3
La récursivité

3.1 Appeler une fonction dans le corps de la fonction

Au chapitre précédent, pour définir la fonction Σ sur une instruction de la forme `f(`$\mathtt{t}_1$`,...,`$\mathtt{t}_n$`);`, nous avons utilisé la fonction Σ sur l'instruction `p`, qui est le corps de la fonction `f`. Cette définition est-elle bien formée ou peut-elle être circulaire ?

Cette définition est clairement bien formée quand l'instruction `p` ne contient pas elle-même d'appels de fonctions, c'est-à-dire quand le programme principal — la fonction `main` — appelle des fonctions qui n'appellent pas, elles-mêmes, de fonctions.

Cette définition est également bien formée quand le programme contient `k` définitions de fonctions $\mathtt{f}_1$, ..., $\mathtt{f}_k$ telles que le corps de la fonction $\mathtt{f}_i$ ne contienne que des appels à des fonctions antérieurement définies, c'est-à-dire à des fonctions $\mathtt{f}_j$ pour $\mathtt{j} < \mathtt{i}$. Certains langages, comme Fortran, ne permettent ainsi d'appeler une fonction `f` dans le corps d'une fonction `g` que si `f` a été définie avant `g`. Dans ce cadre, la définition de la fonction Σ est une récurrence triple : d'abord sur le numéro de la fonction, puis sur le nombre de boucles `while` imbriquées, puis sur la taille de l'instruction. Remarquons qu'un tel ordre sur les fonctions existe toujours si ces fonctions sont introduites à partir du programme principal en isolant des parties de programme l'une après l'autre : on isole une partie $\mathtt{p}_k$ du programme principal, puis une partie $\mathtt{p}_{k-1}$ du programme principal ou de la fonction $\mathtt{p}_k$, ...

Cependant, la plupart des langages de programmation permettent d'écrire des fonctions qui s'appellent elles-mêmes, ou qui appellent des fonctions qui appellent d'autres fonctions, ... qui, finalement, appellent la fonction initiale.

Cette possibilité est implicite dans notre définition de la fonction Σ de Java puisque l'environnement global `G` est global au programme : les fonctions utilisables dans le corps d'une fonction sont toutes celles du programme. Ainsi, rien n'empêche d'appeler une fonction `f` dans le corps d'une fonction `g`, que cette fonction `f` soit définie avant `g`, après `g` ou que ce soit la fonction `g` elle-même.

On appelle *définitions récursives* de fonctions les définitions de fonctions qui appellent la fonction définie elle-même, ou qui appellent des fonctions qui en appellent d'autres, qui, *in fine*, appellent la fonction initiale. Pour les définitions récursives, la définition de la fonction Σ du chapitre précédent peut être circulaire. Par exemple, si `f` est une fonction définie ainsi

```
static void f (final int x) {f(x);}
```

alors la définition de Σ`(f(x);,e,m,G)` utilise la valeur de Σ`(f(x);,e,m,G)`, ce qui est circulaire.

Nous devons donc trouver une autre manière de définir cette fonction Σ.

3.2 Les définitions récursives

3.2.1 Les définitions récursives et les définitions circulaires

Un exemple plus intéressant de définition récursive est celle de la fonction factorielle

```
static int fact (final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

Pour calculer la factorielle du nombre `3`, on doit calculer la factorielle de `2`, ce qui demande de calculer la factorielle de `1`, ce qui demande de calculer la factorielle de `0`. Cette valeur est `1`. La factorielle de `1` est donc obtenue en multipliant `1` par cette valeur, ce qui donne `1`. La factorielle de `2` est obtenue en multipliant `2` par cette valeur, ce qui donne `2`. Et la factorielle de `3` est obtenue en multipliant `3` par cette valeur, ce qui donne `6`.

On dit parfois qu'une définition récursive est une définition qui utilise l'objet qu'elle définit. Cette idée est absurde : les définitions circulaires sont incorrectes, dans les langages de programmation, comme ailleurs. D'ailleurs, s'il était

possible d'utiliser une fonction dans sa propre définition, la fonction factorielle aurait une définition beaucoup plus simple

```
static int f (final int x) {return f(x);}
```

et la définition de la fonction qui multiplie son argument par 4 ou qui l'élève au carré serait absolument identique.

3.2.2 Les définitions récursives et les définitions par récurrence

Une autre tentative de comprendre la définition de la fonction `fact` est d'y voir une définition par récurrence de la suite u_n = `n !` : u_0 = 1, u_{n+1} = `(n + 1) *` u_n. Si cette tentative aboutit pour cette fonction, elle n'aboutit pas dans le cas général, par exemple pour la fonction

