ALGORITHMES, TYPES ET STRUCTURES

Comme mentionné dans l'introduction de l'informatique commune, l'informatique est la science de la construction de l'information par le calcul. Mais, s'il est facile de comprendre qu'on peut calculer sur les nombres, comment peut-on calculer de l'information en général? Calculer sur les entiers est une chose, calculer une information en général en est une autre. La solution réside dans la création de structures de données concrètes qui rendent possible le calcul sur des concepts plus sophistiqués que les nombres.

Ces structures de données sont construites à partir de briques de bases, les types simples. Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées : le choix d'une structure de données plutôt qu'une autre, par exemple choisir un entier long plutôt qu'un flottant ou une liste au lieu d'un tableau, peut rendre inefficace un algorithme selon le choix effectué.

Le concept de type abstrait de données permet d'écrire un algorithme en faisant abstraction de l'implémentation de la structure de données (cf. figure 1). Par exemple, un dictionnaire peut-être implémenté de différentes manières, soit en utilisant un arbre, soit en utilisant une table de hachage. Un même TAD peut être réalisé par plusieurs structures de données dont les performances en complexité temporelle ou mémoire ne sont pas nécessairement équivalentes.

L'étude des types abstraits et des structures de données associées à l'algorithme est donc fondamentale en informatique.

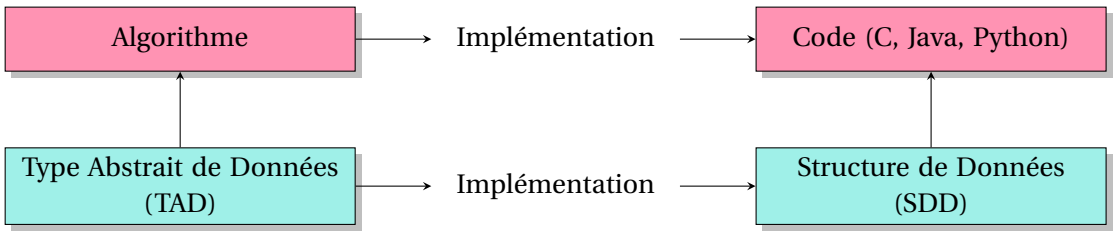

FIGURE 1 – Imbrication des concepts d'algorithme, de type abstrait de données, d'implémentation, de code et de structure de données

A Des types abstraits de données et des structures de données

a Types simples, composés et instances

Les types simples sont généralement les entiers, les flottants et les booléens. S'y ajoutent parfois les caractères qui sont représentés par des entiers.

Les types composés sont les listes, les tableaux, les arbres, les files, les piles. Ce sont des structures composites qui permettent de manipuler l'information sous la forme d'ensembles ordonnés ou non.

■ **Définition 1 — Instance.** En informatique, une instance est un objet matérialisé dans la machine qui représente un exemplaire d'un certain type dans un langage dans la mémoire.

■ **Exemple 1 — Instance et types de base.** En python, la séquence :

- `a = 3` créé une instance d'un type `int` qui vaut 3,
- `f = 3.5` créé une instance du type `float` qui vaut 3.5,
- `b = True` créé une instance du type `bool` qui vaut `True`,

En OCaml, l'équivalent s'exprime :

- `let i = 3`,
- `let f = 3.5`
- `let b = true`

b Type abstrait de données et structure de données

■ **Définition 2 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boite noire.

Un TAD spécifie le quoi, c'est-à-dire le type de données contenues ainsi que les opérations possibles. Par contre, il ne spécifie pas comment les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 3 — Structure de données.** Une structure de données est une mise en œuvre concrète d'un type abstrait, une implémentation d'un type abstrait dans un langage de programmation. On y décrit donc la manière de représenter les données et d'implémenter les opérations en machine.

R Un type abstrait de données est à une structure de données ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

■ **Exemple 2 — Un entier.** Un entier est un TAD qui :

(données) contient une suite de chiffres[a] éventuellement précédés par un signe − ou +,

(opérations) fournit les opérations $+, -, \times, //, \%$.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char`, `short`, `int`, `uint`, `long int` en C,
- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 3 — Un booléen.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

(données) se note Vrai ou Faux,

(opérations) fournit les opérations logiques conjonction, disjonction et négation...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

Les exemples précédents de types abstrait de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 4 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants :

- liste,
- file,
- pile,
- arbre binaire,
- dictionnaire ou tableau associatif,
- ensemble,
- graphe.

c TAD Tableau

Le tronc commun informatique se concentre sur l'utilisation des structures **impératives**, c'est-à-dire essentiellement les tableaux statiques, dynamiques et les variables[1].

