# Langages et programmation

*Notre parcours se poursuit avec la notion de langage de programmation. Apprendre un premier langage de programmation est une étape importante dans la découverte de l'informatique : c'est le moment où l'on devient autonome, en étant capable de construire soi-même quelque chose. Apprendre un langage de programmation demande de comprendre les constructions de ce langage, et, surtout, de savoir les utiliser. Et, en programmation, c'est beaucoup en forgeant que l'on devient forgeron. Ce chapitre propose donc une importante série d'exercices de programmation, qui permettent de mettre en pratique les notions apprises. Le langage choisi comme exemple ici est Java, mais les notions introduites sont universelles. Elles se transposent sans trop de difficultés à presque tous les langages. À la fin du chapitre, nous étendons la problématique des langages de programmation aux langages formels en général, en contrastant le langage Java avec le langage XHTML, qui est un langage formel, mais pas un langage de programmation.*

## Cours

### Le noyau impératif

#### Affectation, déclaration, séquence, test et boucle

La plupart des langages de programmation comportent, parmi d'autres, cinq constructions : l'affectation, la déclaration de variable, la séquence, le test et la boucle. Ces constructions forment le *noyau impératif* de ces langages.

L'*affectation* est une construction qui permet de former une *instruction* avec une variable `x` et une expression `t`. En Java, cette instruction s'écrit `x = t;`, par exemple, `x = y + 3;`. Les *variables* sont des identificateurs, c'est-à-dire des mots formés de une ou plusieurs lettres. Les *expressions* sont formées à

partir des variables et des constantes avec des opérations, comme, par exemple, `+`, `-`, `*`, `/` et `%`, ces deux dernières opérations étant respectivement les quotient et reste de la division euclidienne.

Pour expliquer ce qui se passe quand on exécute l'instruction `x = t;` on doit supposer qu'il y a, dans un recoin de la mémoire d'un ordinateur, une case appelée `x`. Exécuter l'instruction `x = t;` consiste alors à remplir cette case avec la *valeur* de l'expression `t`. La valeur contenue antérieurement dans la case `x` est effacée. Si l'expression `t` est une constante, par exemple `3`, sa valeur est cette même constante. Si c'est une expression sans variables, comme `3 + 4`, sa valeur s'obtient en effectuant quelques opérations arithmétiques, ici une addition. Si l'expression `t` contient des variables, alors il faut aller chercher les valeurs correspondant à ces variables dans les cases de la mémoire de l'ordinateur. L'ensemble des contenus des cases de la mémoire de l'ordinateur s'appelle un *état*.

Il est important de distinguer les expressions, comme `x + 2`, des instructions, comme `y = x + 2;`. Une expression *se calcule*, une instruction *s'exécute*.

Dans ces exemples, les valeurs des expressions sont des nombres entiers relatifs. En fait, dans les langages de programmation, la valeur d'une expression ne peut pas être un entier arbitraire et une telle valeur appartient toujours à un intervalle fini, en Java à l'intervalle $[-2^{31}, 2^{31} - 1]$ – voir le premier chapitre.

Avant de pouvoir affecter une variable `x`, il faut la déclarer, c'est-à-dire associer le nom `x` à une case de la mémoire de l'ordinateur. La *déclaration de variable* est une construction qui permet de former une instruction à partir d'un type, d'une variable, d'une expression et d'une instruction. En Java, cette instruction s'écrit `{int x = t; p}` où `p` est une instruction. Par exemple, `{int x = 4; x = x + 1;}` déclare une variable `x` de type `int` – entier –, de valeur initiale `4`, utilisable dans l'instruction `x = x + 1;`. Plus généralement, la variable `x` peut ensuite être utilisée dans l'instruction `p`, qui s'appelle la *portée* de la variable `x`.

Outre le type `int`, il y a, en Java, trois autres types d'entiers : `byte`, `short` et `long` qui correspondent aux intervalles $[-2^7, 2^7 - 1]$, $[-2^{15}, 2^{15} - 1]$ et $[-2^{63}, 2^{63} - 1]$. Il y a aussi, en Java, d'autres *types de données scalaires* : deux types, `float` et `double`, pour les nombres à virgule, qui s'écrivent en notation « scientifique », par exemple `3.14159`, `666` ou `6.02E23`, un type `boolean` pour les booléens qui s'écrivent `false` et `true` et un type `char` pour les caractères qui s'écrivent entre apostrophes, par exemple `'b'`. Enfin, il y a des *types composites*, comme les tableaux et les chaînes de caractères, sur lesquels nous reviendrons ci-après. Chaque type est muni d'opérations qui servent à construire des expressions de ce type.

Pour déclarer une variable d'un type `T`, on doit remplacer le type `int` par `T`. La forme générale d'une déclaration est donc `{T x = t; p}`. Nous avons choisi dans ce chapitre d'initialiser systématiquement les variables au moment de leur déclaration.

La *séquence* est une construction qui permet de former une instruction à partir de deux instructions $p_1$ et $p_2$. En Java, cette instruction s'écrit $\{p_1\ p_2\}$, par exemple `{x = 1; y = 2;}`. Dans cette instruction, le premier point-virgule appartient à la première affectation `x = 1;` et non à la séquence elle-même. Par convention, l'instruction $\{p_1\ \{p_2\ \{\ \ldots\ p_n\}\ \ldots\}\}$ s'écrit aussi $\{p_1\ p_2\ \ldots\ p_n\}$. Pour exécuter l'instruction $\{p_1\ p_2\}$ dans un état `s`, on exécute d'abord $p_1$ dans l'état `s`, ce qui produit un état `s'`, puis on exécute $p_2$ dans l'état `s'`.

Le *test* est une construction qui permet de former une instruction à partir d'une expression booléenne `b` et de deux instructions $p_1$ et $p_2$. En Java, cette instruction s'écrit `if(b)`$p_1$ `else`$p_2$, par exemple `if(x == 1)y = 2; else y = 5;`. Pour exécuter l'instruction `if (b)`$p_1$ `else`$p_2$ dans un état `s`, on commence par calculer la valeur de l'expression `b` dans l'état `s`, puis, selon que cette valeur est `true` ou `false`, on exécute $p_1$ ou $p_2$ dans l'état `s`.

La *boucle* est une construction qui permet de former une instruction à partir d'une expression booléenne `b` et d'une instruction `p`. En Java, cette instruction s'écrit `while (b) p`, par exemple `while (x <= 1000) x = x * 2;`. Pour exécuter l'instruction `while (b) p` dans un état `s`, on commence par calculer la valeur de `b` dans l'état `s`. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état… Et ce processus se poursuit jusqu'à ce que la valeur de `b` soit `false`.

Avec cette construction apparaît une possibilité de *non-terminaison*. En effet, si l'expression booléenne `b` vaut éternellement `true`, alors l'instruction `p` s'exécute à l'infini et l'exécution de l'instruction `while (b) p` ne termine pas. C'est par exemple le cas de l'instruction

```
int x = 1;
while (x >= 0) {x = 3;}
```

En introduisant une instruction fictive `skip;` dont l'exécution ne fait rigoureusement rien, on peut définir l'instruction `while (b) p` comme une abréviation pour l'instruction infinie

```
if (b) {p if (b) {p if (b) {p if (b) …
                    else skip;}
               else skip;}
```

```
        else skip;}
    else skip;
```

La boucle est donc l'une des multiples notations qui permettent d'exprimer un objet infini avec une expression finie. Et le fait qu'une boucle puisse ne pas terminer est une conséquence du fait que c'est un objet infini.

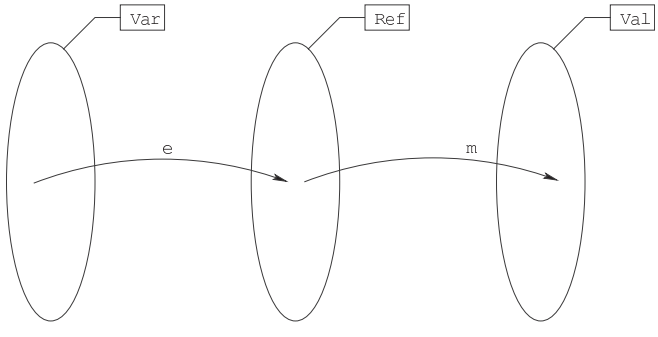
La boucle `for` est juste une notation plus pratique pour un cas particulier de la boucle `while`. L'instruction `for (`$p_1$`; b; `$p_2$`)` $p_3$ est une abréviation pour l'instruction $p_1$`; while (b) {`$p_3$ $p_2$`;}`. Par exemple, l'instruction
`for (i = 1; i <= 10; i = i + 1) s = s + i;`
ajoute successivement à `s` les entiers de `1` à `10`.

### La sémantique du noyau impératif

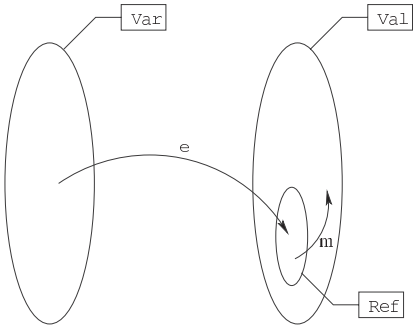
On peut, comme nous l'avons fait ci-avant, expliquer par une phrase en français la *sémantique* de chaque instruction, c'est-à-dire ce qui se passe quand on exécute cette instruction. Toutefois, ces explications deviendront vite compliquées et imprécises, quand nous introduirons d'autres constructions des langages de programmation, en particulier les fonctions. Nous allons donc définir plus précisément la sémantique de ces instructions, et tout d'abord une notion d'état, puis la manière dont l'exécution d'une instruction transforme cet état.

Pour définir la notion d'état, on pose un ensemble infini `Var` dont les éléments sont appelés *variables*. Et on définit un ensemble `Val` de *valeurs* qui contient les nombres entiers, les booléens, etc. Un *état* est une fonction d'une partie finie de l'ensemble `Var` dans l'ensemble `Val`.

Il sera utile, dans la suite de ce chapitre, de considérer cette fonction comme la composée de deux fonctions de domaine fini : la première, l'*environnement,* d'une partie finie de l'ensemble `Var` dans un ensemble intermédiaire `Ref`, dont les éléments sont appelés *références*, et la seconde, la *mémoire*, d'une partie finie de l'ensemble `Ref` dans l'ensemble `Val`.



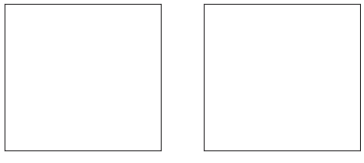
Il sera également utile de définir l'ensemble `Ref` comme un sous-ensemble de `Val`.



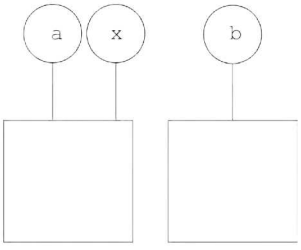

Cela mène à poser deux ensembles infinis `Var` et `Ref` puis à définir un ensemble `Val` qui contient les nombres entiers, les booléens, etc. et les éléments de l'ensemble `Ref`. L'ensemble des *environnements* se définit alors comme l'ensemble des fonctions d'une partie finie de l'ensemble `Var` dans l'ensemble `Ref` et l'ensemble des *mémoires* comme l'ensemble des fonctions d'une partie finie de l'ensemble `Ref` dans l'ensemble `Val`. On écrit $[x = r_1, y = r_2]$ l'environnement qui associe la référence $r_1$ à la variable `x` et la référence $r_2$ à la variable `y`. De même, on écrit $[r_1 = 5, r_2 = 7]$ la mémoire qui associe la valeur `5` à la référence $r_1$ et la valeur `7` à la référence $r_2$.

On définit une fonction de mise à jour d'un environnement notée `+` telle que l'environnement `e + [x = r]` soit la fonction coïncidant partout avec `e` sauf en `x` où elle vaut `r`. De même, on définit une fonction de mise à jour d'une mémoire telle que la mémoire `m + [r = V]` soit la fonction coïncidant partout avec `m` sauf en `r` où elle vaut `V`.

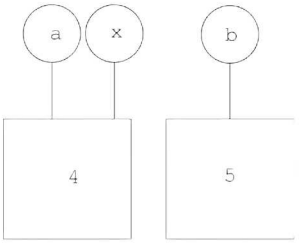
On est souvent amené à représenter les états graphiquement. Chaque référence est représentée par une case dessinée sur le plan. Deux cases situées à des endroits différents du plan représentent toujours des références différentes.



Puis, on représente l'environnement en ajoutant à certaines cases une ou plusieurs étiquettes avec un nom.




Si chaque étiquette est associée à une référence unique, rien n'empêche, en principe, deux étiquettes d'être associées à la même référence. Enfin, on représente la mémoire en remplissant chaque case avec une valeur.




À nouveau, rien n'empêche deux cases de contenir la même valeur.

Pour définir ce qui se passe quand on calcule une expression, on définit alors la fonction Θ qui associe une valeur à chaque triplet formé d'une expression, d'un environnement et d'une mémoire. Par exemple, `Θ(x + 3,[x = r1, y = r2],[r1 = 5,r2 = 6])= 8.`

– `Θ(x,e,m) = m(e(x))`, par exemple si `e = [x = r]` et `m = [r = 5]`, alors

`e(x) = r` et `m(e(x)) = 5`,

– `Θ(c,e,m) = c`, si `c` est une constante, comme `4`, `true`, etc.

– `Θ(t + u,e,m) = Θ(t,e,m) + Θ(u,e,m)`,

– et de même pour les autres opérations.

Enfin, pour définir ce qui se passe quand on exécute une instruction, on définit la fonction Σ qui associe une mémoire à chaque triplet formé d'une instruction, d'un environnement et d'une mémoire. Par exemple, `Σ(x = x + 1;,[x = r],[r = 5]) = [r = 6]`. Cette fonction est partielle et elle n'est pas définie sur le triplet `(p, e, m)` si l'instruction `p` ne termine pas quand on l'exécute dans l'environnement `e` et la mémoire `m`, ou si cette exécution produit une erreur.

– Quand l'instruction `p` est une déclaration de variable de la forme `{T x = t; q}`, la fonction Σ se définit ainsi

Σ({T x = t; q},e,m) = Σ(q,e + [x = r],m + [r = Θ(t,e,m)])
où r est une référence quelconque qui n'apparaît pas dans e et m.
– Quand l'instruction p est une affectation de la forme x = t;, cette fonction se définit ainsi

Σ(x = t;,e,m) = m + [e(x) = Θ(t,e,m)]

ce qui traduit la phrase « Exécuter l'instruction x = t; consiste à remplir la case x avec la valeur de l'expression t ».
– Quand l'instruction p est une séquence de la forme {p1 p2}, la fonction Σ se définit ainsi

Σ({p1 p2 },e,m) = Σ(p2 ,e,Σ(p1 ,e,m))

– Quand l'instruction p est un test de la forme if (b) p1 else p2, la fonction Σ se définit ainsi. Si Θ(b,e,m) = true alors

Σ(if(b) $p_1$ else $p_2$,e,m) = Σ($p_1$,e,m)

Si Θ(b,e,m) = false alors

Σ(if(b) $p_1$ else $p_2$,e,m)= Σ($p_2$,e,m)

– Venons-en maintenant au cas où l'instruction p est une boucle de la forme while (b) q. Nous avons vu qu'en introduisant une instruction fictive skip; telle que Σ(skip;,e,m) = m, on peut définir l'instruction while (b) q comme une abréviation pour une instruction infinie. Pour traiter de tels objets infinis, on cherche souvent à les approcher par des objets finis. Ainsi, en introduisant une instruction fictive giveup; telle que la fonction Σ ne soit jamais définie en (giveup;,e,m), on peut définir une suite d'approximations finies de l'instruction while (b) q.

