# Structures de données abstraites

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

# Table des matières

# Définition

« Une structure de données abstraite est un type muni d'opérations » — Le programme

- Type : donne un nom à la structure
  - *abstrait* ≈ *opaque* : on ne précise **aucun détail concret d'implémentation**
- Opérations : permettent de manipuler les objets du type
  - Un ensemble d'algorithmes dont (au moins) l'une des entrées est du type de la structure
  - Leurs spécifications sont le cœur de la définition de la structure

## Exemple : *pile fonctionnelle*

(Développé plus tard)

- Type : `'a stack`
  - Une valeur `empty_stack : 'a stack`
- Opérations :
  - `push : 'a -> 'a stack -> 'a stack`
  - `pop : 'a stack -> 'a * 'a stack`

Ce sont les spécifications de *push* et *pop* qui préciseront ce que fait la structure

# Exemples ; vocabulaire

## Opérations typiques

- *Constructeur* : créer une nouvelle instance d'une structure (vide ou initialisée)
  - Éventuellement aussi un *destructeur*, qui détruit (libère) une instance
- *Accesseurs* : accéder à des valeurs stockées dans une instance de la structure
  - Suivant des contraintes / possibilités qui dépendent de la structure (la définisse)
- *Transformateurs* : modifier le contenu d'une instance de la structure
  - *Ditto* ; n'existe pas toujours (par exemple pour les structures fonctionnelles/immuables)

## Exemple : *vecteur* (ou *tableau*)

- Constructeur *create* : paramètres : un entier $n$, une valeur $e$ d'un certain type ; crée et renvoie un nouveau vecteur de $n$ éléments d'indices $\in [\![0, n-1]\!]$ et de valeur $e$
- Accesseur *get* : paramètres : vecteur $v$, un indice $i$ ; renvoie la valeur de l'élément de $v$ à l'indice $i$
- Transformateur *set* : paramètres : un vecteur $v$, un indice $i$, une valeur $x$ ; écrit la valeur $x$ à l'indice $i$ de $v$

# Exercice

- Proposez une implémentation de la structure abstraite vecteur d'`int` en C (sans traiter les erreurs)
- Pourquoi ne sait on pas (pour l'instant) faire en OCaml ?

# Correction

## En C

```
int *create(size_t n, int e) = {
    int *v = malloc(n * sizeof(int));
    for (size_t i = 0; i < n; i++) { v[i] = e; }
    return v; }
int  get(int *v, size_t i) { return v[i]; }
void set(int *v, size_t i, int x) { v[i] = x; }
```

## Pourquoi pas en OCaml ?

On ne connaît pas (pour l'instant) de façon pour modifier un objet : `set` impossible à faire

## Remarque

Il ne faut pas confondre le type d'une structure de données abstraite et les types concrets d'implémentation, y compris quand elles ont le même nom ><

- Les `list` Python ne sont pas *que* des structures de données abstraite de liste…

# Structures de données abstraites : pourquoi ?

## La manipulation de données…

- Est une composante présente dans (presque) tous les algorithmes
- Se fait suivant des besoins, des critères plus ou moins restrictifs & exigeants
  - Pouvant se retrouver à l'identique dans de nombreux algorithmes (très différents)

## Abstraction $\rightsquigarrow$ modularité

Découpler spécifications et détails concrets d'implémentation

- Permet de distinguer les *besoins* (spécifications) des réalisations
- Toute réalisation satisfaisant aux besoins est acceptable

Une implémentation concrète d'un algorithme a le choix de l'implémentation concrète de ses structures de données ; une implémentation concrète d'une structure de données peut être utilisée dans plusieurs algorithmes

# Abstraction & modularité : pas que pour les structures de données

## Même chose que la distinction spécifications / implémentation d'un algorithme

Exemple du tri :

