# Programmation Avancée (INF441)

# Paramétrer

François Pottier

3 mai 2016

# (Rappel) Abstraction/∃

Au sens existentiel, abstraire, c'est cacher, encapsuler.

« Moi, module `BinomialHeap`, j'affirme qu'il existe un type des files de priorité, muni d'opérations `empty`, `insert`, `extract`, `merge`, etc. »

# (Rappel) Abstraction/∀

Au sens universel, abstraire, c'est paramétrer.

« Moi, module `ShortestPaths`, j'affirme que pour toute implémentation des files de priorités, et pour toute représentation d'un graphe pondéré, je suis capable de calculer les plus courts chemins, etc. »

# Pourquoi paramétrer ?

En paramétrant un composant,

- on évite de fixer trop tôt certains choix ou certains détails ;
  - on peut implémenter `ShortestPaths` avant d'avoir une file de priorité
- on rend ainsi ce composant plus général, plus ré-utilisable ;
  - y compris plusieurs fois dans un même programme
- et on simplifie son code !

# Par quoi paramétrer ?

On peut paramétrer par :

- un type ;
- une valeur, y compris une fonction ;
- une structure qui regroupe des types et des valeurs (séance 5).

# Un type trop spécialisé

Voici le type des listes dont les éléments sont des entiers :

```
type list =
| Nil
| Cons of int * list
```

Soit, dans la notation de la théorie des types :

$$L = 1 + \mathit{Int} \times L$$

# Du code trop spécialisé

On peut définir une bibliothèque de fonctions utiles :

```
let rec length accu xs =
  match xs with
  | Nil -> accu
  | Cons (_, xs) -> length (accu + 1) xs

let length xs =
  length 0 xs
```

On peut définir aussi `rev`, `append`, `map`, `sort`, et bien d'autres.

La bibliothèque Data.List de Haskell en contient une centaine environ !

Mais si on veut des listes de paires d'entiers, il faut tout recommencer... ?

# Une spécialisation excessive

Ce n'est pas tolérable. En principe, on ne doit jamais dupliquer du code !

- Ce n'est pas maintenable.
- Ce ne serait pas très malin qu'un programme compilé contienne 107 exemplaires de la librairie `List`.

# Un problème lié au typage statique ?

Dans un langage typé dynamiquement, nous n'aurions pas ce problème.

En JavaScript, on a des tableaux de ce qu'on veut :

```
> cars = ['Saab', 'Volvo', 'BMW']
[ 'Saab', 'Volvo', 'BMW' ]
> integers = [1, 2, 3]
[ 1, 2, 3 ]
```

Et même des tableaux de tout et n'importe quoi :

```
> cars = ['Saab', 42]
[ 'Saab', 42 ]
```

Cette discipline est très flexible, mais pas très sûre.

Que faire dans un langage typé statiquement, comme Java ou OCaml ?

# Paramétrer par un type

Heureusement,

- la définition des listes a un sens quel que soit le type des éléments,
- les fonctions `length`, etc. fonctionnent quel que soit ce type.

On peut les considérer comme paramétrées par le type des éléments.

Strachey (1967) appelait cela le polymorphisme « paramétrique ».[1]

1. Il parlait de polymorphisme « ad hoc » lorsque deux fonctions distinctes ont le même nom : par exemple, l'addition des entiers et l'addition des flottants. On appelle cela aussi la surcharge.

# Paramétrer par un type

> The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the

Milner (1978) définit et implémente ML,

- premier langage doté de polymorphisme paramétrique,
- et aussi d'inférence de types !

Ancêtre de Standard ML, Haskell, OCaml.

Le polymorphisme est apparu dans Java 1.5 (2004) et C# 2.0 (2005).

# Un type paramétré

On paramètre le type des listes par le type `'a` des éléments :

```
type 'a list =
| Nil
| Cons of 'a * 'a list
```

Soit, dans la notation de la théorie des types :

$$L(E) = 1 + E \times L(E)$$

# Remarque

C'est ainsi que le type `'a list` est défini dans la bibliothèque d'OCaml.

