Programmation Avancée (INF441)

Présentation du cours
Code, données, coût

François Pottier

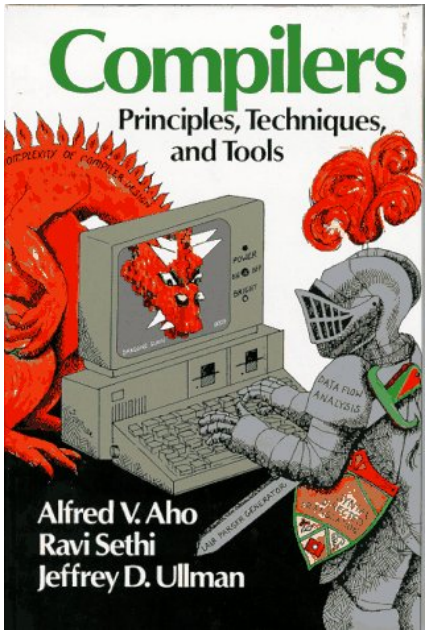
5 avril 2016

Qui suis-je ?

Je suis chercheur à l'INRIA,
spécialiste de la théorie
des langages de programmation.

Je suis également (à temps partiel)
professeur chargé de cours
à l'École Polytechnique.

J'enseigne INF441 et INF564 (Compilation).



Pourquoi suivre ce cours ?

A priori,

- Vous savez programmer ; cf. INF311, INF321, INF411 ;
- Vous disposez d'une boîte à outils algorithmique ; cf. INF421.
- Vous avez peut-être abordé concurrence et distribution ; cf. INF431.

Que vous reste-t-il à apprendre ?

Pourquoi suivre ce cours ?

A priori,

- Vous savez programmer ; cf. INF311, INF321, INF411 ;
- Vous disposez d'une boîte à outils algorithmique ; cf. INF421.
- Vous avez peut-être abordé concurrence et distribution ; cf. INF431.

Que vous reste-t-il à apprendre ?

> *The lyf so short, the craft so long to lerne.*
>
> — *Chaucer (1340–1400)*

Vers la programmation à grande échelle

Le noyau de Linux 3.6 contient 40K fichiers et 15M lignes de code.

Il a plus de 10K auteurs, dont au moins 1K actifs.

Aucun n'en connaît l'intégralité.

Comment est-ce possible ?

Programmation modulaire

Il faut une architecture modulaire, c'est-à-dire

- une organisation en composants
- aussi indépendants que possible les uns des autres
- et que l'on assemble pour former des composants plus complexes.

Programmation modulaire

Idéalement, ces composants doivent être faciles à

- ré-utiliser dans de nouveaux scénarios,
- remplacer par d'autres composants équivalents,
- faire évoluer sans affecter le reste du système,
- tester ou prouver indépendamment les uns des autres.

Abstraction

Pour tout cela, l'abstraction joue un rôle fondamental.

Ce mot a deux significations, duales l'une de l'autre...

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. »

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. »

Programmation « avancée »

Que pouvons-nous faire en huit semaines ?

- Étudier l'abstraction sous ses multiples formes (séances 2–5) ;
  - types abstraits ; objets et clôtures ;
  - polymorphisme ; foncteurs ; interfaces paramétrées ; etc.
- Appliquer ces outils au problème de l'itération (séances 6–8).
  - comment séparer producteur et consommateur,
  - tout en leur permettant de dialoguer ?
  - qui contrôle qui ?
  - folds ; itérateurs ; suspensions ; flots ; continuations ; etc.

Programme modeste dans ses grandes lignes, mais outils puissants.

Présupposés

Je suppose connue la programmation structurée traditionnelle :

- fonctions (récursives) et variables (globales et locales) ;
- structures de contrôle (`if`, `switch`, `while`...).

Un peu de Java...

Je suppose que vous êtes familiers de Java :

```
static int fact (int n) {
  int accu = 1;
  while (n > 0) accu *= n--;
  return accu;
}
```

Je mettrai l'accent sur certains aspects plus avancés :

```
Comparator<Integer> c =
  (n1, n2) -> {
    return n1 < n2 ? 1 : n1 == n2 ? 0 : -1;
  };
```

... Et un peu d'OCaml

Je vous demanderai d'utiliser aussi un fragment d'OCaml :

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

De façon à pouvoir écrire des choses concises mais intéressantes :

```
let map (f : 'a -> 'b) (it : 'a iterator) : 'b iterator =
  fun () ->
    match it() with
    | None ->
        None
    | Some a ->
        Some (f a)
```

OCaml est pour les masses, c'est un high-frequency trader qui le dit !