■ **Définition 4 — TAD tableau.** Un TAD tableau représente une structure finie indiçable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice, par exemple `t[3]`.

(données) le plus souvent des nombres, en tout cas des types identiques : on appelle la donnée l'élément d'un tableau.

(opérations) on distingue deux opérations principales caractéristiques :

- l'accès à un élément via un indice entier via un opérateur de type `[]`,
- l'enregistrement de la valeur d'un élément d'après son indice.

Les implémentations du TAD tableau sont la plupart du temps des structures des données linéaires en mémoire : les données d'un tableau sont rangées dans des zones mémoires **contigües**, les unes derrières les autres.

D'un point de vue de l'implémentation, on peut construire le TAD tableau de deux manières :

1. statique : la taille du tableau est fixée, on ne peut pas ajouter ou enlever d'éléments.
2. dynamique : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

R En Python, il n'existe pas de type tableau statique dans le cœur du langage. Cependant, la liste Python est implémentée par un tableau dynamique et permet donc de pallier ce manque. Néanmoins, pour le calcul numérique, il faut absolument privilégier l'usage des tableaux Numpy qui implémentent le TAD tableau statique.

On qualifie souvent les implémentations du TAD tableau d'**impératives** : il s'agit d'une zone précise de la mémoire que l'on peut **modifier** en effectuant un effet de bord. Les tableaux sont des types **muables**. Par exemple, on peut changer la valeur de la première case d'un tableau en Python par l'instruction `t[0] = 3`.

R Le caractère muable de ces structures impératives est un inconvénient lorsqu'on cherche à prouver la correction d'un programme ou que l'on travaille dans le contexte de la programmation concurrente. Fort heureusement, il est possible de construire des structures **immuables**[a] plus adaptées à ces objectifs : des structures de données inductives et immuables.

a. on dit aussi persistantes

1. références en OCaml

B ℕ : paradigme d'un ensemble inductif dont l'ordre est bien fondé

Les deux pierres angulaires aux fondements de l'informatique sont la théorie des ensembles et la logique. La fin du XIX^{e} siècle a marqué un tournant dans l'histoire des sciences et en particulier pour les mathématiques : la théorie des ensembles et ses paradoxes ont forcé les mathématiciens à mieux formaliser les raisonnements et les démonstrations, c'est-à-dire les types et la logique.

Les entiers naturels, les éléments de l'ensemble ℕ, constituent le fondement de l'informatique. Or, on ne peut pas construire cet ensemble des nombres entiers naturels, ensemble pourtant constitué des objets mathématiques les plus évidents : tout le monde comprend les concepts 0, 1, 2, construire l'ensemble $\{0,1,2,\ldots\}$ nécessite d'expliciter les «…», ce qui n'est pas simple a priori. Les premières définition axiomatiques de ℕ apparaissent à la fin du XIX^{e} siècle, sous l'impulsion de Dedekind et Peano.

■ **Définition 5 — Définition axiomatique de l'ensemble des entiers naturels.** On postule qu'il existe un ensemble dit des «entiers naturels» noté ℕ muni de :

1. l'élément zéro, noté 0 est un entier naturel.
2. une application «successeur» $s : \mathbb{N} \to \mathbb{N}$, vérifiant les axiomes de Peano suivants :
   (a) s est injective,
   (b) Zéro n'est le successeur d'aucun entier naturel,
   (c) (Axiome de récurrence) Soit A un sous-ensemble de ℕ tel que :
   — $0 \in A$,
   — $\forall n \in A, s(n) \in A$ (stabilité de l'ensemble),
   alors $A = \mathbb{N}$.

■ **Définition 6 — Addition.** L'opération addition sur les entiers naturels est définie par :

(B) $a + 0 = a$

(I) $a + s(b) = s(a + b)$

■ **Définition 7 — Multiplication.** L'opération multiplication sur les entiers naturels est définie par :

(B) $a \times 0 = 0$

(I) $a \times s(b) = a \times b + a$

R On peut vérifier que l'addition et la multiplication sont commutatives et que la multiplication est distributive par rapport à l'addition. On en déduit une relation d'ordre $\leqslant$ sur ℕ compatible avec ces opérations :

$$\forall (m,n) \in \mathbb{N}^2, m \leq n \iff \exists k \in \mathbb{N}, n = m + k$$

R En notant $1 = s(0)$, on peut formellement se débarrasser de la fonction s en la remplaçant par $a \to a+1$. On simplifie alors les démonstrations.