$p_0$ = if (b) giveup; else skip;
$p_1$ = if (b) {q if (b) giveup; else skip;} else skip;
…
$p_{n+1}$ = if (b) {q $p_n$} else skip;

L'instruction $p_n$ tente d'exécuter l'instruction while (b) q en faisant au maximum n tours de boucle. Si, après n tours, elle n'a pas terminé, alors elle abandonne. Si la fonction Σ n'est définie en aucun triplet ($p_n$,e,m), cela signifie que, quel que soit le nombre de tours de boucle que l'on s'accorde, le programme ne termine pas. Dans ce cas, on pose que Σ n'est pas définie en (while (b) q,e,m). Sinon, soit n le plus petit entier tel que Σ soit définie en ($p_n$,e,m), on pose Σ(while (b) q,e,m) = Σ($p_n$,e,m).

## Les constructions d'entrée/sortie

Les constructions d'entrée d'un langage permettent de lire des données sur un périphérique, par exemple un clavier, et les constructions de sortie permettent d'afficher des données sur un périphérique, par exemple un écran.

Les constructions d'entrée en Java sont assez complexes, nous utiliserons donc une extension de Java, la classe `Isn` – le ficher `Isn.java` est disponible sur le site de ressources pédagogiques SILO, il suffit de le placer dans le répertoire courant pour pouvoir utiliser les instructions décrites ici. L'évaluation de l'expression `Isn.readInt()` attend que l'utilisateur tape un nombre entier au clavier et produise ce nombre comme valeur. Une utilisation typique est `n = Isn.readInt();`. De manière similaire, la construction `Isn.readDouble` permet de lire un nombre à virgule et la construction `Isn.readChar` un caractère.

L'exécution de l'instruction `System.out.print(t);` affiche à l'écran la valeur de l'expression `t`. L'exécution de l'instruction `System.out.println();` passe à la ligne. L'exécution de l'instruction `System.out.println(t);` affiche à l'écran la valeur de l'expression `t`, puis passe à la ligne.

## La notion de fonction

### Isoler une instruction

La manière la plus simple d'introduire la notion de fonction est de voir cette construction comme un moyen d'éviter les redondances dans un programme. Si une même instruction revient plusieurs fois dans un programme, on peut l'isoler dans une *fonction* et la remplacer dans le programme par un *appel* à cette fonction. L'instruction que l'on isole dans une fonction s'appelle le *corps* de cette fonction. Par exemple, au lieu de recopier plusieurs fois dans un programme l'instruction

```
System.out.println();
System.out.println();
System.out.println();
```

qui permet de sauter trois lignes, on définit une fonction `sauterTroisLignes`

```
static void sauterTroisLignes () {
 System.out.println();
 System.out.println();
 System.out.println();}
```

puis on appelle cette fonction depuis le programme principal

```
sauterTroisLignes();
```

autant de fois que nécessaire.

Organiser un programme en fonctions permet d'éviter les redondances. En outre, cela rend les programmes plus clairs et plus faciles à lire : pour comprendre un programme qui utilise la fonction `sauterTroisLignes`, il n'est pas nécessaire de savoir comment cette fonction est programmée, il suffit de savoir ce qu'elle fait. Enfin, cela permet d'organiser l'écriture du programme. On peut décider d'écrire la fonction `sauterTroisLignes` un jour et le programme principal le lendemain ou organiser une équipe de manière qu'un programmeur écrive la fonction `sauterTroisLignes` et un autre le programme principal.

Dans certains langages de programmation comme les langages d'assemblage ou Basic, la notion de fonction se réduit à ce procédé sommaire d'abréviations.

### Le passage d'arguments

Dans de nombreux cas, l'instruction que l'on cherche à isoler présente quelques variations d'une occurrence à l'autre et on souhaite donner un paramètre, un *argument*, à la fonction. Par exemple, écrire une fonction `sauterDesLignes` qui prend en argument un nombre entier `n` et saute `n` lignes

```
static void sauterDesLignes (int n) {
int i = 0;
for (i = 0; i < n; i = i + 1)
 System.out.println();}
```

puis l'utiliser dans le programme principal sous la forme `sauterDesLignes(3);` ou `sauterDesLignes(10);`. La variable `n` qui figure comme argument dans la définition de la fonction s'appelle un *argument formel* de la fonction. Quand on appelle une fonction `sauterDesLignes(3);` l'expression `3` que l'on donne en argument s'appelle un *argument réel* de l'appel.

## La notion de valeur

Dans de nombreux cas également, on veut que la fonction puisse non seulement recevoir de l'information depuis le programme principal, mais également en transmettre en retour en direction de ce programme. Par exemple, on veut pouvoir écrire une fonction `hypotenuse` qui prend en argument les longueurs des deux petits côtés d'un triangle rectangle, calcule la longueur de l'hypoténuse de ce rectangle et la renvoie au programme principal

```
static double hypotenuse (double x, double y) {
 return Math.sqrt(x * x + y * y);}
```

puis l'utiliser dans le programme principal

```
u = hypotenuse(3,4);
v = hypotenuse(5,12);
```

Une fonction peut d'une part effectuer une action, par exemple afficher quelque chose ou modifier la mémoire, et d'autre part renvoyer une valeur.

En Java, la déclaration d'une fonction est formée du mot-clé `static` suivi du type de la valeur retournée, du nom de la fonction, de la liste des arguments formels, chacun précédé de son type, et du corps de la fonction. Quand la fonction ne renvoie pas de valeur, le type de retour est remplacé par le mot-clé `void`.

Une fonction qui ne renvoie pas de valeur s'appelle une *procédure*. Dans certains langages, comme Pascal ou Ada, les procédures sont distinguées et déclarées par un mot-clé spécial. À l'inverse, dans d'autres langages, comme Caml, une procédure est simplement une fonction qui renvoie une valeur de type `unit`. Comme son nom l'indique, `unit` est un type singleton qui ne contient qu'une valeur, écrite `()`. Une procédure renvoie donc invariablement la valeur `()`, ce qui ne transmet aucune information. Le cas de Java est intermédiaire, puisque l'on y déclare une procédure en remplaçant le type de la valeur retournée par le mot-clé `void`. Contrairement au `unit` de Caml, le `void` de Java n'est pas un type, mais seulement un mot-clé qui indique l'absence de valeur renvoyée : il n'est, par exemple, pas possible de déclarer une variable de type `void`.

L'appel d'une fonction, comme `hypotenuse(3,4)`, est une expression et celui d'une procédure, comme `sauterDesLignes(10);`, est une instruction.

### Les variables globales

Imaginons que nous voulions isoler l'instruction `x = 0;` dans le programme

```
int x = 0;
x = 3;
x = 0;
```

Cela nous amènerait à écrire la fonction

```
static void reset () {x = 0;}
```

puis le programme principal

```
int x = 0;
x = 3;
reset();
```

Mais ce programme est incorrect, car l'instruction `x = 0;` qui apparaît dans le corps de la fonction `reset` n'est plus dans la portée de la variable `x`. Pour que la fonction `reset` puisse avoir accès à la variable `x`, il faut déclarer la variable `x` comme une variable *globale*, c'est-à-dire commune à toutes les fonctions du programme et au programme principal

```
static int x = 0;

static void reset () {x = 0;}
```

puis le programme principal

```
x = 3;
reset();
```

En Java, toutes les fonctions peuvent utiliser toutes les variables globales, que celles-ci soient déclarées avant ou après la fonction.

Un *programme* est donc constitué de la déclaration de variables globales $x_1$, …, $x_n$, de la définition de fonctions $f_1$, …, $f_n$, puis du *programme principal* `p`. Il a été choisi, en Java, de faire du programme principal une fonction qui porte un nom spécial : `main`. La fonction `main` ne doit pas renvoyer de valeur et doit toujours avoir un argument de type `String []`. Outre le mot-clé `static`, la définition de la fonction `main` doit aussi être précédée du mot-clé `public`.

Par ailleurs, il faut donner un nom au programme et introduire ce nom par le mot-clé `class`. La forme générale d'un programme est donc

```
class Prog {
static T1 x1 = t1;
…
static Tn xn = tn;
static … f1 (…) …
…
```

```
static … f_n'   (…) …

public static void main (String [] args) {p}}
```

Par exemple

```
class Hypotenuse {

static double hypotenuse (double x, double y) {
return Math.sqrt(x * x + y * y);}

public static void main (String [] args) {
System.out.println(hypotenuse(3,4));}}
```

### La sémantique des fonctions

Étendre la sémantique du noyau impératif que nous avons définie précédemment demande de prendre en compte plusieurs nouveautés.

Tout d'abord, les fonctions Θ et Σ doivent prendre en arguments, outre une instruction, un environnement et une mémoire, un *environnement global* `G`. Cet environnement global est constitué, d'une part, d'un environnement `e` qui contient la déclaration des variables globales et, d'autre part, des définitions de fonctions. Ensuite, nous devons prendre en compte le fait que, puisqu'une expression peut désormais être un appel de fonction et que l'exécution du corps de cette fonction peut modifier la mémoire, l'évaluation d'une expression peut modifier la mémoire. De ce fait, le résultat Θ`(t,e,m,G)` de l'évaluation d'une expression `t` ne sera plus simplement une valeur, mais un couple formé d'une valeur et d'une mémoire. Il faut donc modifier les définitions ci-avant, afin de prendre en compte ces nouveautés.

Ensuite, il faut ajouter dans la définition des fonctions Θ et Σ le cas des instructions formées d'un appel de fonction.

Pour définir Σ`(f(t_1, …, t_n);,e,m,G)`, on doit, dans un premier temps, évaluer les arguments réels de la fonction, ce qui produit des valeurs `V_1, … V_n` et une nouvelle mémoire `m_n` `((V_1, m_1) = Θ(t_1,e,m,G), (V_2, m_2) = Θ(t_2,e,m_1,G), …, (V_n, m_n) = Θ(t_n,e,m_n-1,G))`. Puis, on étend l'environnement `e'` des variables globales, qui fait partie de l'environnement global `G`, en ajoutant des liaisons entre les arguments formels `x_1, …, x_n` de la fonction et de nouvelles références `r_1, …, r_n`, ce qui donne l'environnement `e" = e' + [x_1 = r_1] + … + [x_n = r_n]`. De même, on étend la mémoire `m_n` en ajoutant des liaisons entre les références `r_1, …, r_n` et les valeurs `V_1, …, V_n`, ce qui donne la mémoire `m" = m_n + [r_1 = V_1] + … + [r_n = V_n]`. On

exécute alors le corps de la fonction dans cet environnement et cette mémoire étendus et le résultat $\Sigma(p,e'',m'',G)$ est une mémoire $m'''$. On pose $\Sigma(f(t_1, \ldots, t_n),e,m,G) = m'''$. Le point essentiel ici est que l'environnement dans lequel on exécute le corps de la fonction contient les déclarations des variables globales et des arguments formels de la fonction, mais pas les variables déclarées dans le programme principal.

Dans la définition des fonctions $\Sigma$ et $\Theta$ que nous avons données ci-avant, les arguments d'une instruction ou expression de la forme $f(t_1, \ldots, t_n)$ sont évalués de la gauche vers la droite : $t_1$ est évalué dans la mémoire $m$, puis $t_2$ est évalué dans la mémoire $m_1$ produite par l'évaluation de $t_1$…

Une alternative aurait été d'évaluer les arguments de la droite vers la gauche : évaluer $t_n$ dans la mémoire $m$, puis $t_{n-1}$ dans la mémoire $m_{n-1}$ produite par l'évaluation de $t_n$… Et ces deux manières de faire ne sont pas équivalentes.

Par exemple, si la fonction `f` renvoie son argument après avoir augmenté de `1` la valeur d'une variable globale `n`

```
static int f (int x) {
n = n + 1;
return x;}
```

la fonction `g` renvoie son argument après avoir multiplié par `2` la valeur de la variable `n`

```
static int g (int x) {
n = 2 * n;
return x;}
```

et la fonction `h` renvoie la somme de ses deux arguments et de la variable `n`, alors l'évaluation de l'expression `h(f(4),g(5))` dans un état où la valeur de la variable `n` est `0`, donne la valeur `11` dans le premier cas et `10` dans le second. En effet, dans le premier cas, la valeur de `n` est d'abord augmentée de `1` avant d'être multipliée par `2`, ce qui donne `2`, alors que, dans le second, elle est d'abord multipliée par `2` avant d'être augmentée de `1`, ce qui donne `1`.

### Le passage d'arguments par valeur et par référence

Si le contenu initial de la variable `x` est `4` et celui de la variable `y` est `7`, après avoir exécuté l'instruction `z = x; x = y; y = z;`, la variable `x` contient la valeur `7` et la variable `y` la valeur `4`. On peut vouloir isoler cette instruction, qui intervertit le contenu des variables `x` et `y`, dans une fonction.

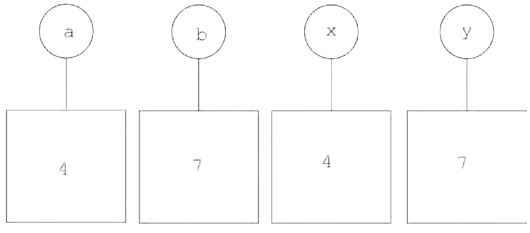
```
class Prog {

static int a = 0;
static int b = 0;
static void echange (int x, int y) {
int z = x; x = y; y = z;}

static public void main (String [] args) {
a = 4;
b = 7;
echange(a,b);
System.out.println(a);
System.out.println(b);}}
```

De manière surprenante, alors que l'on s'attendrait naïvement à ce que les contenus de a et b aient été intervertis et que les nombres 7 puis 4 s'affichent, c'est le nombre 4 qui s'affiche en premier, suivi du nombre 7.

En fait, ce résultat est tout à fait cohérent avec la sémantique définie ci-avant. On part avec un environnement e = [a = $r_1$, b = $r_2$] et une mémoire m = [$r_1$ = 4, $r_2$ = 7]. L'appel de la fonction echange(a,b); demande de calculer les valeurs des expressions a et b dans l'environnement e et la mémoire m. On obtient 4 et 7 respectivement. Puis on construit l'environnement [a = $r_1$, b = $r_2$, x = $r_3$, y = $r_4$] et la mémoire [$r_1$ = 4, $r_2$ = 7, $r_3$ = 4, $r_4$ = 7]



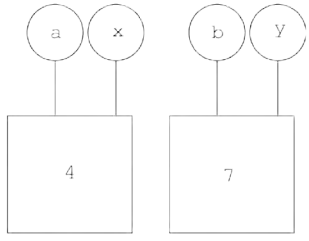

On intervertit ensuite le contenu des variables x et y, en utilisant une variable auxiliaire, ce qui donne la mémoire [$r_1$ = 4, $r_2$ = 7, $r_3$ = 7, $r_4$ = 4, $r_5$ = 4] et on revient au programme principal.

L'environnement est alors e = [a = $r_1$, b = $r_2$] et la mémoire [$r_1$ = 4, $r_2$ = 7,$r_3$ = 7,$r_4$ = 4,$r_5$ = 4]. Le contenu des variables a et b n'a pas changé.

Autrement dit, la fonction `echange` ignore tout des variables `a` et `b`, elle ne peut utiliser que leur valeur au moment de l'appel et, en aucun cas, ne peut modifier leur contenu : exécuter l'instruction `echange(a,b);` donne le même résultat qu'exécuter l'instruction `echange(4,7);`, c'est-à-dire cela ne fait rien.

Le mécanisme de passage des arguments, tel que nous l'avons décrit, et qui s'appelle le *passage par valeur*, ne permet donc pas d'écrire une fonction `echange` qui intervertit le contenu de deux variables.