- Souvent une opération utile, utilisée en « sous-routine » dans un algorithme plus conséquent
- Beaucoup d'implémentations concrètes (c.-à.-d. d'algorithmes de tri)
  - Par exemple ?
- Interchangeables (sans changer la correction, mais possiblement les performances)

## Correspondance spécifications = théorème, implémentation = preuve

Un parallèle avec les maths :

- Spécifications (d'un algorithme, d'une structure) de données : un énoncé de théorème
- Réalisation des spécifications : une preuve

- On peut avoir plusieurs preuves d'un même théorème (parfois *beaucoup*)
- On n'a pas besoin de connaître une preuve pour appliquer un théorème
- Il est utile d'isoler des résultats en tant que théorèmes (ou lemmes, ou…) : on ne veut pas avoir à les reprouver à chaque fois

# Rappel : démarche de conception d'un algorithme

Étant données des spécifications (un théorème à prouver) :

- Quelles sous-routines (lemmes) peuvent être utiles ?
  - *Seulement besoin des spécifications*
- Quelles *structures de données abstraites* (lemmes, aussi) peuvent être utiles ?
  - *ditto*

~ On réfléchit ~

- On décrit l'algorithme, en utilisant sous-routines & structures de données en boîte noire
- Si l'on souhaite implémenter l'algorithme (le définir complètement), il faut faire de même pour les sous-routines & structures de données
  - Qui *doivent* être implémentables, sinon notre algorithme ne l'est pas… (même problème qu'un « théorème » prouvé avec un lemme faux…)
  - Qu'on peut implémenter (« reprouver ») nous-même, ou non en utilisant une bibliothèque (« en admettant la preuve, faite ailleurs »)
  - Dans tous les cas : pas d'impact sur la *correction*, mais impact possible sur le *coût*

# Table des matières

# Deux approches pour les structures de données

## Structures fonctionnelles (ou ≈ immuables)

- **On ne modifie jamais** les objets manipulés : on en crée de nouveaux
- L'approche usuelle pour la programmation en *style fonctionnel* (ex. : OCaml sans effets)
- Par exemple implémentées avec des types récursifs (listes, arbres (cf. semestre prochain)…)

## Structures impératives (ou ≈ mutables)

- Certaines opérations **modifient** les objets manipulés
- L'approche usuelle pour la programmation en *style impératif* (ex. : C ; OCaml avec effets)
- Par exemple implémentées avec des types mutables (tous les types C non qualifiés `const`) ; des références (pointeurs) sur des types immuables

L'approche la plus courante *en général* est l'impérative, mais l'approche fonctionnelle est fréquente dans les langages fonctionnels…

# Comparaison éclaire

## Structures fonctionnelles

Permettent le *partage* entre plusieurs « versions » d'une structure (possible sans soucis car immuable)

- Gain d'espace (et de temps) parfois non triviaux
- Par exemple pratique pour du *backtracking*

Mais nécessite des *indirections* (suivre des pointeurs), plus ou moins sous le capot

## Structures impératives

- Souvent plus efficaces qu'une structure fonctionnelle (pas, ou moins d'indirections),
  - quand pas de partage nécessaire
- Peut entraîner des problèmes d'*aliasing* (plusieurs personnes veulent modifier les mêmes données sans le savoir)
  - qu'on peut souvent résoudre en faisant des copies, mais dans ce cas…

# Table des matières

# Intermède : gestion d'erreur dans les spécifications

Dans tous les exemples qui suivent, on ne précise pas le comportement des fonctions en cas d'utilisation « erronée »

Car :

- Les mécanismes disponibles pour réagir en cas d'erreur dépendent du langage (exceptions, types `option`, UB, pointeurs nuls…)
- Il peut y avoir plusieurs approches possibles (*cf.* semaine dernière, TP)
- De façon générale, il y a moins facilement consensus sur le sujet

