Listes CAML

Table des matières

1 Listes en CAML

Les listes CAML sont des *listes chaînées immuables*.

- immuable : on ne peut pas les modifier ;
- liste chaînée ?

1.1 Une liste est une structure de données

Définition 1.1

En informatique, une *structure de données* est une structure logique destinée à organiser et à agir sur des données.

Définition 1.2

Une *liste* est une collection séquentielle et de taille arbitraire d'éléments de même type.

Exemple 1

- *Liste des étudiants inscrits en CPGE*
- *Liste de courses*
- *Liste de choses à faire.*

Exemple 2

Les listes en PYTHON.

1.2 La structure de liste chaînée

Définition 1.3

Une *liste chaînée* est une liste dont la représentation en mémoire est faite de cellules comportant une donnée et un pointeur vers la cellule suivante.

Exemple 3 (*Dessin*)

Vocabulaire : tête ; queue

Exemple 4

Liste des élèves avec des étiquettes et des bouts de ficelles.

✔

On n'a un accès direct qu'au premier élément. L'accès aux éléments suivants se fait de manière séquentielle : chaque élément permet l'accès au suivant et ainsi de suite.

EXERCICE 1

Expliquer comment réaliser les fonctions suivantes :

1. Tester si la liste est vide
2. Rechercher un élève dans la liste

3. Ajouter un élève dans la liste
4. Modifier le nom d'un élève dans la liste (attention : pas en CAML)

☞ Remarque 1

Rien *à voir avec les « listes »* PYTHON*, malgré la consonance.*

1.3 Implémentation en CAML

En CAML :

- la liste vide s'obtient avec `[]` [1] et s'appelle *nil*.
- la « flèche » s'obtient avec `::`, opérateur infixe qui se lit *conse* pour constructeur de listes.

Exemple 5

OCAML
```
[];;
- : 'a list = []
0 :: [];;
- : int list = [0]
'a' :: 'b' :: 'c' :: [];;
- : char list = ['a'; 'b'; 'c']
[0; 1; 2];;
- : int list = [0; 1; 2]
```

- Le type d'une liste s'écrit `'a list`. Tous les éléments sont de même type.
- La liste vide est « polymorphe » (au sens où `'a` est une variable de type libre).
- Il y a des espaces autour de `::`.
- `[0; 1; 2]` est une *abréviation* pour `0 :: 1 :: 2 :: []`
- Attention, ce sont des points-virgules !

EXERCICE 2

Écrire en CAML une fonction `est_vide : 'a list -> bool` qui teste si une liste est vide.

1. Oui, comme en PYTHON, mais il vaut mieux ne pas faire le parallèle.

OCAML
```
let est_vide liste =
  liste = []
;;
```

EXERCICE 3

Écrire en CAML une fonction `ajoute : 'a -> 'a list -> 'a list` qui ajoute un élément à une liste (en tête bien sûr).

OCAML
```
let ajoute element liste =
  element :: liste
;;
```

1.4 Représentation arborescente

Exemple 6 (*Dessin*)

`[0; 1; 2]`

☞ Remarque 2

Ces arbres sont appelés peignes.

1.5 Filtrage sur les listes

On a vu comment construire des listes. Mais comment accéder aux éléments ?

OCAML
```
match liste with
| [] ->
| tete :: queue ->
```

Exemple 7

Autre manière de définir `est_vide` *:*

OCaml
```
let est_vide liste =
  match liste with
  | [] -> true
  | _ -> false
;;
```

EXERCICE 4

Écrire les fonctions `hd : 'a list -> 'a` et `tl : 'a list -> 'a list` qui renvoient respectivement la tête et la queue d'une liste. Quelle est leur complexité ?

OCaml
```
let hd liste =
  match liste with
  | [] -> failwith "Liste vide"
  | tete :: queue -> tete
;;
```

OCaml
```
let tl liste =
  match liste with
  | [] -> failwith "Liste vide"
  | tete :: queue -> queue
;;
```

☞ **Remarque 3**

Ces fonctions sont définies en CAML*, mais à utiliser le moins possible. Il s'agit de* `List.hd` *et* `List.tl`*.*

2 Quelques fonctions simples sur les listes

2.1 Longueur d'une liste

OCaml
```
let rec list_length liste =
  match liste with
  | [] -> 0
  | tete :: queue -> 1 + (list_length queue)
;;
list_length : 'a list -> int = <fun>
```

Il faut dès maintenant se poser les questions :

- La fonction termine-t-elle ?
- La récursion est-elle correcte ?
- Quelle est la complexité ?

Il y a exactement $n+1$ appels si la longueur de la liste est n et le reste est en $O(1)$, donc la complexité est $O(n)$ (et même $\Theta(n)$) et donc linéaire. Autrement, on obtient la relation de récurrence $C(0) = \Theta(1)$ et pour $n \geq 1$, $C(n) = C(n-1) + \Theta(1)$, ce qui permet de retrouver une complexité $\Theta(n)$.

☞ **Remarque 4**

Cette fonction est prédéfinie en CAML *:* `List.length`*.*

2.2 Appartenance à une liste

OCaml
```
let rec mem element liste =
  match liste with
  | [] -> false
  | tete :: queue ->
     if tete = element then
       true
     else
       mem element queue
;;
mem : 'a -> 'a list -> bool = <fun>
```

ou avec un filtrage directement

OCaml
```
let rec mem element liste =
  match liste with
  | [] -> false
  | element :: queue -> true (* NON !!! *)
;;
```

il faut une garde

OCaml
```
let rec mem element liste =
  match liste with
  | [] -> false
  | tete :: queue when tete = element -> true
  | _ :: queue -> mem element queue
;;
```

ou encore, en utilisant (et en l'indiquant sur la copie) le fait que `||` est paresseux :

OCaml
```
let rec mem element liste =
  match liste with
  | [] -> false
  | tete :: queue -> (tete = element) || (mem element queue)
;;
```

Là encore la complexité est linéaire en la taille de la liste.

