']}`.

Q 8) Si on peut avancer, c'est-à-dire si le champ `z.next` n'est pas vide, on transvase un élément de la suite au chemin, en $\Theta(1)$. On n'a pas besoin de connaître la longueur de l'éditeur, on est à la fin si `z.next` est une liste vide.

OCaml
```
let next z =
  match z.next with
  | [] -> z
  | hd :: tl -> {path = hd :: z.path; next = tl}
```

Q 9) Il suffit d'ajouter l'élément en tête du chemin, en $\Theta(1)$.

OCaml
```
let insert elt z =
  {path = elt :: z.path; next = z.next}
```

Q 10) De même, on supprime la tête du chemin, si possible.

OCaml
```
let backspace z =
  match z.path with
  | [] -> z
  | _ :: tl -> {path = tl; next = z.next}
```

Q 11) C'est immédiat en $\Theta(1)$.

OCaml
```
let of_list lst =
  {path = []; next = lst}
```

Q 12) On renverse le chemin et on le place devant ce qu'il reste. C'est exactement ce que réalise la fonction `List.rev_append`. La complexité est linéaire en la longueur du chemin et donc en la position du curseur.

OCaml
```
let to_list z =
  List.rev_append z.path z.next
```

II.3 Listes doublement chaînées (C)

Q 13) On créé le maillon fictif et on renvoie un pointeur vers cette sentinelle.

C
```
cell* empty_editor(void) {
  cell* cursor = malloc(sizeof(cell));
  cursor->data = '\0';
  cursor->prev = NULL;
  cursor->next = NULL;
  return cursor;
}
```

Q 14) Comme indiqué dans la remarque, on renvoie un pointeur vers le maillon suivant, s'il existe. La complexité est en $\Theta(1)$.

C
```
cell* next(cell* cursor) {
  if (cursor->next != NULL) {
    return cursor->next;
  } else {
    return cursor;
  }
}
```

Q 15) Les 5 erreurs sont :

- `cell` au lieu de `cell*` dans le type de retour ;
- **`sizeof`**`(cell*)` au lieu de **`sizeof`**`(cell)` dans le `malloc` ;
- Manque une parenthèse fermante au `malloc` :
- Utilisation de `<>` au lieu de `!=` (confusion avec CAML) ;
- Oubli du `;` à la ligne `cursor->prev->next = c`.

Q 16) Cette fonction alloue un nouveau maillon `c` qui contient la donnée à insérer. Le maillon précédent est celui qui précédait le curseur et le suivant va être le curseur. On met également à jour le maillon précédent, s'il est non nul, pour qu'il pointe vers ce maillon `c`. De même le précédent du curseur (qui n'est jamais nul) est `c`.

Q 17) La complexité est constante en $\Theta(1)$.

Q 18) On supprime le maillon précédent, s'il n'est pas nul, en n'oubliant pas de mettre à jour le champ précédent du curseur et le suivant du précédent du précédent (s'il est non nul). On n'oublie pas non plus de libérer la mémoire du maillon supprimé. La complexité est en $\Theta(1)$.

C

```c
cell* backspace(cell* cursor) {
  cell* c = cursor->prev;
  if (c == NULL) {
    return cursor;
  }
  cursor->prev = c->prev;
  if (c->prev != NULL) {
    c->prev->next = cursor;
  }
  free(c);
  return cursor;
}
```

Q 19) Il y a plusieurs possibilités. En voici une.

C

```c
void free_editor(cell* cursor) {
  // Avancer jusqu'à la fin
  while (cursor->next != NULL) {
    cursor = cursor->next;
  }
  // Backspace tant que possible
  while (cursor->prev != NULL) {
    cursor = backspace(cursor);
  }
  assert(cursor->prev == NULL && cursor->next == NULL);
  free(cursor);
}
```

On avance jusqu'à la fin et on supprime tant que possible. Il nous reste un éditeur vide formé de la seule sentinelle que l'on libère enfin. La complexité est linéaire en la taille de l'éditeur.

Q 20) On initialise un éditeur vide puis on y insère successivement tous les éléments de la chaîne de caractères. Le curseur se retrouve natuellement à la fin. La complexité est linéaire en la taille de la chaîne de caractères.