```
static int f (final int n) {
 if (n <= 1) return 1;
 if (n % 2 == 0) return (1 + f(n / 2));
 return 2 * f(n + 1);}
```

En effet, le calcul de la valeur de cette fonction en 11 demande le calcul de sa valeur en 12, qui demande le calcul de sa valeur en 6, qui demande le calcul de sa valeur en 3, qui demande le calcul de sa valeur en 4, qui demande le calcul de sa valeur en 2, qui demande le calcul de sa valeur en 1. Ainsi, le calcul de la valeur de la fonction `f` en `n` ne demande pas uniquement celui de sa valeur en `n - 1`, ni même celui de sa valeur en des entiers strictement inférieurs à `n`, mais également celui de sa valeur en des entiers supérieurs à `n`. Pourtant, cette définition est correcte et, pour tout entier `n`, le calcul de la valeur de `f` en `n` donne un résultat.

Un exemple plus intéressant est la fonction suivante

```
static int ack (final int x, final int y) {
 if (x == 0) return 2 * y;
 if (y == 0) return 1;
 return ack(x - 1,ack(x,y - 1));}
```

qui donne toujours un résultat après un nombre fini d'appels, mais qui, d'après un théorème démontré par Wilhelm Ackermann en 1928, ne peut pas être définie en imbriquant des définitions par récurrence.

3.2.3 Les définitions récursives et les programmes infinis

Quand on a une définition récursive de fonction, par exemple la définition de la factorielle, il est possible de transformer cette définition en une autre, non récursive, en remplaçant les appels à la fonction `fact` dans le corps de la fonction `fact` par des appels à une autre fonction `fact1`, identique à `fact`, mais définie avant elle

```
static int fact1 (final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}

static int fact (final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}
```

La définition de la fonction `fact` n'est désormais plus récursive, mais celle de la fonction `fact1` l'est. On peut, de même, remplacer les appels à la fonction `fact1` dans le corps de la fonction `fact1` par des appels à une fonction `fact2`, et ainsi de suite. On aboutit à un programme qui n'est plus récursif, mais qui est infini. Les définitions récursives sont donc, comme la boucle `while`, un moyen d'exprimer des programmes infinis et, comme la boucle `while`, les définitions récursives introduisent une potentialité de non terminaison.

Comme pour le cas de la boucle `while`, on peut introduire une expression fictive `giveup` et approcher ce programme infini par des approximations finies obtenues en remplaçant la définition de la $\mathtt{n}^{\text{ème}}$ copie de la fonction `fact` par la fonction `giveup` et en supprimant les suivantes qui ne sont plus utiles.

Calculer la valeur de la $\mathtt{n}^{\text{ème}}$ approximation du programme `p` consiste à tenter de calculer la valeur du programme `p` en faisant au maximum `n` appels récursifs imbriqués. Si au bout de ces `n` appels, le calcul n'est pas terminé, on l'abandonne.

Il n'est pas difficile de démontrer que pour tout état `e, m`, ou bien la suite $\Sigma(\mathtt{p_n,e,m,G})$ n'est jamais définie ou bien elle est définie à partir d'un certain rang, et dans ce cas, elle est constante sur son domaine. Rappelons que, dans ce second cas, la limite de la suite est la valeur qu'elle prend sur son domaine et que la suite n'a en revanche pas de limite si elle n'est jamais définie.

On peut maintenant définir la fonction Σ en `(p,e,m,G)`

$$\Sigma(\mathtt{p,e,m,G}) = \mathtt{lim_n}\ \Sigma(\mathtt{p_n,e,m,G})$$

Pour généraliser cette idée au cas de plusieurs fonctions mutuellement récursives, on va abandonner l'idée de recopier les fonctions mais introduire un paramètre pour le nombre d'appels de fonctions imbriqués. On définit une famille de fonctions $\Sigma_\mathtt{k}$ telle que $\Sigma_\mathtt{k}(\mathtt{p,e,m,G})$ soit le résultat de l'exécution de

l'instruction p, si l'exécution de cette instruction demande k appels imbriqués au plus. Si au bout de k appels imbriqués, le calcul n'est pas terminé, alors la fonction Σ_k n'est pas définie en (p,e,m,G).

La définition des fonctions Σ_k est tout à fait similaire à celle de la définition de la fonction Σ, sauf dans le cas des appels de fonctions. Pour définir Σ_k(f(t_1,...,t_n);,e,m,G), on commence par définir (v_1,m_1) = Θ_k(t_1,e,m,G), (v_2,m_2) = Θ_k(t_2,e,m_1,G), ..., (v_n,m_n) = Θ_k(t_n,e,m_{n-1},G), puis e'' et m'' comme au chapitre précédent. Ensuite, au lieu de considérer l'objet Σ_k(p,e'',m'',G) on considère l'objet Σ_{k-1}(p,e'',m'',G). Sauf, bien entendu, si k = 0 et dans ce cas, la fonction Σ_0 n'est pas définie pour cette expression.

Une fois la famille de fonctions Σ_k définie, on définit la fonction Σ

$$\Sigma(p,e,m,G) = \lim_k \Sigma_k(p,e,m,G).$$

3.2.4 Les définitions récursives et les équations au point fixe

La définition récursive de la fonction fact