Afin que l'interversion des contenus des variables `x` et `y` intervertisse celui des variables `a` et `b`, on souhaiterait plutôt que le corps de la fonction s'exécute dans l'environnement $[a = r_1, b = r_2, x = r_1, y = r_2]$ et la mémoire $[r_1 = 4, r_2 = 7]$.




Distinguer ces deux états est la principale motivation de la décomposition de l'état en un environnement et une mémoire en introduisant un ensemble intermédiaire de références, comme nous l'avons fait ci-avant.

La plupart des langages de programmation comportent une construction qui permet d'écrire une telle fonction. Mais cette construction est un peu différente dans chaque langage. Certains langages, comme Pascal, comportent un mécanisme primitif appelé le passage d'arguments *par référence*, ou *par variable*. Ainsi, dans la définition de la procédure `echange`, on peut faire précéder chaque argument du mot-clé `var`. Quand un argument d'une fonction est ainsi déclaré en passage par référence, on ne peut appliquer cette fonction qu'à une variable. Ainsi, on peut écrire `echange(a,b)` mais pas `echange(4,7)`, ni `echange(2 * a,b)`. D'autres langages, comme Caml et C, utilisent le fait que les références sont des valeurs, c'est-à-dire que l'ensemble `Ref` est un sous-ensemble de l'ensemble `Val`, et on écrit une fonction `echange` qui prend en argument, non deux entiers, mais deux références. En C, la référence associée à une variable `x` dans l'environnement s'écrit `&x` ($\Theta$`(&x,e,m,G) = e(x)`). En Caml, la référence associée à une variable `x` dans l'environnement s'écrit `x,` – en Caml $\Theta$`(x,e,m,G) = e(x)` et non `m(e(x))` – alors que la valeur associée à cette référence dans la mémoire s'écrit `!x` – $\Theta$`(!t,e,m,G) =`

m(Θ(t,e,m,G)). Le mécanisme utilisé en Java est encore différent, il consiste à utiliser des *types enveloppés*. Nous le présenterons au paragraphe « Les enregistrements », car il utilise des constructions du langage que nous n'avons pas encore introduites.

### La récursivité

Précédemment, pour définir la fonction Σ sur une instruction de la forme $f(t_1, \ldots, t_n);$, 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 $f_1, \ldots, f_k$ telles que le corps de la fonction $f_i$ ne contienne que des appels à des fonctions antérieurement définies, c'est-à-dire à des fonctions $f_j$ pour $j < i$. 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. Remarquons qu'un tel ordre sur les fonctions existe toujours si ces fonctions sont introduites à partir du programme principal en isolant des instructions l'une après l'autre : on isole une instruction $p_k$ du programme principal, puis une instruction $p_{k-1}$ du programme principal ou de la fonction $p_k$ …

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, *in fine*, appellent la fonction initiale. On les appelle *définitions récursives*. Un exemple de définition récursive est celle de la fonction factorielle

```
static int fact (int x) {
if (x == 0) return 1;
else 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`.

Pour les définitions récursives, la définition de la fonction Σ ci-avant peut être circulaire. Par exemple, si `f` est une fonction définie ainsi

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

alors la définition de `Σ(f(x);,e,m,G)` utilise la valeur de `Σ(f(x);,e,m,G)`. Nous devons donc trouver une autre manière de définir cette fonction Σ.

On dit parfois qu'une définition récursive est celle 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 (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. Une autre tentative de comprendre les définitions récursives est d'y voir des définitions par récurrence. Mais, si cette idée marche dans le cas de la fonction factorielle, elle est insuffisante dans le cas général, car rien n'empêche une fonction récursive de s'appeler elle-même sur un argument plus grand que son argument.

Une idée plus intéressante est de transformer une définition récursive, par exemple celle de la fonction `fact`, 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 (int x) {
if (x == 0) return 1;
else return x * fact1(x - 1);}

static int fact (int x) {
if (x == 0) return 1;
else 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`, elles 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 `p` infini par des approximations finies $p_n$ obtenues en remplaçant la définition de la $n^{ème}$ copie de la fonction `fact` par la fonction `giveup` et en supprimant les suivantes qui ne sont plus utiles. Calculer la valeur de la $n^{ème}$ 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.

Si la fonction Σ n'est définie en aucun quadruplet $(p_n,e,m,G)$, cela signifie que, quel que soit le nombre d'appels récursifs imbriqués que l'on s'accorde, le programme ne termine pas. Dans ce cas, on pose que Σ n'est pas définie en `(p,e,m,G)`. Sinon, soit `n` le plus petit entier tel que Σ soit définie en $(p_n,e,m,G)$, on pose $\Sigma(p,e,m,G) = \Sigma(p_n,e,m,G)$.

Une autre manière de définir la sémantique des fonctions récursivement définies est de les voir comme des équations. La fonction `fact` est définie comme une solution de l'équation `fact = if (x == 0) return 1; else return x * fact(x - 1);`. Cette démarche aboutit exactement au même résultat, car pour démontrer l'existence d'une solution de cette équation, on procède à nouveau par approximations successives.

### Programmer sans affectation

En comparant la fonction factorielle écrite avec une boucle

```
static int fact (int x) {
int i = 0;
int r = 1;
for (i = 1; i <= x; i = i + 1) {r = r * i;}
return r;}
```

et récursivement

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

on constate que la première utilise des affectations : `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'affectation.

Plus généralement, on peut considérer un sous-langage de Java dans lequel on supprime l'affectation. Dans ce cas, la séquence et la boucle deviennent inutiles. Il reste un noyau de Java formé de la déclaration de variables, 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. Après son noyau impératif, nous voyons donc apparaître un nouveau sous-langage de Java. Et on peut, de même, définir le noyau fonctionnel de beaucoup de langages de programmation. De manière surprenante, ces deux sous-ensembles de Java ont la même puissance que Java tout entier. L'ensemble des fonctions que l'on peut programmer, en Java, dans le noyau impératif de Java ou dans son noyau fonctionnel est le même. 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.

## Les enregistrements

Dans les programmes que nous avons décrits ci-avant, chaque variable contient un nombre entier, un nombre à virgule, un booléen ou un caractère. Une telle variable ne peut pas contenir un objet formé de plusieurs nombres, booléens ou caractères, comme, par exemple, un nombre complexe formé de deux nombres à virgule, un vecteur formé de plusieurs coordonnées ou une chaîne formée de plusieurs caractères. Nous allons maintenant présenter une construction, les *enregistrements*, qui permet de construire des types *composites*, c'est-à-dire des types construits comme produits cartésiens d'autres types.

### Les n-uplets à champs nommés

Mathématiquement, un n-uplet est une fonction dont le domaine est un segment initial de $\mathbb{N}$. Dans les langages de programmation, les n-uplets sont la plupart du temps *à champs nommés*, c'est-à-dire que ce sont des fonctions dont le domaine est, non un segment initial de $\mathbb{N}$, mais un ensemble fini quelconque, dont les éléments sont appelés *étiquettes*. Un tel n-uplet à champs nommés s'appelle un *enregistrement*.

Par exemple, si l'on se donne un ensemble d'étiquettes `latitude`, `longitude` et `altitude`, on peut construire l'enregistrement `{latitude = 45.83, longitude = 6.86, altitude = 4810.0}`.

En Java, on définit un nouveau type d'enregistrements en indiquant l'étiquette et le type de chacun de ses champs. Par exemple, le type `Point` se définit ainsi

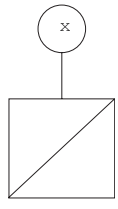
```
class Point {
double latitude;
double longitude;
double altitude;}
```

Une telle définition s'écrit avant l'introduction du nom du programme par le mot-clé `class`.

Une fois un tel type défini, on peut donner le type `Point` à une variable.

```
Point x = null;
```

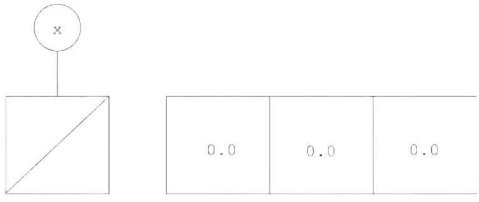
Comme pour toute déclaration de variable, cela ajoute un couple à l'environnement associant une référence `r` à cette variable et un couple à la mémoire associant une valeur à la référence `r`. Dans cet exemple, on déclare cette variable en lui donnant comme valeur initiale une valeur spéciale appelée `null`. On représente ainsi un état, dans lequel la variable `x` est associée dans l'environnement à une référence `r`, elle-même associée dans la mémoire à la valeur `null`.




En Java, la référence `r` n'est jamais associée directement à un enregistrement dans la mémoire. La case `r` est toujours une « petite » case qui ne peut contenir que `null` ou une autre référence. Pour associer un enregistrement à la variable `x`, il faut donc commencer par créer une case suffisamment grande pour contenir trois nombres à virgule. Cela se fait avec une nouvelle construction : `new`

```
new Point()
```

L'évaluation de l'expression `new Point()` crée une nouvelle référence `r'` et associe cette référence à un enregistrement, par défaut `{latitude = 0.0, longitude = 0.0, altitude = 0.0}`. La valeur de cette expression est la référence `r'`.

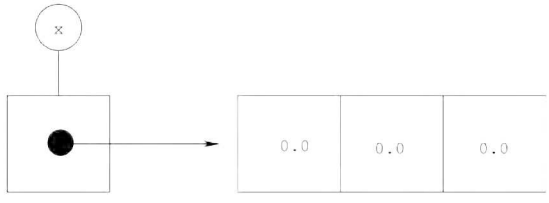



Une référence qui a été ajoutée à la mémoire par la construction `new` s'appelle une *cellule*. L'ensemble des cellules de la mémoire s'appelle le *tas*. L'opération consistant à ajouter une nouvelle cellule à la mémoire s'appelle l'*allocation* d'une cellule.

Quand on exécute l'instruction

```
x = new Point();
```

on associe la référence `r'` – à droite sur le dessin – à la référence `r` – à gauche sur le dessin – dans la mémoire. L'environnement est alors `[x = r]` et la mémoire `[r = r', r' = {latitude = 0.0, longitude = 0.0, altitude = 0.0}]`.




La valeur de l'expression `x` dans cet environnement et cette mémoire est alors `m(e(x))`, c'est-à-dire la référence `r'`.

Construire de telles mémoires dans lesquelles une référence `r'` est associée à une référence `r` est la principale motivation pour considérer les références comme des valeurs, c'est-à-dire pour prendre `Ref` comme un sous-ensemble de `Val`.

Il est, bien entendu, possible de déclarer la variable `x` en lui donnant une cellule comme valeur initiale

```
Point x = new Point();
```

Si la valeur de l'expression `t` est une référence `r'` associée dans la mémoire à un enregistrement et `l` est une étiquette, la valeur de l'expression `t.l` est le champ `l` de cet enregistrement. Ainsi, l'instruction

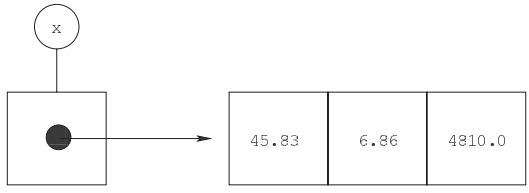
```
System.out.println(x.longitude);
```

affiche `0.0`. En particulier, quand `t` est une variable `x`, sa valeur est `m(e(x))` et donc la valeur de l'expression `x.latitude` est le champ `latitude` de l'enregistrement `m(m(e(x)))`.

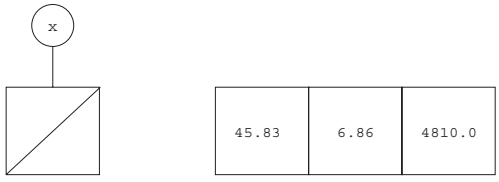
Pour affecter un champ d'un enregistrement, on utilise une nouvelle instruction `t.l = u;`, où `l` est une étiquette et `t` et `u` sont des expressions. Si la valeur de l'expression `t` est une référence `r` associée dans la mémoire à un enregistrement qui a un champ `l`, alors, quand on exécute l'instruction `t.l = u;`, le champ `l` de cet enregistrement reçoit la valeur de l'expression `u`. Ainsi, en exécutant les instructions

```
x.latitude = 45.83;
x.longitude = 6.86;
x.altitude = 4810.0;
```

on construit l'état




Enfin, quand une cellule n'est plus utilisée par un programme, elle peut être supprimée de la mémoire. Cette opération s'appelle la *désallocation* de la cellule. Ainsi, si on exécute l'instruction `x = null;` dans l'environnement `[x = r]` et la mémoire `[r = r', r' = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}]`, on obtient la mémoire `[r = null, r' = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}]`,



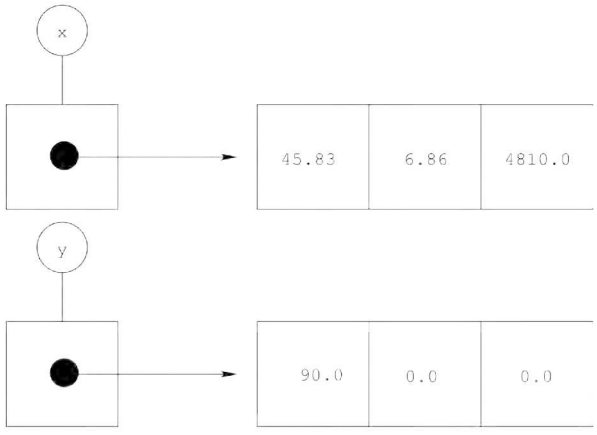

dans laquelle la cellule `r'` est devenue inutile. On peut la désallouer, afin d'obtenir la mémoire `[r = null]`. En Java, cette opération est automatique ; dans d'autres langages, c'est au programmeur de décider quelles cellules désallouer.

Le mécanisme des enregistrements dans un langage de programmation, comme Java, est donc constitué de cinq constructions qui permettent :
– de définir un type, `class` en Java,
– d'allouer une cellule, `new` en Java,
– d'accéder à un champ, `t.l` en Java,
– d'affecter un champ, `t.l = u;` en Java,
– de désallouer une cellule, automatique en Java.

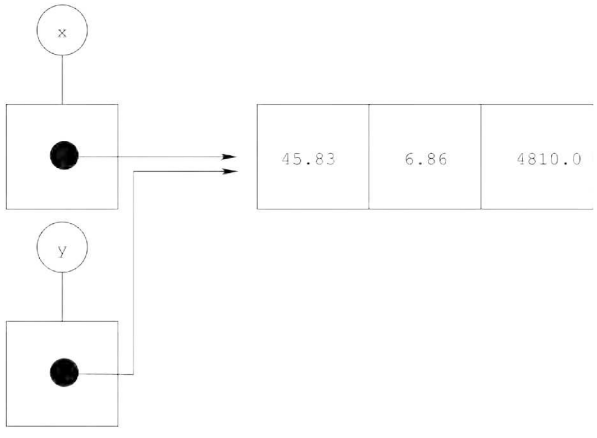
Comprendre le mécanisme des enregistrements dans un nouveau langage de programmation consiste à comprendre les constructions qui servent à définir un type, à allouer et désallouer une cellule, à accéder à un champ et à l'affecter.

### Le partage

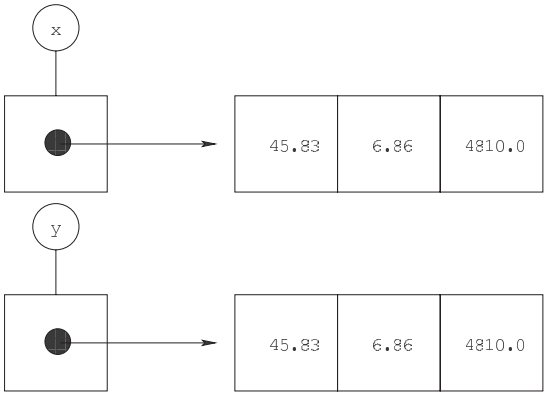
Supposons que `x` et `y` soient deux variables de type `Point`, associées à deux références $r_1$ et $r_2$ dans l'environnement. Supposons, en outre, que, dans la mémoire, $r_1$ soit associée à une référence $r_3$, elle-même associée à un enregistrement `{latitude = 45.83, longitude = 6.86, altitude = 4810.0}` et $r_2$ à une référence $r_4$, elle-même associée à l'enregistrement `{latitude = 90.0, longitude = 0.0, altitude = 0.0}`. Ainsi, `e = [x =` $r_1$`, y =` $r_2$`]`, `m = [`$r_1$ `=` $r_3$`,` $r_2$ `=` $r_4$`,` $r_3$ `= {latitude = 45.83, longitude = 6.86, altitude = 4810.0}, `$r_4$ `= {latitude = 90.0, longitude = 0.0, altitude = 0.0}]`.




Quand on exécute l'instruction `y = x;`, on calcule la valeur de `x`, qui est la référence $r_3$ et on associe cette valeur à la référence $r_2$. La mémoire devient [$r_1$ = $r_3$, $r_2$ = $r_3$, $r_3$ = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}].




En revanche, si l'on exécute les instructions `y.latitude = x.latitude; y.longitude = x.longitude; y.altitude = x.altitude;`, on obtient la mémoire [$r_1$ = $r_3$, $r_2$ = $r_4$, $r_3$ = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}, $r_4$ = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}].




Si l'on affecte ensuite le champ `latitude` de l'enregistrement `x` : `x.latitude = 23.45;` et que l'on affiche le champ `latitude` de `y`, on obtient `23.45` dans le premier cas, et `45.83` dans le second. On dit, dans le premier cas, que les variables `x` et `y` *partagent* la cellule $r_3$. Toute modification de la cellule associée à `x` entraîne automatiquement une modification de celle associée à `y` et réciproquement.

Si `a` et `b` sont deux expressions de type `Point`, leur valeur est une référence et l'expression `a == b` vaut `true` uniquement quand ces deux références sont identiques. C'est-à-dire quand `a` et `b` partagent une même cellule. On appelle cette relation l'égalité *physique*.

Il est cependant possible d'écrire une fonction qui teste l'égalité *structurelle* de deux enregistrements, c'est-à-dire l'égalité champ à champ

