

LANGAGES DE PROGRAMMATION

IFT-3000

BÉCHIR KTARI

Département d'informatique et de génie logiciel
Université Laval
Québec, QC, G1V 0A6, Canada

Avant propos

Ce document constitue les notes du cours «Langages de programmation» enseigné à l'Université Laval.

Première partie

Programmation fonctionnelle

Chapitre 1

Fondements de la programmation fonctionnelle

La programmation fonctionnelle est basée sur le paradigme fonctionnel de calcul qui considère un programme comme une fonction mathématique. Elle s'oppose à la programmation impérative qui, elle, est basée sur les notions de commande, d'affectation et d'exécution séquentielle.

L'idée de base est de considérer une fonction comme une boîte noire ayant une entrée (les paramètres de la fonction) et une sortie (le résultat de la fonction). Ainsi le programmeur se concentre sur l'application de règles simples de programmation et sur l'essence de son application, alors que le style de programmation impérative le force à se soucier des détails d'implémentation et de fonctionnement de l'ordinateur (adressage, espace mémoire, etc.).

De plus, une approche fonctionnelle et la mise à l'écart de l'affectation (effets de bord) permettent d'avoir un ensemble simple et homogène de concepts. Une fonction ne pourra en aucun cas produire un effet de bord comme il est possible de le faire dans un langage impératif comme le langage C. En effet, en mathématiques, étant donnée une fonction f , l'expression

$$f(1) - f(1)$$

vaudra toujours zéro, quel que soit l'ordre d'évaluation de cette expression (de droite à gauche ou de gauche à droite). Cependant, en considérant un langage à effet de bord (impératif) comme le C, non seulement ce type d'expression ne s'évaluera pas toujours à zéro mais en plus sa valeur pourra dépendre de l'ordre d'évaluation. Voici un exemple, en C, illustrant ces faits :

```
int x = 1;

int f(int y)
{
    x = x + y;
    return x;
}
```

Dans ce cas, si on évalue de gauche à droite, $f(1)-f(1)$ s'évalue en -1 ; si c'est de droite à gauche, ça s'évalue en 1 !

Ce chapitre est composé de deux sections. La première présente à l'aide d'exemples les fondements de la programmation fonctionnelle. Suit une description des caractéristiques des langages fonctionnels ainsi que leur historique.

1.1 Concepts

La principale composante de la programmation fonctionnelle est la fonction. Cette dernière est définie par trois composantes de base : le domaine, le co-domaine et la définition¹. La première est représentée par un ensemble de valeurs sur lesquelles la fonction peut être appliquée. Cet ensemble constitue le type de l'argument de la fonction. Le co-domaine est un ensemble contenant toutes les valeurs qui sont susceptibles d'être retournées par l'application de la fonction. Il représente le type du résultat de la fonction. La définition spécifie, à partir du domaine, comment les éléments du co-domaine sont construits. Une dernière composante optionnelle est le nom de la fonction.

Ainsi, pour définir une fonction dont le nom est *double*, il suffit de spécifier ses quatre composantes :

- Nom : *double*
- Domaine : *int*
- Co-domaine : *int*
- Définition : $x + x$.

Dans cet exemple, le terme *int* représente le type de tous les entiers naturels, c'est-à-dire \mathbb{N} . La notation mathématique équivalente pour représenter cette fonction est :

$$\begin{aligned} \text{double} : \text{int} &\longrightarrow \text{int} \\ x &\mapsto x + x \end{aligned}$$

Une fois définie, la fonction est perçue comme une boîte noire ayant une entrée de type *int* et une sortie de type *int* égale au double de l'entrée. L'utilisation de cette fonction consiste à lui fournir une valeur entière et à recevoir un résultat. L'application de cette fonction à une valeur du domaine, soit 3, s'écrit :

$$\text{double}(3)$$

Le résultat de cette expression est la valeur entière 6.

Dans la définition de cette fonction, l'opérateur d'addition “+” correspond aussi à une fonction. Une autre notation qui peut clarifier cette notion est la notation «préfixée» :

$$+(x, y)$$

Dans ce cas, il est clair que l'opérateur d'addition est une fonction. Ainsi, le corps d'une fonction correspond à un ensemble d'application de fonctions, déjà définies, à des valeurs appropriées (respectant le domaine des fonctions).

Une autre technique pour définir et appliquer les fonctions est le λ -calcul (lambda-calcul) développé par Alonzo Church dans les années 1930. La nouvelle forme de la définition d'une fonction est :

$$[\langle \text{nom de la fonction} \rangle \equiv] \lambda \langle \text{arguments} \rangle. \langle \text{definition} \rangle$$

Dans ce cas, la spécification d'un nom est optionnelle, ce qui permet de définir des fonctions sans avoir besoin de leur attribuer des noms. Par exemple, la définition de la fonction *double* devient :

$$\lambda x. x + x$$

L'application de cette fonction à la valeur 3 s'écrit :

$$(\lambda x. x + x)(3)$$

Toute variable, utilisée dans le corps d'une fonction, qui apparaît parmi les arguments de la fonction est désignée par “variable liée”. Lors de l'application d'une fonction à une expression donnée, cette variable est liée à la valeur résultant de l'évaluation de cette expression. Dans cet exemple, la variable x est liée à la valeur 3.

Une des principales caractéristiques des fonctions est la capacité de les composer entre elles pour en former une qui leur est équivalente. Soit *abs* une fonction qui calcule la valeur absolue d'un entier relatif. Elle est définie comme suit :

1. Couramment désignée par corps de fonction.

$$abs \equiv \lambda x. |x|$$

Mathématiquement, pour définir une fonction qui retourne la valeur absolue du double d'un entier relatif, il faut écrire :

$$f(x) \equiv abs (double (x))$$

En λ -calcul, cette composition est définie par un opérateur de composition de fonctions. En utilisant la notation du λ -calcul, la fonction f représentant la composition de abs et de $double$ est définie comme suit :

$$f \equiv abs \circ double$$

1.2 Historique et caractéristiques

C'est en 1958, que Mac Carthy inventa le premier langage de programmation fonctionnelle appelé Lisp (*List Processing*) dont les programmes ont une similarité avec les termes du λ -calcul. Quelques années plus tard, et plus précisément en 1966, P. Landin proposa le langage ISWIM (*If You See What I Mean*) qui constituera la base pour plusieurs langages fonctionnels futurs. Puis en 1978, J. Backus présenta le langage FP (*Functional Programming*) ayant comme principale caractéristique l'absence de noms de variables. La même année, R. Milner (Prix Turing²) proposa un langage appelé ML (*Meta Language*) qui est inspiré de ISWIM et qui possède un système de types original basé sur la dite synthèse de types. Depuis, ML est devenu le prototype des langages fonctionnels.

Tous ces langages partagent les caractéristiques suivantes :

- Un ensemble de fonctions primitives : celles-ci correspondent à des fonctions préalablement définies dans le langage. Elles peuvent être appliquées à des arguments ou utilisées pour la construction d'autres fonctions. Par exemple, l'opérateur $+$ est une fonction primitive.
- Un ensemble de formes fonctionnelles : celles-ci sont des fonctions qui acceptent d'autres fonctions comme paramètres. Elles sont utilisées pour la construction de nouvelles fonctions à partir des fonctions déjà définies. Par exemple, la composition des fonctions est une forme fonctionnelle. Elle peut être définie comme suit :

$$\circ \equiv \lambda f. (\lambda g. (f (g)))$$

De sorte que :

$$(abs \circ double) (x) \equiv abs (double (x))$$

- Un opérateur d'application : il correspond à une forme fonctionnelle spéciale qui prend comme paramètres une fonction et un ensemble de valeurs de paramètres, et retourne le résultat de l'application de la fonction aux paramètres. Par exemple, la fonction suivante prend en argument une autre fonction qu'elle applique au deuxième argument :

$$F \equiv \lambda f. (\lambda x. (f (x)))$$

L'expression $F (double) (4)$ est équivalente à $double (4)$. Les deux s'évaluent en la valeur entière 8.

- Un ensemble de données primitives et de constructions : ces données primitives correspondent aux types de base tels que les entiers, les réels, les caractères, etc. De même, il est nécessaire d'avoir des constructeurs pour manipuler des données plus complexes telles que les listes ou les tableaux de valeurs.
- Des opérateurs de liaison : ils correspondent aux opérateurs **let**, **rec** et λ . Ils lient des identificateurs à des valeurs fonctionnelles.

2. Ce prix est considéré comme étant l'équivalent du prix Nobel en informatique.

- Un mécanisme de gestion implicite de la mémoire : en effet, la programmation fonctionnelle fait abstraction des effets de bord produits, par exemple, par l'affectation. Il est donc nécessaire de disposer d'un mécanisme qui gère en arrière-plan la mémoire.
- Les fonctions sont des valeurs de première classe : dans ce cas, les fonctions sont traitées comme n'importe quel objet du langage. En particulier, elles peuvent être utilisées comme paramètres et retournées comme résultat d'évaluations d'autres fonctions. Par exemple, la définition suivante

$$P \equiv \lambda f.(\lambda g.(\lambda x.(f (g (x)))))$$

définie une fonction qui prend un paramètre fonctionnel f et retourne une fonction $\lambda g.(\lambda x.(f (g (x))))$ qui prend à son tour un paramètre fonctionnel g et retourne une autre fonction $\lambda x.(f (g (x)))$. Cette dernière attend un paramètre x et retourne l'expression $f (g (x))$, qui correspond à l'application de f au résultat de l'application de la fonction g à x .

Mais la principale caractéristique de ces langages est qu'ils partagent le même modèle mathématique. Ce modèle repose sur deux concepts majeurs : l'abstraction et l'application. Les deux chapitres qui suivent présentent un langage de programmation, nommé OCaml, basé sur le paradigme fonctionnel.

Chapitre 2

ML

En 1978, Robin Milner développa, à Edinburgh, un langage appelé ML qui est inspiré des travaux de Landin (voir section 1.2, page 5). À l'origine, ce langage était le méta-langage du système de preuves LCF (*Logic for Computable Functions*) : Définition de stratégies de preuves, manipulation d'objets de type «théorème», «règle», etc. Depuis, ML est devenu le prototype des langages fonctionnels.

En 1980, la standardisation de ML eut lieu en empruntant quelques idées du langage fonctionnel HOPE développé par Burstall et al. Cette standardisation donna naissance au langage SML (*Standard ML*). D'autres versions de ML existent : LML (Göteborg), Caml et OCaml (Paris), etc., et beaucoup plus récemment F# et Reason¹. Une des originalités de ces langages est leur système de types basé sur la dite synthèse de types (*type inference*).

Concernant ML², qui fait l'objet de ce chapitre, ses principales caractéristiques sont :

- Langage de programmation fonctionnelle : les fonctions sont des valeurs de première classe : elles peuvent être utilisées comme paramètres, retournées comme résultat d'évaluation d'autres fonctions et stockées dans des structures de données (listes de fonctions, arbres contenant des fonctions, enregistrements comprenant des fonctions, etc.).
- Langage interactif : chaque expression est analysée (typée), compilée et évaluée. Finalement, la valeur ainsi que son type sont retournés.
- Fortement typé : pour éviter toute sorte d'erreurs à la phase d'exécution, ML associe à chaque expression du langage un type qui est déterminé automatiquement à la compilation. De plus, la conversion implicite de type est interdite.
- Un système de typage polymorphe : un type est polymorphe lorsqu'il comporte des variables de types. Ainsi, une fonction qui prend en argument une valeur de type polymorphe, peut être appliquée à des valeurs de différents types. Par exemple, dans un tel système, on attribue à la fonction identité le type $'a \rightarrow 'a$. Ce type signifie que la fonction prend une valeur de type $'a$ et rend une valeur de même type. Le type $'a$ signifie que nous sommes en présence d'un type polymorphe. Grâce au polymorphisme, il est possible d'appliquer cette fonction à des valeurs de différents types : *int*, *float*, *bool*, etc.
- Portée statique des identificateurs : ML résout les références des identificateurs à la compilation, assurant ainsi des programmes plus efficaces et plus modulaires.
- Mécanisme de traitement d'exceptions : les exceptions permettent de traiter les effets prévus durant l'exécution.
- Modules : les modules permettent de structurer et de définir proprement des interfaces. Un module est une collection de déclarations vue comme un tout. Il permet la construction de programmes par incrémentation, en utilisant et en liant un ensemble de modules différents.

1. De nos jours, on retrouve les aspects du paradigme fonctionnel dans la plupart des langages de programmation couramment utilisés : Java, C++, C#, Python, Javascript, Ruby, Scala, Go (de Google), etc.

2. Quand on mentionne ML, on fait référence à toute la famille des langages qui en sont issus, en particulier du langage présenté dans ce chapitre : OCaml.