```
static int fact (final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

ne peut pas se lire comme une définition, c'est-à-dire comme une opération consistant à attribuer un nom fact à un objet. On peut en revanche la considérer comme une équation dont la variable est la fonction fact. En effet, la fonction fact est l'unique fonction f des entiers naturels dans les entiers naturels qui vérifie l'équation

$$f = (x \mapsto \text{si } (x == 0) \text{ alors } 1 \text{ sinon } x * f(x - 1))$$

Cette équation a la forme f = G(f), c'est donc une équation au point fixe.

Certaines équations au point fixe, par exemple, l'équation

$$f = (x \mapsto 1 + f(x))$$

qui correspond à la définition récursive

```
static int boucle (final int x) {
 return 1 + boucle(x);}
```

n'ont pas de solutions, quand on se restreint aux fonctions totales des entiers dans les entiers.

Toutefois, nous avons vu que les fonctions récursivement définies peuvent ne pas terminer. C'est donc parmi les fonctions partielles des entiers dans les

entiers qu'il faut chercher des solutions. Et, dans cet ensemble, l'équation au point fixe

$$\texttt{f = (x} \mapsto \texttt{1 + f(x))}$$

a une solution, qui est la fonction de domaine vide.

On peut démontrer que, de manière générale, une telle équation au point fixe a toujours au moins une solution dans l'ensemble des fonctions partielles des entiers dans les entiers. Ces différentes fonctions peuvent être ordonnées par la relation d'inclusion de leurs graphes et on peut démontrer que, parmi toutes ces fonctions, l'une d'elles est la plus petite. C'est cette fonction qu'une définition récursive définit.

Il y a donc ici une manière alternative de définir la fonction Σ. Cette définition alternative est cependant rigoureusement équivalente à celle de la section 3.2.3, car ce théorème d'existence d'un point fixe se démontre précisément en construisant une solution comme la limite d'une suite de fonctions.

Exercice 3.1

Quelles sont toutes les solutions de l'équation suivante ?

$$\texttt{f = (x} \mapsto \texttt{f(x))}$$

Quelle est la plus petite ?

Soit `boucle` la fonction ainsi définie

```
static int boucle (final int x) {
 return boucle(x);}
```

Quelle est la valeur de l'expression `boucle(4)` ?

Exercice 3.2

Quelles sont toutes les solutions de l'équation suivante ?

$$\texttt{f = (x} \mapsto \texttt{2 * f(x))}$$

Quelle est la plus petite ?

Soit `boucle` la fonction ainsi définie

```
static int boucle (final int x) {
 return 2 * boucle(x);}
```

Quelle est la valeur de l'expression `boucle(4)` ?

Exercice 3.3

Dans l'ensemble des fonctions partielles des entiers relatifs dans les entiers relatifs, quelles sont toutes les solutions de l'équation suivante ?

`f = (x ↦ si (x == 0) alors 1 sinon x * f(x - 1))`

Quelle est la plus petite ?

En supposant un type `int` fictif pour les entiers relatifs, que renvoie l'appel `fact(-1)` ?

Dans l'ensemble des fonctions partielles d'un intervalle dans lui-même, quelles sont toutes les solutions de cette équation ?