```
static boolean equal (Point x, Point y) {
return (x.latitude == y.latitude)
&& (x.longitude == y.longitude)
&& (x.altitude == y.altitude);}
```

### Les types enveloppés

Un type *enveloppé* est un type enregistrement à un seul champ. Un exemple est le type `Integer`

```
class Integer {
int c;}
```

À première vue, le type `Integer` peut sembler redondant avec le type `int`, puisque l'enregistrement, c'est-à-dire le mono-uplet, `{c = 4}` n'est pas très différent de l'entier `4`. Et il est vrai que le programme

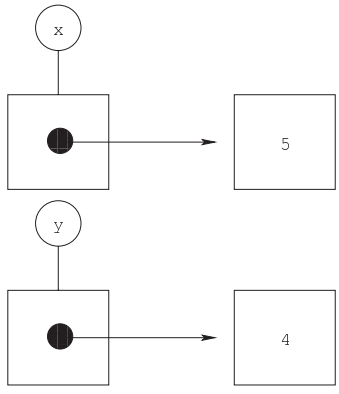
```
Integer x = new Integer();
x.c = 4;
Integer y = new Integer();
y.c = x.c;
x.c = 5;
System.out.println(y.c);
```

peut se réécrire en

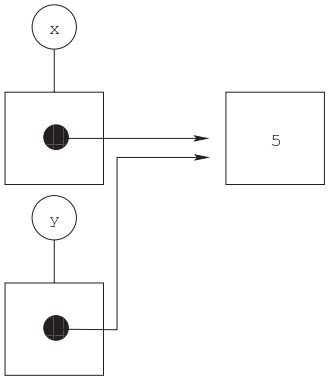
```
int x = 4;
int y = x;
x = 5;
System.out.println(y);
```

qui produit le même résultat : l'un comme l'autre affichent le nombre `4`.

Cependant, si on remplace l'instruction `Integer y = new Integer(); y.c = x.c;` par `Integer y = x;` au lieu d'obtenir l'état

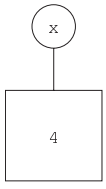



on obtient l'état

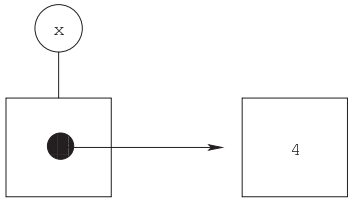



et le programme affiche `5`, et non `4`.

Plus généralement, au lieu d'avoir un état dans lequel une variable `x` est associée à une référence `r` associée à une valeur, par exemple, `4`




les types enveloppés permettent d'avoir un état dans lequel une variable `x` est associée à une référence `r`, associée à une référence `r'`, elle-même associée à une valeur `4`.




Cela permet, en particulier, à plusieurs variables de partager une valeur.

Comme nous l'avons vu précédemment, en Java, il n'est pas possible d'écrire une fonction qui intervertit le contenu de deux arguments de type `int`. En revanche, cela est possible pour des arguments de type `Integer`.

```
static void echange (Integer x, Integer y) {
int z = x.c;
x.c = y.c;
y.c = z;}
```

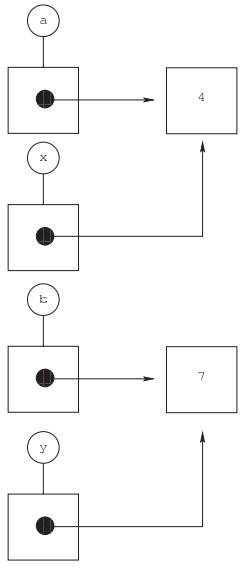
et le programme

```
public static void main (String [] args) {
a = new Integer();
a.c = 4;
b = new Integer();
b.c = 7;
echange(a,b);
System.out.println(a.c);}
System.out.println(b.c);}
```

affiche `7` et `4`.

Les types enveloppés permettent donc de réaliser le passage d'arguments par référence en Java.

En effet, quand on appelle la fonction `echange`, on construit l'état




et intervertir les contenus de `x` et `y` intervertit bien ceux de `a` et `b`.

## Les tableaux

Les langages de programmation permettent aussi de construire des n-uplets dont les champs ne sont pas nommés, mais indicés par des entiers. On les appelle des *tableaux*.

Les champs d'un tableau, contrairement à ceux d'un enregistrement, sont tous du même type.

Le nombre de champs d'un tableau est déterminé au moment de l'allocation du tableau, et non au moment de la déclaration de son type, comme c'est le cas pour un enregistrement. Cela permet de calculer, au cours de l'exécution d'un programme, un entier `n` et d'allouer un tableau de taille `n`. Il est également possible de calculer, au cours de l'exécution d'un programme, un entier `k` et d'accéder au $k^{ème}$ champ d'un tableau, ou de l'affecter. En revanche, une fois le tableau alloué, il n'est plus possible de changer sa taille. La seule possibilité est d'allouer un nouveau tableau et de recopier l'ancien dans le nouveau.

En Java, un tableau dont les éléments sont de type `T` est de type `T []`.

On alloue un tableau avec l'opération `new`

```
new int [10]
```

L'évaluation de l'expression `new int [u]`, où `u` est une expression dont la valeur est un entier `n`, crée une nouvelle référence `r'` et associe cette référence à un n-uplet ne contenant que des valeurs par défaut : `0` dans ce cas. Les champs sont numérotés de `0` à `n - 1`.

Si la valeur de l'expression `t` est une référence associée dans la mémoire à un tableau et la valeur de l'expression `u` est un entier `k`, la valeur de l'expression `t[u]` est la valeur contenue dans le $k^{ème}$ champ de ce tableau.

Pour affecter le $k^{ème}$ champ d'un tableau, on utilise une nouvelle instruction `t[u] = v`; où `t` est une expression dont la valeur est une référence associée dans la mémoire à un tableau, `u` est une expression dont la valeur est un entier `k` et `v` une expression de même type que les éléments du tableau. Quand on exécute cette instruction, le $k^{ème}$ champ du tableau reçoit la valeur de l'expression `v`.

Ainsi, le programme

```
int [] t = new int [10];
int k = 5;
t[k] = 4;
System.out.println(t[k]);
```

affiche `4`.

Pour indicer des n-uplets par des couples ou des triplets d'entiers, typiquement pour représenter des matrices ou des images, une possibilité est de construire un tableau dont les éléments sont eux-mêmes des tableaux. Ainsi, l'élément d'indice `(i,j)` du tableau `t` s'écrit `t[i][j]`. Un tel tableau a le type `T [][]`. L'allocation d'un tel tableau à entrées multiples se fait cependant en une seule opération

```
int [][] t = new int [20][20];
```

## Les types de données dynamiques

Les enregistrements permettent de construire des données composées de plusieurs nombres ou caractères. Mais toutes les valeurs d'un tel type de données sont formées du même nombre de champs. Il est impossible de définir un type enregistrement qui contienne plusieurs nombres ou caractères sans que l'on sache *a priori* combien. On peut représenter de tels objets par des tableaux, mais la taille d'un tableau est déterminée une fois pour toutes au moment de son allocation. Nous allons à présent voir comment utiliser des données dont la taille n'est pas bornée, si ce n'est par la taille de la mémoire de l'ordinateur utilisé, qui est nécessairement finie. On appelle *données dynamiques* de telles données composites dont la taille n'est pas connue *a priori* et peut évoluer au cours de l'exécution du programme.

Précédemment, nous avons construit un type enregistrement dont les champs étaient d'un type scalaire `double`. Il est également possible de définir un type enregistrement `T` dont les champs sont eux-mêmes d'un type enregistrement, en particulier le type `T` lui-même. Par exemple, un triplet d'entiers `(a,b,c)` peut se définir comme le couple `(a,(b,c))`, et plus généralement une *liste* non vide d'entiers peut se définir comme un couple formé d'un entier, la *tête* de la liste, et d'une liste plus courte, la *queue* de la liste. Cela amène à définir le type `IntList` ainsi

```
class IntList {
int hd;
IntList tl;}
```

La tête de la liste `1, 2, 3, 4`, par exemple, est l'entier `1`. La queue de cette liste est la liste `2, 3, 4` – et non l'entier `4`.

Le type `IntList` est donc un type enregistrement récursif, c'est-à-dire dont l'un des champs est lui-même de type `IntList`. Parmi les types enregistrements récursifs, certains ont un unique champ récursif et d'autres plusieurs.

On appelle *types de listes* les différents types qui ont un unique champ récursif, quel que soit le nombre de champs non récursifs, et *types d'arbres* les types qui en ont plusieurs.

Les éléments du type `IntList` sont soit :

– la valeur `null`, qui représente, par convention, la liste vide,

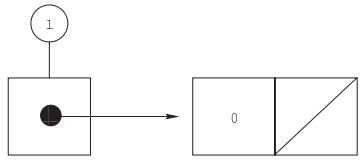
– un élément de `int × IntList`, qui représente une liste non vide.

Autrement dit, `IntList` vérifie la propriété `IntList = {null} ∪ (int × IntList)`.

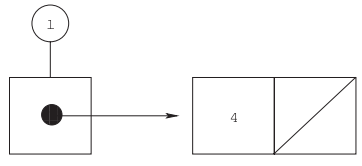
Observons ce qui se passe quand on exécute le programme

```
IntList l = new IntList();
l.hd = 4;
l.tl = new IntList();
l.tl.hd = 5;
l.tl.tl = null;
```

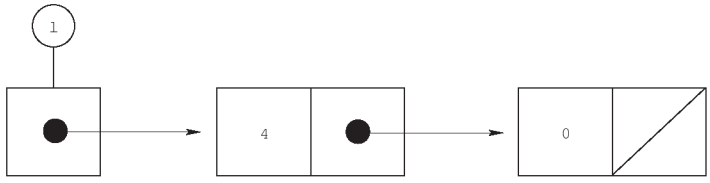
La déclaration `IntList l = new IntList();` alloue une cellule `r'`, remplit cette cellule avec les valeurs par défaut `0` et `null`, associe la variable `x` à une cellule `r` dans l'environnement et la référence `r` à la référence `r'` dans la mémoire



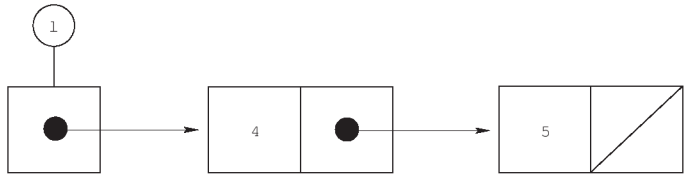

L'instruction `l.hd = 4;` affecte le champ `hd` de la cellule avec la valeur `4`.




L'instruction `l.tl = new IntList();` alloue une nouvelle cellule



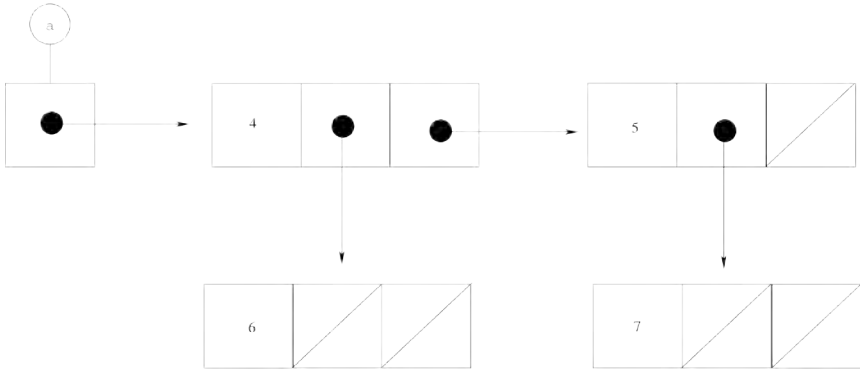

Enfin, les deux dernières instructions affectent les champs de cette cellule



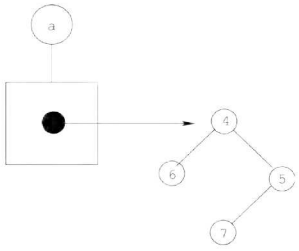

De manière similaire, nous pouvons définir le type des arbres binaires

```
class Arbre {
int val;
Arbre gauche;
Arbre droit;}
```

Un *arbre* est une valeur de ce type, c'est ou bien la valeur `null` appelée l'*arbre vide* ou bien une référence associée dans la mémoire à un enregistrement formé d'un nombre entier et de deux arbres, appelés le sous-arbre gauche et le sous-arbre droit de l'arbre `a`.



Nous utilisons souvent une notation plus pratique pour les arbres. Chaque cellule est représentée simplement par un cercle et les flèches vers les autres cellules seront remplacées par de simples segments. Ainsi, ce même état sera représenté ainsi




Nous avons « défini » ci-avant le type `IntList`, comme `IntList = {null} ∪ (int × IntList)`. Le fait que `IntList` apparaisse à droite du signe = est dû à ce que la définition est récursive. Encore une fois, les définitions récursives nous apparaissent comme des définitions circulaires. Et, encore une fois, une manière de briser cette circularité est de définir un ensemble `L` qui est solution de l'équation `X = {null} ∪ (int × X)` et de procéder par approximations successives, en définissant l'ensemble $L_i$ des valeurs de type `IntList` que l'on peut construire en `i` étapes au plus

$L_0 = \emptyset$
$L_1 = \{null\} \cup (int \times L_0) = \{null\}$
$L_2 = \{null\} \cup (int \times L_1)$
$L_3 = \{null\} \cup (int \times L_2)$
...

puis de définir l'ensemble `L` comme la réunion de ces ensembles : $L = \bigcup_i L_i$

La valeur `null` est essentielle dans cette construction. La même construction avec l'équation `X = int × X` donnerait un type vide.

L'ensemble `L` ainsi construit n'est pas l'unique solution de l'équation `X = {null} ∪ (int × X)`, l'ensemble de toutes les suites finies ou infinies en est également une, mais l'ensemble construit est la plus petite de ces solutions pour la relation d'inclusion.

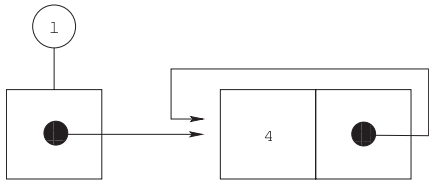
### Les valeurs infinies

Le type `IntList` est également solution de l'équation `X = {null} ∪ (int × X)`, mais ce n'est pas exactement la plus petite. En effet, ce type

contient des valeurs qui ne peuvent pas se construire en un nombre fini d'étapes. Par exemple, le programme