Ces caractéristiques se retrouvent dans les différentes parties de ce chapitre. La syntaxe présentée dans ce chapitre est celle de OCaml. Il ne s’agit pas d’une description complète et définitive du langage. Pour le lecteur intéressé, voici quelques références³ :

1. Manuel d’utilisation du langage.
Disponible sur le Web : <http://caml.inria.fr/pub/docs/manual-ocaml>
2. Pierre Weis et Xavier Leroy⁴, Le langage Caml.
Excellent livre qui se limite à la partie fonctionnelle et impérative du langage (ne traite pas les modules, ni l’aspect orienté objet).
Disponible sur le Web : <http://caml.inria.fr/pub/distrib/books/llc.pdf>
3. Philippe Narbel. Programmation fonctionnelle, générique et objet - Une introduction avec le langage Ocaml, Vuibert informatique, 2005.
Disponible à la bibliothèque scientifique de l’Université Laval : QA 76.6 N218 2005.
4. Tomas Petricek, Jon Skeet. Real-World Functional Programming, Manning, 2010 (<http://www.manning.com/petricek>). Excellent ouvrage qui traite de la programmation fonctionnelle dans l’environnement .NET (F# et C#).
5. Page principale du langage : <http://www.ocaml.org>

Le chapitre est organisé comme suit :

- une première section présente très brièvement le langage, notamment l’interpréteur ;
- plusieurs sections sont consacrées à l’aspect «Données» du langage : types de base, types composés, types algébriques, abréviation de types, et définition d’identificateurs ;
- plusieurs sections sont consacrées à l’aspect «Comportements» du langage : filtrage par motifs et fonctions, ainsi que plusieurs notions adjacentes ;
- une section présente les aspects impératifs du langage : références, structures de contrôle, tableaux et exceptions ;
- une section présente la notion de modules qui réunit les aspects «Données» et «Comportements» : modules, signatures et foncteurs ;
- avant de conclure, une section présente quelques commandes utiles disponibles dans l’interpréteur.

2.1 Premiers pas avec OCaml

La plupart des implémentations de ML sont interactives. À l’instar du plusieurs langages modernes, cette interaction est basée sur une boucle *lire, évaluer et afficher*, c’est-à-dire que ML lit une expression saisie par l’utilisateur, l’analyse, la compile, l’évalue, puis affiche le résultat à l’écran ou sur la console. Ce résultat est composé d’une information statique (le type de l’expression analysée) et d’une information dynamique (la valeur du résultat).

À chaque fois que l’évaluateur est prêt pour traiter une nouvelle expression, il affiche le caractère «#», communément appelé *prompt*. Une fois l’expression saisie, l’utilisateur doit taper les caractères «;;» suivi d’un retour chariot pour indiquer à l’évaluateur que l’expression est prête pour être évaluée. Par exemple :

```
# 3 + 3;;
- : int = 6
#
```

Dans cet exemple, l’expression $3 + 3$ est évaluée. Le résultat de cette évaluation est composé de trois parties :

3. Hormis 2., toutes ces références traitent aussi de l’aspect orienté objet du langage, aspect traité dans le chapitre consacré à **Objective ML**.

4. Xavier Leroy est un des concepteurs des langages Caml et OCaml.

- Le tiret «`-`» précise que le résultat de l'expression n'est liée à aucun identificateur. Il est aurait été possible que l'expression à évaluer ait comme résultat de lier une valeur à un identificateur. Par exemple :

```
# let n = 4;;
val n : int = 4
```

Dans ce cas, l'interpréteur OCaml précise que le résultat de l'évaluation est un identificateur *n* de type *int* et est lié à la valeur 4. Si on interroge de nouveau l'interpréteur au sujet du contenu de *n*, il retourne alors sa valeur :

```
# n;;
- : int = 4
```

et nous précise cette fois-ci que le résultat ne produit aucun nouvel identificateur.

- La valeur 6 précédée par l'opérateur d'égalité `=` correspond à l'évaluation dynamique de l'expression. Ainsi, l'addition des valeurs entières 3 et 3 vaut 6.
- Le type *int* précédé par deux points (`:`) correspond à l'évaluation statique de l'expression. Ce type indique que le résultat de l'addition des deux valeurs entières 3 et 3 est un entier.

La dernière ligne représente l'évaluateur en état d'attente : le symbole «`#`» indique à l'utilisateur qu'il peut à nouveau saisir une expression.

Il est possible aussi de saisir une expression sur plusieurs lignes. En fait, tant que l'utilisateur ne saisit pas les caractères «`;;`» suivi du retour chariot, signifiant la fin d'une expression, l'évaluateur n'intervient pas. Par exemple :

```
# if (4 > 5)
then
  5
else
  6;;
- : int = 6
```

Par ailleurs, l'évaluateur signale les erreurs syntaxiques ou sémantiques (en général des erreurs de typage) lors de l'analyse d'une expression :

```
# 3 + 2.0;;
Characters 4-7:
  3 + 2.0;;
  ----
Error: This expression has type float but an expression was expected of type int
```

Dans cet exemple, l'évaluateur indique que 2.0 est une valeur de type *float* alors qu'elle est utilisée comme une valeur de type *int* (puisqu'additionnée, avec l'opérateur `+` défini sur les entiers, à l'entier 3).

Par contre, les expressions suivantes sont acceptées par l'évaluateur :

```
# 3 + 2;;
- : int = 5

# 3.0 +. 2.0;;
- : float = 5.

# 3. +. 2.;;
- : float = 5.
```

Ces trois exemples évoquent le fait que le langage OCaml ne supporte pas la propriété appelée surcharge (ou *overloading*). En effet, l'opérateur `+` est utilisé pour additionner deux entiers alors que `+.` est utilisé pour additionner deux nombres réels.

Inférence de types : Hormis pour quelques exceptions (comme pour la définition d'un enregistrement, section 2.3.2, page 23), Ocaml ne nécessite pas que le programmeur précise des types lorsqu'il définit un identificateur (lié à une valeur quelconque) ou une fonction. Le type le plus général⁵ est inféré automatiquement. Par exemple, si vous écrivez une fonction de tri et que votre code s'avère générique, OCaml inférera le type le plus général pour cette fonction, soit un type générique.

Typage explicite : Notons qu'il est possible de typer explicitement n'importe quelle expression ou valeur :

```
# 2;;
- : int = 2

# (2 : int);;
- : int = 2

# 2 + 3;;
- : int = 5

# (2 : int) + (3 : int);;
- : int = 5

# ((2 : int) + (3 : int) : int);;
- : int = 5

# ((+): int -> int -> int) (2 : int) (3 : int);;
- : int = 5

# let n : int = 4;;
val n : int = 4

# let n : int = (4 : int);;
val n : int = 4
```

Clairement, Ocaml supporte les deux extrêmes : le typage explicite, à la sous-expression près ; l'absence de typage et l'inférence de types.

Abréviation de types : Le mot-clé **type** offre la possibilité au programmeur de définir ses propres types, notamment des abréviations de types existants :

```
# type entier = int;;
type entier = int

# let n : entier = 5;;
val n : entier = 5
```

Les sections qui suivent présentent plus en détail les différentes caractéristiques de ce langage en les illustrant avec des exemples.

2.2 Données : Types de base

Un type n'est qu'une collection de valeurs. Les types servent à classer des valeurs. À titre d'exemple, le type *int* sert à classer les entiers relatifs, le type *float* sert à classer les nombres réels, le type *bool* sert à classer les booléens, le type *string* sert à classer les chaînes de caractères, etc. Nous disons que la valeur 2 est de type *int*, que 2.0 est de type *float*, alors que "Hello" est de type *string*.

5. La notion de «type le plus général» est étudiée plus en détail au chapitre 4.

Le type d'une expression est déterminé par un ensemble de règles qui garantissent que si cette expression s'évalue en une valeur, alors le type de cette valeur correspond au type assigné à l'expression. Par exemple, le système de types de OCaml assigne à l'expression `3 + 4` le type *int* (puisque'il s'agit de la somme de deux entiers). Lorsque cette expression est évaluée, elle retourne la valeur 7 qui a un type conforme au type inféré, soit *int*.

Cette section présente brièvement les différents types de base supportés par le langage OCaml. Chaque type est présenté avec ses propriétés et les principales fonctions (ou opérateurs) qui lui sont définies.

2.2.1 Type *unit*

Le type *unit* consiste en une seule valeur notée `()`. Il est utilisé lorsqu'une expression n'a pas de valeur ou lorsqu'une fonction n'a pas d'arguments. Prenons l'exemple suivant :

```
# print_string;;
- : string -> unit = <fun>
```

Dans cet exemple, on demande à l'interpréteur d'évaluer l'expression `print_string`. Cette dernière correspond à l'identificateur d'une fonction prédéfinie qui affiche à l'écran une chaîne de caractères passée en argument. La réponse de l'interpréteur est que la valeur de cette expression est une fonction et que son type est *string -> unit*. Ce type précise que l'argument de la fonction est une chaîne de caractères tandis que son résultat est une valeur de type *unit*. Dans ce cas là on devrait parler de procédure ou de commande `print_string` plutôt que de fonction. Voici un exemple d'utilisation de cette commande :

```
# print_string "Hello_World" ;;
Hello World- : unit = ()
```

Comme on peut le constater :

- le message est affiché;
- le résultat de l'expression est `()`;
- son type est *unit*.

L'exemple qui suit permet de mieux distinguer le résultat de l'évaluation de l'expression de l'effet de bord produit par cette évaluation, c'est-à-dire l'affichage à l'écran de la chaîne de caractères :

```
# print_string "Hello_World\n" ;;
Hello World
- : unit = ()
```

2.2.2 Booléens

Le type *bool* est constitué des valeurs `true` et `false` qui correspondent aux valeurs booléennes usuelles vrai et faux. En général, une valeur booléenne résulte de l'évaluation d'une comparaison entre deux objets. Par exemple :

```
# 4 = 4;;
- : bool = true

# 1.5 > 2.5;;
- : bool = false
```

OCaml propose six opérateurs de comparaisons ou opérateurs relationnels :

= pour l'égalité,
 <> pour l'inégalité,
 < pour l'infériorité stricte,
 > pour la supériorité stricte,
 <= pour l'infériorité ou égal,
 >= pour la supériorité ou égal.

Ces opérateurs, et notamment les opérateurs d'égalité et d'inégalité s'appliquent à la plupart des types d'objets en OCaml : entiers, réels, etc. Cependant, il n'est pas possible de les appliquer pour comparer des fonctions, même si ces dernières retournent les mêmes résultats pour les mêmes arguments. Par exemple, appliqués aux fonctions prédéfinies `min` et `max`, on obtient une erreur (une exception est soulevée ; voir section 2.7.5, page 69) :

```
# min = max;;
Exception: Invalid_argument "compare: functional value".
```

Plusieurs opérateurs booléens sont définis en OCaml, permettant d'évaluer des expressions composées de sous-expressions booléennes :

- **Conjonction** : L'opérateur «standard» `&&` permet de décrire des expressions conjonctives, c'est-à-dire des expressions qui ne sont vraies que si toutes les sous-expressions qui la composent sont vraies. Par exemple, la notation mathématique suivante permet de tester si une valeur n est comprise entre deux valeurs entières 0 et 10 :

$$n \in [0, 10[\quad \text{ou} \quad 0 \leq n < 10$$

En OCaml, il faut connecter deux expressions booléennes, comme suit :

```
# let n = 4;;
val n : int = 4

# 0 <= n && n < 10;;
- : bool = true

# 0 <= n-5 && n < 10;;
- : bool = false
```

La première expression, `let n = 4`, permet de définir un identificateur `n` en lui associant une valeur 4. La deuxième expression, `0 <= n && n < 10` teste si cette valeur appartient à l'intervalle délimité par les valeurs entières 0 et 10. Le résultat de l'évaluation de cette expression est `true`. Pour aboutir à ce résultat, l'évaluateur procède comme suit :

- il évalue la sous expression `0 <= n`. Celle-ci est vraie pour une valeur `n` égale à 4 ;
- puis il évalue la sous expression `n < 10` ; de même, elle est vraie pour une valeur `n` égale à 4 ;
- il retourne le résultat de l'évaluation de l'expression intermédiaire `true && true`, c'est-à-dire `true`.