Pourquoi deux langages ?

Ceci n'est en principe ni un cours de Java, ni un cours d'OCaml.

Un bon programmeur connaît plusieurs langages
et en assimile facilement de nouveaux (Scheme, Haskell, Scala, ...).

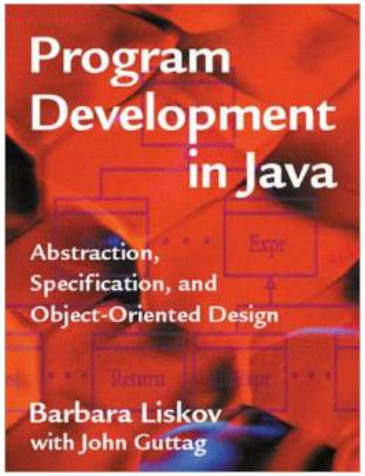
Java et OCaml semblent proposer des cadres conceptuels différents,

- « programmation orientée objets », versus
- « programmation fonctionnelle » ;

mais sont en réalité beaucoup plus proches qu'il n'y paraît :

- organisation de la mémoire (pile, tas) ; modèle de coût ; (séance 1)
- fonctions/objets mariant données et comportement ; (séances 1 et 3)
- etc.

Ouvrages recommandés



Liskov, Guttag.
Program Development in Java.

Conchon, Filliâtre.
Apprendre à programmer avec OCaml.

Et le poly INF441, bien sûr ! avec beaucoup d'exercices.

Détails pratiques

Huit séances, du 5 avril 2016 au 31 mai 2016.

Contrôle classant le mardi 7 juin 2016.

Projet optionnel. (Un projet INF4** exigé pour le PA Info.)

$M = \max(\, CC,\, (PI + 2 * CC)/3 \,)$

Détails pratiques

Quatre sujets de projet, disponibles en ligne dès maintenant.

Le projet peut être fait en binôme.

Réfléchissez-y et travaillez-y **dès maintenant !**

Faites connaître votre choix avant le 20 avril 2016.

- par email à `francois.pottier@inria.fr`
- (depuis un email `@polytechnique.edu`)

Rendu du projet et d'un court rapport le 31 mai 2016.

Soutenance orale à partir du 7 juin 2016.

Menu du jour

Aujourd'hui :

- Rappel de l'organisation du code et des données.
- Modèle de coût (temps, espace).
- Valables pour Java, OCaml, et beaucoup d'autres langages.

Comportement = code

On programme en écrivant du code dans un langage de haut niveau.

Une fois compilé, c'est une suite d'instructions en langage machine.

Il est immuable, donc accessible en lecture seule.

Le langage de programmation nous offre (au moins) deux formes d'abstraction si fondamentales qu'on y pense rarement :

- abstraction procédurale ;
- abstraction vis-à-vis de la machine.

Abstraction procédurale

Le code est toujours découpé en de nombreuses fonctions (ou méthodes).

Celles-ci peuvent être regroupées dans des bibliothèques ré-utilisables.

Ces idées remontent à Wheeler (1952) :

> `If library sub-routines exist [...] then the task of constructing the remaining part of the programme is naturally very much less [...].`

Abstraction procédurale

Une fonction est une abstraction, au sens où l'utilisateur sait ce qu'elle fait, pas comment elle le fait :

> `Even after [a library sub-routine] has been coded and tested there remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily.`

En un sens, une fonction est un composant indépendant :

> `Thus the sub-routines can be more easily coded and tested in isolation from the rest of the programme.`

Abstraction vis-à-vis de la machine

À l'époque de Wheeler, on programmait en langage machine et sans OS :

> `Are [the sub-routines] going to be stored on punched paper tape or [...] in the auxiliary store of the machine?`

Aujourd'hui, le compilateur (Java, OCaml, etc.) et le système d'exploitation se chargent de ces détails.

Cette portabilité est une autre forme d'abstraction.

Zones de données

Les données sont réparties en deux zones, toutes deux de taille variable :

- la pile ;
- le tas.

Les variables globales (appelées « champs statiques » en Java) forment une troisième zone, de taille fixe.

La pile

La pile contient les variables locales des fonctions en cours d'exécution.

Ce sont des données dont la durée de vie est l'exécution d'une fonction.