C

```c
cell* from_string(char* str) {
  cell* cursor = empty_editor();
  int i = 0;
  while (str[i] != '\0') {
    cursor = insert(str[i], cursor);
    i++;
  }
  return cursor;
}
```

Q 21) Il suffit ensuite de revenir au début en itérant tant que nécessaire la fonction `back`. Cela rajoute une passe en $O(n)$, si n est la taille de l'éditeur et ne change pas la complexité asymptotique.

Q 22) C'est un peu délicat. Il faut commencer par calculer le nombre d'éléments dans l'éditeur pour pouvoir allouer la mémoire de la chaîne de caractères. La complexité est encore linéaire en la taille de l'éditeur, même s'il y a plusieurs passes.

C

```c
char* to_string(cell* cursor) {
  // On va à la toute fin
  while (cursor->next != NULL) {
    cursor = cursor->next;
  }
  // On compte le nombre de maillons en revenant au début.
  // On compte aussi le maillon fictif pour terminer notre
  // chaîne de caractère par la sentinelle
  int n = 1;
  while (cursor->prev != NULL) {
    cursor = cursor->prev;
    n++;
  }
  // (suite page suivante)
```

C

```c
  // (suite de la page précédente)
  // On créé notre chaîne et on remplit avec le contenu de
  // tous les maillons, y compris le maillon fictif.
  char* str = malloc(n);
  for (int i = 0; i < n; i++) {
    str[i] = cursor->data;
    cursor = cursor->next;
  }
  assert (cursor == NULL);
  return str;
}
```

III Le zipper pour les arbres (CAML)

Q 23) On obtient, avec le même `p` que dans l'exemple :

OCAML

```
{path = Right (E, 'd', p); next = N (E, 'e', E)}
```

Q 24) On a directement

OCAML

```
let of_tree t = {path = Top; next = t}
```

Q 25) De même, et avec les mêmes noms de variables :

OCAML

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

Q 26) On utilise les informations du chemin pour reconstruire l'arbre.

OCAML

```
let move_up z =
  match z.path with
  | Top -> z
  | Left (p, x, r) ->
     {path = p; next = N (z.next, x, r)}
  | Right (l, x, p) ->
     {path = p; next = N (l, x, z.next)}
```

Q 27) On remonte le zipper tout en haut pour placer le curseur sur la racine.

OCAML

```
let rec to_tree z =
  if z.path = Top then
    z.next
  else
    to_tree (move_up z)
```

IV Comparaison d'arbres binaires de recherche (CAML)

Q 28) Question de cours.

Q 29) C'est presque une question de cours à ceci près que l'on utilise la fonction `cmp` passée en argument plutôt que l'opérateur polymorphe `<` pour comparer les éléments et que l'on renvoie une option plutôt qu'un booléen. Il y a au plus autant d'appels récursifs que la hauteur de l'arbre.

OCAML

```
let rec get cmp x t =
  match t with
  | E -> None
  | N (l, y, r) ->
     let c = cmp x y in
     if c = 0 then Some y
     else if c < 0 then get cmp x l
     else get cmp x r
```

Q 30) Cette fonction est de type `('a -> 'a -> int) -> 'a list -> 'b tree` (rien ne force le type des éléments de l'arbre vide à être le même que celui des

éléments de la liste). Visiblement, cette fonction cherche à insérer tous les éléments d'une liste dans un arbre binaire de recherche, mais a été écrite avec les pieds : elle renvoie toujours l'arbre vide ! Insistons bien sur le fait que la ligne `insert cmp tete t;` n'a pas d'effet de bord et, comme son résultat est ignoré, ne sert absolument à rien (il y a d'ailleurs un message d'avertissement car son type n'est pas `unit`). Voici une version correcte :

OCaml
```
let rec to_set cmp liste =
  match liste with
  | [] -> E
  | tete :: queue -> insert cmp tete (to_set cmp queue)
```

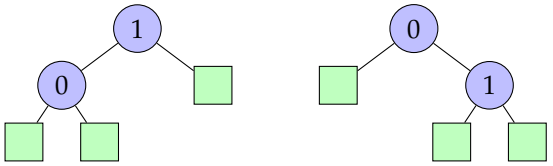
Q 31) C'est difficile et délicat mais c'est du cours ! On commence par chercher la position du nœud à supprimer. Si ce nœud n'a pas deux fils non vides alors on peut directement le supprimer. Sinon, on va chercher le minimum de son sous-arbre droit que l'on supprime récursivement (on peut car ce minimum ne peut pas avoir deux fils non vides) pour remplacer l'étiquette du nœud. Encore une fois on veille à minimiser les appels à `cmp`.