Théorème 1 Toute partie non vide de N admet un plus petit élément.

Démonstration. Par l'absurde. Soit A une partie non vide de $\mathbb{N}$ n'admettant pas de plus petit élément. Considérons l'ensemble B défini par $B = \{n \in \mathbb{N}, \forall k \leq n, k \notin A\}$. Montrons que $B = \mathbb{N}$ par récurrence, ce qui impliquera $A = \emptyset$.

(Initialisation) 0 est dans B, car dans le cas contraire il serait dans A et constituerait le plus petit élément de $A \subset \mathbb{N}$, ce qui contredirait notre hypothèse.

(Hérédité) Supposons que n soit un élément quelconque de B. On cherche à montrer que $n+1$ est dans B. On procède par l'absurde. Supposons que $n+1$ appartient à A. Si $n+1 \in A$, comme aucun entier inférieur à $n+1$ n'est dans A (par hypothèse de récurrence), alors $n+1$ est le plus petit élément de A. C'est absurde d'après notre hypothèse. Donc $n+1 \notin A$. On a donc $\forall k \leq n+1, k \notin A$. D'où $n+1 \in B$.

(Conclusion) Par récurrence, $B = \mathbb{N}$.

On en déduit que $A = \emptyset$, ce qui est absurde puisqu'on a supposé que A n'était pas vide. ■

Théorème 2 — Principe d'induction. Soit $\mathcal{P}_n$ une propriété fonction d'un entier naturel n. Lorsque les deux conditions suivantes sont vérifiées :

(B) $\mathcal{P}_0$ est vraie,

(I) $\mathcal{P}_n$ implique $\mathcal{P}_{n+1}$ pour tout n,

alors pour tout entier naturel n, $\mathcal{P}_n$ est vraie.

Démonstration. Par l'absurde. Soit $\mathcal{P}_n$ une propriété dépendant d'un entier naturel n vérifiant les conditions (B) et (I). Supposons qu'il existe des entiers k pour lesquels la proposition $\mathcal{P}_k$ soit fausse et notons l'ensemble de ces entiers X : $X = \{k \in \mathbb{N}, \mathcal{P}_k \text{ est fausse }\}$.

Si X est vide, alors $\mathcal{P}_n$ est vraie pour tout entier naturel.

Si X est non vide, il admet un plus petit élément v d'après le théorème 1.

— soit $v = 0$ et alors $\mathcal{P}_0$ est fausse, ce qui contredit la propriété (B) et notre hypothèse. C'est absurde.

— soit $v > 0$, $\mathcal{P}_v$ est fausse et $\mathcal{P}_{v-1}$ est vraie. Donc (I) n'est pas vérifiée par $\mathcal{P}_n$, ce qui contredit notre hypothèse. C'est absurde.

On en conclut que $\mathcal{P}_n$, à partir du moment où elle vérifie les propriétés (B) et (I), est vraie pour tout entier naturel n. ■

R L'élément qui cristallise l'intérêt des informaticiens pour le principe d'induction, c'est qu'il est **constructif** : il permet, à partir d'une information simple (cas de base B) et de règles d'induction simples (I), de calculer un ensemble d'information.

■ **Définition 8 — Ordre bien fondé.** Soit $(E, \leqslant)$ un ensemble muni d'une relation d'ordre. On dit que l'ensemble E est **bien fondé** s'il n'existe pas de suite infinie strictement décroissante d'éléments de E.

■ **Définition 9 — Ensemble bien ordonné.** Soit $(E, \leqslant)$ un ensemble muni d'une relation d'ordre **total**, c'est-à-dire un ensemble ordonné. On dit que l'ordre sur E est **bien ordonné** s'il est également bien fondé.

Théorème 3 — Caractérisation des ordres bien fondés. Soit $(E, \leqslant)$ un ensemble. L'ordre de E est bien fondé si et seulement si toute partie non vide F de E admet un élément minimal.