Mais OCaml a quelques notations spéciales pour les listes :

- `Nil` est noté `[]`
- `Cons(x, xs)` est noté `x :: xs`
- `[x; y; z]` signifie `x :: y :: z :: []`

# Des listes de ce qu'on veut...

Une variable de types `'a` dénote un type inconnu.

On peut l'instancier (la remplacer) plus tard par un type arbitraire.

On peut construire des listes d'entiers, des listes de paires d'entiers, etc.

Ici, dans une session interactive `ocaml` :

```
# 4 :: 3 :: [];;
- : int list = [4; 3]
# (4, 3) :: (2, 1) :: [];;
- : (int * int) list = [(4, 3); (2, 1)]
```

# ... pas des listes de tout et n'importe quoi

Toutefois, pas de listes hétérogènes :

```
# 4 :: true :: [];;
Error: This expression has type bool
       but an expression was expected of type int
```

En effet, nous avons posé

$$L(E) = 1 + E \times L(E)$$

donc, par dépliage à l'infini,

$$L(E) = 1 + E \times (1 + E \times (1 + E \times (\ldots)))$$

Tous les éléments ont un même type $E$.

# Du code polymorphe

Ce code fonctionne quel que soit le type des éléments de la liste `xs` :

```
let rec length accu xs =
  match xs with
  | Nil -> accu
  | Cons (_, xs) -> length (accu + 1) xs

let length xs =
  length 0 xs
```

Le type de `length` est donc $\forall E.\ L(E) \to \mathit{Int}$.

On écrit dans le fichier list.mli :

```
val length: 'a list -> int
```

La quantification universelle est implicite, mais elle est bien là.

# Inférence de types

OCaml infère les types. On peut omettre :

- le type de l'argument et le type du résultat d'une fonction
- le type d'une variable locale introduite par `let` ou `match`

Cela nous permet d'écrire du code polymorphe sans même y réfléchir :

```
let compose f g x = g (f x)
```

# Détails syntaxiques

Dans un fichier `.mli`, la quantification universelle est implicite :

```
val compose: ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
```

Dans un fichier `.ml`, si on le souhaite, on peut quantifier explicitement :

```
let compose: 'a 'b 'c . ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
= fun f g x -> g (f x)
```

Ce style est imposé quand on utilise la récursivité polymorphe.

# Définition d'une classe paramétrée

Depuis Java 1.5, une classe peut être paramétrée par une ou plusieurs variables de types, ici `A` et `B`.

```
public class Pair<A, B> {
  public final A a;
  public final B b;
  public Pair (A a, B b) { this.a = a; this.b = b; }
}
```

Le compilateur examine cette définition sous l'hypothèse que `A` et `B` sont des types inconnus.

# Définition d'une interface paramétrée

La même idée s'applique aux interfaces, comme `Comparator<T>` :

```
public interface Comparator<T> {
  int compare (T o1, T o2);
}
```

Cf. aussi `Function<T, R>`, rencontré la semaine dernière.

# Instanciation d'une classe paramétrée

On instancie une classe paramétrée en remplaçant les paramètres par des types quelconques : (*)

```
Pair<Integer, Boolean> p =
  new Pair<Integer, Boolean> (42, true);
Pair<Integer, Pair<Integer, Boolean>> q =
  new Pair<Integer, Pair<Integer, Boolean>> (21, p);
```

C'est lourd. Heureusement, on peut demander un peu d'inférence :

```
Pair<Integer, Boolean> p =
  new Pair<> (42, true);
Pair<Integer, Pair<Integer, Boolean>> q =
  new Pair<> (21, p);
```

C'est documenté par Oracle ici.