Que renvoie l'appel `fact(-100)` si on définit la fonction `fact` de la manière suivante ?

```
static byte fact (final byte x) {
 if (x == 0) return 1;
 return (byte) (x * fact((byte) (x - 1)));}
```

Pourquoi ? Et l'appel `fact(-100)` si on définit la fonction `fact` de la manière suivante ?

```
static double fact (final byte x) {
 if (x == 0) return 1.0;
 return x * fact((byte) (x - 1));}
```

Pourquoi ?

3.3 Caml

En Caml, on exécute le corps de la fonction dans l'environnement dans lequel cette fonction a été déclarée, étendu par la déclaration de ses arguments. De ce fait, seules les fonctions définies avant la fonction `f` *sont accessibles dans le corps de* `f`*, et la définition*

```
let fact x = if x = 0 then 1 else x * fact(x - 1)
in print_int (fact 6)
```

est incorrecte.

Pour pouvoir utiliser la fonction `fact` *dans sa propre définition il faut utiliser une nouvelle construction* `let rec f` x_1 `...` x_n `= t in p`

```
let rec fact x = if x = 0 then 1 else x * fact(x - 1)
in print_int (fact 6)
```

et la définition de la fonction `fact` *est alors ajoutée, à chaque appel, à l'environnement dans lequel on exécute le corps de la fonction.*

Quand deux fonctions sont mutuellement récursives, on ne peut pas les définir ainsi

```
let rec pair x = if x = 0 then true else impair(x - 1)
in let rec impair x = if x = 0 then false else pair(x - 1)
in print_bool(pair 7)
```

car l'environnement dans lequel on exécute la fonction `pair` *ne contient pas la fonction* `impair` *et on doit utiliser une construction spéciale pour les fonctions mutuellement récursives* `let rec f` x_1 `...` x_n `= t and g` y_1 `...` y_p `= u and ...` *Par exemple*

```
let rec pair x = if x = 0 then true else impair (x - 1)
and impair x = if x = 0 then false else pair (x - 1)
in print_bool (pair 7)
```

3.4 C

En C, comme en Caml, on exécute le corps de la fonction dans l'environnement dans lequel cette fonction a été déclarée, étendu par la déclaration des arguments. De ce fait, seules les fonctions définies avant une fonction `f` *sont accessibles dans le corps de* `f`*.*

Cependant, afin de permettre la récursivité, la définition de la fonction `f` *est ajoutée, à chaque appel, à l'environnement dans lequel on exécute le corps de la fonction. On fait donc exactement la même chose qu'en Caml pour un* `let rec`*, ou autrement dit, la définition d'une fonction en C correspond au* `let rec` *de Caml et non au* `let`*. Ainsi, le programme*

```
int fact (const int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}

int main () {
 printf("%d\n",fact(6));
 return 0;}
```

est correct et affiche 720*.*

Quand les définitions de plusieurs fonctions, par exemple de deux fonctions `f` *et* `g`*, sont mutuellement récursives, on doit commencer par* prototyper *la*

fonction `g` *pour permettre l'appel de* `g` *dans le corps de* `f`*. Prototyper une fonction signifie annoncer le type de ses arguments et de sa valeur de retour et s'engager à définir cette fonction plus bas dans le programme.*

```
int impair (const int);

int pair (const int x) {
 if (x == 0) return 1;
 return impair(x - 1);}

int impair (const int x) {
 if (x == 0) return 0;
 return pair(x - 1);}

int main () {
 printf("%d\n",pair(7));
 return 0;}
```

3.5 Programmer sans affectation

En comparant la fonction factorielle écrite avec une boucle

```
static int fact (final int x) {
 int i;
 int r;

 r = 1;
 for (i = 1; i <= x; i = i + 1) {r = r * i;}
 return r;}
```