La dernière expression, `0 <= n-5 && n < 10`, est évaluée en la valeur booléenne `false`. Dans cette expression, seule la sous-expression `0 <= n-5` sera en réalité évaluée puisque l'opérateur `&&` bénéficie de la propriété suivante :

false && e \equiv false

Cette propriété stipule que pour une expression conjonctive, dès qu'une de ses sous-expressions (en évaluant les sous-expressions de gauche à droite) est évaluée à `false`, il n'est pas nécessaire d'évaluer le reste de l'expression. L'exemple qui suit en est une illustration :

```
# 3 < 2 && (print_string "Hello\n"; true);;
- : bool = false
```

- **Disjonction** : L'opérateur «standard» `||` permet de décrire les expressions disjonctives, c'est-à-dire celles composées de sous-expressions séparées par cet opérateur. Une expression

disjonctive est vraie si au moins une de ces sous-expressions est vraie. En reprenant l'exemple précédent avec la même valeur de n , nous avons :

```
# (0 <= n - 5) || (n < 10);;
- : bool = true

# 0 <= n - 5;;
- : bool = false
```

De la même manière que pour l'opérateur précédent, quelque soit l'évaluation d'une expression booléenne e :

$$\text{true} \parallel e \equiv \text{true}$$

C'est-à-dire que dès qu'on évalue une sous-expression (de gauche à droite) à **true**, il n'est pas nécessaire d'évaluer le reste de l'expression disjonctive.

- **Négation** : Le mot-clé **not** permet de retourner l'inverse d'une expression booléenne. Si elle est vraie, alors l'opérateur de négation appliqué à l'expression retourne **false** et vice-versa. Par exemple :

```
# not false;;
- : bool = true

# not (n < 10);;
- : bool = false
```

Les valeurs booléennes résultantes de l'application de l'un de ces opérateurs à des expressions booléennes e_1 et e_2 sont résumées dans le tableau 2.1.

e_1	e_2	$e_1 \&\& e_2$	$e_1 \parallel e_2$	not e_2
false	false	false	false	true
false	true	false	true	false
true	false	false	true	
true	true	true	true	

TABLE 2.1 – Table de vérité.

En général, une expression booléenne est utilisée dans l'expression conditionnelle

if e **then** e_1 **else** e_2

Cette expression s'évalue comme suit :

- évaluation de l'expression booléenne e :
 - si cette évaluation vaut **true** alors l'évaluateur retourne le résultat de l'évaluation de l'expression e_1 ;
 - sinon, il retourne le résultat de l'évaluation de e_2 .

L'évaluateur statique s'assure que le type de e est booléen et que les expressions e_1 et e_2 ont le même type. On peut exprimer cette règle de manière plus formelle :

$$\frac{\mathcal{E} \vdash e : \text{bool} \quad \mathcal{E} \vdash e_1 : t \quad \mathcal{E} \vdash e_2 : t}{\mathcal{E} \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t} \text{ (if)}$$

Ceci est dû au fait que cette forme syntaxique est une expression et non une commande. Ainsi, il est possible de lier à un identificateur le résultat d'une expression conditionnelle :

```
# let n = if false then false else true;;
val n : bool = true
```

En présence de deux types identiques (dans cet exemple, il s'agit du type *bool*), l'évaluateur statique affecte le type *bool* à cet identificateur.

Par contre, si le type de e_1 est différent de celui de e_2 , alors il ne peut typer l'expression conditionnelle :

```
# if 3 < 2 then false else ();;
Characters 25-27:
  if 3 < 2 then false else ();;
  ~~~~~
Error: This variant expression is expected to have type bool
       There is no constructor () within type bool
```

À l'instar d'autres langages, il est possible d'utiliser une expression conditionnelle en ne précisant que sa clause **then**. La seule contrainte est que le bloc de cette clause doit être de type *unit* :

```
# if 5 = 2 + 3 then print_string "Je_suis_de_type_unit\n";;
Je suis de type unit
- : unit = ()

# if 5 = 2 + 3 then "Je_suis_de_type_string";;
~~~~~
Error: This expression has type string but an expression was expected of type unit
       because it is in the result of a conditional with no else branch
```

Voici d'autres exemples d'utilisation de l'expression conditionnelle :

```
# if (false || true) then true else false;;
- : bool = true

# if true
then
  if false then true else false
else
  true;;
- : bool = false
```

2.2.3 Entiers

Le type *int* représente un sous-ensemble des entiers relatifs, c'est-à-dire \mathbb{Z} . Plusieurs opérateurs sont définis sur les entiers :

- Addition : L'opérateur binaire $+$ permet de calculer la somme de deux entiers passés en argument. Par conséquent, il n'est pas possible d'utiliser cet opérateur pour additionner un entier avec un réel. Par exemple :

```
# 3 + 4;;
- : int = 7

# 4 + (-6);;
- : int = -2

# 2 + 2.0;;
Characters 4-7:
2 + 2.0;;
~~~~~
Error: This expression has type float but an expression was expected of type int
```

- Multiplication : L'opérateur binaire $*$ permet de calculer le produit de deux entiers⁶. Son type est identique à celui de $+$. Par exemple :

6. À ne pas confondre avec l'opérateur sémantique $*$ qui est utilisé pour représenter le type des tuples.

```
# 5 * 6;;
- : int = 30

# 5 * (3 + 7);;
- : int = 50
```

- Soustraction : L'opérateur binaire `-` permet de soustraire deux entiers. Par exemple :

```
# 4 - 6;;
- : int = -2

# 4 - -2;;
- : int = 6

# 4 - - 2;;
- : int = 6
```

- Division : L'opérateur binaire `/` calcule la division entière de deux entiers. Cet opérateur provoque une erreur (exception) si le deuxième argument est égal à zéro. Par exemple :

```
# 8 / 2;;
- : int = 4

# 7 / 4;;
- : int = 1

# 4 / 0;;
Exception: Division_by_zero.
```

- Reste : Pour calculer le reste d'une division, il faut utiliser l'opérateur binaire `mod` comme suit :

```
# 7 mod 3;;
- : int = 1

# 8 mod 4;;
- : int = 0

# 5 mod 0;;
Exception: Division_by_zero.
```

Cet opérateur provoque aussi une erreur si le deuxième argument vaut zéro. Par ailleurs, il peut être utilisé pour tester si un entier est un multiple d'un autre :

```
# 3600 mod 225;;
- : int = 0
```

Puisque le résultat de la division entière de la valeur 3600 par la valeur 225 est égal à 0, alors la première valeur est un multiple de la deuxième. Ainsi, pour décider si un entier `n` est pair, il suffit d'utiliser cet opérateur comme suit : $(n \bmod 2) = 0$. Ce test permet de vérifier si la division de l'entier `n` par la valeur 2 a un reste nul. Si c'est le cas, il est pair sinon il est impair.

D'autres fonctions sont définies sur les entiers telles que `max`, `min` et `abs` qui permettent respectivement de calculer la valeur maximum et minimum parmi deux entiers, et la valeur absolue d'un entier donné :

```
# min 3 5;;
- : int = 3

# max 3 5;;
- : int = 5

# abs 4;;
- : int = 4

# abs (-4);;
- : int = 4
```

Notez que OCaml offre plusieurs autres fonctions qui s'appliquent sur les entiers. Il offre aussi deux modules, `Int32` et `Int64`, permettant de manipuler de grands nombres entiers. Pour bénéficier des fonctions définies dans ces modules, il suffit d'ouvrir le module (section 2.8, page 72) comme suit :

```
# open Int32;;
```

2.2.4 Réels

Les valeurs réelles sont représentées par le type *float*. À l'instar des entiers, plusieurs opérateurs et fonctions sont définis sur ce type. Pour additionner, soustraire, multiplier ou diviser deux valeurs réelles, il suffit d'utiliser les opérateurs `+`, `-`, `*`, et `/`. Les exemples qui suivent illustrent l'utilisation de ces opérateurs :

```
# 3.0 +. 4.2;;
- : float = 7.2

# 2.0 +. -2.0;;
- : float = 0.

# 5.1 *. 1.;;
- : float = 5.1

# 8. /. 2.25;;
- : float = 3.55555555555555536

# 5. -. 3.4;;
- : float = 1.6

# abs_float 4.5;;
- : float = 4.5

# abs_float (-4.);;
- : float = 4.
```

D'autres fonctions propres aux valeurs réelles sont définies :

- `cos` : calcule le cosinus d'une valeur donnée.
- `sin` : calcule le sinus d'une valeur donnée.
- `log` : calcule le logarithme d'une valeur donnée.
- `sqrt` : calcule la racine carrée d'une valeur donnée.
- `exp` : calcule l'exponentiel d'une valeur donnée.

Pour illustrer l'utilisation de ces fonctions, voici quelques exemples :

```
# (cos (45.) *. cos (45.)) +. (sin(45.) *. sin(45.));
- : float = 1.

# log 1.;;
- : float = 0.

# sqrt 16.;;
- : float = 4.

# exp 1.;;
- : float = 2.71828182845904509

# exp 0.;;
- : float = 1.

# exp 2;;
  Characters 4-5:
    exp 2;;
    ~
Error: This expression has type int but an expression was expected of type float
Hint: Did you mean '2.'?

# log 0.;;
- : float = neg_infinity
```

Les deux derniers exemples illustrent respectivement que la fonction `exp` n'accepte que des valeurs réelles et que la fonction `log`, appliquée à la valeur 0.0, retourne la valeur mathématique `neg_infinity`. Il est effectivement possible d'utiliser deux valeurs constantes représentant $+\infty$ et $-\infty$:

```
# infinity;;
- : float = infinity

# 1. +. infinity;;
- : float = infinity

# neg_infinity;;
- : float = neg_infinity

# infinity +. neg_infinity;;
- : float = nan

# nan;;
- : float = nan
```

La valeur constante `nan` représente un nombre non défini (*not a number*).

Deux fonctions sont définies pour permettre de transformer des valeurs réelles en valeurs entières et vice-versa : `float_of_int` et `int_of_float`. Les exemples qui suivent illustrent l'utilisation de ces fonctions :

```
# float_of_int 2;;
- : float = 2.

# float_of_int (-3);;
- : float = -3.

# int_of_float 1.5;;
- : int = 1

# int_of_float (-2.5);;
- : int = -2
```

```
# float_of_int (int_of_float 2.0);;
- : float = 2.

# sqrt (float_of_int 16);;
- : float = 4.
```

2.2.5 Caractères

Le type de base, *char*, permettant de représenter les caractères.

Pour transformer un nombre ASCII en un caractère, il faut utiliser la fonction `chr` comme suit (remarquons qu'il est nécessaire d'ouvrir le module `Char` pour bénéficier de cette fonction; autrement, il aurait fallu utiliser la notation point, c'est-à-dire `Char.chr`) :

```
# open Char;;

# chr 77;;
- : char = 'M'

# chr 76;;
- : char = 'L'

# chr 0;;
- : char = '\000'

# chr 127;;
- : char = '\127'
```

La fonction duale à celle-ci est la fonction `code` qui retourne le code ASCII d'un caractère donnée :

```
# code '{';;
- : int = 123

# code '}';;
- : int = 125

# code (chr 123);;
- : int = 123
```

Il existe aussi des fonctions équivalentes : `char_of_int` et `int_of_char` :

```
# char_of_int 77;;
- : char = 'M'

# int_of_char 'M';;
- : int = 77
```

2.2.6 Chaînes de caractères

Le type *string* consiste en l'ensemble des chaînes finies de caractères. Les chaînes de caractères sont écrites entre guillemets. Par exemple :

```
# "Hello";;
- : string = "Hello "

# "";;
- : string = ""
```

La deuxième expression, "", représente la chaîne vide, c'est-à-dire composée d'aucun caractère ou dont la taille est nulle. Les caractères qui composent une chaîne de caractères correspondent à ceux du code ASCII dont les valeurs appartiennent à l'intervalle [0, 255]. Il est possible de spécifier le code ASCII d'un caractère à l'intérieur d'une chaîne de caractères grâce à l'utilisation du symbole «\» suivi du code ASCII sur trois chiffres :

```
# "\077\076" ;;
- : string = "ML"
```

D'autres fonctions sont définies sur les chaînes de caractères :

- Taille : La fonction **length** calcule la taille d'une chaîne de caractères. Son utilisation est simple comme l'illustre les exemples suivants :

```
# open String ;;

# length "" ;;
- : int = 0

# length "Hello" ;;
- : int = 5

# length "\077\076" ;;
- : int = 2

# length "077076" ;;
- : int = 6

# length "\\077\\076" ;;
- : int = 8
```

L'avant dernier exemple calcule la taille d'une chaîne composée des caractères M et L dont le code ASCII est respectivement égal à \077 et \076. Par contre, le dernier exemple calcule la taille d'une chaîne composée des caractères 0, 7, 7, 0, 7 et 6.

- Concaténation : Pour concaténer deux chaînes de caractères, il suffit d'utiliser l'opérateur préfixé ^ comme suit :