# Instanciation d'une interface paramétrée

De même, on instancie une interface paramétrée en remplaçant les paramètres par des types quelconques. (*)

Ici, on définit des fonctions de comparaison :

```
Comparator<Integer> increasing =
  (x, y) -> (x < y) ? -1 : (x == y) ? 0 : 1;
Comparator<Integer> decreasing =
  (x, y) -> (x > y) ? -1 : (x == y) ? 0 : 1;
```

# (*) Un détail

Pour des raisons techniques et historiques, une variable de types `X` ne peut pas être instanciée par un type primitif.

On peut écrire `Pair<Integer, Boolean>`, mais pas `Pair<int, boolean>`.

C'est l'une des raisons de l'existence de la classe `Integer` :

```
public class Integer { private final int value; ... }
```

Pour éviter trop de douleur, le compilateur convertit automatiquement, dans les deux sens, entre `int` et `Integer`.

Le transparent précédent en tirait déjà parti.

# Exemples

La bibliothèque de Java contient des classes et interfaces paramétrées :

| | | |
|---|---|---|
| `PriorityQueue<E>` | implémente | `Queue<E>` |
| `HashMap<K, V>` | implémente | `Map<K, V>` |
| `TreeMap<K, V>` | implémente | `Map<K, V>` |
| `LinkedList<E>` | implémente | `Collection<E>` |
| | qui hérite de | `Iterable<E>` |

# Exemple d'utilisation

Nous en avons déjà utilisé certaines, par exemple lors de la séance 1 :

```
Queue<HuffmanTree> q = new PriorityQueue<> ();
```

Ici, on instancie la variable de types `E` en la remplaçant par `HuffmanTree`.

La méthode `add` de l'interface `Queue<E>` attend un argument de type `E`, donc, ici, `q.add` attend un argument de type `HuffmanTree` :

```
HuffmanTree t = ...;
q.add(t);
```

# Méthodes polymorphes

Parfois, une méthode seule est paramétrée, sans que la classe le soit :

```
public class PrintingUtilities {
  static <E> void printCollection (Collection<E> v) {
    int i = 0;
    for (E e : v)
      System.out.println("Element #" + i++ + ": " + e);
  }
}
```

Lorsqu'on appelle cette méthode, l'instanciation de `E` est inférée :

```
Collection<Integer> c = java.util.Arrays.asList(1, 2, 3, 4);
PrintingUtilities.printCollection(c);
```

Ici, `E` est automatiquement instancié par `Integer`.

Ceci est analogue à une fonction polymorphe d'OCaml.

# « Polymorphisme » en Java pre-1.5

En Java avant 1.5, on utilisait `Object` comme un type fourre-tout.

- Une variable de type `X` quelconque a aussi le type `Object`.
- Inversement, on peut convertir de `Object` vers `X` grâce à un cast,
- qui peut échouer (`ClassCastException`), donc est dangereux.

La situation est analogue en C, où le type fourre-tout est `void*`, et où un cast incorrect est encore plus dangereux (comportement anormal, crash).

# « Polymorphisme » en Java pre-1.5

Le type `LinkedList` est défini comme « liste d'`Object` ».

Lorsqu'on insère un élément dans une liste, on oublie donc son type.

Lorsqu'on extrait un élément de la liste, un test dynamique est nécessaire :

```
LinkedList xs = new LinkedList ();
xs.add(new Integer (42));
Boolean b = (Boolean) xs.poll();
```

Ce code échoue ! On a inséré un entier et on espère extraire un Booléen.

