TD 3 : Structures de données et effets de bord

Octobre-Novembre 2012

Vue d'ensemble

On parle de *structure de données* pour nommer les différentes manières dont on peut organiser un ensemble de données. Suivant l'organisation choisie, on obtient des propriétés variées, parfois étonnantes, comme l'accés "rapide" à n'importe quel élément ou au plus petit élément de l'ensemble de données.

Voyons voir ce que vous réserve Maple :

Les suites ou n-upplets : Déjà rencontrées pendant le dernier TD, on peut en construire facilement avec `seq` et la transformer ensuite en une liste ou un ensemble.

Les listes : Une structure de données permettant de maintenir un ordre sur des éléments et d'en insérer facilement. Malgré la syntaxe `liste[n]` qui permet d'accéder au $n^{\text{ème}}$ élément, il est difficile de les voir comme des tableaux : au delà d'une centaine d'élèments, on ne peut pas modifier le contenu d'une case.

Les ensembles : Comme son nom l'indique, cette structure représente un ensemble au sens mathématique, c'est à dire une "suite" sans ordre ni doublon. Pour créer un ensemble, il suffit d'entourer une suite par des accolades :

```
> a := 3, 6, 8, 9, 8, 2, 3, 5;
                          a := 3, 6, 8, 9, 8, 2, 3, 5
> b := { a };
                           b := { 2, 3, 5, 6, 8, 9 }
```

Vous pouvez extraire un élément ou un interval avec la même syntaxe qu'une liste.

```
> b[3..5];
                                  { 5, 6, 8 }
```

Les opérations ensemblistes `union`, `intersect`, `minus`, `subset` sont importantes et s'emploie ainsi :

```
> { 2, 3 } union { 5, 7, 2 };
                                { 2, 3, 5, 7 }
```

Pour tester l'appartenance d'un élément à un ensemble, vous pouvez utiliser l'opérateur `in` :

```
> is(5 in { 2 , 9, 7 });
                                  false
> evalb(k in { i, j, k });
                                  true
```

Les tableaux : C'est une structure où les éléments sont indexés par des entiers. L'accés à un élément est sensé se faire en temps constant et on peut les modifier. Un tableau posséde une dimension d et chaque dimension $1 \leq i \leq d$ à une taille l_i. Le nombre de cases d'un tableau est alors $\prod_{i=1}^{d} l_i$. Pour fabriquer un tableau, on utilise le constructeur `Array` :

```
> a := Array([ 5, 6, 8, 9, 10, 42 ]);
                        a := [ 5 6 8 9 10 42 ]
> Array(1..2, 1..3, [[z,x,e],[p,h,k]]);
                                | z x e |
                                | p h k |
```

Le premier tableau est unidimensionel, construit à partir d'une liste tandis que le second tableau, de dimension 2 et de taille 2×3, est construit à partir d'une liste de liste. Il est aussi possible de fabriquer un tableau à partir d'une fonction. Pour accéder à un élément d'un tableau, on utilise la même syntaxe que pour les listes, en donnant une suite de coordonnées si le tableau est multidimensionel.

Les tableaux associatifs ou *map* : Nommés `table` en maple, il s'agit de tableaux où les éléments peuvent être indexés par n'importe quoi (des chaines de caractères, des équations, ...). Pour en fabriquer, on utilise la syntaxe suivante :

```
> a := table(["x" = 5, "réponse" = 42, "bonjour" = 76]);
                    a := table(["x" = 5, "réponse" = 42, "bonjour" = 76])
> a["réponse"];
                                  42
```

Les effets de bord

On peut voir la plupart des procédures que l'on a écrit jusqu'à présent comme des boîtes noires hermétiques : on fournit des valeurs à la procédure qui travaille dessus puis elle nous renvoit une valeur. La seule interaction de la procédure avec l'environnement (le contexte) passe par les arguments que nous lui envoyons et la valeur qu'elle nous retourne. En fait ce n'est pas tout à fait vrai. Nous avons vu dans le TD2 que la fonction `print` permettait d'agir sur l'environnement (afficher quelque chose à l'écran en l'occurence) sans que cela passe via les arguments ou la valeur retournée. On appelle cela un *effet de bord*. Il y a deux principaux effets de bord que nous allons manipuler :