OCaml
```
let rec minimum t =
  match t with
  | E -> failwith "arbre vide"
  | N (E, y, _) -> y
  | N (l, _, _) -> minimum l

let rec delete cmp x t =
  match t with
  | E -> E
  | N (l, y, r) ->
     let c = cmp x y in
     if c < 0 then N (delete cmp x l, y, r)
     else if c > 0 then N (l, y, delete cmp x r)
     else
       if l = E then r
       else if r = E then l
       else
         let z = minimum r in
         N (l, z, delete cmp z r)
```

Q 32) Question de cours : la liste est triée.

Q 33) Il suffit de donner un contre-exemple simple :



Q 34) Les deux fonctions sont de type `'a zipper -> 'a zipper option`. La première, que l'on pourrait appeler `leftmost` va chercher le nœud le plus à gauche du curseur pour y placer le curseur (on renvoie une option sur ce zipper). Si un tel nœud n'existe pas, ce qui n'est possible que si le curseur était sur un arbre vide, elle renvoie `None`. La deuxième, que l'on pourrait appeler `left_ancestor` recherche le première ancêtre du curseur pour lequel on provient d'un fils gauche et y place le curseur (on renvoie une option sur ce zipper). Si un tel nœud n'existe pas (ce qui n'est possible que si on se trouve sur la branche la plus à droite) elle renvoie `None`.

Q 35) On peut en déduire une fonction qui fait avancer le curseur sur le nœud qui serait son successeur dans un parcours infixe de l'arbre. Si le nœud possède un fils droit, on va chercher le nœud le plus à gauche de celui-ci. Sinon, on remonte jusqu'au premier ancêtre pour lequel on est un fils gauche.

OCaml
```
let rec next_infix z =
  match z.next with
  | E -> None
  | N (_, _, E) -> first_left_ancestor z
  | N (_, _, r) -> leftmost (move_right z)
```

Q 36) On suit les indications.

OCaml
```
let compare_tree cmp t1 t2 =
  let zo1 = leftmost (of_tree t1) in
  let zo2 = leftmost (of_tree t2) in
  let rec parallel_infix zo1 zo2 =
    (* suite page suivante *)
```

OCaml
```
(* suite de la page précédente *)
    match zo1, zo2 with
    | None, None -> 0
    | None, _ -> -1
    | _, None -> 1
    | Some z1, Some z2 ->
       let x1 = label z1.next in
       let x2 = label z2.next in
       let c = cmp x1 x2 in
       if c = 0 then
         parallel_infix (next_infix z1) (next_infix z2)
       else
         c
  in
  parallel_infix zo1 zo2
```

V Parcours d'arbres iteratifs (C)

Q 37) Voir corrigé du TP n° 02, question 17.

Q 38) Voir corrigé du TP n° 02, question 24.

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

Q 39)
```
type main = carte tree
```

Q 40)
```
type stats = (main * int) tree
```

Q 41) La fonction de comparaison n'utilise que la première composante du couple. Les deux appels aboutissent sur le même nœud d'étiquette `(m, 42)` et renvoient donc `Some (m, 42)`. En effet, on a bien pris soin dans la fonction `get` ci-dessus de renvoyer `Some y` et non `Some x`. Même si `compare_key x y` vaut 0, `x` et `y` ne sont pas les mêmes. C'est tout l'intérêt d'utiliser une fonction de comparaison pour identifier certains objets.

Q 42) On peut appeler la fonction `get` avec la fonction de comparaison `compare_key` pour rechercher un élément suivant la première composantes : la clé. La deuxième composante n'étant jamais utilisée, on peut mettre n'importe quelle valeur du bon type, par exemple 0.

OCaml
```
let nb_parties m s =
  match get compare_key (m, 0) s with
  | None -> 0
  | Some (_, i) -> i
```

Q 43) On incrémente le nombre de parties si la main était déjà présente (en réalité on remplace, c'est bien le comportement de la fonction `insert`) ou on l'ajoute avec une valeur 1 sinon.

OCaml
```
let ajoute_partie m s =
  match get compare_key (m, 0) s with
  | None -> insert compare_key (m, 1) s
  | Some (_, i) -> insert compare_key (m, i + 1) s
```

Q 44) On utilise de même presque directement la fonction `delete`.

OCaml
```
let supprime_main m s =
  delete compare_key (m, 0) s
```

— FIN DU CORRIGÉ —