☞ **Remarque 5**

Cette fonction est prédéfinie en CAML *:* `List.mem`.

2.3 Concaténation

OCaml
```
let rec concat liste1 liste2 =
  match liste1 with
  | [] -> liste2
  | tete1 :: queue1 -> tete1 :: (concat queue1 liste2)
;;
concat : 'a list -> 'a list -> 'a list = <fun>
```

La complexité et en $\Theta(n)$ où n est la longueur de la première liste (indépendante de la taille de la deuxième !).

En CAML, on dispose de l'opérateur infixe `@` pour la concaténation.

OCaml
```
[1; 2; 3] @ [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

⚠

Sa complexité *n'est pas* $O(1)$, mais linéaire en la taille de la première liste !

- Proscrire son utilisation
- Ou alors s'engager à en tenir compte dans les calculs de complexité

2.4 Miroir

Problème : Comment renverser l'ordre des éléments dans une liste ?

Première idée : On renverse la queue puis on insère la tête à la fin de la liste.

OCaml
```
let rec ajoute_en_fin element liste =
  match liste with
  | [] -> [element]
  | tete :: queue -> tete :: (ajoute_en_fin element queue)
;;
ajoute_en_fin : 'a -> 'a list -> 'a list = <fun>
ajoute_en_fin 5 [2; 3; 1; 6];;
- : int list = [2; 3; 1; 6; 5]

let rec rev liste =
  match liste with
  | [] -> []
  | tete :: queue -> ajoute_en_fin tete (rev queue)
;;
rev : 'a list -> 'a list = <fun>
rev [2; 3; 1; 6];;
- : int list = [6; 1; 3; 2]
```

Quelle est la complexité ?

La fonction `ajoute_en_fin` vérifie encore une équation de récurrence $C_{ajoute}(n) = \Theta(1) + C_{ajoute}(n-1)$ et est donc de complexité linéaire en la taille de la liste. Pour la fonction `rev` on trouve la relation $C_{rev}(n) = C_{rev}(n-1) + C_{ajoute}(n-1) = C_{rev}(n-1) + \Theta(n)$. Ainsi $C_{rev}(k) - C_{rev}(k-1) = \Theta(k)$, donc $\sum_{k=1}^{n} C_{rev}(k) - C_{rev}(k-1) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2)$ et donc, par télescopage on trouve $C_{rev} = \Theta(n^2)$ et la complexité est quadratique.

Peut-on faire mieux ?

On va utiliser la notion *d'accumulateur*.

OCaml
```
(* Concatène le miroir de `liste` et `accumulateur` *)
let rec rev_append liste accumulateur =
  match liste with
  | [] -> accumulateur
  | tete :: queue -> rev_append queue (tete :: accumulateur)
;;
rev_append : 'a list -> 'a list -> 'a list = <fun>

let rev liste =
  rev_append liste []
;;
rev : 'a list -> 'a list = <fun>
```

Pour la complexité de `rev_append` : si la liste est vide on trouve $\Theta(1)$. Pour une liste de taille $n \geq 1$ il y a un appel à `rev_append` avec une liste de taille $n-1$ et un accumulateur dont la taille a augmenté. Mais la taille de l'accumulateur n'entre nul part en compte dans ce calcul. L'équation de récurrence est à nouveau $C_{rev\_append}(n) = C_{rev\_append}(n-1) + \Theta(1)$ et la complexité est linéaire.

L'appel à `rev` consiste en un simple appel à `rev_append` et sa complexité est donc la même (avec un $\Theta(1)$ négligeable en plus).

☞ **Remarque 6**

On peut définir la fonction auxiliaire comme fonction locale :

OCaml
```
let rev liste =
  let rec rev_append liste accumulateur =
    match liste with
    | [] -> accumulateur
    | tete :: queue -> rev_append queue (tete ::
      ↪ accumulateur)
  in
  rev_append liste []
;;
```

Dans tous les cas, donner la signature (le type) de la fonction auxiliaire, sa spécification et une explication claire.

3 Une fonctionnelle simple sur les listes

EXERCICE 5

Écrire la fonction qui :

1. Ajoute 1 à tous les éléments d'une liste
2. Renvoie la liste des longueurs des chaînes de caractères d'une liste de chaînes

OCAML
```
let rec ajoute_1 liste =
  match liste with
  | [] -> []
  | tete :: queue -> (tete + 1) :: (ajoute_1 queue)
;;
ajoute_1 : int list -> int list = <fun>
ajoute_1 [3; 2; 1];;
- : int list = [4; 3; 2]
```

OCAML
```
let rec longueurs_chaines liste =
  match liste with
  | [] -> []
  | tete :: queue -> (String.length tete) :: (longueurs_chaines
   ↪  queue)
;;
longueurs_chaines : string list -> int list = <fun>
longueurs_chaines ["Aulagnier"; "Larochette"];;
int list = [9; 10]
```

Quel est le lien entre ces deux fonctions ? Elle suivent un schéma général de la forme :

OCAML
```
let rec nom_fonction liste =
  match liste with
  | [] -> []
  | tete :: queue -> (f tete) :: (nom_fonction queue)
;;
```

La fonctionnelle `map` (**m**ultiple **ap**plication) applique une fonction donnée en entrée (dans l'exemple précédent : `f`) à tous les éléments d'une liste.

OCAML
```
let rec map f liste =
  match liste with
  | [] -> []
  | tete :: queue -> f tete :: map f queue
;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Exemple 8

Autre solution à l'exercice précédent :

1.
2.