La taille de la pile varie. À une constante près, elle est égale au nombre d'appels de fonctions (imbriqués) en cours à cet instant.

```
let rec hanoi n src dst tmp =
  if n > 0 then begin
    hanoi (n - 1) src tmp dst;
    Printf.printf "Move a disk from %d to %d.\n" src dst;
    hanoi (n - 1) tmp dst src
  end
```

L'appel `hanoi n 1 2 3` exige un temps $O(2^n)$ et un espace $O(n)$ en pile.

Si cela ne vous semble pas évident, démontrez-le par récurrence sur n.

Le tas

La pile ne contient que des données atomiques et à durée de vie limitée.

Où et sous quelle forme stocker une donnée composite, p. ex. un arbre ?

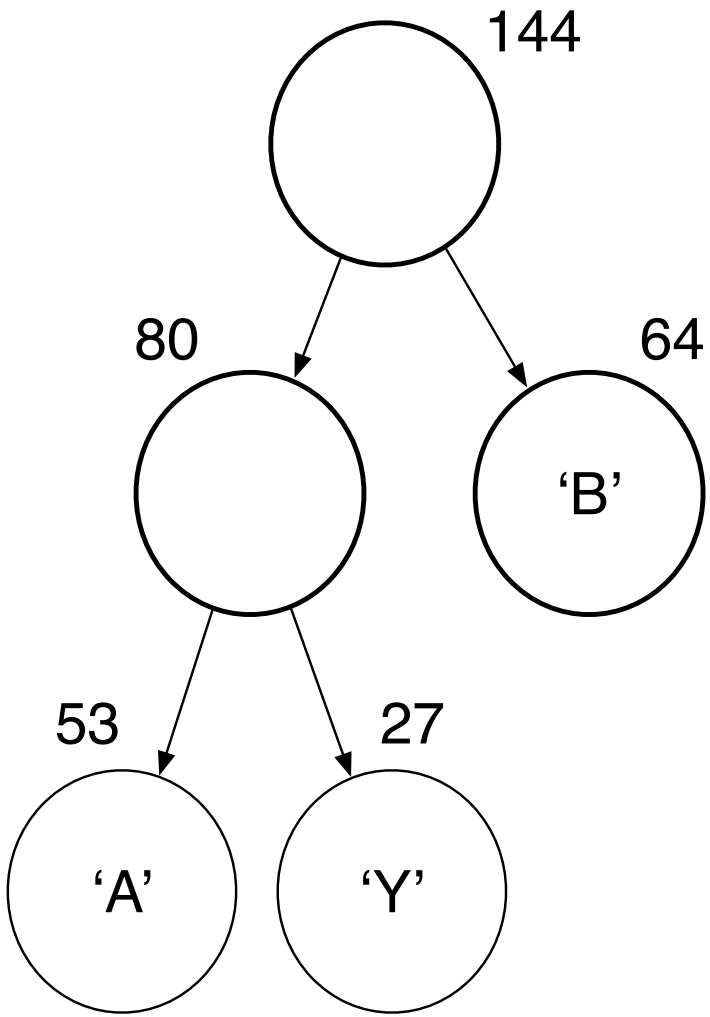
Où stocker une donnée à durée de vie illimitée ?

Le tas répond à ces questions.

Exemple : arbres de Huffman

Dans un arbre de Huffman, on a :

- des feuilles, qui portent un symbole et une fréquence,
- des nœuds internes binaires, qui portent deux sous-arbres et une fréquence.



Le tas

Le tas est une zone de données. Il est disjoint de la pile.