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.Boolean
```

On a en fait une combinaison de typage statique et dynamique.

Style encore permis en Java aujourd'hui, mais déconseillé.

# Rechercher un élément fixé

Cette fonction teste si la liste `ys` contient l'entier 42 :

```
let rec contains42 (ys : int list) =
  match ys with
  | [] -> false
  | y :: ys -> y = 42 || contains42 ys
```

Son type est `int list -> bool`. Elle est très (trop) spécialisée.

# Rechercher un élément donné

Pour un peu plus de généralité, on paramètre par l'élément `x` recherché :

```
let rec contains x (ys : int list) =
  match ys with
  | [] -> false
  | y :: ys -> y = x || contains x ys
```

Le type de `contains` est `int -> int list -> bool`.

C'est un peu mieux : on peut tester si une liste contient 63, par exemple.

# Rechercher un élément donné

Pour un peu plus de généralité, on paramètre par l'élément `x` recherché :

```
let rec contains x (ys : int list) =
  match ys with
  | [] -> false
  | y :: ys -> y = x || contains x ys
```

Le type de `contains` est `int -> int list -> bool`.

C'est un peu mieux : on peut tester si une liste contient 63, par exemple.

Mais si on recherche un élément supérieur ou égal à 42 ?

Ou bien un élément impair et supérieur à 42 ?

Etc.

# Rechercher un élément satisfaisant

Pour plus de généralité encore, on paramètre la fonction de recherche par une propriété que doit satisfaire `y` :

```
let rec exists (satisfactory : int -> bool) (ys : int list) =
  match ys with
  | [] -> false
  | y :: ys -> satisfactory y || exists satisfactory ys
```

On écrit ainsi une fonction paramétrée par une fonction :

```
val exists : (int -> bool) -> int list -> bool
```

C'est possible parce qu'une fonction est une valeur comme une autre.

# Rechercher un élément satisfaisant

Exemples d'utilisation, ici dans une session interactive :

```
# exists (fun y -> y = 42) [ 1; 42; 2 ];;
- : bool = true
# exists (fun y -> y mod 2 = 1 && y > 42) [ 1; 42; 2 ];;
- : bool = false
```

La recherche d'un élément `x` donné devient un cas particulier :

```
let contains x ys =
  exists (fun y -> y = x) ys
```

Une clôture a accès aux variables locales définies à l'extérieur ; ici, `x`.

# Recherche polymorphe

Bonus : le code n'est plus fondamentalement restreint au type `int`.

```
let rec exists (satisfactory : 'a -> bool) (ys : 'a list) =
  match ys with
  | [] -> false
  | y :: ys -> satisfactory y || exists satisfactory ys
```

On peut déclarer dans le fichier `.mli` :

```
val exists : ('a -> bool) -> 'a list -> bool
```

`exists` est paramétrée par un type `'a` et par une fonction `satisfactory`.

# La généralité favorise la généralité

En cherchant à paramétrer le code par des valeurs,
on le rend souvent polymorphe – paramétré par des types.

Par exemple, si on cherche à généraliser ceci :

```
val increment_each: int list -> int list
  (* add 1 to each element *)
val decrement_each: int list -> int list
  (* subtract 1 from each element *)
```

on arrivera probablement à ceci :

```
val map: ('a -> 'b) -> 'a list -> 'b list
```

# Digression : l'égalité polymorphe

Notre fonction `contains`, elle aussi, est polymorphe :

```
let contains x ys =
  exists (fun y -> y = x) ys
```

En effet, OCaml infère le type suivant :

```
val contains: 'a -> 'a list -> bool
```

N'est-ce pas surprenant ?

# Digression : l'égalité polymorphe

C'est le cas parce que la fonction primitive = est polymorphe :

```
val (=) : 'a -> 'a -> bool
```

Elle compare en profondeur deux valeurs de n'importe quel type :

```
# (2, 2) = (2, 3);;
- : bool = false
# [1;2;3] = [1;2;4];;
- : bool = false
# (fun x -> x) = (fun x -> x + 1);;
Exception: Invalid_argument "equal: functional value".
```

Elle n'est pas (et ne peut pas être) implémentée en OCaml.

# Digression : l'égalité polymorphe

Cette égalité polymorphe est parfois pratique, mais :

- peut être coûteuse,
- peut échouer, et surtout
- viole l'abstraction.

Si `q1` et `q2` sont des files de priorité fournies par le module `LeftistHeap`, que signifie la comparaison `q1 = q2` ?

- elle peut renvoyer `false` alors que `q1` et `q2` ont les mêmes éléments !
- elle est permise ! alors qu'on pensait que `empty`, `insert`, `extract`, `merge` étaient les seules opérations exposées à l'utilisateur.

Mieux vaut ne l'appliquer qu'à des types primitifs, comme `int`.

# Listes

Considérons des listes simplement chaînées immuables.

Elles sont représentées par trois classes paramétrées (poly §3.1.2) :