```
# "Hello" ^ " " ^ "World" ;;
- : string = "Hello World"
```

- Sous-chaîne : Pour extraire une sous-chaîne de caractères d'une chaîne de caractères donnée, il faut utiliser la fonction **sub**. Cette fonction admet trois paramètres : le premier désigne la chaîne de caractères en question ; le deuxième paramètre désigne la position dans la chaîne à partir de laquelle il faut extraire ; le troisième paramètre spécifie le nombre de caractères à extraire à partir du caractère dont la position est spécifiée par le deuxième paramètre :

```
# sub "Hello" 2 2 ;;
- : string = "ll"

# "Y" ^ (sub "Hello" 1 4) ^ "w" ;;
- : string = "Yellow"

# sub "Hello" 0 1 ;;
- : string = "H"
```

Notons que la première position d'une chaîne de caractères est 0 et non 1.

- Ordre lexicographique : Les opérateurs relationnels définis pour les entiers et les réels (<, <=>, =, ...) peuvent être utilisés pour comparer les chaînes de caractères.

Pour comparer deux entiers, il faut se baser sur l'ordre défini sur les entiers relatifs : 2 est plus petite que 3 qui est plus petite que 4 et ainsi de suite. Concernant les chaînes de caractères, il faut se baser sur l'ordre des codes ASCII associés aux différents caractères qui composent la chaîne. Par exemple :

```
# "a" < "b";;
- : bool = true

# "A" < "a";;
- : bool = true

# "ab20" > "ab19";;
- : bool = true

# "Hello" = "H" ^ "\101110";;
- : bool = true
```

Puisque le code ASCII du caractère a est inférieur à celui de b, il en est de même concernant les chaînes de caractères "a" et "b". Pour comparer les deux chaînes "ab20" et "ab19", il faut parcourir d'une manière parallèle les deux séquences de caractères et les comparer une à une. Dès que l'un des caractères est inférieur, supérieur, ou égal à celui qui lui correspond (même rang dans les deux chaînes), alors il en est de même pour les chaînes correspondantes. Avant de clore cette section, notons qu'il existe un ensemble de fonctions permettant de créer un objet de type *string* à partir d'objets de type *int*, *float* ou *bool*. Par exemple :

```
# string_of_int (10 + 15);;
- : string = "25"

# string_of_float (sqrt 9.);;
- : string = "3."

# string_of_bool ("a" < "b");;
- : string = "true"

# "The_assertion_4<6_is_" ^ (string_of_bool (4 < 6));;
- : string = "The assertion 4 < 6 is true"
```

Par ailleurs, il existe une fonction, nommée `print_string` qui permet d'afficher à l'écran des objets de type *string*. Combinée aux fonctions précédentes, ceci permet d'afficher des objets de types *bool*, *int* et *float*.

Aussi, pour afficher les différents objets d'une liste d'entiers, il est nécessaire d'appliquer cette fonction à chaque élément de la liste. La section 2.3 présente les types composés de OCaml, en particulier le type *list*.

La section qui suit présente l'utilisation des identificateurs et des déclarations dans OCaml. Ils permettent d'associer ou de lier à des identificateurs des valeurs résultantes d'évaluations d'expressions.

2.3 Données : Types composés

Les types composés permettent de représenter des objets du monde réel. En effet, les entiers peuvent décrire des prix, des quantités ou des tailles. Les chaînes de caractères permettent de décrire des noms, des adresses, etc. Seulement, le monde réel est composé aussi d'objets plus complexes tels que :

- listes d'employés,
- individus,
- positions géographiques,
- etc.

Par exemple, pour représenter une position géographique, il est nécessaire de disposer d'une structure permettant d'accueillir deux réels (ou trois suivant le nombre de dimensions choisi) représentant les positions dans chaque axe géographique. Ainsi, la paire (3.2,5.4) désigne une position

géographique située à la position 3.2 sur le premier axe (axe des x), et à la position 5.4 sur le deuxième axe (axe des y).

Pour représenter des objets tels que des individus, il est adéquat d'avoir au sein d'une même structure la liste des différentes caractéristiques (ou champs) concernant cet individu. Par exemple, les champs «nom», «prenom», «adresse», «numero_tel», etc. conviennent pour décrire un individu. Une telle structure est appelée enregistrement.

Pour représenter une liste d'employés, il est nécessaire de disposer d'une structure de données permettant de créer et de manipuler des listes d'objets.

Ces trois structures sont définies dans le langage OCaml. Elles correspondent respectivement aux tuples, aux enregistrements et aux listes, et font l'objet de cette section. À cela, OCaml ajoute le type algébrique et inductif qui permettent de définir tous les types de données usuels.

Polymorphisme : Toutes les données ne sont pas forcément composées d'éléments ayant un type de base : un élément peut lui-même être d'un type composé ; il peut aussi être d'un type générique ou polymorphe, c'est-à-dire pouvant être défini avec des valeurs de différents types.

À l'instar d'autres langages, OCaml utilise des paramètres de type, ou variables de types, afin de préciser qu'un type est polymorphe. De telles variables de types correspondent à : 'a, 'b, 'c, etc. (qu'on nomme *alpha*, *beta*, *gamma*, etc.). Pour indiquer qu'un type composé est polymorphe, on précède son nom par la liste des variables de types ; ces dernières sont naturellement utilisées dans la description des types de certains éléments du type composé.

Dans les sous-sections qui suivent, plusieurs exemples de types polymorphes sont présentés.

2.3.1 Tuples

En OCaml, une séquence d'expressions séparées par des virgules et, optionnellement, le tout entouré par deux parenthèses, représente un tuple de valeurs. Ces valeurs correspondent au résultat de l'évaluation de chacune de ces expressions :

(e_1, \dots, e_n) ou e_1, \dots, e_n

L'évaluation dynamique de cette forme syntaxique est le tuple de valeurs suivant :

(v_1, \dots, v_n) ou v_1, \dots, v_n

où v_i correspond à l'évaluation dynamique de e_i . L'évaluation statique est de la forme :

$t_1 * \dots * t_n$

où t_i correspond à l'évaluation statique de e_i , c'est-à-dire au type de v_i , et ce grâce à la propriété de cohérence de la sémantique statique vis à vis de la sémantique dynamique.

Un tuple composé de deux éléments est appelé paire. Les exemples qui suivent présentent cette structure de données :

```
# (min 3 5, ());;
- : int * unit = (3, ())

# ("Hello", not false);;
- : string * bool = ("Hello ", true)

# "Hello", not false;;
- : string * bool = ("Hello ", true)

# (123, ("jean", "tremblay"), 123 mod 20);;
- : int * (string * string) * int = (123, ("jean ", "tremblay "), 3)
```

Le dernier exemple illustre l'utilisation d'un tuple dans un autre tuple. En effet, la paire ("Jean", "tremblay") correspond au deuxième élément du triplet (123, ("Jean", "tremblay"), 3).

Concernant le type, l'utilisation du symbole «*» est due au fait que le type d'un tuple correspond au produit cartésien des ensembles correspondant aux types des éléments de ce tuple. Par exemple,

si une paire comprend un élément de type *int* et un élément de type *bool*, alors son type correspond à l'ensemble des valeurs égales au produit cartésien d'un sous-ensemble de \mathbb{Z} et de l'ensemble composé des valeurs **true** et **false**.

Deux fonctions sont disponibles et permettent d'accéder au premier et au deuxième élément d'une paire (pour accéder aux éléments d'un tuple, il faut utiliser le filtrage par motifs, section 2.5, page 37) :

```
# fst (3,5);;
- : int = 3

# snd (3,5);;
- : int = 5
```

Les opérateurs = et <> peuvent être utilisés pour comparer deux tuples de même type, c'est-à-dire de même arité et dont les différents éléments de même position sont de même type. Ainsi, deux tuples sont égaux si leurs éléments de même position sont égaux. Par exemple :

```
# (2,3) = (4,6);;
- : bool = false

# (3,()) = (min 3 5,());;
- : bool = true

# (4,5,6) <> (6,5,4);;
- : bool = true

# 2,3 = 4,6;;
- : int * bool * int = (2, false, 6)
```

Polymorphisme : Il est possible de définir un type tuple polymorphe. Par exemple :

```
1 # type ('a, 'b) paire = 'a * 'b;;
2 type ('a, 'b) paire = 'a * 'b
3
4 # let p : ('a, 'b) paire = (6,7.1);;
5 val p : (int, float) paire = (6, 7.1)
6
7 # let p : (int, float) paire = (6,7.1);;
8 val p : (int, float) paire = (6, 7.1)
9
10 # let p : ('a, 'b) paire = ((3,"a"),(4.0, 'x', true));;
11 val p : (int * string, float * char * bool) paire =
12   ((3, "a"), (4., 'x', true))
```

- À la ligne 1, on définit, à l'aide du mot-clé **type**, un type polymorphe *paire* représentant des paires de valeurs de types quelconques. Ce type est doublement polymorphe puisqu'il admet deux paramètres de type, ou variables de type : *'a* et *'b*. Le fait d'utiliser deux variables de type différentes indiquent que le type de la première valeur des paires représentées n'est pas forcément identique à celui de la deuxième valeur de ces paires. Remarquons aussi la syntaxe utilisée : les variables de types sont à la gauche du type défini ; elles sont entourées de parenthèses et séparées de virgules.
- À la ligne 4, on définit un identificateur *p*, lié à une paire d'entiers, en indiquant explicitement qu'il s'agit d'une valeur de type *('a, 'b) paire*. Le type inféré par OCaml précise qu'il s'agit, dans ce cas précis, d'une valeur de type *(int, float) paire* puisqu'en réalité la paire est composée d'un entier et d'un réel (*float*) ; on dit alors que les variables de types *'a* et *'b* ont été instanciées par les types *int* et *float*. Dit autrement, elles sont considérées dans ce contexte (la paire (6,7.1)) respectivement comme un entier et un réel.

- À la ligne 7, on donne un exemple pour indiquer qu’il est possible, lors de la définition de l’identificateur `p`, d’expliciter de manière plus concrète son type : *(int, float) paire* (on aura donc implicitement instancié les deux variables de type). Dans ce cas, le type inféré est le même.
- Le dernier exemple illustre comment les deux variables de type peuvent être instanciées par des types composés (respectivement paire et triplet).

Voici un autre exemple de définition d’un type tuple polymorphe :

```
# type 'a coord3 = 'a * 'a * 'a;;
type 'a coord3 = 'a * 'a * 'a

# let c : 'a coord3 = (12,15,33);;
val c : int coord3 = (12, 15, 33)

# let c' : 'a coord3 = (26.2,14.1,16.9);;
val c' : float coord3 = (26.2, 14.1, 16.9)
```

Dans ce cas, on définit un type polymorphe *coord3* en précisant qu’il s’agit d’un triplet de valeurs d’un type quelconque ; le fait d’utiliser une unique variable polymorphe, *'a*, nous permet de préciser que les trois valeurs du triplet doivent être du même type. Ainsi, dans un cas, on définit un triplet d’entiers ; dans l’autre, il s’agit d’un triplet de réels. Les types inférés sont respectivement *int coord3* et *float coord3*.

Si on définit un triplet ayant des valeurs de types différents, on obtient évidemment une erreur :

```
# let c'' : 'a coord3 = (12,15,10.2);;
Characters 29-33:
let c'' : 'a coord3 = (12,15,10.2);;
-----
Error: This expression has type float but an expression was expected of type int
```

Notons ici que l’erreur est provoquée par le fait qu’on indique explicitement le type de l’identificateur *c''* que l’on veut définir ; en précisant qu’il doit être de type *'a coord3*, on impose que ces trois valeurs soient de même type. Sans cette contrainte, OCaml considère qu’on définit un triplet quelconque, pouvant évidemment avoir des valeurs de types différents :

```
# (12,15,10.2);;
- : int * int * float = (12, 15, 10.2)
```

Pour terminer, si parmi les éléments d’un tuple qu’on définit, figurent des valeurs polymorphes, OCaml l’indique dans le type inféré :

```
# (fst, snd);;
- : ('a * 'b -> 'a) * ('c * 'd -> 'd) = (<fun>, <fun>)
```

Dans cet exemple, il infère que le type de la paire *(fst, snd)*, formée des fonctions prédéfinies *fst* et *snd*, est un produit de deux types de fonctions polymorphes (section 2.6.2, page 51) ; il utilise différentes variables de types pour distinguer les types des arguments de ces deux fonctions. Par contre, il précise, pour *fst*, que le type du premier élément de la paire est le même que celui du résultat ; pour *snd*, il précise que le type du deuxième élément de la paire est le même que celui du résultat.

2.3.2 Enregistrements

Les enregistrements en OCaml sont similaires à ceux du langage C. Cependant, les valeurs d’un enregistrement sont définies par un ensemble d’associations de la forme $l = e$ où l est un nom de champ et e une expression. Ce type d’associations assigne au champ nommé l la valeur correspondant à l’évaluation de e . Par conséquent, un enregistrement a la forme suivante :

$$\{l_1 = e_1; \dots; l_n = e_n\}$$

L'évaluation dynamique de cette expression produit la valeur :

$$\{l_1 = v_1; \dots; l_n = v_n\}$$

où chaque v_i correspond à l'évaluation dynamique de e_i . L'évaluation statique de cette expression produit le type suivant :

$$\{l_1 : t_1; \dots; l_n : t_n\}$$

où chaque t_i correspond au type de v_i .

Notez que pour la définition d'une valeur enregistrement, contrairement aux tuples et aux listes (section suivante), pour lesquels il est possible de définir des valeurs *à la volée*, pour définir un enregistrement, il est nécessaire, au préalable, de définir, à l'aide du mot-clé **type**, le type d'un tel enregistrement :