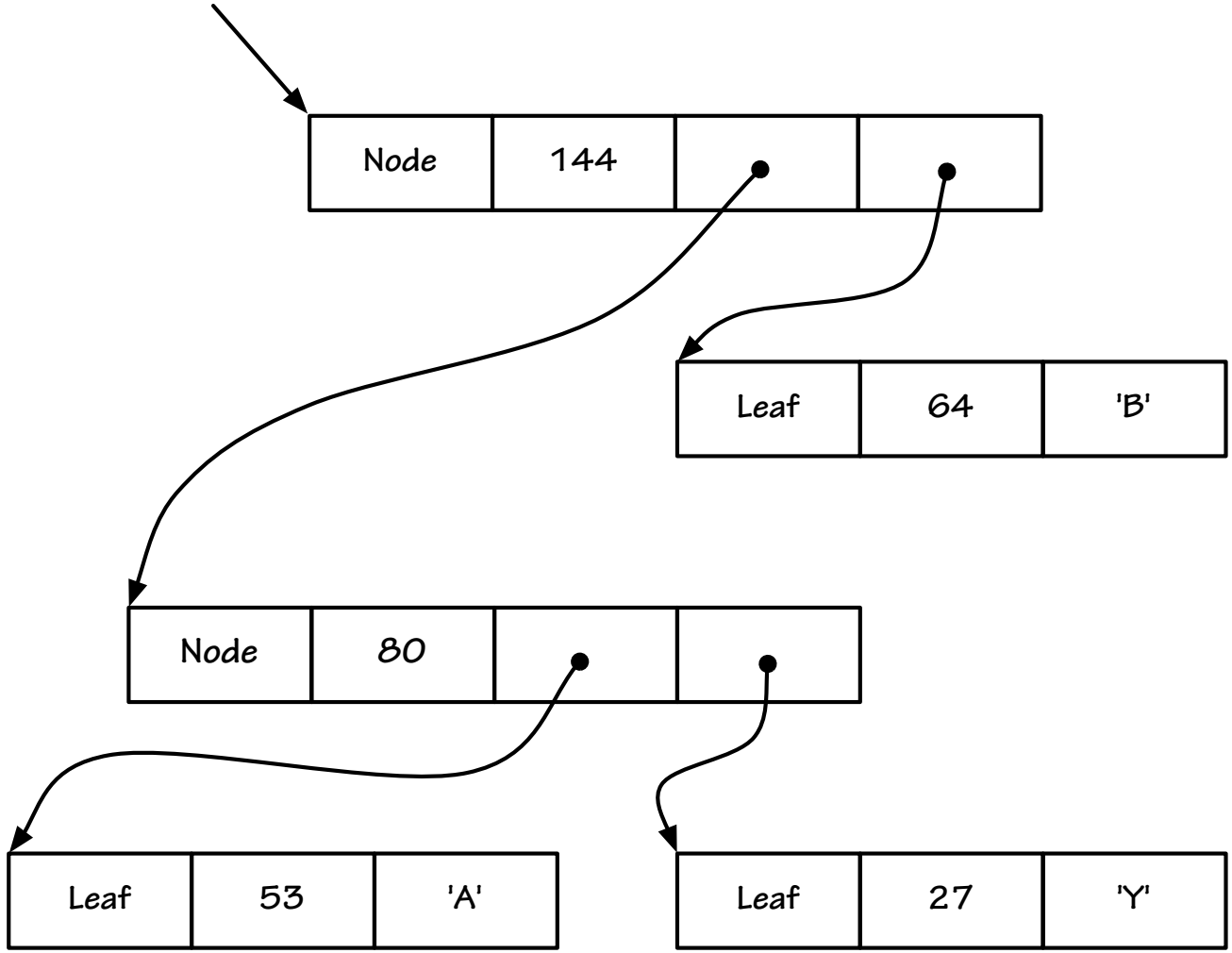
Sa taille et sa durée de vie sont a priori illimitées.

On peut à tout moment y allouer un nouveau bloc de mémoire.

Un bloc est composé d'une étiquette et de zéro, un ou plusieurs champs.

François
Pottier

Structure des blocs dans le tas



Types sommes et produits

L'étiquette d'un bloc est soit `Leaf`, soit `Node`.

C'est une disjonction – une somme.

Si c'est `Leaf`, on a un champ `freq` et un champ `c`.

Si c'est `Node`, on a un champ `freq` et un champ `left` et un champ `right`.

C'est une conjonction – un produit.

Types primitifs et types récursifs

Certains champs contiennent un entier, un caractère, etc.
    Leur type (`int`, `char`, etc.) est primitif.

D'autres champs contiennent (un pointeur vers) un arbre.
    Le type des arbres est récursif.

Types algébriques

Types primitifs, sommes, produits, récursivité.

On appelle types algébriques les types obtenus à l'aide de ces notions.

Notation concise

Le type T de nos arbres de Huffman peut être décrit ainsi :

$$T = (Int \times Char) + (Int \times T \times T)$$

À titre de comparaison, voici le type des listes d'éléments de type E :

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

Et le type des leftist heaps que nous rencontrerons la semaine prochaine :

$$H = 1 + (Int \times E \times H \times H)$$

Les types offrent un langage de description des données.

Interlude

Représentation canonique des arbres, dans le tas, en Java.
(Code en ligne)
Poly §1.1.1.

Interlude

Représentation canonique des arbres, dans le tas, en OCaml.
(Code en ligne)
Poly §1.1.2.

Accès à une somme de produits

Il faut analyser l'étiquette avant d'accéder aux champs.