```
public abstract class List<E> {}
public class Nil<E> extends List<E> {}
public class Cons<E> extends List<E> {
  private final E head;
  private final List<E> tail;
  public Cons (E head, List<E> tail) { ... } // (Trivial.)
}
```

# Éliminer les doublons

On souhaite supprimer les doublons d'une liste supposée triée.

C'est-à-dire construire une nouvelle liste, sans doublons.

Pour que le code soit ré-utilisable, il faut :

- abstraire vis-à-vis d'une relation d'ordre (ou d'égalité) ;
- abstraire vis-à-vis du type des éléments ;
- le premier étant nécessaire au second.

# Éliminer les doublons

Le code sera paramétré par le type `E` et par un ordre `c` sur ce type :

```
public abstract class List<E> {
  // Assuming the list is sorted, remove duplicate elements.
  public abstract List<E> noDup (Comparator<E> c);
  // Assuming the list is sorted, remove duplicate elements
  // and all leading occurrences of the element e.
  protected abstract List<E> noDupAux
    (Comparator<E> c, E e);
}
```

La seconde méthode est auxiliaire.

# Implémentation

L'implémentation des deux méthodes dans la classe `Nil` est triviale :

```
public List<E> noDup (Comparator<E> c) {
  return this;
}
protected List<E> noDupAux (Comparator<E> c, E e) {
  return this;
}
```

# Implémentation

Dans la classe `Cons`, la méthode `noDup` préserve le premier élément :

```
public List<E> noDup (Comparator<E> c) {
  return new Cons<E> (
    head,
    tail.noDupAux(c, head)
  );
}
```

et appelle `noDupAux` pour traiter la suite.

# Implémentation

C'est ici que sert l'objet `c` :

```
protected List<E> noDupAux (Comparator<E> c, E e) {
  if (c.compare(e, head) == 0)
    return tail.noDupAux(c, e);
  else
    return noDup(c);
}
```

Notons que nos deux méthodes sont mutuellement récursives.

Exercices (faciles) :

- Terminent-elles ? Quelle est leur complexité en temps et en espace ?
- Quel est l'équivalent de ce code en OCaml ?

# Utilisation

Ce code est relativement facile d'emploi :

```
Comparator<Integer> increasing =
  (x, y) -> (x < y) ? -1 : (x == y) ? 0 : 1;
List<Integer> xs =
  new Cons<> (1,
  new Cons<> (1,
  new Nil<> ()
  ));
List<Integer> ys = xs.noDup(increasing);
```

# Autre exemple

La bibliothèque `Arrays` propose une fonction de tri :

```
static <T> void sort (T[] a, Comparator<T> c);
```

Elle est polymorphe : elle fonctionne pour tout type `T`.

Elle attend deux arguments :

- un tableau `a` d'éléments de type `T` ;
- une fonction `c` capable de comparer deux éléments.

Comment appelle-t-on cette fonction ? C'est tout simple...

# Utilisation

Supposons qu'on aie sous la main un tableau d'entiers :

```
Integer[] a = { 2, 1, 3 };
```

On peut le trier par ordre croissant :

```
Arrays.sort(a, (x, y) -> (x < y) ? -1 : (x == y) ? 0 : 1);
```

ou par ordre décroissant :

```
Arrays.sort(a, (x, y) -> (x > y) ? -1 : (x == y) ? 0 : 1);
```

Ou bien, si on a déjà nommé ces fonctions de comparaison :