```
# type paire_int = {l1:int; l2:int};;
type paire_int = { l1 : int; l2 : int }

# let e = {l1=2;l2=4};;
val e : paire_int = {l1 = 2; l2 = 4}

# let e' = {l3=5};;
_____
Error: Unbound record field l3
```

- Opérateurs relationnels : Deux enregistrements sont égaux si leurs champs de même noms sont égaux. Puisque chaque composante de l'enregistrement est identifiée par un nom de champ, leur ordre n'importe pas comme l'illustrent les exemples suivants :

```
# {l1=2;l2=4} = {l1=2;l2=4};;
- : bool = true

# {l1=2;l2=4} = {l2=4;l1=2};;
- : bool = true

# {l1=4;l2=2} = {l2=4;l1=2};;
- : bool = false
```

- Sélecteurs : L'accès à un champ d'un enregistrement se fait de manière tout à fait standard :

```
# type personne =
  { nom: string;
    prenom: string;
    age: int
  };;
type personne = { nom : string; prenom : string; age : int; }

# let e1 = {nom="Tremblay"; prenom="Pierre"; age=30};;
val e1 : personne = {nom = "Tremblay"; prenom = "Pierre"; age = 30}

# e1.nom;;
- : string = "Tremblay"

# (e1.age) / 10;;
- : int = 3
```

- Définition par copie : Il est possible de définir un enregistrement à partir d'un autre :

```
# let e2 = {e1 with prenom = "Jean"; age = 45};;
val e2 : personne = {nom = "Tremblay"; prenom = "Jean"; age = 45}

# e1;;
- : personne = {nom = "Tremblay"; prenom = "Pierre"; age = 30}
```

Polymorphisme : Il est possible de définir un type enregistrement polymorphe. Par exemple :

```
# type 'a enr = { nom: string; prenom: string; profil: 'a };;
type 'a enr = { nom : string; prenom : string; profil : 'a; }

# let e1 = { nom="tat"; prenom="toto"; profil=(175,45,"ok") };;
val e1 : (int * int * string) enr =
  {nom = "tat"; prenom = "toto"; profil = (175, 45, "ok")}

# let e2 = { nom="tat"; prenom="toto"; profil='b' };;
val e2 : char enr = {nom = "tat"; prenom = "toto"; profil = 'b'}
```

Ce type, *enr*, est paramétré par une variable de type, *'a*, puisqu'un de ses éléments (champs), *profil*, est déclaré polymorphe. Ainsi, lors de la définition de l'enregistrement *e1*, Ocaml nous indique que son type est *(int * int * string) enr*, c'est-à-dire une valeur de type *enr* dans lequel la variable de type *'a* est instanciée par le type représentant un triplet de valeurs de types respectifs *int*, *int* et *string*. En effet, le champ *profil* de l'enregistrement *e1* est défini comme un triplet de valeurs de tels types respectifs.

L'enregistrement *e2* est de type *char enr* puisque dans ce cas, le champ *profil* est défini par un caractère.

Évidemment, il est possible qu'un type soit paramétré par plusieurs variables de types :

```
# type ('a,'b,'c) enr = {mat: int; identite: 'a; profil: 'b; adr1: 'c; adr2: 'c};;
type ('a, 'b, 'c) enr = {
  mat : int;
  identite : 'a;
  profil : 'b;
  adr1 : 'c;
  adr2 : 'c;
}
```

Ce type est triplement polymorphe (comme pour le type de tuples polymorphes, remarquons comment les trois variables de type sont précisées à gauche du nom du type défini, *enr*, entourées de parenthèses et séparées par des virgules) : son champ *identite* est polymorphe, son champ *profil* est polymorphe, ainsi que ses deux derniers champs, *adr1* et *adr2*. Le choix des variables de type est important : en utilisant trois variables de type différentes, *'a*, *'b* et *'c*, on précise que les champs *adr1* et *adr2* n'ont pas forcément le même type que celui du champ *identite* et celui de *profil* ; aussi, on précise que les champs *adr1* et *adr2*, bien qu'ayant un type polymorphe, ont le même type. Autrement dit, si dans le champ *adr1*, on utilise, par exemple, une valeur de type *string*, le champ *adr2* devra forcément comprendre une valeur de même type, soit *string*.

L'exemple qui suit illustre la définition d'un enregistrement de ce type :

```
# let e1 =
  { mat = 123;
    identite = {nom="tata"; prenom="toto"; age=12};
    profil = 'a';
    adr1 = "Ste-Foy";
    adr2 = "Quebec"
  };;
val e1 : (personne, char, string) enr =
  {mat = 123; identite = {nom = "tata"; prenom = "toto"; age = 12};
  profil = 'a'; adr1 = "Ste-Foy"; adr2 = "Quebec"}
```

Le type inféré pour cette valeur (liée à *e1*) indique qu'il s'agit d'une valeur de type *enr* dans lequel la première variable de type *'a* a été instanciée par (ou est considérée comme) le type *personne* (défini précédemment), la variable de type *'b* a été instanciée par le type *char* et la variable de type *'c* par *string*. Autre exemple :

```
# let e2 =
  { mat = 007;
    identite = ("James", "Bond");
    profil = "secret";
    adr1 = {l1=2; l2=5};
    adr2 = {l1=5; l2=12}
  };;
val e2 : (string * string, string, paire_int) enr =
  {mat = 7; identite = ("James", "Bond"); profil = "secret";
   adr1 = {l1 = 2; l2 = 5}; adr2 = {l1 = 5; l2 = 12}}
```

Dans ce cas, la valeur définie est de type $(string * string, string, paire_int) enr$ (le type $paire_int$ a été défini précédemment).

2.3.3 Listes

Une liste est une séquence d'objets de même type. Sa forme générale est :

$[e_1; \dots; e_n]$

L'évaluation dynamique d'une liste d'expressions est une liste de valeurs correspondant à l'évaluation de chaque expression :

$[v_1; \dots; v_n]$

L'évaluation statique de cette forme syntaxique est :

$t\ list$

c'est-à-dire le type d'une liste dont les éléments sont de types t . Par exemple, le type $int\ list$ correspond aux listes d'entiers, $bool\ list\ list$ correspond aux listes de listes de valeurs booléennes, $(int * int)\ list$ désigne les listes de paire d'entiers, alors que $(int \rightarrow int)\ list$ représente les listes de fonctions qui prennent comme argument un entier et retournent comme résultat un entier :

```
# [2-1; max 3 2; 5];;
- : int list = [1; 3; 5]

# [2-1, max 3 2, 5];;
- : (int * int * int) list = [(1, 3, 5)]

# [[true;true];[false;true;true]];
- : bool list list = [[true; true]; [false; true; true]]

# [3,4;5,7;6,8];;
- : (int * int) list = [(3, 4); (5, 7); (6, 8)]

# [(3,4);(5,7);(6,8)];;
- : (int * int) list = [(3, 4); (5, 7); (6, 8)]

# [max;min];;
- : ('a -> 'a -> 'a) list = [<fun>; <fun>]
```

Le dernier exemple indique que la liste définie est polymorphe : elle comprend des éléments qui correspondent à des fonctions polymorphes (section 2.6.2, page 51).

Par ailleurs, la différence entre un tuple et une liste est qu'un tuple a un nombre d'éléments fixe et de types différents, alors qu'une liste a un nombre d'éléments variable et dont les types sont identiques.

Il existe une autre notation pour la construction d'une liste : Il s'agit de l'utilisation de la liste vide `[]` et du constructeur de listes `<::>`. Une liste peut être décrite sous la forme suivante :

$e :: l$

où e correspond au premier élément de la liste, et l correspond au reste de la liste. Le type du constructeur `<::>` est :

$'a * 'a \text{ list} \rightarrow 'a \text{ list}$

Il prend deux arguments, l'un de type $'a$ (type polymorphe) et l'autre de type $'a \text{ list}$ correspondant à une liste de valeurs de types $'a$, et retourne une liste de type $'a \text{ list}$ correspondant à la première liste augmentée d'un premier élément correspondant au premier argument. Par exemple :

```
# [];;
- : 'a list = []

# 3 :: [4;5;6];;
- : int list = [3; 4; 5; 6]

# "Hello" :: "the" :: "World" :: [];;
- : string list = ["Hello"; "the"; "World"]

# (3 :: []) :: (4 :: 5 :: []) :: ([]) :: [];;
- : int list list = [[3]; [4; 5]; []]

# [3] :: [4;5] :: [] :: [];;
- : int list list = [[3]; [4; 5]; []]

# max :: min :: [];;
- : ('a -> 'a -> 'a) list = [<fun>; <fun>]
```

Plusieurs opérateurs et fonctions sont définis sur les listes :

- Deux fonctions, **hd** et **tl**, permettent d'accéder au premier élément d'une liste et au reste de la liste. Elles correspondent aux fonctions *car* et *cdr* du langage Lisp. Par exemple⁷ :

```
# open List;;

# hd [(3, 4); (5, 7); (6, 8)];;
- : int * int = (3, 4)

# tl [(3, 4); (5, 7); (6, 8)];;
- : (int * int) list = [(5, 7); (6, 8)]

# hd (3 :: 4 :: []);;
- : int = 3

# tl (3 :: 4 :: []);;
- : int list = [4]

# hd (tl ["Hello"; "the"; "World"]);;
- : string = "the"
```

Le dernier exemple illustre la manière d'accéder à un élément quelconque de la liste. Il suffit d'appliquer plusieurs fois la fonction **tl** jusqu'à obtenir une liste dont le premier élément est celui désiré. En appliquant la fonction **hd** à cette liste, on accède alors à cet élément. Une manière plus simple de procéder est d'utiliser la fonction **nth** qui permet d'accéder directement au n-ième élément d'une liste :

```
# nth ["Hello"; "the"; "World"] 2;;
- : string = "World"

# nth (nth [[3]; [4;5]; []] 1) 0;;
- : int = 4

# (nth [max;min] 0) 5 6;;
- : int = 6
```

À l'instar des chaînes de caractères, le premier élément d'une liste possède la position zéro.

7. Remarquez que l'utilisation de ces fonctions nécessite l'ouverture du module `List`.

- L'opérateur @ permet de concaténer deux listes de même type. Les exemples qui suivent illustrent son utilisation :

```
# [1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

# (1 :: 2 :: []) @ [3;4;5];;
- : int list = [1; 2; 3; 4; 5]
```

La liste vide ([]) constitue un élément neutre pour l'opérateur de concaténation. En effet :

```
# [1;2;3] @ [];;
- : int list = [1; 2; 3]

# [] @ [1;2;3];;
- : int list = [1; 2; 3]
```

- La fonction `length` permet de calculer la taille d'une liste, c'est-à-dire le nombre d'éléments que comprend une liste. Par exemple :

```
# length [];;
- : int = 0

# length [1;2;3];;
- : int = 3

# length [[1;2;3];[];[4;5];[3]];;
- : int = 4
```

- Inverse : La fonction `rev` retourne une liste dont les éléments sont dans l'ordre inverse de ceux de la liste passée en arguments. Par exemple :

```
# rev [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]

# hd (rev [1;2;3;4;5]);;
- : int = 5
```

Le deuxième exemple illustre une méthode d'accès au dernier élément d'une liste.

- Opérateurs relationnels : Les opérateurs `=` et `<>` sont définies aussi pour les listes. Deux listes sont égales si leurs éléments ayant la même position dans la liste sont égales. Par exemple :

```
# [] = [];;
- : bool = true

# [[1];[1;2;3]] = [1::[];[1;2]@[3]];;
- : bool = true

# [1;2;3] = [3;2;1];;
- : bool = false
```

Polymorphisme : Il est possible de définir des types de listes polymorphes ou de manipuler de tel type de listes :