# Liste fonctionnelle

## Spécifications

- Une valeur (immuable) pour la liste vide
  - Éventuellement une fonction *is_empty* qui teste si son argument est la liste vide
- Un constructeur *cons* : paramètres : une valeur *x*, une liste *xs* ; construit une *nouvelle* liste où *x* a été ajouté en tête de *xs*
- Un accesseur *head* : paramètre : une liste *x* ; renvoie l'élément en tête de *x* s'il existe
- Un accesseur *tail* : paramètre : une liste *x* ; renvoie la liste *x* privée de sa tête si *x* est non vide
- Pas de transformateurs

## Exemple d'implémentation

```
type 'a fl = Nil | Cons of 'a * 'a fl
let is_empty x = (x = Nil)
let cons x xs = Cons (x, xs)
let head = function Cons(x, xs) -> x  | _ -> failwith "Empty list"
let tail = function Cons(x, xs) -> xs | _ -> failwith "Empty list"
```

# Pile

*En anglais :* **stack**

## Principe

Dans une pile, on peut :

- ajouter (empiler) un élément en sommet de pile
- si non vide, extraire (dépiler) un élément

Avec la contrainte « LIFO » (*last in, first out*) qu'on dépile le dernier élément empilé encore présent sur la pile (celui actuellement en son sommet)

## Utilité

Permet de stocker des éléments à traiter, dans un ordre précis. Utile par exemple pour des parcours de graphe, des algorithmes à base de pile (…). *Cf.* futurs TDs, TPs, DSs…

# Pile fonctionnelle

## Spécifications

- Une valeur (immuable) pour la pile vide
  - Éventuellement une fonction *is_empty* qui teste si son argument est la pile vide
- Un constructeur *push* : paramètres : une valeur $x$, une pile $s$ ; construit une *nouvelle* pile où $x$ a été ajouté en sommet de $s$
- Une fonction *pop* : paramètre : une pile $s$ ; si $s$ est non vide, renvoie le couple constitué du sommet de $s$ et d'une *nouvelle* pile identique au reste de $s$
- Éventuellement un accesseur *peek* : paramètre : une pile $s$ ; renvoie le sommet de $s$ s'il existe

# Pile fonctionnelle : implémentation « abstraite »

Les spécifications correspondent exactement à celles de la liste fonctionnelle, où *pop* combine *head* et *tail*

- On peut implémenter une pile fonctionnelle à partir d'une liste fonctionnelle :

```
type 'a fs = 'a fl
let push = cons
let pop s = head s, tail s
let peek = head
```

- Fonctionne pour *n'importe quelle* implémentation d'une liste fonctionnelle (la notre, ou…)
- Pour « notre » implémentation de liste, toutes les opérations sont en temps constant

En pratique (en OCaml), on utilisera directement les `list`

# Pile impérative

## Spécifications

- Un constructeur de pile vide
  - Maintenant (quasiment) nécessaire sous la forme d'une fonction !
  - Pourquoi ?
- Une fonction *is_empty* qui teste si son argument est une pile vide
- Un transformateur *push* : paramètres : une valeur $x$, une pile $s$ ; *modifie* $s$ pour ajouter $x$ en son sommet
- Un accesseur & transformateur *pop* : paramètre : une pile $s$ ; si $s$ est non vide,
  - renvoie l'élément en sommet de $s$
  - **et modifie s en supprimant celui-ci**
- Éventuellement un accesseur *peek* : paramètre : une pile $s$ ; renvoie le sommet de $s$ s'il existe

# Pile impérative : implémentation 1

On utilise le module `Stack`…

- https://ocaml.org/manual/5.4/api/Stack.html

## À savoir utiliser

- `create`
- `is_empty`
- `push`
- `pop`
- L'exception `Empty`

Plus de détails plus tard…

# Pile impérative : implémentation 2

- Avec un tableau (dynamique)
  - *Cf.* TP

# Pile impérative : implémentations 3 & 4

- Avec une *référence* sur une liste fonctionnelle
  - *Cf.* TD
- Avec une liste impérative