```
IntList l = new IntList();
l.hd = 4;
l.tl = l;
```

construit la liste




c'est-à-dire la liste `4, 4, 4, 4, 4`... qui est infinie.

Le type `IntList` contient en fait toutes les listes *rationnelles*, c'est-à-dire les listes finies ou infinies qui n'ont qu'un nombre fini de sous-listes distinctes. Toutes les listes finies sont rationnelles, mais il y a des listes rationnelles infinies, comme celle-ci.

## Abstraire un type de données

Lorsqu'on écrit un programme, il est souvent utile d'abstraire la définition des types de données, en cachant la définition effective de ce type et en utilisant un nombre limité de fonctions qui permettent de manipuler les éléments de ce type. On distingue ainsi souvent les fonctions de construction, ou *constructeurs*, qui servent à définir de nouvelles valeurs du type considéré, et les primitives d'accès, ou *sélecteurs*, qui servent à connaître ces valeurs. Cette abstraction permet de construire des programmes plus modulaires, plus faciles à modifier et à réutiliser. Par exemple un type `Suite`, des suites d'entiers, se spécifie ainsi.

– Constructeurs :

• `Suite vide ();` est la suite vide, qui ne contient aucun élément,

• `Suite ajoutTete (Suite s, int e);` est la suite dont le premier élément est `e` et le reste `s`.

– Sélecteurs :

• `boolean estVide (Suite s);` vaut `true` si et seulement si `s` est la suite vide,

• `int premier (Suite s);` est le premier élément de la suite `s`,

• `Suite fin (Suite s);` est la suite `s` privée de son premier élément.

Bien sûr, `premier` et fin ne sont pas définis pour une suite vide.

Différentes solutions sont alors possibles pour implémenter ce type `Suite`, par exemple en utilisant un tableau d'entiers ou une liste. Le point essentiel est que, pour utiliser ce type `Suite`, nous n'avons pas besoin de savoir s'il est défini comme un type de listes ou un type de tableaux, mais uniquement de connaître ces cinq fonctions.

## La notion générale de langage

Un langage de programmation, nous l'avons vu, a cette caractéristique d'être compréhensible à la fois par une machine et par un être humain. De nombreux autres langages partagent cette caractéristique. Pour définir un tel langage, on commence toujours par définir sa syntaxe, c'est-à-dire l'ensemble des chaînes de caractères qui appartiennent à ce langage. Pour commencer par un exemple simple, nous pouvons définir la syntaxe du langage qui contient toutes les chaînes de caractères formées d'un nombre quelconque de fois la lettre `a` suivie de la lettre `b`. Ce langage contient les chaînes `b, ab, aab`… mais pas la chaîne `aba`. L'ensemble *L* de ces chaînes de caractères peut se définir par les deux règles suivantes.

– La chaîne de caractères `b` appartient à *L*.

– Si la chaîne `S` appartient à *L*, alors la chaîne `a S` aussi.

Ces règles peuvent s'écrire dans une notation plus compacte

*L* = `b|a` *L*

Nous pouvons, de même, décrire la syntaxe des instructions du noyau impératif de Java. La description que nous proposons n'est qu'indicative, la véritable syntaxe de Java est bien entendu plus complexe.

*I* = *T V* = *E*; *I* | *V* = *E*; | {*II*} | if (*E*) *I* else *I* | while (*E*) *I*

*T* = byte | short | int | long | float | double | boolean | char

*E* = *une expression permettant de calculer une valeur d'un type donné*

*V* = *un identificateur en Java*

Nous retrouvons l'idée qu'une instruction est une déclaration (*T V* = *E*; *I*), une affectation (*V* = *E*;), une séquence ({*II*}), un test (if (*E*) *I* else *I*) ou une boucle (while (*E*) *I*). Dans cette définition, nous utilisons trois catégories grammaticales annexes : celle des types (*T*) et celles des expressions (*E*) et des identificateurs (*V*) qui restent à définir.

La syntaxe, telle que nous l'avons définie, ne nous contraint pas à affecter une variable avec une expression du même type, ainsi l'instruction `int x; x = "toto";` est syntaxiquement correcte. Elle ne déclenchera pas une erreur

de syntaxe, mais une erreur de type. De même, en français, le verbe « dormir » s'emploie avec un sujet qui désigne une personne, mais la phrase « Une idée dort » est syntaxiquement correcte, contrairement à la phrase « Une malgré dort ».

Cette manière de décrire la syntaxe d'un langage peut s'utiliser avec d'autres langages que les langages de programmation. Un exemple est le langage HTML (*Hypertext Markup Language*) qui sert à décrire des pages web. Dans la suite de ce paragraphe, nous considérerons une version de HTML, appelée XHTML 1.0, dont la syntaxe est plus simple que celle du langage HTML proprement dit. En effet, le langage HTML permet de nombreuses libertés, ce qui complique l'écriture de sa syntaxe et des programmes l'interprétant, les navigateurs web.

La manière la plus simple de décrire une page web est d'indiquer la chaîne de caractères qui doit s'afficher sur cette page, par exemple la chaîne `Bonjour tout le monde`. Cela mène à une syntaxe simple

*B* = ε | *C B*

*C* = `a | b | c | d | ... | z`

qui spécifie qu'un élément de *B* est ou bien la chaîne vide ε ou bien une chaîne non vide, formée d'un caractère, *C*, et d'une chaîne *B*.

Mais, rapidement, nous nous rendons compte que nous voulons, dans notre page web, mettre une certaine partie du texte en gras et une autre en italique. Pour mettre, par exemple, le mot `tout` en gras, il faut insérer la balise `<b>` au début du mot et la balise `</b>` à la fin de ce mot, et de même avec les balises `<i>` et `</i>` : `Bonjour <b>tout</b> le <i>monde</i>`.Ce qui nous mène à la syntaxe suivante

*B* = ε| *CB* | *EB*

*C* = `a | b | c | d | ...`

*E* = `<b>` *B* `</b>` | `<i>` *B* `</i>`

qui exprime qu'un élément du langage *B* est une suite composée de caractères ou d'éléments du langage entourés de balises, comme `<b>tout</b>`. L'expression `<b><i>tout</i> le monde</b>` appartient au langage, mais pas l'expression `<i>tout <b>le</i> monde</b>`, car la seule imbrication autorisée est qu'une balise soit ouverte et fermée entre l'ouverture et la fermeture d'une autre balise, même si les êtres humains que nous sommes ont souvent une bonne idée de la sémantique qu'il faudrait attacher à cette seconde expression. Pour représenter, dans la syntaxe définie ci-avant, un document dont la sémantique serait d'écrire le mot `tout` en italique, `le` en gras et italique, puis `le monde` en gras, il faut par exemple écrire l'expression `<i>tout <b>le</b></i> <b>monde</b>`.

Nous pouvons ensuite ajouter d'autres types d'éléments, par exemple les ancres pour les liens hypertextes et les titres de paragraphes

*E* = `<b>` *B* `</b>` | `<i>` *B* `</i>` | `<a` *A*`>` *B* `</a>` | `<h1>` *B* `</h1>`

La balise `<h1>...</h1>` (*Heading 1*) signifie que le texte qu'elle contient est un titre de paragraphe. En XHTML, il existe 6 niveaux de titres, de `h1` à `h6`. Concernant la balise `<a>` (*Anchor*), il faut ajouter un attribut, *A*, qui sert à indiquer l'adresse de la page web vers laquelle le navigateur devra s'orienter si nous cliquons sur ce lien

*A* = `href = "`*U*`"`
*U* = *une adresse web*

où *U*, qui reste à décrire, est la syntaxe des adresses sur le Web. Cette syntaxe est disponible à l'adresse: `http://www.w3.org/Addressing/URL/5_BNF.html`. Elle fait environ 120 lignes. Ainsi l'expression `Bonjour <a href = "http://science-info">tout</a> le <b>monde</b>` appartient au langage.

Enfin, nous voulons pouvoir inclure dans la description d'une page web des informations, par exemple un titre, qui n'apparaîtront pas sur la page. Pour cela la page contient un en-tête. Et comme le langage XHTML peut évoluer, il faut aussi indiquer, au début du fichier, la version du langage utilisée, chaque version correspondant à une grammaire bien spécifique.

Cela nous amène à définir, en utilisant le langage *B* défini ci-avant, un langage *L*

*L* = *V* `<html> <head>` *H* `</head> <body>` *B* `</body> </html>`

Pour simplifier, nous pouvons supposer que *V* est le langage qui ne contient que la chaîne de caractères

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

et que *H* se limite à des expressions de la forme `<title>` *C* `</title>`. Ainsi nous arrivons à la conclusion que l'expression

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head><title>Ma page web</title></head>
<body>
Bonjour <a href = "http://science-info-lyca.fr">tout</a>
le <b>monde</b>
```

```
</body>
</html>
```

est une expression bien formée en XHTML.

La sémantique d'une telle expression est d'afficher la page

Bonjour tout le **monde**

dont le titre est `Ma page web` et où un clic sur le lien tout mène à la page `http://science-info-lyca.fr`. Le fait que le texte contenu entre les balises `<a>...</a>` soit cliquable et mène vers une autre page web fait partie de la sémantique du langage XHTML, de même que le texte entre les balises `<title>...</title>` correspond au titre du document.

Toutefois, le *rendu visuel* de ces éléments sémantiques est paramétrable : il existe un langage formel, appelé CSS (*Cascading Style Sheets*) qui permet de définir le rendu de la plupart des balises du langage XHTML et, par exemple, de décider que les liens cliquables sont en bleu au lieu d'être soulignés, ou que les titres de paragraphes doivent être d'une certaine police et taille.

Le langage XHTML, comme le langage Java, est compréhensible par une machine. Un programme qui doit traiter un texte écrit en XHTML ou en Java, comme un navigateur ou un compilateur, commence en général par transformer ce texte en un arbre – cette transformation est l'*analyse syntaxique* du texte –, car un arbre est une structure beaucoup plus facile à traiter qu'un texte linéaire.

Ces langages sont également compréhensibles par les êtres humains, même si nous avons souvent du mal à ne pas oublier une parenthèse ou une balise quand nous cherchons à nous exprimer dans un langage aussi contraint. C'est pour cela que, après avoir débuté comme langage produit par les humains dans les années 1990, (X)HTML est devenu un langage essentiellement produit par des programmes, comme, par exemple, les éditeurs de sites web, ou encore les traitements de texte. En sauvegardant un document édité avec un traitement de texte en format (X)HTML, le document sauvegardé sur le disque est un document texte possédant comme annotations de mise en forme des instructions HTML et CSS. Un tel document peut ainsi être lu par un navigateur, ce qui lui confère une bien meilleure portabilité que l'utilisation d'un format propriétaire.

## Exercice corrigé et commenté

Le but de cet exercice est d'écrire un programme qui permet d'extraire le contour d'une image. Par exemple, extraire le contour de l'image

 produit l'image 

### Le type des images

Comme nous l'avons vu dans le premier chapitre, une image est un ensemble de pixels. Nous représentons donc une image comme un tableau `tab` à double entrée, l'élément du tableau `tab[x][y]` étant la valeur du pixel de coordonnées `x, y`. Dans cet exercice, nous utilisons des images en noir et blanc, à niveau de gris. Chaque pixel est représenté par un nombre entier compris entre `0` – noir – et une certaine valeur `max` – blanc – qui, en pratique, sera souvent égale à `255`. Une situation plus simple serait d'utiliser des images en noir et blanc mais sans gris, où chaque pixel est représenté par un booléen. Une situation plus complexe serait d'utiliser des images en couleur, où chaque pixel est représenté par un triplet de nombres entiers.

Une image se définit donc comme un enregistrement composé de deux nombres entiers : la largeur et la hauteur de l'image ; un troisième nombre entier : la valeur maximale que peut prendre un pixel ; et enfin le principal : le tableau des pixels.

```
class Image {
 int l;
 int h;
 int max;
 int [][] tab;}
```

Pour fabriquer, transformer et lire une image, nous écrivons trois fonctions. La première prend en arguments trois nombres entiers `l`, `h` et `max`, crée et renvoie une image de largeur `l`, hauteur `h` et valeur maximale `max`. La valeur des pixels de cette image n'est pas spécifiée. En pratique, il est probable que le tableau soit rempli par défaut de `0` et donc que l'image soit entièrement noire.

```
static Image creeImage (int l, int h, int max) {
 Image p = new Image ();
 p.l = l;
 p.h = h;
```

```
 p.max = max;
 p.tab = new int [l][h];
 return p;}
```

La deuxième fonction permet de transformer une image `p` en affectant le pixel de coordonnées `x, y` avec la valeur `k`.

```
static void affectePixel (Image p, int x, int y, int k){
 p.tab[x][y] = k;}
```

La dernière permet d'accéder à la valeur d'un pixel de coordonnées `x, y` dans une image `p`. Si jamais les valeurs `x` et `y` correspondent à un point hors de l'image, la fonction n'échoue pas, mais renvoie la valeur du pixel le plus proche.

```
static int valPixel (Image p, int x, int y) {
 return p.tab[Math.min(Math.max(x,0),p.l-1)]
 [Math.min(Math.max(y,0),p.h-1)];}
```

Dans la suite de l'exercice, nous nous astreindrons à créer, transformer et accéder aux images en utilisant ces trois fonctions uniquement. Ainsi, si nous souhaitons, un jour, transformer la manière dont les images sont représentées, il nous suffira de transformer ces trois fonctions et le reste du programme ne sera pas affecté.