```

# [];;
- : 'a list = []

# [[];[fst;snd]];;
- : ('a * 'a -> 'a) list list = [[]; [<fun>; <fun>]]

# [[];[fst,snd]];;
- : (('a * 'b -> 'a) * ('c * 'd -> 'd)) list list = [[]; [<fun>, <fun>]]

# type ('key, 'value) list_paires = ('key * 'value) list;;
type ('key, 'value) list_paires = ('key * 'value) list

# let l : ('key * 'value) list = [(1,"a"); (2,"b"); (3,"c")];;
val l : (int * string) list = [(1, "a"); (2, "b"); (3, "c")]

```

- Dans le premier exemple, on constate que le type inféré pour la liste vide est `'a list` ; en effet, n'admettant aucun élément (qui viendrait préciser le type des éléments que pourrait contenir cette liste), OCaml considère donc qu'il s'agit d'une liste polymorphe.
- Le deuxième exemple est plus complexe : on définit une liste qui comprend deux sous-listes ; la première est la liste vide : par conséquent, aucune contrainte sur le type des valeurs possibles. La deuxième comprend les fonction `fst` et `snd`. Contrairement à l'exemple présenté à la page 23, dans ce cas, ces deux fonctions étant membres d'une même liste, leur type doivent être identiques : on passe donc des types polymorphes `('a * 'b -> 'a)` et `('c * 'd -> 'd)` à l'unique type polymorphe `('a * 'a -> 'a)`. OCaml infère donc, comme type le plus général pour cette liste, le type `('a * 'a -> 'a) list list`.
- Dans le troisième exemple, il s'agit plutôt d'une liste de paires de fonctions ; le polymorphisme au niveau de chacune des deux fonctions est préservé ; OCaml infère donc, comme type le plus général pour cette liste, le type `('a * 'b -> 'a) * ('c * 'd -> 'd) list list`.
- Le quatrième exemple illustre la définition d'un type de listes polymorphes ; doublement polymorphe puisque le type comprend deux variables de types : `'key` et `'value` (on n'est donc pas limité par des caractères uniques pour l'utilisation des noms des variables de types).
- Le dernier exemple illustre la définition d'une liste ayant le type polymorphe défini précédemment ; OCaml infère le type de cette liste, conformément aux éléments qui y sont définis.

2.3.4 Types somme, produit, algébrique et inductif

Le mot-clé **type**, permettant de définir des types produits (pour les «enregistrements» et les «tuples»), permet aussi de définir des types somme, algébrique et inductif. La forme générale de cette définition est :

$$\text{type } ('a_1, \dots, 'a_n) \text{ id_type} = \text{exp_type};;$$

dans laquelle `id_type` est un identificateur qui est désormais lié à l'expression de type `exp_type` ; si le type est polymorphe, `id_type` est précédé par le nombre de variable de type requis (`'a1, ..., 'an`). `exp_type` est composée d'une suite (somme) de constructeurs de données pouvant admettre des paramètres (auquel cas, on parle de somme de produits). On appelle type algébrique un type qui est défini comme la somme de produits de types ; aussi, parmi ces paramètres, on peut retrouver `id_type` : on parle alors de types algébriques récurrents, ou tout simplement de types inductifs.

Type somme : somme de constructeurs sans paramètres ou types énumérés

Ces types de données sont souvent déclarés en énumérant leurs éléments : on parle de «somme» puisqu'il s'agit de choix entre plusieurs alternatives. Par exemple, le type prédéfini `bool` est défini en énumérant ses deux valeurs `true` et `false`. De même, il serait possible de définir un type `booléen` comme suit :

```
# type booleen = Vrai | Faux;;
type booleen = Vrai | Faux

# Vrai;;
- : booleen = Vrai

# if 2 > 1 then Vrai else Faux;;
- : booleen = Vrai
```

La partie «définition du type» est composée de deux constructeurs : Vrai et Faux. Ainsi, toute valeur booléenne vaut soit Vrai, soit Faux.

L'exemple qui suit définit, par énumération, un type *direction* et un type *couleur* :

```
# type direction = Nord | Sud | Est | Ouest;;
type direction = Nord | Sud | Est | Ouest

# type couleur = Rouge | Vert | Bleu;;
type couleur = Rouge | Vert | Bleu
```

Type algébrique : somme de produits ou constructeurs avec paramètres

Les constructeurs de données peuvent correspondre à des constantes (comme pour les types énumérés), mais peuvent aussi admettre des paramètres. L'exemple qui suit permet de définir un type *couleur* admettant quatre constructeurs sans paramètres (constantes) et un constructeur (RGB) admettant trois paramètres (un triplet) de type entiers :

```
# type couleur = Bleu | Rouge | Vert | Jaune | RGB of int * int * int;;
type couleur = Bleu | Rouge | Vert | Jaune | RGB of int * int * int

# RGB(125,112,0);;
- : couleur = RGB (125, 112, 0)

# RGB(255,255,255);;
- : couleur = RGB(255,255,255)

# Jaune;;
- : couleur = Jaune
```

Pour le constructeur RGB, on parle de produit puisqu'il comprend simultanément trois composantes de type entiers. Et pour le type *couleur*, on dit qu'il est défini comme une somme de produits puisqu'il s'agit d'un choix (somme) entre plusieurs constructeurs dont un est défini comme un produit de trois entiers.

Polymorphisme : Il est possible de définir des types algébriques polymorphes :

```
# type ('a,'b) message = Get of 'a | Send of 'b;;
type ('a, 'b) message = Get of 'a | Send of 'b

# Send "hello";;
- : ('a, string) message = Send "hello"

# Get 5;;
- : (int, 'a) message = Get 5
```

Notons que les deux valeurs définies dans cet exemple, en apparence différentes, peuvent figurer dans une même liste :

```
# [Send "hello"; Get 5];;
- : (int, string) message list = [Send "hello"; Get 5]
```

En effet, il s'agit alors d'une liste d'éléments de même type, soit de type *(int, string) message*.

Type *option* : Lorsqu'on veut décrire les deux états «rien du tout» ou «une valeur d'un certain type», un type algébrique serait tout à fait approprié. OCaml propose un type algébrique prédéfini, *'a option*, permettant de représenter ces deux états. Il est défini comme suit :

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

Le constructeur (la constante) **None** permet de représenter l'état «rien du tout»; le constructeur **Some** permet de représenter «une certaine valeur d'un certain type». Ce type sera utile lorsque, par exemple, on voudra définir (sans avoir à utiliser des exceptions) une fonction qui, pour certains cas, ne devrait pas retourner de valeur; alors que pour d'autres cas, elle en retournerait.

Type inductif : type algébrique récursif

Évidemment, il est possible de définir des types récursifs. Par exemple, pour définir un type représentant des arbres dont les feuilles comprennent des entiers :

```
# type arbre_entiers = Feuille of int | Noeud of arbre_entiers * arbre_entiers;;
type arbre_entiers = Feuille of int | Noeud of arbre_entiers * arbre_entiers

# Feuille(6);;
- : arbre_entiers = Feuille 6

# Feuille 2;;
- : arbre_entiers = Feuille 2

# Noeud(Feuille 1, Feuille 2);;
- : arbre_entiers = Noeud (Feuille 1, Feuille 2)

# Noeud(Noeud(Feuille 1, Feuille 2), Noeud(Feuille 3, Feuille 4));;
- : arbre_entiers = Noeud(Noeud(Feuille 1, Feuille 2), Noeud(Feuille 3, Feuille 4))
```

On parle alors de type inductif puisqu'il est construit à l'aide de sommes, de produits et de récursivité :

- somme (choix entre) deux constructeurs : *Feuille* et *Noeud* ;
- le constructeur *Feuille* admet un paramètre de type *int* ;
- le constructeur *Noeud* admet deux paramètres (produit de deux composantes), tous deux de type *arbre_entiers* (récursion).

Par ailleurs, il est possible de définir différemment ce type en ajoutant une constante pour représenter les arbres vides, et en considérant des entiers dans les nœuds :

```
# type arbre_int = Vide | F of int | N of int * arbre_int * arbre_int;;
type arbre_int = Vide | F of int | N of int * arbre_int * arbre_int

# Vide;;
- : arbre_int = Vide

# F 3;;
- : arbre_int = F 3

# N(1, F 2, Vide);;
- : arbre_int = N (1, F 2, Vide)

# N(1, N(2, F 3, F 4), N(5, F 6, F 7));;
- : arbre_int = N (1, N (2, F 3, F 4), N (5, F 6, F 7))
```

Les types inductifs permettent de définir tous les types de données usuels ; ainsi, il est possible de définir les listes d'entiers comme suit :

```
# type liste_int = Vide | Cons of int * liste_int;;
type liste_int = Vide | Cons of int * liste_int

# Vide;;
- : liste_int = Vide

# Cons(1, Cons(2, Cons(3, Vide)));;
- : liste_int = Cons (1, Cons (2, Cons (3, Vide)))
```

Aussi, il est possible de combiner les différents types de données :

```
# type arbre = F of int * char | N of sous_arbre
and sous_arbre = {v: int * char; fils: arbre list};;
type arbre = F of int * char | N of sous_arbre
and sous_arbre = { v : int * char; fils : arbre list; }

# let a1 = N {v=(1, '1'); fils=[F (2, '2'); F (3, '3'); F (4, '4')]};;
val a1 : arbre = N {v = (1, '1'); fils = [F (2, '2'); F (3, '3'); F (4, '4')]}
```

Dans cet exemple, on définit un type d'arbres particuliers ; il est composé :

- soit d'une feuille comprenant une paire d'entiers et de caractères ;
- soit d'un nœud comprenant des sous arbres ; ces derniers correspondent à des enregistrements comprenant un champ v correspondant à une paire d'entiers et de caractères, et un champ fils comprenant une liste d'arbres (des sous arbres).

Comme le type d'un enregistrement doit être explicitement défini (n'est pas inféré automatiquement), et que les types *sous_arbre* et *arbre* se font référence mutuellement (on parle de définition récursive croisée), on utilise le mot-clé **and** pour définir ces deux types inter-reliés.

Polymorphisme : Il est possible de définir des types inductifs. L'exemple qui suit définit un type permettant de représenter des formules propositionnelles :

```
# type 'a expr =
| Base of 'a
| Const of bool
| And of 'a expr * 'a expr
| Or of 'a expr * 'a expr
| Not of 'a expr;;
type 'a expr =
Base of 'a
/ Const of bool
/ And of 'a expr * 'a expr
/ Or of 'a expr * 'a expr
/ Not of 'a expr
```

Il est alors possible de définir différentes valeurs ayant ce type :

```
# And(Base "il_fait_beau", Const true);;
- : string expr = And (Base "il fait beau", Const true)

# Not(Base 'p');;
- : char expr = Not (Base 'p')
```

Par ailleurs, il est assez simple d'étendre les définitions inductions d'arbres, définies précédemment, pour considérer des arbres polymorphes :

```
# type 'a arbre =
  | Vide
  | Feuille of 'a
  | Noeud of 'a * 'a arbre * 'a arbre;;
type 'a arbre = Vide / Feuille of 'a / Noeud of 'a * 'a arbre * 'a arbre
```

On peut alors définir des arbres de différents types :

```
# let a_int = Noeud(1, Feuille 2, Feuille 5);;
val a_int : int arbre = Noeud (1, Feuille 2, Feuille 5)

# let a_string = Noeud("a", Feuille "b", Vide);;
val a_string : string arbre = Noeud ("a", Feuille "b", Vide)

# let a_prop = Noeud(Const true, Feuille(Base 'p'), Feuille(Base 'q'));;
val a_prop : char expr arbre =
  Noeud (Const true, Feuille (Base 'p'), Feuille (Base 'q'))
```

Pour terminer, un exemple de définition de type d'arbres doublement polymorphe (les éléments dans les feuilles n'ont pas forcément le même type que ceux dans les nœuds) :

```
# type ('a, 'b) tree =
  | Empty
  | Leaf of 'a
  | Node of 'b * ('a, 'b) tree * ('a, 'b) tree;;
type ('a, 'b) tree =
  Empty
  / Leaf of 'a
  / Node of 'b * ('a, 'b) tree * ('a, 'b) tree
```

Quelques exemples d'arbres de ce type :

```
# let t1 = Node(1, Empty, Empty);;
val t1 : ('a, int) tree = Node (1, Empty, Empty)

# let t2 = Node(1, Leaf "a", Leaf "b");;
val t2 : (string, int) tree = Node (1, Leaf "a", Leaf "b")

# let t3 = Node(And(Base "p", Base "q"), Leaf 'a', Leaf 'b');;
val t3 : (char, string expr) tree =
  Node (And (Base "p", Base "q"), Leaf 'a', Leaf 'b')

# let t4 = Node(a_int, Leaf a_string, Empty);;
val t4 : (string arbre, int arbre) tree =
  Node (Noeud (1, Feuille 2, Feuille 5),
    Leaf (Noeud ("a", Feuille "b", Vide)), Empty)