```
Arrays.sort(a, increasing);
Arrays.sort(a, decreasing);
```

On écrirait essentiellement la même chose en OCaml.

# Classes abstraites et héritage

Déclarer une méthode abstraite, qui sera définie dans une sous-classe, est un autre moyen d'écrire du code ouvert, où tout n'est pas fixé :

```
public abstract class Sort<T> {
  // To be defined by a sub-class:
  protected abstract int compare (T o1, T o2);
  // To be called by the client:
  public void sort (T[] a) { ... }
}
```

La relation d'ordre n'est plus un paramètre, mais un « membre abstrait ».

On a en général le choix entre ces deux approches.

Ci-dessus, le type `T` est resté un paramètre. Toutefois, en Scala...

# Classes abstraites et héritage

En Scala, le type `T` aussi peut devenir un membre abstrait :

```
trait Sort {
  // To be defined by a sub-class:
  type T
  def compare (o1 : T, o2 : T) : Int
  // To be called by the client:
  def sort (a : Array[T]) { ... }
}
```

Scala offre le choix entre paramètres et membres abstraits, pour les valeurs et pour les types.

# Récursivité polymorphe

Une fonction récursive s'appelle elle-même avec une valeur différente.

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

Évident. Mais que pensez-vous de ceci :

Une fonction polymorphe récursive peut s'appeler elle-même avec un type différent !

Après tout, elle fonctionne pour n'importe quel type. Donc, pourquoi pas ?

Un exemple suit...

# Listes ordinaires

Une liste simplement chaînée (immuable) permet :

- insertion et retrait en tête en temps $O(1)$ ;
- accès au $i$-ième élément en temps $O(i)$.

# « Random access lists »

Pourrait-on modifier la structure de données pour avoir plutôt :

- insertion et retrait en tête en temps $O(\log n)$ ;
- accès au $i$-ième élément en temps $O(\log i)$.

C'est ce que permettent les binary random access lists (Okasaki, 1995).

# Inspiration

La structure est inspirée de la numération en base 2.

L'écriture binaire d'un entier $n$ est :

- '0', si $n$ est nul ;
- '0', suivi de l'écriture binaire de $\frac{n}{2}$, si $n$ est pair ;
- '1', suivi de l'écriture binaire de $\frac{n-1}{2}$, si $n$ est impair.

(On écrit ici le bit de poids faible à gauche.)