# Liste impérative : spécifications (base)

## Spécifications (base)

- Un constructeur de liste vide
- Une fonction *is_empty* qui teste si son argument est une liste vide
- Un transformateur *insert_head* : paramètres : une valeur v, une liste x ; modifie x pour y ajouter un élément contenant v en tête
  - (*push* d'une pile impérative)
- Un transformateur *remove_head* : paramètres : une liste x ; modifie x pour en supprimer le premier élément (et éventuellement le renvoyer) s'il existe
  - (*pop* d'une pile impérative si renvoyé, sinon en conjonction avec *head*)
- Un accesseur *get_head* : paramètres : une liste x ; renvoie le premier élément de x si non vide
  - (*peek* d'une pile impérative)
- Un accesseur *get_tail* : paramètres : une liste x ; renvoie la queue xs de x (la liste formée de tous ses éléments sauf sa tête)

# Liste impérative : spécifications (Extension)

## Spécifications (Extension)

- Un transformateur *insert_after* : paramètres : une valeur $v$, une élément $e$ d'une liste $x$ ; modifie $x$ pour y ajouter un élément contenant $v$ entre $e$ et son (éventuel) successeur
- Un transformateur *remove* : paramètres : une liste $x$, un élément $e$ de $x$ ; modifie $x$ pour en supprimer l'élément $e$
- Un accesseur *next* : paramètres : un élément $e$ d'une liste $x$ ; renvoie l'élément suivant $e$ s'il existe

# Liste impérative : implémentation 1

```
#include <sys/queue.h>
```

- Pas au programme

# Liste impérative : méta-implémentation 2

Comme une liste chaînée, qui « chaîne » (…) des « maillons » avec des pointeurs (ou des références…), et maintient une référence séparée sur la tête

## Schéma global

- Nécessite typiquement une certaine récursivité dans les types
  - En OCaml : *usual stuff*
  - En C : par le fait qu'un type de genre `struct` peut contenir un champ de type pointeur vers lui-même

# Liste impérative : implémentation 2 en C (exemple)

En utilisant le type concret :

```c
struct slist_e
{
    int dat; // ou void *dat pour être générique
    struct slist_e *nxt;
};
```

pour un maillon, et :

```c
struct slist_e*
```

pour la liste « toute entière »

# Liste impérative : implémentation 2 en C (exemple) *bis*

Exemples de fonctions :

```c
struct slist_e *make_empty(void) {
    return NULL;
}

void insert_head(int v, struct slist_e **x) {
    assert(x != NULL);

    struct slist_e *new_e = ckd_malloc(sizeof(struct slist_e));
    new_e->dat = v;
    new_e->nxt = *x;
    *x = new_e;

    return;
}
```

# Liste impérative : implémentation 2 en OCaml (exemple)

En utilisant le type concret :

```
type 'a slist_e = { mutable dat : 'a;
                    mutable nxt : 'a slist_e option }
```

pour un maillon et :

```
type 'a slist = 'a slist_e option ref
```

pour la liste « toute entière »

Utilisent des aspects impératifs du langage (pas encore vus ensemble)

# Liste impérative : implémentation 2 en OCaml (exemple) *bis*

Exemples de fonctions :

```
let make_empty () = (ref None : 'a slist)

let insert_head (v : 'a) (x : 'a slist) =
    x := Some {dat = v ; nxt = !x}
```

# Liste impérative : implémentation 2 : comparaison des exemples/langages

```c
void insert_head(int v, struct slist_e **x) {
    assert(x != NULL);
    struct slist_e *new_e = ckd_malloc(sizeof(struct slist_e));
    new_e->dat = v;
    new_e->nxt = *x;
    *x = new_e;
}
```

```
let insert_head (v : 'a) (x : 'a slist) =
    x := Some {dat = v ; nxt = !x}
```