En OCaml, l'analyse de cas est explicite :

- `match t with Leaf (f, c) -> ... | Node (f, t1, t2) -> ...`

En Java, elle est souvent implicite :

- une méthode abstraite est déclarée dans la classe parent
- et définie dans chaque sous-classe,
- où le type de `this` est plus précis, ce qui permet l'accès aux champs.

Données versus comportement

Jusqu'ici, nous avons :

- des données stockées dans les variables et dans le tas ;
  - par exemple, un arbre ;
- du code qui alloue, lit, modifie ces données ;
  - par exemple, une fonction qui insère un élément dans un arbre.

Code et données sont séparés.

Marier données et comportement

On appelle objet une entité dotée d'un état et d'un comportement.

- Il répond à une requête (exprimée par un appel de méthode)
- d'une manière qui peut dépendre de son état actuel
- et (si nécessaire) modifie son état.

Ce n'est donc ni purement du code, ni purement des données.

Exemple

Soit un compteur doté de deux méthodes `next` et `reset`.

Il fournit un service que l'on peut résumer en Java via une interface.

On implémente ce service en Java sous forme d'une classe.

Interface

Le service rendu :

```
public interface ICount {
  int next ();
  void reset ();
}
```

Implémentation

Voici une manière d'implémenter ce service :

```
public class Count implements ICount {
  // Fields. (Data.)
  // Invariant: step divides (next - init).
  private int next;
  private final int init, step;
  // Constructor.
  public Count (int init, int step) {
    this.next = this.init = init; this.step = step;
  }
  // Methods. (Code.)
  public int next() { int n = next; next += step; return n; }
  public void reset() { next = init; }
}
```

Dynamic dispatch

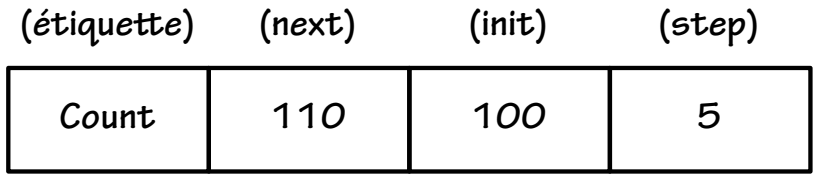
Ci-dessous, on ne sait pas à quelle classe appartient l'objet `c` :

```
void f (ICount c) {
  c.reset();
  while (c.next() < 100) { ... }
}
```

Lors de l'appel `c.reset()` ou `c.next()`, la machine consulte l'étiquette de `c` pour savoir quelle méthode doit être appelée.

L'étiquette détermine donc (indirectement) des adresses dans le code.

Mariage entre données et comportement



Un objet de type `ICount` contient donc

- des données (étiquette et champs), mais aussi
- des pointeurs de code (codés dans l'étiquette).

Et en OCaml ?

Les clôtures marient données et comportement de façon similaire.

```
(* "Constructor". *)
let count init step =
  let next = ref init in
  (* "Methods". *)
  let next() =
    let n = !next in
    next := n + step;
    n
  and reset() =
    next := init
  in
  (* Return a tuple of the methods. *)
  (next, reset)
```

Nous y reviendrons lors de la séance 3.

Modèle de coût

Le modèle de coût de Java et d'OCaml est très simple. En bref,

- Temps : chaque opération élémentaire coûte $O(1)$.
  - opération primitive (addition, comparaison, etc.)
  - appel de fonction ; accès à une variable
  - allocation d'un bloc ; accès à un champ
- Espace (pile) : chaque fonction active occupe un espace $O(1)$.
- Espace (tas) : chaque bloc occupe un espace $O(1)$.
  - `new A (...)` en Java, `A (...)` en OCaml
  - `x -> x + y` en Java, `fun x -> x + y` en OCaml

L'allocation d'un tableau coûte $O(n)$ en temps et en espace dans le tas.

Modèle de coût

Le garbage collector (GC) libère les blocs devenus inaccessibles.

- Estimer l'espace occupé par les blocs accessibles est parfois délicat.
- Estimer le temps consommé par le GC est très difficile.
  Nous ignorerons cette question.

À retenir

Quelques points importants :

- Organisation des données en sommes de produits.
  - étiquette et champs
- L'étiquette d'un objet a un double rôle :
  - elle désigne une branche d'une somme ;
  - elle détermine implicitement des pointeurs de code.

En TD aujourd'hui :

- types algébriques (= arbres) en Java et en OCaml.

Aujourd'hui, si possible **venez dès 13h** au bâtiment Turing (1er étage, salles 1101, 1106 et ... ?).