- ▷ Les entrée-sortie (IO). Elles correspondent à toutes les actions d'interaction avec un utilisateur ou un système de fichier, c'est à dire afficher à l'écran, lire une entrée depuis le clavier, enregistrer ou récupérer des données dans un fichier. On retiendra pour l'instant la fonction `print` qui permet d'afficher n'importe quelle valeur et éventuellement la fonction `printf` qui permet de formatter le texte en sortie (= mettre en forme le texte).
- ▷ Les modifications de variables globales. On a vu au TD précédent que l'on pouvait déclarer des variables avec le mot-clé `global` (en opposition avec `local`). Contrairement aux variables locales, les variables globales sont des variables de l'environnement que la fonction peut modifier. Leur principal intéret est d'éviter la copie de trés grosse quantité de données. En effet, une valeur passée en argument doit être copiée pour être modifiée puis renvoyée ce qui n'est pas souhaitable sur de gros tableaux par exemple. Il faut par contre être prudent avec les variables globales car elles peuvent avoir des effets inattendus si on ne fait pas attention : une même variable globale employée dans deux buts distinct peut mener à des situations abérantes.

Un peu de complexité

Pour mesurer l'efficacité d'un programme, on pourrait se contenter de mesurer le temps d'éxécution sur plusieurs entrée. Cependant, le matériel changeant beaucoup d'une machine à une autre, cela ne constitue pas une échelle adéquate. On va donc essayer de caractériser le nombre d'opérations effectuées par un algorithme en fonction de la "taille" de son entrée : pour une liste ou un tableau il s'agit de sa longueur, pour un entier trés gros de son nombre de bits en binaire, pour un polynôme on pourrait envisager le degrés maximal ou le nombre de monômes... Ensuite, on considère qu'une instruction atomique (affectation, calcul pas trop compliqué, test dans une structure de contrôle) s'éxécute en $\mathcal{O}(1)$ et on calcule ainsi la *complexité* de la fonction. Voici des exemples montrant comment calculer la complexité dans les cas de base :

▷ Calcul de base en $\mathcal{O}(n)$:

```
bidule := proc(n)
  local x;
  x := 0;
  for i from 0 to n do
    x := x + 1;
  end do;
end proc;
```

▷ Voici un exemple en $\mathcal{O}(n * m)$ (les deux boucles sont imbriquées) :

```
chose := proc(n,m)
  local a,b;
  for a from 1 to n do
    for b from 1 to m do
      print(a*b);
    end do;
  end do;
end proc;
```

▷ Dans cet exemple, on est en $\mathcal{O}(n + m)$ (les boucles ne sont pas imbriquées) :

```
truc := proc(n,m)
  local i;
  i := 0;
  while i < n do print(i); end do;
  while i < m do print(i); end do;
end proc;
```

Exercices

Exercice 1 (*Suites récurrentes*)

On considère une fonction $f : \mathbb{R} \to \mathbb{R}$ et une suite $(u_n)_{n\geq 0}$ définie par

$$\begin{cases} u_0 \in \mathbb{R} \\ u_{n+1} = f(u_n) \end{cases}$$

1. *Écrivez une procédure qui prend en entrée f, u_0 et n et qui renvoit la liste $(u_k)_{0\leq k\leq n}$.*
2. *Écrivez une procédure qui calcule la suite $(v_k)_{k\geq 0}$ définie par :*

$$\begin{cases} v_0 = (u_0, 0) \\ v_{2k+1} = (u_k, u_{k+1}) & \text{pour } k \geq 0 \\ v_{2k+2} = (u_{k+1}, u_{k+1}) & \text{pour } k \geq 0 \end{cases}$$

3. *Testez vos fonctions pour f, u_0 et n bien choisis en traçant la suite $(v_k)_{k\geq 0}$.*

Exercice 2 (*Tri par insertion*)

On veut trier une liste ou un tableau de n éléments. Pour cela, on considère la taille du conteneur :
- ▷ *Si $n = 0$ ou 1 le conteneur est trié*
- ▷ *Si les $n-1$ premiers éléments du conteneur sont déjà triés, on peut trier tout le conteneur en insérant le $n^{\text{ème}}$ élément à la bonne place*

Utilisez cette idée pour implémenter un algorithme de tri, sur un tableau ou sur une liste comme vous le souhaitez.