# Node((+), Node((-), Empty, Empty), Leaf fst);;
- : ('a * 'b -> 'a, int -> int -> int) tree =
  Node (<fun>, Node (<fun>, Empty, Empty), Leaf <fun>)
```

- Dans le premier exemple, comme seul un élément d'un nœud est précisé, le type de l'arbre demeure polymorphe au niveau des feuilles.
- Le troisième et le quatrième exemples illustrent la puissance du polymorphisme : arbre de propositions et de caractères ; arbre d'arbres d'entiers et d'arbres de chaînes de caractères.
- Le dernier exemple représente un arbre comprenant, aux nœuds, des fonctions de type $int \rightarrow int \rightarrow int$; aux feuilles, il comprend des fonctions de type polymorphe $'a * 'b \rightarrow 'a$.

Variants polymorphes

Comme précisé précédemment, lorsque l'on veut utiliser un constructeur d'un type algébrique ou inductif, il est nécessaire que ce dernier soit préalablement déclaré :

```
# One;;
...Error: Unbound constructor One

# type t = One | Succ of t;;
type t = One | Succ of t

# One;;
- : t = One

# Succ One;;
- : t = Succ One
```

OCaml offre cependant la possibilité d'utiliser des constructeurs sans avoir à les déclarer préalablement dans une définition à l'aide du mot-clé **type**. Ces constructeurs, qu'on nomme «variants polymorphes», ont la particularité d'être préfixés par une anti-apostrophe (`'`). Voici quelques exemples :

```
# 'One;;
- : [> 'One ] = 'One

# 'Entier 3;;
- : [> 'Entier of int ] = 'Entier 3

# 'Paire ("x",3);;
- : [> 'Paire of string * int ] = 'Paire ("x", 3)
```

Bien sûr, il est possible de définir des listes utilisant des variants polymorphes :

```
# [ 'One; 'Two; 'Three ];;
- : [> 'One | 'Three | 'Two ] list = [ 'One; 'Two; 'Three ]

# [ 'One; 'Two; 'One ];;
- : [> 'One | 'Two ] list = [ 'One; 'Two; 'One ]

# [ ("x", 'I(5)); ("y", 'I(8)); ("z", 'B(true)) ];;
- : (string * [> 'B of bool | 'I of int ]) list =
  [ ("x", 'I 5); ("y", 'I 8); ("z", 'B true) ]
```

Dans la section 2.6.8, page 61, nous expliquons le sens du symbole `<` dans le type inféré d'un variant polymorphe, et nous présenterons un type, appelé type ouvert, qui utilise plutôt le symbole `>`.

2.3.5 Abréviations de types

Comme nous l'avons illustré tout au long des sections précédentes, le mot-clé **type** permet l'abréviation de types. Par exemple, il est possible de définir un type nommé `paire_entiers` pour désigner les paires d'entiers :

```
# type paire_entiers = int * int;;
type paire_entiers = int * int
```

De même, il est possible de définir des abréviations à partir d'autres abréviations :

```
# type liste_paire_entiers = paire_entiers list;;
type liste_paire_entiers = paire_entiers list
```

2.3.6 Mot-clé «type»

En résumé, en le comparant au langage C, le mot-clé d'OCaml **type** permet de faire ce que les mots-clés **typedef**, **enum**, **struct** et **union**, du C, permettent de faire, soit :

- l'abréviation de types,
- la définition de nouveaux types (énumérés, tuples, enregistrements, algébriques et inductifs).

2.4 Données : Définition d'identificateurs, liaisons et déclarations

Comme dans tout langage de programmation, il est d'usage d'utiliser des données à travers des identificateurs ; on parle de variables dans la plupart des langages de programmation. Contrairement à la plupart de ces langages, dans les langages de programmation fonctionnels purs, on ne retrouve pas de notion de variables ; on parle plutôt d'identificateurs, qui s'apparentent à des constantes dans les langages usuels ; en effet, l'idée de lier une donnée (valeur) à un identificateur est simplement de pouvoir l'utiliser de manière plus adaptée. Par exemple, si la donnée «3.14» représente le nombre pi, à la place de manipuler cette valeur comme telle dans tout un programme, on peut simplement la lier à un identificateur qu'on nommera «pi» et l'utiliser via cet identificateur dans tout le reste du programme :

```
# let pi = 3.14;;
  val pi : float = 3.14

# pi;;
- : float = 3.14
```

Évidemment, tout identificateur doit être défini avant d'être utilisé ; défini, en étant lié à une valeur. En OCaml, contrairement à la notion de variable qui peut être déclarée sans être initialisée, il n'est pas possible de déclarer un identificateur sans le lier à une valeur.

Une façon d'introduire un identificateur est la liaison (*binding* en anglais) à une valeur. Le mot-clé **let** permet de définir cette liaison. Ainsi, **let pi = 3.14** signifie que l'identificateur pi est lié à la valeur 3.14. La forme générale de la liaison est :

let *id* = *e*

Pour évaluer cette commande, OCaml effectue les deux étapes suivantes :

- il évalue en premier lieu la partie droite de l'équation, soit *e*, obtenant ainsi la valeur *v*⁸ ;
- puis il associe cette valeur à l'identificateur *id*, qui vaut désormais *v* et a le type *t* de *e*.

Voici un autre exemple illustrant la notion de liaison :

```
# let x = max 3 4;;
  val x : int = 4

# x;;
- : int = 4
```

Les identificateurs d'OCaml sont similaires aux déclarations **const** de C. En d'autres termes, lier un identificateur à une valeur n'est pas une affectation. Une fois cet identificateur liée, sa valeur ne peut plus être modifiée. Pour y parvenir, il faut utiliser la notion de référence (ou pointeur) décrite à la section 2.7.1, page 63.

En fait, à chaque fois qu'un identificateur est lié à une valeur, OCaml crée un nouvel identificateur qui n'a rien à voir avec toute déclaration précédente du même identificateur (même nom). En

⁸. Comme on va le constater dans la suite de ce chapitre, et celui consacré au paradigme orienté objet, les fonctions et les objets étant des valeurs de première classe, il sera possible aussi de lier des identificateurs à ce type de valeurs.

effet, une nouvelle déclaration peut attribuer une valeur dont le type est différent de celui défini pour ce même identificateur dans une déclaration précédente. Les exemples qui suivent illustrent ces propos :

```
# let x = "Pierre";;
  val x : string = "Pierre"

# let y = x;;
  val y : string = "Pierre"

# let x = true;;
  val x : bool = true

# y;;
- : string = "Pierre"
```

La première expression lie à un identificateur `x` une chaîne de caractères. Puis un identificateur `y` est lié à l'évaluation de l'identificateur `x`, soit cette même chaîne. La troisième expression crée un nouvel identificateur `x` à laquelle elle associe la valeur booléenne `true`. L'ancienne valeur associée à `x` n'est plus accessible. Aussi, la valeur associée à `y` n'est pas affectée par la nouvelle valeur associée à `x`.

Plusieurs déclarations d'identificateurs peuvent être effectuées simultanément grâce à l'utilisation du mot-clé **and**. Dans ce cas, OCaml évalue simultanément (en parallèle) les différentes liaisons comme l'illustrent ces exemples :

```
# let x = 12;;
  val x : int = 12

# let x = true and y = x;;
  val x : bool = true
  val y : int = 12

# let z = "Hello" and y = z;;
  -;;
Error: Unbound value z
```

Le deuxième exemple définit deux identificateurs `x` et `y` à l'aide des mots-clés **let** et **and**. Il évalue tout d'abord les parties droites des deux déclarations, obtenant les valeurs `true` et 12 respectivement pour la première et la deuxième déclaration. Puis il lie ces valeurs respectivement aux identificateurs `x` et `y`. Le dernier exemple illustre le cas où la valeur d'un identificateur non déclaré, `z`, est liée à un autre identificateur, `y` dans ce cas. En effet, bien que l'identificateur `z` soit déclaré, l'évaluateur ne le "perçoit" pas en évaluant la déclaration `y = z`. Ceci est dû au fait que l'évaluation se fait en parallèle.

Ces déclarations permettent de définir des identificateurs globaux dont la portée est celle de tout le programme. Dans certains cas, il est nécessaire d'effectuer des déclarations d'identificateurs dont la portée est restreinte à un contexte donné : Il s'agit alors d'identificateurs locaux. Pour de telles déclarations, OCaml propose l'expression suivante :

let *id* = *e*₁ **in** *e*₂

Cette expression s'évalue comme suit :

- Tout d'abord l'expression *e*₁ est évaluée. La valeur qui en résulte est liée à l'identificateur *id*.
- Puis, l'évaluation de l'expression *e*₂ est amorcée et chaque occurrence de l'identificateur *id* est remplacée par la valeur à laquelle il est lié.
- Le résultat de cette dernière évaluation est retourné comme valeur de l'évaluation de l'expression au complet.

Dans cette expression, la portée des identificateurs déclarés est locale à l'expression e_2 . Les exemples qui suivent illustrent l'utilisation de cette forme syntaxique :

```
# let x = true;;
  val x : bool = true

# let x = 10 in x*x;;
- : int = 100

# x;;
- : bool = true

let x = 5 in
let x = x + 1 in
  x;;
- : int = 6

# let x = 5 and y = x in (x < 10) && y;;
- : bool = true
```

```
# let x = 5 in let y = x in x*y;;
- : int = 25

# let x = 1 in let x = 2 and y = x in x+y;;
- : int = 3

# x;;
- : bool = true

# y;;
-
Error: Unbound value y
```

La première expression déclare un identificateur global x lié à la valeur **true**. Dans les expressions suivantes, cet identificateur est redéclaré mais à un niveau local. Par conséquent, l'identificateur global x n'est pas affecté. Jusqu'à la dernière expression, sa valeur vaut **true**. Par contre, dans les expressions faisant intervenir la constructeur syntaxique **let—in**, sa valeur vaut tantôt 5 et tantôt 1. Ces valeurs ne sont pris en compte qu'à l'intérieur de l'expression, c'est-à-dire entre le mot clé **in** et la fin de l'expression. La dernière expression s'évalue en une erreur puisque OCaml ne trouve pas d'identificateur global y déclaré, malgré que celui-ci soit déclaré dans les expressions **let**.

Cette forme de déclaration locale permet d'éviter de ne déclarer que des identificateurs globaux. Dans certains cas, il est nécessaire de ne pas redéfinir un identificateur global. En le déclarant localement, il est possible de redéfinir sa valeur sans affecter sa valeur globale.

2.5 Comportements : Filtrage par motifs

Dans les sections précédentes, nous avons présenté différentes notions liées à l'aspect «Données» d'un programme ; on y a présenté différentes manières de définir ou de construire des données (de base ou composées). La présente section, et celle qui suit, présentent des notions liées à l'aspect «Comportements» d'un programme. La première notion est le filtrage par motifs (*pattern matching*) grâce auquel il sera possible de «déconstruire» une donnée en vue de manipuler ces composantes.

La forme générale du filtrage par motifs est ⁹ :

```
match e with
| motif1 -> e1
```

9. La barre verticale, |, pour le *motif*₁ est optionnelle.