### Un exemple d'image

Première question de l'exercice : créer une image, par exemple, l'image ci-avant qui est la réunion du carré dont les points extrêmes sont de coordonnées `50`, `50` et `200`, `200` et du rectangle dont les points extrêmes sont de coordonnées `150`, `100` et `300`, `300`. Il faut pour cela créer une image avec la fonction `creeImage`, puis affecter chacun de ses pixels avec la valeur `0` ou `255` selon que le pixel est dans l'un de ces deux rectangles ou non.

```
static Image exempleImage () {
 Image p = creeImage(400,400,255);
 int x = 0; int y = 0;
 for (x = 0; x < 400; x = x + 1) {
 for (y = 0; y < 400; y = y + 1) {
 if (((50 <= x) && (x <= 200) && (50 <= y) && (y <= 200))
    ||
    ((150 <= x) && (x <= 300) && (100 <= y) && (y <= 300)))
 affectePixel(p,x,y,0);
```

```
 else affectePixel(p,x,y,255);}}
 return p;}
```

Pour afficher cette image, nous écrivons une fonction `afficheImage` qui, à nouveau, affiche l'image pixel par pixel.

```
static void afficheImage(Image p) {
 Isn.initDrawing(«Contour»,10,10,p.l,p.h);
 int x = 0;
 int y = 0;
 for (x = 0; x < p.l; x = x + 1) {
 for (y = 0; y < p.h; y = y + 1) {
 Isn.drawPixel(x,y,valPixel(p,x,y));}}}
```

Nous pouvons enfin écrire le programme principal, qui consiste simplement à créer cette image et à l'afficher.

```
public static void main (String [] args) {
 afficheImage(exempleImage());}
```

### Extraire le contour

Venons-en maintenant à l'algorithme d'extraction de contour proprement dit. Il s'agit de créer une nouvelle image `q` à partir d'une image `p`. Un point `x, y` est noir dans l'image `q` s'il appartient au contour de l'image `p`, c'est-à-dire s'il est très différent de l'un des deux points situés à sa droite et en dessous de lui. Très différent signifie, par exemple, que la différence de valeur entre ces deux pixels est supérieure à `20`.

```
static Image contour (Image p) {
 int x = 0;
 int y = 0;
 int a = 0;
 int b = 0;
 int c = 0;
 Image q = creeImage(p.l,p.h,255);
 for (x = 0; x < q.l; x = x + 1) {
  for (y = 0; y < q.h; y = y + 1) {
  a = valPixel(p,x,y);
  b = valPixel(p,x+1,y);
  c = valPixel(p,x,y+1);
```

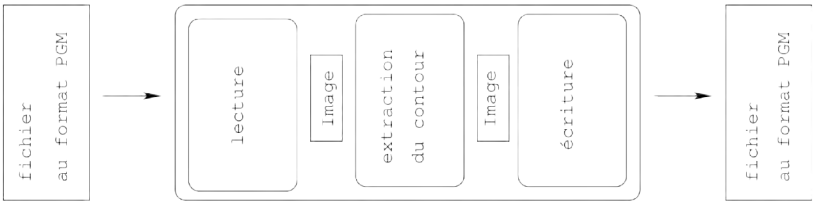
```
  if ((Math.abs(a - b) > 20) || (Math.abs(a - c) > 20))
  affectePixel(q,x,y,0);
  else affectePixel(q,x,y,255);}}
 return q;}
```

Nous pouvons alors transformer notre programme principal pour extraire le contour de notre image.

```
public static void main (String [] args) {
 afficheImage(contour(exempleImage()));}
```

### Le format PGM

Avec ce programme, nous pouvons extraire le contour de l'image que nous avons créée, mais pas extraire le contour d'une image créée par d'autres. Pour ce faire, nous devons remplacer la fonction `exempleImage` par une fonction qui crée une valeur de type `Image` à partir d'un fichier qui décrit une image dans un format standard et, de même, remplacer la fonction `afficheImage` par une fonction qui écrit une valeur de type `Image` dans un tel fichier. Ce qui nous donne un programme organisé de la manière suivante




Nous choisissons d'utiliser le format PGM (*portable graymap*). Il existe deux variantes du format PGM : la variante ASCII et la variante brute. Nous choisissons la première. Il est possible de transformer au format PGM des images de formats très variés en utilisant un logiciel de traitement d'images, par exemple le logiciel `gimp`.

Un fichier au format PGM, dans sa variante ASCII, est constitué de

– sur la première ligne, la chaîne « P2 »,

– une ligne de commentaire, entre les caractères # et retour à la ligne,

– une ligne qui contient deux entiers, la largeur `l` et la hauteur `h` de l'image,

– une ligne qui contient un entier, la valeur maximale `max` utilisée pour coder les niveaux de gris,

– `l × h` entiers compris entre `0` et `max` pour chacun des pixels, séparés par des espaces ou des retours à la ligne. L'ordre des pixels est ligne par ligne de haut en bas et de gauche à droite.

En fait, le format PGM est un peu plus général que cela, car il est possible de mettre une ligne de commentaire non seulement à la deuxième ligne, mais n'importe où dans le fichier. En pratique cependant, la plupart des fichiers PGM suivent cette grammaire plus restreinte.

Nous écrivons donc une fonction qui lit une image à ce format

```
static void lisLigne () {
 char c = Isn.readChar();
 while (c != '\n') c = Isn.readChar();}

static Image lisImage () {
 int x = 0;
 int y = 0;
 lisLigne();
 lisLigne();
 int l = Isn.readInt();
 int h = Isn.readInt();
 int max = Isn.readInt();
 Image p = creeImage(l,h,max);
 for (y = 0; y < h; y = y + 1) {
  for (x = 0; x < l; x = x + 1) {
  affectePixel(p,x,y,Isn.readInt());}}
 return p;}
```

Et une autre qui écrit une image à ce format.

```
static void ecrisImage (Image p) {
 int x = 0;
 int y = 0;
 System.out.println("P2");
 System.out.println("#");
 System.out.print(p.l);
 System.out.print(" ");
 System.out.println(p.h);
 System.out.println(p.max);
 for (y = 0; y < p.h; y = y + 1) {
  for (x = 0; x < p.l; x = x + 1) {
 System.out.println(valPixel(p, x,y));}}}
```

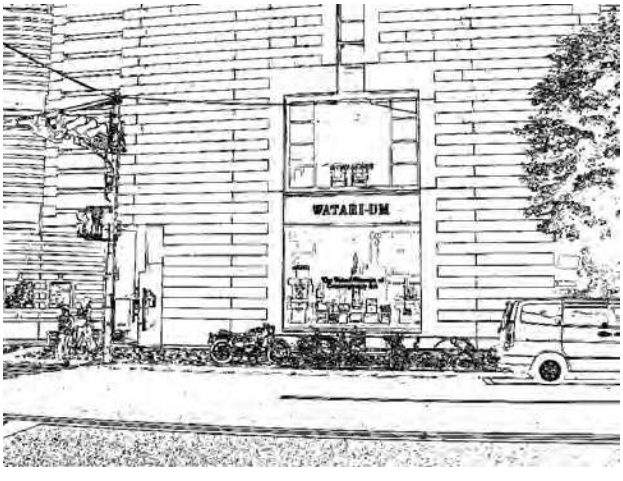
Il nous suffit alors de transformer le programme principal.

```
public static void main (String [] args) {
 ecrisImage(contour(lisImage()));}
```

Pour utiliser notre programme avec une image décrite dans un fichier `f1.pgm` et en récupérant l'image produite dans un fichier `f2.pgm`, il suffit de rediriger l'entrée et la sortie du programme, par exemple avec la commande

```
java Contour < f1.pgm > f2.pgm
```

Nous pouvons ensuite visualiser ces images en utilisant n'importe quel logiciel de traitement d'image.

## Exercices non corrigés

### Compréhension pas à pas du cours. Compréhension du noyau impératif – exercices « papier »

#### Exercice 1. La valeur des expressions : la fonction Θ

Si `t` et `u` sont deux expressions, on a `Θ(t & u,e,m) = Θ(t,e,m) & Θ(u,e,m)`. Donner de façon similaire :

1. `Θ(t | u,e,m)`,
2. `Θ(t && u,e,m)`,
3. `Θ(t || u,e,m)`.

#### Exercice 2. L'exécution des instructions : la fonction Σ

Donner, pour chacun des triplets ci-après, la mémoire qui lui est associée par la fonction Σ en appliquant pas à pas les définitions :

1. `Σ(x = 7;,[x = r,y = r'],[r = 5,r'= 6])`,
2. `Σ({x = 7; y = x + 1; y = y + x + 2;},[x = r,y = r',z = r"],[r = 5, r'= 6,r"= 4])`,
3. `Σ(if (x == 5) {y = x + 1; x = 4;} else x = 5;, [x = r, y = r'], [r = 5, r'= 7])`.

#### Exercice 3. Le test

Donner, pour chacun des triplets ci-après, la mémoire qui lui est associée par la fonction Σ :

1. `Σ(if (x == 1) x = x + 1;,[x = r],[r = 1])`,
2. `Σ(if (x == 1) if (y == 7) x = 2; else x = 3;,[x = r, y = r'], [r = 1, r'= 5])`,
3. `Σ(if (x == 1) {if (y == 7) x = 2;} else x = 3, [x = r, y = r'], [r = 1, r'= 5])`.

#### Exercice 4. Boucles

1. Soit l'instruction `while (x > 0) x = x -1;`. Écrire explicitement les approximations $p_0$, $p_1$, $p_2$, $p_3$ et $p_4$. Calculer `Σ(while (x > 0) x = x-1;,[x = r],[r = 3])`.

2. Calculer
   - `Σ(do x = x-1; while (x > 0),[x = r],[r = 3])`,
   - `Σ(do x = x -1; while (x > 0),[x = r],[r =-2])`,
   - `Σ(while (x > 0) x = x - 1;,[x = r],[r = -2])`.

3. Soit l'instruction `while (x > 0) {while (y < 5) {x = x - 1; y = y + 1;} x = x + 2;}` exécutée dans l'environnement `[x = r, y = r']` et la mémoire `[r = 4, r'= 0]`. Décrire le déroulement de son exécution.

4. Calculer : – Σ({y = 1; for(x = 1;;x < 10; x = x + 1;) y = y*x;}, [x=r, y=r'], [r=3,r'=7]),

– Σ({x = 0; for(i = 1;i <11; i = i + 1;) for(j = 1;j < 6; j = j + 1) x = x + 1;}, [x=r, i=r', j=r''],[r=0, r'=0, r''=0]).

### Exercice 5. Programmes équivalents

On considère les deux programmes `p1` et `p2` suivants dans lesquels `b` est une expression et `q1` et `q2` sont des affectations ou des séquences d'affectations. On suppose que les seules variables utilisées dans `b`, `q1` et `q2` sont `x` et `y`.

```
static public void main (String [] args){ /* p1 */
int x=0;
int y=0;
if (b) q1 else q2}

static public void main (String [] args) { /* p2 */
int x=0;
int y=0;
if (b) q1;
if (!b) q2}
```

Donner des exemples possibles de `b`, `q1` et `q2` pour que, après exécution de `p1` et `p2` dans des environnements identiques :

1. les mémoires obtenues soient identiques,
2. les mémoires obtenues soient différentes.

## Fonctions, procédures, récursivité

### Exercice 6. Passage de paramètres

Donner les affichages produits lors de l'exécution des programmes suivants – on supposera que l'évaluation des arguments se fait toujours de la gauche vers la droite :

1.
```
class essaiPassage{
static int y=3;

static int g(int u, int v) {
int x=2;
return (x+u+v*v);}

public static void main(string[]args) {
system.out.println (g(1, y));
system.out.println (g(1, 3));}}
```

2.
```
class essaiMasquage {
static int y=1;

static int f (int x) {
 return (x+x);}

static int g(int z) {
 int x=7;
 return (z+x);}

static int h(int z) {
 int x=7;
 return (z+x);}

public static void main (String [] args) {
 System.out.println(f(x));
 System.out.println(g(x) + « « + x);
 System.out.println(h(x) + « « + x);}}
```

3.
```
class Essai-OrdreEvaluation {
static int n = 0;

static void reset () {
 n = 0;}

static int f(int x) {
 n = n+1 ;
 return (x + n);}

static int g (int y) {
 n = 2*n;
 return y + n);}

public static void main (String [] args) {
 reset () ;
 System.out.println (f(4) + g(4));
 reset () ;
 System.out.println (g(4) + f(4));}}
```

### Exercice 7. Fonctions récursives

1. On considère la fonction `biz` suivante. Évaluer `biz(25)`, `biz(24)`, `biz(23)`.

L'évaluation termine-t-elle toujours ? Exprimer le détail de la fonction Θ d'évaluation sur l'appel `biz(15)`.

```
static int biz (int x) {
 if (x < 2) return 0;
 else if (x % 5 = 0) return biz (x / 5) ;
 else return (3 + biz(x + 1));}
```

2. Écrire en Java la fonction `L` suivante, qui prend en argument un entier et rend un entier :

```
si x <2
alors L(x) = 0
sinon si x est pair
alors L(x) = 1 + L(x /2)
sinon L(x) = 1 + L((x-1) /2)
```

Décrire l'évolution des mémoires au cours de l'évaluation de `L(37)`.

3. On considère la fonction `Pr` ci-après. Donner l'évolution des mémoires lors de l'évaluation de `Pr(45)` et de `Pr(17)`. Que fait en général cette fonction ?

```
static boolean pr (int n) {
 if (n < 2) return false;
 else return nd(n,2);}

static boolean nd (int n, int k) {
 if (n % k = 0) return false;
 else if ((k+1)*(k+1) <= n) return nd(n, k+1);
 else return true;}
```

## Écriture de programmes

### Exercice 8. Parcours d'une suite

Dans cet exercice, `l` est une suite d'entiers pour laquelle on considérera deux implémentations possibles : tableau et liste dynamique.

1. Trouver l'élément maximal de `l`.
2. Calculer la somme des éléments de `l`.

3. Calculer la « moyenne olympique » des éléments de `l` – la moyenne de `l` privée de ses éléments de valeur minimale et maximale.

4. Trouver le nombre d'éléments de `l` supérieur à une valeur `k`.

5. Trouver la valeur du « second maximum » de `l` – le plus grand élément de `l` strictement inférieur à son élément maximal.

6. Calculer la longueur de la plus longue sous-suite croissante de `l`.

### Exercice 9. Compter les mots

On considère une suite de caractères dans laquelle chaque élément est soit une *lettre*, soit un *séparateur* – espace, tabulation, retour à la ligne – soit une *ponctuation* – virgule, point, etc. On définit un *mot* comme une suite de lettres située entre deux caractères séparateurs ou de ponctuation, exception faite du premier mot du texte, lequel n'est pas nécessairement précédé d'un tel caractère. Une telle suite de caractères peut être obtenue par la lecture, caractère par caractère, d'un fichier texte.

1. Compter les mots du texte – attention, il peut y avoir plusieurs séparateurs entre chaque mot.

2. Calculer la fréquence de chaque mot du texte.

### Exercice 10. Interclassement

On considère deux tableaux d'entiers `T1[n1]` et `T2[n2]` dont les valeurs sont triées par ordre croissant : les suites `T1[0], T1[1] … T1[n1-1]` et `T2[0], T2[1] … T2[n2-1]` sont croissantes. En parcourant simultanément ces deux suites, construire dans un tableau `T[n1+n2-1]` la suite croissante obtenue par interclassement des deux suites initiales.

### Exercice 11. Schéma de Hörner, changement de base

Un nombre entier naturel `N` est représenté en base `10` par une chaîne de caractères, dont chaque élément est compris entre `'0'` et `'9'`, et terminé par une marque de fin.

1. Écrire une fonction qui convertit une telle chaîne en entier.

2. Afficher la valeur du carré de `N`.

3. Afficher la chaîne de caractères représentant `N` en base `b`, pour `b` donné.

### Exercice 12. Chaînes de caractères

On considère deux chaînes de caractères $S_1$ et $S_2$, de longueurs respectives $l_1$ et $l_2$, représentées dans des tableaux `T1` et `T2`.

1. Déterminer si $S_1$ est un palindrome.

2. Déterminer si $S_1$ est une projection de $S_2$, c'est-à-dire si tous les éléments de $S_1$ sont présents dans $S_2$ dans le même ordre.

3. Déterminer si $S_1$ et $S_2$ sont des anagrammes.

4. Calculer le nombre minimal de caractères à ajouter à $S_1$, à n'importe quelle position, pour que $S_1$ soit un palindrome.

### Exercice 13. À propos d'ensembles

On considère le type ensemble d'entiers sur {0, 1, 2, …, N-1} et deux représentations possibles pour ce type :

– Chaque ensemble E est représenté par un tableau de k entiers, où k est le cardinal de E : E est l'ensemble des k éléments du tableau.