et récursivement

```
static int fact (final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

on constate que la première utilise des affectations : `r = 1 ;`, `i = 1 ;`, `i = i + 1 ;` et `r = r * i ;`, mais pas la seconde. Il est donc possible de programmer la fonction factorielle sans utiliser d'affectations.

Plus généralement, on peut considérer un sous-langage de Java dans lequel on supprime l'affectation. Dans ce cas, toutes les variables peuvent être déclarées finales et, dans la définition de la fonction Σ, la mémoire est toujours vide.

La séquence et la boucle sont de ce fait devenues inutiles. Il reste un noyau de Java formé de la déclaration de variables finales, de la définition récursive de fonctions, de l'appel de fonctions, des opérations arithmétiques et logiques et du test. Ce sous-langage s'appelle le *noyau fonctionnel* de Java. On peut, de même, définir le noyau fonctionnel de beaucoup de langages de programmation.

De manière surprenante, ce noyau fonctionnel a la même puissance que Java tout entier. Essayons de préciser cela. À chaque expression `t` de Java, on associe la fonction partielle des entiers dans les entiers qui à l'entier `n` associe la valeur `v` telle que `(v,m') =` Θ`(t,[x = n],[],G)`, et à chaque instruction `p` de Java on associe la fonction partielle des entiers dans les entiers qui à l'entier `n` associe la valeur `v` telle que `(return,v,m) =` Σ`(p,[x = n],[],G)`. Une fonction partielle `f` des entiers dans les entiers est dite *programmable* en Java s'il existe une expression `t` ou une instruction `p` telle que `f` soit la fonction ainsi associée à `t` ou à `p`.

On peut démontrer que l'ensemble des fonctions programmables en Java, dans le noyau impératif de Java ou dans son noyau fonctionnel est le même : c'est l'ensemble des fonctions calculables — voir l'exercice 1.12.

Bien entendu, ce résultat n'est vrai que parce que les définitions de fonctions peuvent être récursives. La boucle et la récursivité sont donc deux moyens essentiellement redondants de construire des programmes infinis, et chaque fois que l'on a besoin de construire un programme infini, on peut choisir d'utiliser ou bien une boucle ou bien une définition récursive.

Exercice 3.4

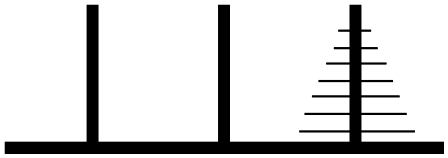
Écrire la définition de la fonction Σ pour le noyau fonctionnel de Java.

Exercice 3.5 (Les tours de Hanoï)

Les tours de Hanoï est un jeu inventé par Édouard Lucas en 1883. Il est formé de sept disques de tailles différentes répartis en trois colonnes. Au départ, tous les disques sont sur la colonne de gauche en taille croissante.

Les seuls mouvements possibles sont les déplacements d'un disque situé au sommet d'une colonne vers le sommet d'une autre colonne, à condition que la colonne d'arrivée soit vide ou que le disque déplacé soit plus petit que son sommet. On note `n -> n'` le déplacement d'un disque de la colonne `n` vers la

colonne `n'`. Le but du jeu est de déplacer tous les disques vers la colonne de droite.

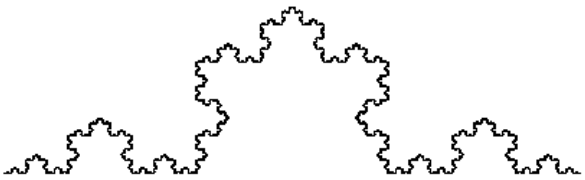


Écrire un programme qui affiche une solution du jeu sous la forme d'une suite de mouvements.

Indication : dans une variante, le jeu n'est formé que de six disques. Si l'on sait résoudre le jeu à six disques, comment résoudre le jeu à sept disques ?

Exercice 3.6 (Le flocon de Von Koch)

Le flocon, courbe définie par Helge Von Koch en 1906, est un exemple de courbe continue partout et dérivable nulle part. C'est aussi un exemple d'ensemble fractal, c'est-à-dire dont la dimension de Haussdorf n'est pas un nombre entier.

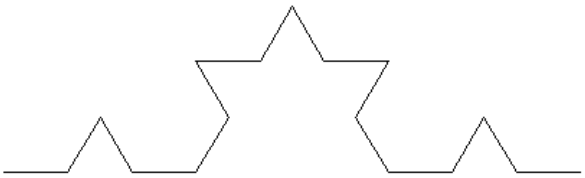


Cette courbe se définit comme la limite de la suite de courbes dont le premier élément est un segment

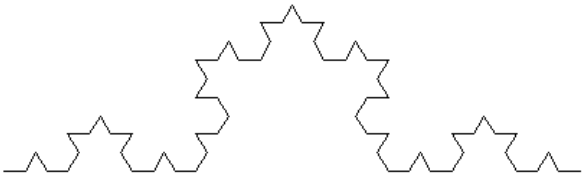
---

et dans laquelle on passe d'un élément au suivant en divisant chaque segment en trois et en remplaçant le morceau du milieu par deux segments formant un triangle équilatéral avec le segment supprimé. Le deuxième élément est donc

le troisième



et le quatrième



Écrire un programme qui dessine le $\mathtt{n}^{\text{ème}}$ élément de cette suite.