R On déduit des théorèmes 3 et 1 que l'ordre $\leqslant$ associé à l'ensemble $\mathbb{N}$ des entiers naturels est bien fondé.

R $(\mathbb{N}, \leqslant)$ est un ensemble bien ordonné mais $(\mathbb{Z}, \leqslant)$ ne l'est pas.

C De l'induction à l'induction structurelle

Cette section étend le principe d'induction à tout ensemble construit de manière inductive.

■ **Définition 10 — Définition inductive d'un ensemble.** Soit E un ensemble. Une définition inductive d'une partie X de E consiste à se donner :

(B) un sous ensemble non vide $\mathcal{B}$ de E appelé ensemble de base,

(I) et d'un ensemble de règles $\mathcal{R}$: chaque règle $r_i \in \mathcal{R}$ est une fonction r_i de $E^{n_i} \to E$ telle que $\forall x_1, \cdots x_{n_i} \in X \Rightarrow r_i\left(x_1, \cdots, x_{n_i}\right) \in X$ pour $n_i > 1$.

On dit que X est alors défini inductivement. On appelle les r_i les constructeurs de X.

Théorème 4 — Théorème du point fixe. À une définition inductive d'un ensemble correspond un plus petit ensemble qui vérifie les propriétés suivantes :

1. il contient $\mathcal{B}$, c'est-à-dire $\mathcal{B} \subset X$,
2. il est stable pour les règles de $\mathcal{R}$: pour chaque règle $r_i \in \mathcal{R}$, pour tout $x_1, \cdots, x_{n_i} \in X$, on a $r_i\left(x_1, \cdots, x_{n_i}\right) \in X$.

Démonstration. Soit E un ensemble défini inductivement comme en 10 à l'aide d'un ensemble de base $\mathcal{B}$ et des règles $\mathcal{R}$.

Soit $\mathcal{P}$ l'ensemble des parties de E vérifiant (B) et (I).

$\mathcal{P}$ est non vide car il contient au moins l'ensemble E qui vérifie les conditions (B) et (I) : $E \in \mathcal{P}$.

Considérons alors X l'ensemble défini comme l'intersection de tous les éléments de $\mathcal{P}$:

$$X = \bigcap_{Y \in \mathcal{P}} Y$$

Par définition, $\mathcal{B}$ est inclus dans chaque $Y \in \mathcal{P}$. $\mathcal{B}$ est donc inclus dans X et X vérifie la condition (B).

On peut montrer que X vérifie également la condition (I). Soit une règle $r_i \in \mathcal{R}$ et des $x_1, \cdots, x_{n_i} \in X$. On a $x_1, \cdots, x_{n_i} \in Y$ pour chaque $Y \in \mathcal{P}$. Pour chaque tel Y, puisque Y est stable par la règle r_i, on doit avoir $r\left(x_1, \cdots, x_{n_i}\right) \in Y$. Puisque cela est vrai pour tout $Y \in \mathcal{P}$, on a aussi $r\left(x_1, \cdots, x_{n_i}\right) \in X$. Donc X est stable par la règle r_i.

Enfin, X est le plus petit ensemble qui vérifie les conditions (B) et (I), car, de par sa définition, il est **inclus dans tout autre ensemble** vérifiant les conditions (B) et (I). ■

Une structure définie inductivement selon 10 possède donc un plus petit élément contenu dans toutes les constructions possibles. On peut montrer que cette structure possède un ordre bien fondé.

R Puisqu'on sait qu'il existe, on considère alors **le plus petit de ces ensembles** défini inductivement qui contient $\mathcal{B}$ et qui est stable par les règles de construction $\mathcal{R}$.

En effet, si on définit l'ensemble des entiers pairs P par 0 et la règle $\forall n \in P, n+2 \in P$, alors on constate que $\mathbb{N}$ vérifie bien ces deux propriétés. Néanmoins, ce n'est pas le plus petit des ensembles caractérisés par cette définition. L'ensemble des nombres pairs est donc le plus petit de ces ensembles définis inductivement.

■ **Définition 11 — Principe d'induction structurelle.** Soit E un ensemble défini **inductivement**, de termes de base $\mathcal{B}$ et de règles de construction $\mathcal{R}$

Soit $\mathcal{P}$ une propriété sur les termes de E.

Si $\mathcal{P}$ est satisfaite pour chaque terme de base de $\mathcal{B}$ (cas de base) et pour chaque constructeur de $\mathcal{R}$ (pas d'induction), alors $\mathcal{P}$ est satisfaite pour tous les termes de E.