– Chaque ensemble E est représenté par un tableau de booléens T[s0..N-1] tel que T[x] est true si et seulement si x ∈ E.

1. Écrire une fonction qui transforme une représentation en l'autre.

2. Écrire dans ces deux cas les fonctions suivantes : initialisation d'un ensemble vide, ajout et suppression d'un élément, test de l'ensemble vide, union et intersection d'ensembles.

### Exercice 14. Point fixe d'une suite

Étant donné un entier x, soit d(x) et c(x) les entiers obtenus en ordonnant les chiffres de l'écriture en base 10 de x par ordre décroissant, resp. croissant. On considère la suite récurrente (X_i) suivante :

$X_0$ est un nombre de N chiffres en base 10.

$X_{i+1} = d(X_i) - c(X_i)$.

On suppose que les nombres s'écrivent toujours avec N chiffres, avec éventuellement des 0 non significatifs en tête. Écrire un programme qui affiche oui ou non selon que la suite $X_i$ comporte ou non un point fixe – $X_i$ comporte un point fixe s'il existe un rang n tel que $X_n = X_{n+1}$.

Questions :

1. L'existence d'un point fixe dépend de la valeur de N – le nombre de chiffres – et de celle de $X_0$. S'il y a un point fixe, le trouve-t-on au bout d'un temps fini ? S'il n'y a pas de point fixe, comment éviter que le programme boucle et savoir répondre non ?

2. À partir de quelle valeur de N le programme n'aura-t-il pas le comportement attendu ? Pourquoi ?

### Exercice 15. Expressions arithmétiques

On considère un arbre binaire a représentant une expression arithmétique :

1. les feuilles sont étiquetées par des nombres entiers,

2. les nœuds internes sont étiquetés par des opérateurs arithmétiques binaires (+, -, *, …).

Écrire une fonction qui calcule la valeur de cette expression arithmétique. Écrire une action qui affiche à l'écran l'expression représentée par a sous différentes formes : préfixée, infixée et postfixée.

Dans un second temps, on pourra chercher à minimiser le nombre de parenthèses en supprimant les parenthèses inutiles.

## Questions d'enseignement

### La programmation

Un programme est un texte qui exprime un *algorithme* transformant de l'*information* et qui est écrit dans un *langage* de programmation afin d'être exécuté par une *machine*. Clé de voûte où les quatre arcs qui structurent l'informatique se rejoignent, la programmation a naturellement une place privilégiée dans un cursus de découverte de l'informatique.

La programmation est, en outre, un élément de ce cursus souvent apprécié des élèves, car elle les place dans une situation active et créative, dans laquelle ils peuvent eux-mêmes fabriquer un objet. Même si les programmes écrits par les élèves sont de bien plus petite taille que ceux qu'ils utilisent quotidiennement – navigateurs, messageries, jeux vidéo… –, écrire un programme soi-même permet d'extrapoler à partir de sa propre expérience et d'imaginer comment ont été écrits les programmes que l'on utilise. Les élèves acquièrent alors un sentiment d'autonomie, en prenant conscience qu'ils pourraient avoir écrit ces programmes eux-mêmes. Cependant, la programmation est réputée difficile à enseigner.

Tout d'abord, enseigner la programmation demande de choisir un langage. Surviennent alors parfois des querelles de chapelle et de longs débats sur les avantages et inconvénients de tel ou tel langage. Le choix d'un langage mène souvent à prendre conscience du nombre de langages existants et de leur fugacité. La question de l'intérêt d'apprendre un langage particulier et éphémère se pose alors naturellement. Apprendre un langage demande aussi d'apprendre un certain nombre de détails dépourvus d'intérêt, par exemple, que l'affectation s'écrit = et la comparaison ==, à moins que – tout dépend du langage choisi – l'affectation ne s'écrive := et la comparaison =.

Ensuite, comme tout savoir technique, c'est-à-dire qui consiste à mobiliser des connaissances dans le but de fabriquer un objet, la programmation demande du temps pour être apprise.

Enfin, enseigner la programmation demande de trouver un vocabulaire conceptuel adéquat pour verbaliser ce qu'il se passe quand un programme est exécuté. Il faut expliquer aux élèves pourquoi leurs programmes font ce qu'ils font sans personnifier la machine, qui « voudrait » ou « ne voudrait pas » fonctionner comme on s'y attend, et en évitant l'écueil d'un trop grand niveau de détail, en expliquant, par exemple, ce qu'il se passe quand on exécute telle ou telle instruction, en termes de signaux circulant sur un bus ou de transistors basculant d'un état à un autre.

Cependant, avec le temps, des réponses à ces légitimes questions ont été apportées.

## Enseigner la programmation v.s. enseigner un langage

La première réponse est sans doute que choisir un langage ou un autre n'est pas si important. À l'exception de langages spécialisés – utilisés par exemple pour écrire des programmes parallèles –, tous les langages sont organisés autour d'un petit nombre de fonctionnalités qui sont les mêmes de l'un à l'autre et qui sont relativement stables dans le temps.

Enseigner la programmation demande de se concentrer sur l'essentiel, c'est-à-dire sur les notions universelles de déclaration, d'affectation, de séquence, de test, de boucle, de fonction, de récursivité, puis, après le lycée, d'enregistrement, d'exception, de module, d'objet… et non sur les bizarreries de tel ou tel langage. C'est seulement ainsi que peut apparaître la portée des notions enseignées.

## Décrire la sémantique

La question de la conceptualisation de la sémantique des programmes mène, comme nous l'avons fait dans ce chapitre, à donner un rôle central à la notion d'état et aux idées qu'une instruction est toujours exécutée dans un certain état et que cette exécution produit un autre état. L'exécution d'une instruction transforme un état en un autre état. Autrement dit, il faut passer de l'idée qu'un programme fait quelque chose, à celle qu'il fait quelque chose *à quelque chose*.

La manière dont on décrit les états doit alors être adaptée aux instructions dont on veut décrire la sémantique : une simple structure associant une valeur à chaque variable est suffisante pour expliquer la sémantique de la déclaration, de l'affectation, de la séquence, du test et de la boucle. Décomposer l'état en un environnement et une mémoire est en revanche nécessaire pour expliquer la notion de fonction et le passage d'arguments. Cette notion d'état pourra être introduite informellement ou précisément définie, elle pourra être écrite, dessinée, figurée par des boîtes de carton… ces choix dépendent bien entendu des élèves auxquels on s'adresse, mais, quels qu'ils soient, introduire cette notion d'état et de transformation d'état donne le cadre conceptuel dans lequel exprimer clairement et confortablement ce qu'il se passe quand un programme est exécuté.

Exprimer ainsi la sémantique des instructions mènera naturellement à distinguer les instructions des expressions et les expressions de leur valeur.

Cela mènera aussi naturellement à prendre conscience que les programmes font ce qu'ils font, mais qu'ils pourraient aussi faire autre chose. Par exemple, quand on exécute la séquence `{x = 4; y = x + 1; x = 10;}`, on construit un état dans lequel la variable `y` contient la valeur `5` et non la valeur `11` – l'affectation `x = 10;` ne modifie pas rétroactivement la valeur de `y`. Mais une autre sémantique serait de mettre dans la case `y` non la valeur de l'expression `x + 1` mais l'expression `x + 1` elle-même, si bien que l'affectation `x = 10;` changerait rétroactivement la valeur de `y`. Cette seconde sémantique est différente de la première, mais elle n'est pas absurde, à tel point que c'est la sémantique de quelques langages de programmation, notamment ceux inclus dans les systèmes de calcul formel.

De même, l'évaluation de l'expression booléenne `t & u` consiste à évaluer les expressions `t` et `u` puis à effectuer la conjonction des deux valeurs booléennes obtenues. Une solution alternative serait de commencer par évaluer l'expression `t`. Si cette valeur est `true`, on doit évaluer `u`, mais si c'est `false`, on peut éviter de le faire et renvoyer la valeur `false`. Cette seconde sémantique est différente, car l'évaluation de `u` peut ne pas terminer ou produire une erreur. Si c'est le cas, la première sémantique ne donnera pas de valeur pour l'expression `t & u` alors que la seconde donnera la valeur `false` si l'évaluation de `t` produit la valeur `false`. Cette seconde sémantique n'est cependant pas absurde, à tel point que de nombreux langages de programmation proposent les deux opérateurs – `&` et `&&` en Java.

Enfin, une question difficile est souvent posée par les élèves : un état est-il un état de la machine physique et la sémantique du langage décrit-elle la réalité du processus physique qui se déroule à l'intérieur de la machine ?

La seule réponse honnête à cette question est négative. L'architecture des machines est devenue si complexe que l'explication que l'on donne est de plus en plus éloignée de la réalité. Par exemple, on imagine souvent que lorsque l'environnement associe une référence `r` à une variable `x` et la mémoire la valeur `3` à la référence `r`, cette référence peut être assimilée à une adresse physique dans la mémoire de l'ordinateur. Mais cela est de moins en moins vrai : afin d'optimiser la gestion de la mémoire, il se peut très bien que la valeur `3` soit en fait stockée dans une mémoire située à l'intérieur du processeur ou au contraire dans une mémoire externe ou encore qu'elle migre d'un lieu à un autre au cours de l'exécution du programme. Plutôt que de mentir aux élèves en leur disant que l'on décrit la réalité ou de leur avouer le mensonge, il est plus intéressant de les amener à réfléchir sur la notion d'adéquation d'un modèle à la réalité. Le modèle que constitue la sémantique du langage est adéquat *observationnellement*, c'est-à-dire que *vu*

*de l'extérieur* tout se passe comme si les choses se passaient comme on les a décrites.

Loin de spécifier la manière dont les choses doivent se dérouler à l'intérieur de la machine, la sémantique d'un langage ne fait que définir un contrat que doivent remplir les concepteurs des compilateurs et des machines. Pour remplir ce contrat, ils ont au contraire toute liberté de s'organiser comme ils le souhaitent.

## La classe terminale

La compréhension des constructions des langages de programmation peut grossièrement se décomposer en quatre étapes :

1. le noyau impératif – déclaration, affectation, séquence, test et boucle – les tableaux,
2. les notions de fonction et de récursivité,
3. les enregistrements et les types dynamiques,
4. les modules et les objets.

En arrivant en terminale, les lycéens devraient avoir déjà franchi la première étape, au cours de l'enseignement d'algorithmique inclus dans le cours de mathématiques de seconde et par l'usage de calculatrices programmables – même s'il n'est pas exclu que quelques révisions soient nécessaires.

En ce qui concerne la programmation, le programme de terminale est donc essentiellement centré sur la deuxième étape : les notions de fonctions et de récursivité – les troisième et quatrième étapes étant, quant à elles, hors programme.

La notion de fonction est une notion très riche, mais elle peut s'introduire étape par étape, chacune pouvant être illustrée par un exemple simple

– isolation d'une instruction – fonction `sauterTroisLignes` ci-avant

– passage d'argument – fonction `sauterDesLignes`

– le retour de valeur – fonction `hypotenuse`

– la notion de variable globale et de portée des variables – fonction `reset`

– la différence entre passage d'arguments par valeur et par référence – fonction `echange`

– la récursivité – fonction `fact`.

Les trois premières étapes sont en général rapidement assimilées par les élèves et les trois dernières demandent davantage de temps. La notion de variable globale est d'autant mieux comprise que les notions de déclaration et de portée d'une variable et la distinction entre l'ordre des instructions dans le texte du programme de l'ordre dans lequel elles sont exécutées, sont assimilées – mais précisément elles le sont rarement, car elles ne deviennent

réellement indispensables que lorsque cette notion de fonction est introduite. La différence entre passage d'arguments par valeur et par référence est difficile pour tout le monde, car c'est le premier contact avec la notion de partage. La récursivité est bizarrement vite assimilée par certains et reste longtemps difficile pour d'autres – ce qui reflète sans doute une difficulté pour les seconds à penser localement, sans essayer de comprendre la globalité d'un processus.

## Les langages de programmation

Dans un cursus de découverte de l'informatique, les notions relatives aux langages de programmation sont le plus souvent abordées à travers l'activité de programmation elle-même. Il est cependant possible d'aller un tout petit peu plus loin et d'inciter les élèves à réfléchir à un certain nombre de particularités des langages de programmation.

La première particularité de ces langages est qu'ils obéissent à une double contrainte : être compréhensibles par l'être humain qui écrit le programme et par la machine qui l'exécute. Tant que nous n'utilisions pas de machines pour exécuter des algorithmes, nous exprimions ces algorithmes – l'algorithme de la multiplication, le triangle de Pascal, le pivot de Gauss… – en langue naturelle, et les langages de programmation ne sont apparus que lorsque nous avons commencé à utiliser des machines pour exécuter ces algorithmes.

Les premiers langages de programmation, que l'on appelle *langages machine*, étaient surtout compréhensibles par les machines et les êtres humains devaient faire beaucoup d'efforts pour s'y exprimer. Une grande partie de l'histoire de la théorie des langages de programmation peut se raconter comme l'effort de concevoir des langages toujours de plus haut niveau, c'est-à-dire toujours plus faciles à utiliser pour les humains. Certains ont imaginé que concevoir des langages de programmation de plus en plus faciles à utiliser mènerait, *in fine*, à programmer les ordinateurs en langue naturelle. Mais, finalement, il semble que les langues naturelles ne sont peut-être pas si bien adaptées à l'expression des algorithmes.

Par exemple, l'histoire montre qu'au cours des siècles le langage mathématique s'est progressivement éloigné des langues naturelles – la grande rupture dans cette évolution étant l'introduction de la notion de variable, au XVI$^{e}$ siècle. De même, les musiciens ne cherchent pas spécialement à rapprocher la notation musicale d'une langue naturelle. De ce fait, l'usage d'un langage formel pour exprimer des algorithmes est peut-être une bonne chose, indépendamment du fait que ces algorithmes doivent être exécutés par des machines.

Découvrir l'informatique est l'occasion pour les lycéens de prendre conscience de l'importance des langages artificiels – le langage mathématique, la notation musicale, la nomenclature chimique… Cette incursion vers le langage mathématique sera aussi l'occasion de remarquer que la notion de variable d'un langage de programmation a peu à voir avec la notion mathématique de variable.

Dès le lycée, on peut apprendre à distinguer la notion de *syntaxe*, qui décrit la manière dont une instruction s'écrit, de sa *sémantique* qui décrit ce qu'il se passe quand on l'exécute. La syntaxe se divisant elle-même en *syntaxe abstraite,* qui décrit les ingrédients dont une instruction est composée – par exemple un test est toujours composé d'une expression `b` et de deux instructions `p` et `q` –, et *syntaxe concrète* qui décrit comment cette instruction s'écrit littéralement – le test s'écrivant alors `if (b) p else q` ou `if b then p else q` selon les langages.

Dès que l'on a introduit les fonctions et la récursivité, il devient possible d'identifier deux fragments dans le langage que l'on utilise : le fragment impératif – formé de la déclaration, de l'affectation, de la séquence, du test et de la boucle – et le fragment fonctionnel – formé de la déclaration, du test, des fonctions et de la récursivité. Chacun de ces fragments est complet, c'est-à-dire que tous les programmes peuvent s'y exprimer. On peut inciter les élèves à écrire les mêmes petits programmes dans chacun de ces fragments.

## Questions d'organisation

Lors de la préparation d'une séance de travaux pratiques, plusieurs questions d'organisation se posent. Premièrement, faut-il donner une ébauche de programme aux élèves, qu'ils doivent alors compléter, ou faut-il ne rien leur donner ? Ici encore, la réponse dépend des élèves auxquels on s'adresse, mais l'expérience semble montrer que, s'il est difficile de ne partir de rien pour écrire un programme, il est plus difficile encore d'insérer son travail dans un programme existant. Il vaut donc mieux, au début, laisser les élèves écrire leurs programmes entièrement, quitte à leur proposer des ébauches quand ils commencent à acquérir un peu de maturité.

