Algorithmique
Correction Contrôle n° 1

INFO-SUP S1 – EPITA

Solution 1 **(Types Abstraits : Listes récursives –** ***5 points*****)**

1. L'opération `rechercher` n'est définie que si l'élément recherché existe, d'où précondition. Ensuite les trois axiomes appliquant l'observateur `est-présent` aux opérations internes `liste-vide` et `cons`. Dans l'ordre : l'élément e n'existe pas dans une liste-vide, l'élément e existe dans une liste dont il est égal au premier élément et dans le cas contraire... Essaie encore (il existe peut-être dans la liste restante). Enfin l'axiome expliquant que la place retournée par `rechercher(e,`λ`)` est celle qui contient e.

   PRÉCONDITIONS

   rechercher(e, λ) **est-défini-ssi** *est-présent*(e,λ)= vrai

   AXIOMES

   est-présent $(e, listevide)$ = faux
   $e = e' \Rightarrow$ *est-présent* $(e, cons(e', \lambda))$ = vrai
   $e \neq e' \Rightarrow$ *est-présent* $(e, cons(e', \lambda))$ = *est-présent* (e, λ)
   $contenu(rechercher(e, \lambda)) = e$

2. Deux axiomes suffiront, le premier explique que le résultat de la concaténation d'une liste vide avec une liste λ est la liste λ, ce qui signifie que l'on conserve les éléments en ordre et nombre de la deuxième liste. Le second axiome explique que l'on conserve aussi les éléments en ordre et nombre de la première liste. Comment ? En montrant que si l'on fait la concaténation avant ou après la construction (cons), le résultat est le même, ce qui signifie que la concaténation ne modifie ni l'ordre, ni les éléments.

   AXIOMES

   concaténer $(listevide, \lambda 2) = \lambda 2$
   concaténer $(cons(e, \lambda), \lambda 2) = cons(e,$ *concaténer* $(\lambda, \lambda 2))$

Solution 2 **(Suppression) –** ***4 points***

Spécifications :

Écrire la fonction `delete` x *list* qui supprime de la liste *list* triée en ordre croissant la première occurrence de la valeur x (si elle est présente).

```
# let rec delete x = function
       [] -> []
     | h::q when h > x -> h :: q
     | h::q when h = x -> q
     | h::q -> h::delete x q ;;
 val delete : 'a -> 'a list -> 'a list = <fun>

# delete 4 [1; 2; 2; 3; 4; 4; 4; 5];;
 - : int list = [1; 2; 2; 3; 4; 4; 5]
```

Solution 3 **(Insertion à la $i^{ème}$ place – *5 points*)**

Spécifications : Écrire la fonction `insert_nth` *x i list* qui insère la valeur x à la $i^{ème}$ place dans la liste *list*.
La fonction devra déclencher une exception `Invalid_argument` si i est négatif ou nul, ou une exception `Failure` si la liste est trop courte.

```
# let insert_nth  x i list =
     if i < 1 then
       invalid_arg "negative rank"
     else
       let rec insert = function
           (1, list)    -> x :: list
         | (_, [])   -> failwith "out of bound"
         | (i, e::q) -> e :: insert(i-1, q)
       in
         insert (i, list);;

 val insert_nth : 'a -> int -> 'a list -> 'a list = <fun>
```

Solution 4 **(Recherche) –** ***4 points***

Spécifications :
Écrire la fonction `search_both` *list a b* qui vérifie si les deux valeurs a et b distinctes sont présentes dans la liste *list*.

```
# let search v1 v2 l =
    let rec aux1 v l = match l with
        [] -> false
       |e::l -> v = e || aux1 v l
    in
    let rec aux2 l = match l with
        [] -> false
       |e::l -> if e = v1 then
                    aux1 v2 l
                else
                if e = v2 then
                    aux1 v1 l
                else
                    aux2 l
    in aux2 l
;;
 val search : 'a -> 'a -> 'a list -> bool = <fun>
```

Une autre version :

```
# let search v1 v2 l =
    let rec aux l found1 found2 = match l with
        [] -> false
```

```
      |e::l -> if e = v1 then
                  found2 || aux l true false
              else
              if e = v2 then
                  found1 || aux l false true
              else
                  aux l found1 found2
    in aux l false false
;;
 val search : 'a -> 'a -> 'a list -> bool = <fun>
```

Solution 5 **(Mystery – *2 points*)**

1. **Spécifications :**
   Donner les résultats des évaluations successives des phrases suivantes.

```
# let go = function
        [] -> []
      | e::list ->
        let rec what x = function
            [] -> []
          | e::list -> (e * x)::(what e list)
        in
        what e list;;
 val go : int list -> int list = <fun>

# go [1; 1; 1; 1; 1] ;;
 - : int list = [1; 1; 1; 1]

# go [42] ;;
 - : int list = []

# go [1; 2; 3; 4; 5] ;;
 - : int list = [2; 6; 12; 20]

# go [2; 21; 2; 21; 2; 21] ;;
 - : int list = [42; 42; 42; 42; 42]
```