Plus formellement,

$$\left.\begin{array}{ll} \forall b \in \mathcal{B}, & \mathcal{P}(b) \\ \forall r \in \mathcal{R}, \forall x_1, x_2, \ldots x_n \in E, & \mathcal{P}(r(x_1, x_2, \ldots, x_n)) \end{array}\right\} \Longrightarrow \forall x \in E, \mathcal{P}(x) \qquad (1)$$

Théorème 5 — Fonction définie inductivement. Soit $X \subset E$ un ensemble défini inductivement de façon non ambigüe à partir de l'ensemble de base $\mathcal{B}$ et des règles $\mathcal{R}$. Soit Y un ensemble.

Pour qu'une application f de X dans Y soit parfaitement définie, il suffit de se donner :

(B) la valeur de $f(x)$ pour chacun des éléments $x \in B$,

(I) pour chaque règle $r_i \in \mathcal{R}$, la valeur de $f(x)$ pour $x = r_i\left(x_1, \cdots, x_{n_i}\right)$ en fonction de la valeur $x_1, \ldots, x_{n_i}, f(x_1), \ldots$ et $f\left(x_{n_i}\right)$.

Démonstration. On cherche à montrer que la fonction f ainsi définie associe bien une unique valeur dans Y à chaque $x \in X$. On procède par induction.

— Cas de base : la valeur associée à chaque $x \in \mathcal{B}$ est bien unique d'après (B).

— Pas d'induction : supposons que cela est vrai pour $x_1, \cdots, x_{n_i}$. On cherche à montrer que cela est vrai en $x = r_i\left(x_1, \cdots, x_{n_i}\right)$, c'est-à-dire que les $f(x_1), \ldots, f(x_{n_i})$ sont définis de manière unique. Comme x ne peut être obtenu que par la règle r_i à partir de $x_1, \ldots, x_{n_i}$, $f(x)$ est donc parfaitement définie par la contrainte pour la règle (I).

■

Pour un ensemble X défini inductivement, une fonction est donc parfaitement définie par le théorème 5 sur X.

■ **Définition 12 — Factorielle, définie inductivement.** Comme on a définit l'ensemble des entiers naturels $\mathbb{N}$ inductivement (cf. définition 5), la fonction factorielle peut être définie inductivement par :

Fact : $\mathbb{N} \to \mathbb{N}$

(B) $\text{Fact}(0) = 1$,

(I) $\text{Fact}(s(n)) = s(n) \times \text{Fact}(n)$, ce qui peut également s'écrire : $\text{Fact}(n+1) = (n+1) \times \text{Fact}(n)$.

D Structures de données définies par induction

La capacité de construire des calculs sur les entiers naturels de manière inductive est fascinante : pour un mathématicien, cet ensemble bien ordonné engendre une infinité de preuves et permet de définir des fonctions sur $\mathbb{N}$; pour un informaticien, cet ensemble est le modèle qui permet la construction d'autres ensembles. Définir une structure de données de manière inductive, c'est pouvoir **construire et calculer** sur les éléments de cet ensemble, c'est-à-dire la raison d'être de l'informatique. Ces **structures inductives** irriguent toutes les parties du programme officiel de l'option informatique et la plupart des développements de ce cours : les listes, les arbres, formules logiques, et les expressions régulières.

a Exemple TAD Liste

■ **Définition 13 — TAD liste.** Un TAD liste représente **une séquence finie d'éléments d'un même type** qui possède un **rang** dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est **dynamique**, c'est à dire qu'on peut ajouter ou enlever des éléments.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut alors zéro. La tête de liste (head) est le premier élément de la liste. La queue de liste (tail) est la liste privée de son premier élément.

(données) de type simple ou composé

(opérations) on peut trouver[a] :

— un constructeur de liste vide,

— un opérateur de test de liste vide,
— un opérateur pour ajouter en tête de liste,
— un opérateur pour ajouter en fin de liste,
— un opérateur pour déterminer et/ou retirer la tête de la liste,
— un opérateur pour déterminer et/ou retirer la queue de la liste (tout sauf la tête),
— un opérateur pour accéder au ième élément.
— un opérateur pour accéder au dernier élément de la liste.

a. Toutes les implémentations ne proposent pas nécessairement toutes ces opérations !