Le nombre de bits dans l'écriture de $n$ est $O(\log n)$.

# Inspiration

On conserve ce schéma pour représenter une séquence de $n$ éléments.

Une séquence de $n$ éléments sera :

- `Nil`, si $n$ est nul ;
- `Zero`, suivi d'une séquence de $\frac{n}{2}$ paires d'éléments, si $n$ est pair ;
- `One`, suivi d'un élément et d'une séquence de $\frac{n-1}{2}$ paires d'éléments, si $n$ est impair.

# Un type algébrique

Cela nous conduit au type algébrique paramétré suivant :

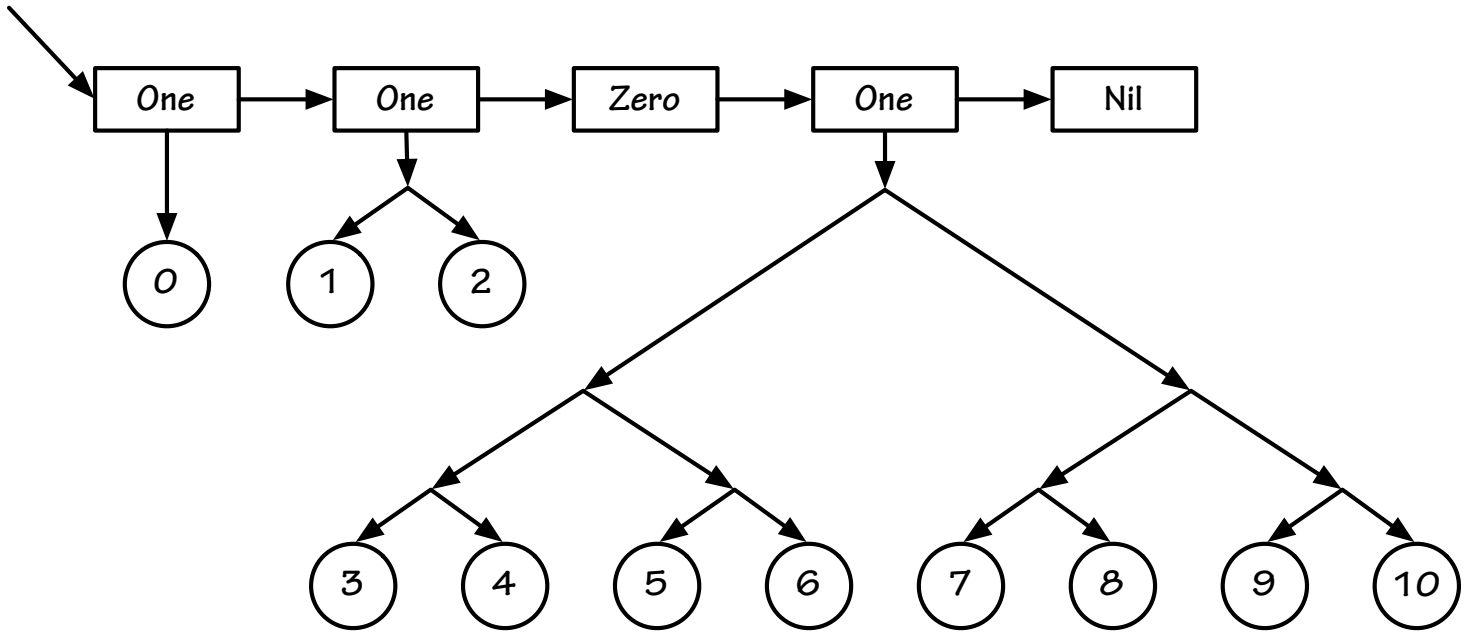
```
type 'a seq =
| Nil
| Zero of     ('a * 'a) seq
| One of 'a * ('a * 'a) seq
```

Ce type paramétré fait référence à lui-même avec un autre paramètre !

Une séquence d'éléments contient une séquence de paires d'éléments.

Cela va nous conduire à écrire des fonctions récursives polymorphes qui s'appellent elles-mêmes avec un type différent...

# Vision graphique




Un chiffre 1 situé au rang $k$ porte une paire de paires de ... paires d'éléments, qui forme un arbre binaire complet à $2^k$ feuilles.

# Interlude

Implémentation des « random access lists » en OCaml.
(Code en ligne)
(Pas dans le poly.)

# Polymorphisme = macro-expansion ?

Le polymorphisme nous évite de devoir dupliquer du code.

Est-ce là tout ?

- Peut-on se passer de polymorphisme
  si l'on est prêt à dupliquer du code ?
- Est-ce que le polymorphisme implique que le compilateur
  duplique le code à notre place ?

# Polymorphisme = macro-expansion ?

La réponse à ces questions dépend du langage et de son compilateur.

- En C++, le compilateur duplique le code pour nous, avant l'exécution ;
- En C#, le code est dupliqué pendant l'exécution, si besoin ;
- En Java et OCaml, pas de duplication : c'est bien le même code qui fonctionne pour n'importe quel type.

C++ ne permet pas la récursivité polymorphe : il faudrait un nombre infini de copies du code...

# À retenir

En paramétrant un fragment de code vis-à-vis d'un ou plusieurs types et valeurs, on le rend plus général, donc (en principe) plus ré-utilisable.

En TD aujourd'hui :

- un type abstrait « hash set » paramétré par le type des éléments et par des fonctions d'équivalence et de hachage.
- une application au partage maximal ou « hash-consing ».