Exercice 3 (*Min, Max & Rang*)

1. *Écrivez une fonction qui calcule le* min *et/ou le* max *d'une liste.*
2. *Écrivez une fonction qui calcule les deux plus petits élèments d'une liste.*
3. *Écrivez une fonction qui calcule le $k^{\text{ème}}$ élément d'une liste pour n'importe quel k valide (vous pouvez utiliser l'exercice précédent). On dit que cet élèment est de rang k.*
4. Bonus : *Trouvez un algorithme linéaire pour trouver l'élément de rang k.*

Exercice 4 (*Crible d'ératosthène*)

Le crible d'hératosthène est un algorithme permettant de calculer assez efficacement tous les nombres premiers dans $[\![1; n]\!]$. Le principe de cet algorithme est d'écrire tous les nombres de $[\![1; n]\!]$ puis pour chaque nombre k non-rayé entre 1 et un nombre que vous déterminerez, de rayer tous les multiples de k. Implémentez le crible d'ératosthène en employant un tableau de booléens avec `true` *si le nombre n'est pas rayé et* `false` *sinon.*

Exercice 5 (*Effets de bord ambigüs*)

Essayez de déterminer ce que pourrait renvoyer `f()` *et* `g()` *dans les codes suivant :*

```
f := proc()
  local x, inc;
  inc := proc()
    global x, y;
     y := x;
     x := y +1 ;
     y;
  end proc;
  x := 2;
  inc()* inc() + inc()/inc();
end proc;
```

```
g := proc()
  local a, b;
  a := proc(x)
    global a,b;
    a := x -> 42; x + 5;
  end;
  b := proc(y)
    global a,b;
    a := z -> a(z + 3); b := 5;
  end;
  a(a(b(42)));
end;
```

Exercice 6 (*Programmation dynamique*)

Donner un algorithme pour trouver la plus longue sous-séquence monotone croissante d'une séquence de n nombres qui soit en temps $\mathcal{O}(n^2)$.

La notation de Landau $\mathcal{O}()$

Soit deux fonctions f et $g : \mathbb{N} \to \mathbb{N}$, on dit que $f = \mathcal{O}(g)$ s'il existe $N \in \mathbb{N}$ et $C > 0$ tel que $\forall n > N, f(n) \leq C \cdot g(n)$.

Quelques propriétés importantes :

▷ Si $f = \mathcal{O}(K \cdot g)$ où $K > 0$ est une constante alors $f = \mathcal{O}(g)$

▷ Si $f = \mathcal{O}(g)$ et $g = \mathcal{O}(h)$ alors $f = \mathcal{O}(h)$

▷ Si $f_1 = \mathcal{O}(g_1)$ et $f_2 = \mathcal{O}(g_2)$ alors $f_1 + f_2 = \mathcal{O}(g_1 + g_2)$

En pratique, rappelez-vous surtout qu'un polynôme $P(n)$ est en $\mathcal{O}(n^{dP})$ où dP est le degré du polynôme.

Parlons de typage

Une commande dans Maple est une entrée sous forme d'une chaine de caractères. Le polynôme $x^2 - 1$ est représenté par le mot "`x^2-1`". Lorsque l'on tape une ligne en Maple, celle-ci est ensuite interprétée : Maple transforme la ligne tapée en un ensemble de données qu'il va pouvoir traiter.

Afin de mieux représenter en mémoire ces données, celles-ci sont classées en plusieurs "types". Chaque type représente une brique de base permettant de construire des expressions beaucoup plus complexes.

La commande `whattype(...)` permet de connaitre le type d'une expression. Appliquée à une expression simple, elle renvoie le type de l'expression en question ; appliquée à une expression composée, le type de l'opération de plus haut niveau (celle de plus basse priorité).

Les types sont très variables : types numériques (integer, float, fraction, ...), algébriques (+, *, ^), d'égalités (= , <>, <=), de relations logiques, de liste, d'ensemble, de tableau, ...

Exercice 7

Quel est le type des expressions suivantes ?

▷ *Types numériques :* `3 ; 4.2 ; 113/3 ; I`

▷ *Types algébriques :* $x + y$ `;` $x - y$ `;` $x * y$ `;` x/y `;` x^y

▷ *Relations logiques :* `(A or B) ; (A and B) ; (not A)`

▷ *Structures de données :* `[5, 6, 7] ; [8, 0], [7] ; Array(seq(4-i,i=1..4))`

▷ *Autres :* $\sin(x)$; `(proc(x) x^2; end)` ; $2..8$; `'x' ; ''sardine''; Pi ; evalf(Pi)`