Une question liée est celle de l'opportunité de faire travailler les élèves individuellement, deux par deux ou en groupe. Ici encore, il n'y a pas de réponse universelle, mais l'expérience semble montrer qu'écrire un programme en groupe – c'est-à-dire définir des modules que chacun doit écrire et parvenir à assembler ces modules – est difficile pour les débutants. C'est donc un mode d'organisation qu'il vaut mieux n'utiliser qu'avec des élèves qui ont acquis un peu de maturité. En revanche, faire travailler les élèves, y compris débutants, deux par deux présente de nombreux avantages, en particulier celui de les forcer à verbaliser leurs idées avant de les mettre en œuvre.

Enfin, il est important de donner une exigence de résultat élevée aux élèves qui se contentent souvent de tester leurs programmes sur un seul exemple. Leur proposer un jeu de test assez complet, comme les inciter à lire et comprendre les messages d'erreur, à instrumenter leurs programmes avec des impressions à chaque étape, voire à utiliser des outils de mise au point, est un moyen de les mener progressivement à réfléchir sur les questions de qualité du logiciel. De même, rendre les programmes lisibles par une indentation soignée et quelques commentaires doit être valorisé : un programme est écrit une fois, mais lu et modifié de nombreuses fois.

## Compléments

### Le partage

Comprendre le passage d'argument des fonctions, mais aussi la notion d'enregistrement, nous a menés à introduire une notion de *partage* qui est une notion très générale. On peut la comprendre de manière simple en constatant qu'il se peut qu'un jour il fasse 22 °C à Paris et 22 °C à Rome. Dans ce cas, on pourra dire que la phrase « la température à Rome est identique à la température à Paris » est vraie. Ce même jour, la phrase « la température à Rome est identique à la température dans la capitale de l'Italie » sera vraie également. Mais, bien entendu, ces deux phrases sont vraies pour des raisons tout à fait différentes. Les logiciens expriment cette différence en disant que les expressions « la température à Rome » et « la température à Paris » ont la même dénotation, mais pas le même sens. Alors que les expressions « la température à Rome » et « la température dans la capitale de l'Italie » ont le même sens, et donc la même dénotation – opposition qui est très proche de l'opposition référent/signifié des linguistes. De même, on pourrait dire que deux variables associées, dans l'environnement, à deux références elles-mêmes associées, dans la mémoire, à une unique valeur ont la même dénotation, mais qu'elles n'ont pas le même sens. En revanche, deux variables associées, dans l'environnement, à la même référence ont le même sens, et donc la même dénotation.

Le dilemme partager/recopier se pose de manière constante quand on réalise, par exemple, une page web. Si on veut mettre sur le site web de son lycée les horaires des bus qui permettent d'accéder à l'établissement, on peut mettre un lien vers la page de la compagnie de bus de la ville, ou alors recopier les informations qui se trouvent sur cette page sur une page du site du lycée. Dans le premier cas, les informations ne seront plus accessibles si le

site de la compagnie des bus est réorganisé, en revanche, elles seront automatiquement mises à jour avec celle du site de la compagnie de bus.

## États et transitions

Notre description de la sémantique des instructions des langages de programmation nous a menés à introduire une notion d'état et une notion de transition entre états. Par exemple, la sémantique de l'instruction `x = 2 ;` est un ensemble de transitions de l'état `([x = r],[r = 1])` à l'état `([x = r],[r = 2])`, de l'état `([x = r],[r = 4])` à `([x = r],[r = 2])`,...

Ces notions d'état et de transition permettent de décrire formellement de nombreux objets, par exemple les règles de beaucoup de jeux. Au jeu d'échecs par exemple, un état décrit essentiellement la position de chaque pièce sur l'échiquier – c'est, formellement, une relation entre les pièces et les cases de l'échiquier – et les règles définissent des transitions possibles entre états. Dans ce cas, cependant, il faut ajouter quelques informations à la description d'un état : à quel joueur est-ce le tour de jouer ? combien de coups ont-ils déjà été joués depuis le début de la partie ? quels sont les joueurs qui ont déjà roqué ? afin que l'ensemble des transitions possibles à partir d'un état dépende uniquement de cet état, et non de l'histoire qui a mené à cet état.

De même, une tortue graphique se définit par un état – sa position et son azimut – et des transitions – avancer, tourner. Une machine également se définit par un état – la valeur de toutes les mémoires, registres, compteur ordinal – et des transitions qui sont exécutées à chaque cycle de l'horloge.

Un tel système défini par un ensemble d'états et un ensemble de transitions s'appelle un *automate*. Cette notion est fondamentale pour l'étude des langages en général, et de l'interprétation et de la compilation des langages de programmation.

## La notion de licence

Avec le développement de logiciels, c'est-à-dire de programmes de grande taille qui demandent donc beaucoup de travail et d'ingéniosité, et l'émergence d'un marché de ces logiciels, est apparue la nécessité de préciser les conditions de leur exploitation par des licences.

L'exploitation des logiciels repose sur un dilemme, qui a son origine dans le fait que les logiciels, biens immatériels, sont des biens *non rivaux* : leur utilisation par une personne n'empêche pas leur utilisation par une autre. De ce fait, si on donne à l'auteur d'un logiciel le monopole de son exploitation et la possibilité d'exploiter ce droit, on limite la diffusion du logiciel

qui pourrait, du fait de sa non-rivalité, être utilisé par tous. À l'inverse, si on lui refuse ce droit, on limite les revenus qu'il peut tirer de son travail. La société doit donc contradictoirement encourager les créateurs et favoriser la diffusion de leur création. Le droit des brevets, dont relèvent les inventions et les activités industrielles, et le droit d'auteur, dont relèvent les arts et les lettres, sont pleinement conscients de ce dilemme puisqu'ils donnent aux inventeurs et aux auteurs le monopole de l'exploitation de leur création et leur permettent de vendre ce droit, mais pour une durée limitée seulement – même si l'on constate une tendance à l'allongement de cette durée.

Second dilemme : la publication du texte du logiciel – du programme source. L'auteur d'un logiciel peut en effet donner à d'autres le droit d'utiliser ce logiciel, sans pour autant en diffuser le source. Cela lui évite de voir ses idées reprises par d'autres. Mais cela limite la possibilité pour d'autres d'améliorer ce logiciel, ou plus simplement de l'examiner pour s'assurer qu'il est conforme à certaines exigences de sûreté et de sécurité – par exemple, qu'il ne laisse pas fuir d'informations confidentielles. Cette question de la publication ne se pose pas dans les domaines des arts et des lettres, où les œuvres ne peuvent pas être exploitées sans être rendues publiques. Dans le domaine des brevets, en revanche, la publication de l'invention est la condition nécessaire à l'acquisition du droit d'exploitation de son invention.

Pour les logiciels, la législation retenue est une forme dérivée du droit d'auteur. En effet, le droit des brevets ne constitue pas un cadre adéquat pour une activité cumulative qui consiste à agencer de nombreuses « briques de base ». Comme il est impossible d'écrire un programme sans reprendre de nombreuses idées déjà formulées par d'autres, le droit des brevets aurait contraint les programmeurs à payer des redevances trop nombreuses. Cela étant, dans certains pays, les logiciels sont brevetables, ce qui pose parfois de sérieux problèmes.

L'idée de propriété intellectuelle remonte au XVI^e^ siècle. Les droits d'auteur sont nés pour répondre à des intérêts de réglementation de la concurrence dans l'édition et l'impression. Dans sa forme juridique moderne, le droit d'auteur a été créé à la veille de la Révolution française par les auteurs de théâtre qui se considéraient spoliés par le monopole d'exploitation de la Comédie-Française. Il n'existe en fait de problème de propriété intellectuelle que là où il y a un marché : Shakespeare, Molière plagiaient sans entraves et la musique d'un de leurs contemporains pouvait librement inclure une appropriation de la musique d'un autre compositeur, ce qui les mènerait aujourd'hui directement au tribunal.

Ces questions du monopole de l'exploitation d'un logiciel et de la diffusion de son source sont celles qui distinguent les deux principales formes de licence utilisées aujourd'hui : les licences *propriétaires* et les licences *libres*. Avec une licence propriétaire, l'auteur du logiciel vend un droit d'utilisation. Il garde le source du logiciel secret afin d'éviter de voir ses idées reprises par d'autres – ce qui peut être considéré comme une régression par rapport au droit des brevets qui impose la publication des inventions. Avec une licence libre, en revanche, l'auteur d'un logiciel donne à ses utilisateurs le droit d'utiliser ce logiciel pour quelque usage que ce soit, d'en étudier le fonctionnement et de l'adapter à ses propres besoins, d'en redistribuer des copies sans limitation, de le modifier et diffuser la version modifiée. L'auteur doit donc donner accès au source de son logiciel.

À première vue, l'auteur d'un logiciel libre semble abandonner la possibilité de voir son travail rémunéré et, de fait, certains logiciels libres ne rapportent aucun revenu à leurs auteurs – c'est par exemple le cas de nombreux logiciels développés par des chercheurs dans les universités. Cependant, il est aussi fréquent pour une entreprise de diffuser un logiciel librement, tout en développant un modèle commercial autour de cette libre diffusion, qui consiste souvent en la vente de biens complémentaires au logiciel. Ainsi, certaines entreprises diffusent librement et gratuitement un logiciel, tout en faisant payer une activité de conseil ou de formation à l'utilisation de ce logiciel. Dans d'autres cas, enfin, une entreprise, ou plus souvent un groupement d'entreprises, paie une autre entreprise pour développer un logiciel libre, sans espoir d'autre retour que celui de l'utilisation du logiciel.

Parmi les licences libres, on en distingue de deux types. Les licences *copyleft*, comme la *General Public Licence* (GPL) ou certaines versions de la licence *CEA CNRS INRIA Logiciel Libre* (CeCILL), imposent que toutes les versions diffusées du logiciel, modifiées ou non, soient distribuées sous la même licence, afin de faire bénéficier les autres des mêmes libertés que celles dont on a soi-même bénéficié – la licence CeCILL résout les problèmes posés en droit français par la licence GPL. Cela empêche que le cycle de la vertu ne soit interrompu par une appropriation privée du logiciel. D'autres licences, comme *Berkeley Software Distribution* (BSD), donnent au contraire la liberté aux utilisateurs du logiciel de le modifier et de diffuser la version modifiée sous une licence propriétaire.

Les licences libres ont permis un nouveau mode de développement des logiciels, par une démarche coopérative. Le système d'exploitation Linux, par exemple, a été développé par des centaines de programmeurs aux quatre coins du monde, qui ont pu travailler ensemble, à cette échelle, grâce au

réseau. Cette démarche, qui présente un certain nombre d'analogies avec la recherche scientifique – coopération internationale, publication, validation par les pairs, liberté de critiquer et d'amender… alors que, dans les temps anciens, Pythagore interdisait à ses disciples de divulguer théorèmes et démonstrations –, participe de l'émergence d'une nouvelle forme d'intelligence collective, qui répond à un besoin créé par l'augmentation de la complexité des objets industriels – développer un logiciel de plusieurs millions de lignes n'est plus à la portée d'une unique équipe de développeurs ou d'une unique entreprise.

Au-delà des logiciels, des licences libres existent aussi pour d'autres contenus immatériels : encyclopédies, ressources pédagogiques… Elles constituent des réponses en termes de droit d'auteur ayant l'objectif de fluidifier la circulation des documents et de faciliter le travail en commun. Elles sont des adaptations du droit des auteurs qui fournissent un cadre juridique au partage sur le Web des œuvres de l'esprit. À la manière de la GPL, les licences *Creative Commons* renversent le principe de l'autorisation obligatoire. Elles permettent à l'auteur d'autoriser par avance, et non au coup par coup, certains usages et d'en informer le public. Les droits donnés doivent correspondre à la nature de la ressource. Autant on peut enrichir collectivement un scénario de travaux pratiques au lycée, et donc permettre de le modifier, autant modifier un article d'opinion n'a pas de sens. Méta-licence, *Creative Commons* permet aux auteurs de se fabriquer des licences, dans une espèce de jeu de Lego simple constitué de seulement quatre briques. Première brique, Attribution : l'utilisateur, qui souhaite diffuser une œuvre, doit mentionner l'auteur. Deuxième brique, Commercialisation : l'auteur indique si son travail peut faire l'objet ou pas d'une utilisation commerciale. Troisième brique, Non-dérivation : un travail, s'il est diffusé, ne doit pas être modifié. Quatrième brique, Partage à l'identique : si l'auteur accepte que des modifications soient apportées à son travail, il impose que leur diffusion se fasse dans les mêmes termes que l'original, c'est-à-dire sous la même licence.

Cette nouvelle manière de produire et de diffuser des objets industriels préfigure peut-être une évolution plus globale de l'industrie et de la notion de propriété, dans un monde dans lequel de plus en plus de biens sont complexes et immatériels.

## Pour aller plus loin

Poursuivre la description des fonctionnalités des langages de programmation donnée dans ce chapitre nous mènerait à aborder d'autres fonction-

nalités : en particulier, la notion d'exception et la notion d'objet. La notion d'objet est relativement facile à comprendre une fois les notions de fonction et d'enregistrement assimilées, car un objet est essentiellement un enregistrement dont certains champs sont des fonctions. L'exemple le plus simple d'objet est un enregistrement à quatre champs : un champ `val` qui est un entier, un champ `print` qui est une fonction qui prend en argument un enregistrement et affiche la valeur de son champ `val`, un champ `reset` qui prend en argument un enregistrement et affecte la valeur `0` à son champ `val` et un champ `add` qui prend en argument un enregistrement et ajoute `1` à son champ `val`.

Ainsi, si `t` est un tel objet, le programme

```
t.reset(t);
t.add(t);
t.add(t);
t.print(t);
```

affiche la valeur `2`. Cet objet est un compteur, mais contrairement à un enregistrement qui aurait un unique champ `val`, il « sait » « lui-même » comment se réinitialiser, s'ajouter `1` ou s'afficher, puisque ces fonctions font partie de sa propre définition.

Bien souvent, les langages de programmation proposent une syntaxe spéciale pour appliquer un champ fonctionnel d'un objet à cet objet même et, dans la définition d'une telle fonction, une syntaxe spéciale pour désigner l'objet lui-même – un pronom réfléchi.

Les exceptions, les objets sont, par exemple, décrits, dans un style proche de celui adopté dans ce chapitre, dans

*Gilles Dowek, Les principes des langages de programmation, Les éditions de l'École polytechnique, 2008.*

Poursuivre l'étude des langages de programmation nous mènerait à aborder des langages plus spécialisés qui permettent par exemple de programmer des machines parallèles – des ordinateurs constitués de plusieurs petits ordinateurs qui effectuent des calculs en même temps. Un cas particulier est celui où chaque machine n'a qu'un nombre fini d'états possibles et, à chaque instant, évolue d'un état à un autre. Ces « systèmes dynamiques discrets » peuvent être *asynchrones*, ce qui signifie que chacun calcule à son rythme, ou *synchrones*, ce qui signifie qu'ils partagent une horloge commune. Entrent dans ce cadre les automates cellulaires, les réseaux de neurones, les pro-

grammes écrits dans des langages synchrones… Ces questions sont abordées dans

`http://fr.wikipedia.org/wiki/Automate_cellulaire` et `http://en.wikipedia.org/wiki/Esterel`.

Un dernier aspect de cette étude concerne la manière dont on réalise des interpréteurs et des compilateurs pour des langages que l'on conçoit soi-même. Ces questions sont abordées dans

Gilles Dowek, *Introduction à la théorie des langages de programmation,* Les éditions de l'École polytechnique, 2006.