Le TAD liste peut être implémenté par un tableau de manière impérative[2] mais il peut également l'être par une structure définie inductivement.

b Structure de liste inductive

■ **Définition 14 — Liste construite de manière inductive.** Une liste d'éléments est soit :

($\mathcal{B}$) Vide

($\mathcal{C}$) une liste construite en AJOUTANT EN TÊTE un élément à une liste

R Les éléments d'une structure définie de manière inductive sont appelés **termes**.

R Un constructeur possède une certaine **arité** : il permet de construire un terme à partir de un ou plusieurs termes. Dans l'exemple de la liste, AJOUT EN TÊTE est un constructeur d'arité 1 car il ne prend qu'un seul paramètre.

R Cette définition inductive engendre un ordre dans les termes. **Cet ordre partiel et bien fondé est appelé ordre structurel.**

E Induction structurelle

Il est possible de démontrer des propriétés des structures inductives grâce à l'**induction structurelle**.

■ **Définition 15 — Principe d'induction structurelle.** Soit $\mathcal{I}$ un ensemble inductif de termes de base $\mathcal{B}$ et de constructeurs $\mathcal{C}$. Soit $\mathcal{P}$ une propriété sur les termes de $\mathcal{I}$.

Si $\mathcal{P}$ est satisfaite pour chaque terme de base de $\mathcal{B}$ et pour chaque constructeur de $\mathcal{I}$, alors $\mathcal{P}$ est satisfaite pour tous les termes de $\mathcal{I}$.

2. comme c'est le cas en Python

Plus formellement,

$$\left.\begin{array}{ll} \forall b \in \mathcal{B}, & \mathcal{P}(b) \\ \forall c \in \mathcal{C}, \forall t_1, t_2, \dots t_n \in \mathcal{I}, & \mathcal{P}(c(t_1, t_2, \dots, t_n)) \end{array}\right\} \Longrightarrow \forall t \in \mathcal{I}, \mathcal{P}(t) \tag{2}$$

F Calculer des fonctions sur une structure inductive

On peut facilement définir des fonctions sur les ensembles inductifs en s'appuyant sur leur définition. Cela permet de faire des calculs sur l'information qu'ils représentent.

■ **Exemple 5 — Longueur d'une liste.** On peut définir la longueur d'une liste en s'appuyant sur la définition inductive : la longueur d'une liste vide est nulle et la longueur d'une liste à laquelle on ajoute en tête un élément est la longueur de cette liste plus un.

On obtient alors une fonction LONGUEUR définie récursivement :

(Cas de base) LONGUEUR(Vide) = 0

(Règle de construction) LONGUEUR(AJOUTER EN TÊTE(L,a)) = 1 + LONGUEUR(L)

G Pourquoi OCaml ?

Comme on peut le voir ci-dessous et comme on le verra dans les chapitres suivants, le langage OCaml est particulièrement adapté aux structures inductives pour plusieurs raisons :

1. les types en OCaml sont nativement récursifs,
2. les types OCaml peuvent être des types somme | ou produit $*$. On les appelle des types algébriques.
3. OCaml procure la syntaxe dite du filtrage de motifs,
4. OCaml permet d'écrire des fonctions récursives.

Combiner ces quatre fonctionnalités permet de traduire directement la plupart des concepts mathématiques exprimés sous la forme d'un ensemble inductif. Les **fonctions** dont les paramètres peuvent être des types algébriques concrétisent les calculs sur les termes des ensembles inductifs.

Ces fonctionnalités de OCaml[3] expliquent en grande partie le choix du langage OCaml en CPGE pour l'option informatique.

Vocabulary 1 — Pattern matching ⇜⇝ Filtrage de motifs

```
type ingredient =
        | Sel | Poivre | Beurre | Sucre | Farine
        | Fondre of ingredient
        | Melanger of ingredient*ingredient
```

3. Tous les langages ne disposent pas de ces fonctionnalités : Python ne dispose pas de types algébriques récursifs et son paradigme est plutôt impératif, pas fonctionnel.

12

```
let rec masse = function
        | Sel —> 30
        | Poivre —> 10
        | Beurre —> 250
        | Sucre —> 250
        | Farine  —> 10
        | Fondre i —>  masse i
        | Melanger (i1,i2) —> masse i1 + masse i2

let kouignamann = Melanger(Beurre, Sucre) in masse(kouignamann)
```