```

| ...
| motifn -> en

```

Cette expression s'évalue comme suit :

- Tout d'abord l'expression e est évaluée pour retourner une valeur v .
- Ocaml tente par la suite de filtrer cette valeur v avec un des motifs $motif_i$.
- Pour le premier motif k pour lequel cette action de filtrage réussit, Ocaml évalue l'expression e_k associée et retourne, comme résultat de toute l'expression, la valeur v_k résultante.

Évidemment, puisqu'un filtrage par motifs (forme générale précisée ci-dessus) est une expression, son évaluation retourne donc une valeur et un type (type de la valeur) ; par conséquent, toutes les expressions e_1, \dots, e_n doivent s'évaluer en des valeurs de même type.

Par ailleurs, notez que si le programmeur oublie de préciser un motif, Ocaml s'en aperçoit et affiche un avertissement (*warning*) ; un point que les autres langages ne traitent pas en général, avec l'instruction correspondante **switch-case**.

Dans la suite de cette section, nous présentons le filtrage par motifs afin de déconstruire les différentes structures de données étudiées dans les sections précédentes.

Tuples : Voici quelques exemples illustrant la manière avec laquelle on déconstruit une donnée de type «tuple» :

```

1  # let p = (2,3);;
2  val p : int * int = (2, 3)
3
4  # match p with
5  | (x,y) -> x + y;;
6  - : int = 5
7
8  # match p with
9  | (x,y) -> x;;
10 - : int = 2
11
12 # match p with (x,_) -> x;;
13 - : int = 2
14
15 # let t = (1,"a",3.14,abs);;
16 val t : int * string * float * (int -> int) = (1, "a", 3.14, <fun>)
17
18 # match t with
19 | (_,_,_,f) -> f (-5);;
20 - : int = 5
21
22 # match t with
23 | _,_,pi, _ -> pi;;
24 - : float = 3.14
25
26 # let t' = ((2,4),(max,min),"toto");;
27 val t' : (int * int) * (('a -> 'a -> 'a) * ('b -> 'b -> 'b)) * string =
28   ((2, 4), (<fun>, <fun>), "toto")
29
30 # let res = match t' with (x1,x2),(f1,f2),_ -> f1 x1 x2, f2 x1 x2;;
31 val res : int * int = (4, 2)

```

- Aux lignes 4 et 5, on filtre la paire p , correspondant à la paire $(2,3)$, avec le motif (x,y) ; puisque le filtrage réussit, à la droite du symbole $->$, les identificateurs x et y sont liés respectivement aux valeurs 2 et 3. Ainsi le résultat de toute l'expression (lignes 4 et 5) est le résultat de la somme $x + y$, soit 5 (ligne 6).

Notons que les identificateurs x et y ne sont visibles que dans la partie à droite du symbole $->$. Ainsi :

```
# match p with
| (x,y) -> x + y;;
- : int = 5
```

```
# x;;
-
Error: Unbound value x
```

- L'expression définie aux lignes 8 et 9 illustre le fait qu'on peut utiliser seulement un identificateur, parmi ceux filtrés par l'expression ; dans cet exemple, seule la valeur de l'identificateur `x` est utilisée et est retournée comme résultat de toute l'expression.
- L'expression définie à la ligne 12 permet de préciser qu'on peut utiliser le symbole `_` pour filtrer n'importe quelle valeur. Dans cet exemple, le motif `(x, _)` précise que la valeur à filtrer doit être une paire de valeurs : la première valeur de la paire sera liée à l'identificateur `x` ; la deuxième valeur de la paire filtre, à coup sûr le motif `_`, mais sa valeur ne sera liée à aucun identificateur (elle sera ignorée dans la partie droite du symbole `->`).
Autrement dit, l'exemple illustre une manière d'accéder au premier élément d'une paire ; il est évidemment possible d'accéder à n'importe quel élément d'un tuple quelconque (voir exemple suivant, aux lignes 18 et 19).
Par ailleurs, cet exemple illustre le fait que toute l'expression peut être saisie sur une ligne. Aussi, que la barre verticale, `|`, pour le premier motif (ici, il n'y en a qu'un), est optionnelle.
- Aux lignes 18 et 19, on filtre le tuple `t`, correspondant à `(1,"a",3.14,abs)`, avec le motif `(_,_,_,f)` ; par conséquent, pour qu'une valeur filtre ce motif, il faut qu'elle corresponde à un quadruplet. Aussi, dans cet exemple, les trois premières valeurs du quadruplet sont ignorées ; la quatrième est liée à l'identificateur `f`, et est appliquée (partie droite de `->`) à la valeur `-5` pour retourner comme résultat final, la valeur absolue de `-5`, soit `5` (ligne 20).
- L'exemple des lignes 22 et 23 permet d'accéder au troisième élément du quadruplet, et de le lier à l'identificateur `pi` ; l'exemple permet aussi de constater que les parenthèses, pour désigner un quadruplet dans un motif, sont facultatifs.
- Le dernier exemple illustre un motif plus avancé qui correspond à un triplet dont les deux premiers éléments sont des paires (respectivement, paire d'entiers, et paire de fonctions). Dans cet exemple, grâce au filtrage par motifs, on récupère les deux valeurs entières et les deux fonctions, et on applique (partie droite de `->`) les deux fonctions aux deux valeurs, pour obtenir, au final, une paire de résultats, qui est liée, grâce au **let**, à l'identificateur global `res`.

Pour terminer, si le filtrage ne réussit, Ocaml produit un message d'erreur :

```
# match p with (x,y,_) -> x + y;;
Characters 13-20:
  match p with (x,y,_) -> x + y;;
                  ^
Error: This pattern matches values of type 'a * 'b * 'c
      but a pattern was expected which matches values of type int * int
```

Enregistrements : Voici quelques exemples illustrant la manière avec laquelle on déconstruit une donnée de type «enregistrement» :

```
# type personne = { nom: string; prenom: string; age: int };;
type personne = { nom : string; prenom : string; age : int; }

# let e1 = {nom="Tremblay"; prenom="Pierre"; age=30};;
val e1 : personne = {nom = "Tremblay"; prenom = "Pierre"; age = 30}

# match e1 with
| { nom=n; prenom=p; age=a } -> p ^ "_" ^ n ^ ",_" ^ (string_of_int a) ^ "_ans";;
- : string = "Pierre Tremblay, 30 ans"
```

Dans cet exemple, on filtre tous les champs de l'enregistrement `e1`, on les lie à des identificateurs, et on les traite en conséquence.

Contrairement aux tuples, les enregistrements ayant des noms de champs pour distinguer ces différents composantes, il n'est pas nécessaire de suivre l'ordre précis des champs ; aussi, au niveau des champs, si certains champs de l'enregistrement ne nous intéressent pas, on peut ignorer leur présence dans le motif :

```
# match e1 with { nom=n; prenom=p } -> p ^ "_" ^ n;;
- : string = "Pierre Tremblay"

# match e1 with
| { age=a; prenom=p } -> p ^ "_a_" ^ (string_of_int a) ^ "_ans";;
- : string = "Pierre a 30 ans"
```

Le premier exemple est définie sur une ligne (sans utilisation de la barre verticale, `|`, puisque optionnelle) : il illustre un cas où seuls certains champs sont précisés dans un motif. Le deuxième exemple illustre que, dans un motif, l'ordre entre les champs n'importe guère.

Par ailleurs, lorsque l'identifiant qu'on utilise dans un motif correspond exactement au nom d'un champ, il n'est pas nécessaire de préciser deux fois ce nom ; les deux derniers exemples peuvent alors être décrits comme suit :

```
# match e1 with { nom; prenom } -> nom ^ "_" ^ prenom;;
- : string = "Pierre Tremblay"

# match e1 with
| { age; prenom } -> prenom ^ "_a_" ^ (string_of_int age) ^ "_ans";;
- : string = "Pierre a 30 ans"
```

Finalement, en reprenant l'exemple de l'enregistrement `e1`, de la page 25, il est possible d'imbriquer différents filtrage par motifs :

```
# match e1 with
| identite ->
  match identite with {nom;prenom} -> prenom ^ ",_" ^ nom;;
- : string = "Pierre , Tremblay"
```

Listes : Voici quelques exemples illustrant la manière avec laquelle on déconstruit une donnée de type «liste» :

```
1 # let liste = [1;2;3;4;5];;
2 val liste : int list = [1; 2; 3; 4; 5]
3
4 # match liste with
5 | [] -> "liste_vide"
6 | e::r -> string_of_int e;;
7 - : string = "1"
8
9 # match [] with
10 | [] -> "liste_vide"
11 | e::r -> string_of_int e;;
12 - : string = "liste vide"
13
14 # match liste with
15 | [] -> "liste_vide"
16 | e::_ -> string_of_int e;;
17 - : string = "1"
```

- Le motif à la ligne 5 désigne une liste vide, alors que celui de la ligne 6 désigne une liste dont le premier élément sera lié à un identificateur nommé `e`, et le reste de la liste sera lié à un identificateur nommé `r`. Les deux motifs permettent de filtrer tous les cas de figure d'une liste : soit la liste est vide, soit elle comprend forcément un premier élément et un reste (ce dernier pouvant correspondre à une liste vide dans le cas où la liste à filtrer comprendrait un seul élément).
Par conséquent, le filtrage, dans cet exemple, permet de retourner la chaîne de caractères "liste vide" dans le cas où la liste à filtrer est vide ; elle retourne le premier élément de la liste, transformé en chaînes de caractères (afin que dans les deux cas de figure, on ait une valeur retournée ayant un même type, soit *string*), dans le cas contraire.
- L'exemple aux lignes 9 à 11 permet de constater le comportement de l'expression lorsque la liste à filtrer est vide.
- Le filtre à la ligne 16 illustre l'utilisation du symbole `_` lorsque la valeur filtrée n'est pas utile ou utilisée ; dans ce cas, puisqu'on veut accéder au premier élément de la liste, le reste de la liste est précisé par le motif `_`.

Voici d'autres exemples illustrant le filtrage par motifs de listes d'éléments :

```

1  # let liste = [1;2;3;4;5];;
2  val liste : int list = [1; 2; 3; 4; 5]
3
4  # match liste with
5  | [] -> -1
6  | _::e2::_ -> e2;;
7  Warning 8 [partial-match]: this pattern-matching is not exhaustive.
8  Here is an example of a case that is not matched:
9  _::[]
10 - : int = 2
11
12 # match liste with
13 | [] -> -1
14 | [_] -> -1
15 | _::e::_ -> e;;
16 - : int = 2

```

- L'exemple aux lignes 4 à 6 permet d'illustrer qu'Ocaml est toujours vigilant en ce qui a trait à la «complétude» des cas de motifs précisés. En effet, dans cet exemple, l'objectif est d'accéder au deuxième élément de la liste ; le motif à la ligne 6, `_::e2::_`, permet en effet d'accéder à cette valeur en la liant à un identificateur nommé `e2`. Ce motif peut filtrer n'importe quelle liste ayant un premier élément (filtré par le premier `_`), un deuxième élément (filtré et lié à `e2`), et un reste de liste (filtré par le deuxième `_`) ; autrement dit, il ne permet de filtrer que des listes ayant au moins deux éléments. Ce motif, combiné avec celui défini à la ligne 5, permettent de filtrer des listes vides ou des listes comprenant au moins deux éléments : ils ne permettent donc pas de filtrer (traiter) n'importe quelle liste. Ocaml le signale par un avertissement (cet avertissement n'empêche pas Ocaml de retourner la bonne valeur résultante de l'évaluation de toute l'expression, soit le deuxième élément de la liste : 2) ; de plus, cet avertissement nous précise un exemple de motif de valeurs qui ne sont pas filtrées : `_::[]`. ce motif représente en effet les listes à un seul élément.
- L'exemple suivant, lignes 15 à 18, corrige la situation en traitant trois cas de figures :

1. la liste vide ;
2. la liste à un élément (dont le motif aurait pu être défini comme suit : `_::[]`) ;
3. la liste à deux éléments et plus.

Notons qu'il est possible de combiner et traiter plusieurs motifs en même temps :

```

1 # match liste with
2 | [] | _::[] -> -1
3 | _::e::_ -> e;;
4 - : int = 2

```

Dans cet exemple, la ligne 2 résume les lignes 16 et 17 de l'exemple précédent. L'exemple précédent aurait pu être défini comme suit :

```

1 # match [1;2] with
2 | [] | [] -> -1
3 | [_;e] -> e
4 | _ -> -1;;
5 - : int = 2

```

Dans ce cas, on aura traité quatre cas de figure :

1. les deux premiers cas sont traités en même temps (ligne 2) et précisent des listes vides ou à un élément ;
2. le troisième cas, ligne 3, précise une liste à deux éléments : le deuxième élément, étant celui qui nous intéresse, est lié à un identificateur `e` ;
3. le dernier cas traite de tous les autres types de listes.

Ainsi, on aura traité tous les cas de figure ; en omettant un de ces cas, par exemple le dernier, `OCaml` nous le rappelle par un avertissement (voir exemple qui suit), en précisant un exemple de motif non traité (ici, à la ligne 11, il nous décrit un motif, `_::_::_`, représentant une liste comprenant au moins trois éléments) :

```

1 # match [1;2] with
2 | [] -> -1
3 | [] -> -1
4 | [_;e] -> e;;
5 Warning 8 [partial-match]: this pattern-matching is not exhaustive.
6 Here is an example of a case that is not matched:
7 _::_::_
8 - : int = 2

```

Finalement, les motifs peuvent combiner différentes formes de types (dans l'exemple qui suit, on considère des listes de paires d'éléments) :

```

1 # match [(1,"a"); (2,"b"); (3,"c")] with
2 | [] -> "liste_vide"
3 | (_,s)::_ -> s;;
4 - : string = "a"

```

Types algébriques et inductifs : Voici quelques exemples illustrant la manière avec laquelle on déconstruit une donnée de type «algébrique» :

```

1 # type couleur = Bleu | Rouge | Vert | Jaune | RGB of int * int * int;;
2 type couleur = Bleu | Rouge | Vert | Jaune | RGB of int * int * int
3
4 # let c = Rouge;;
5 val c : couleur = Rouge
6
7 # match c with
8 | Bleu -> "bleu"
9 | Rouge -> "rouge"
10 | Vert -> "vert"
11 | Jaune -> "jaune"
12 | RGB _ -> "autre:_RGB_...";;
13 - : string = "rouge"

```

Dans cet exemple, à la ligne 12, le symbole `_` permet d'abstraire le triplet argument du constructeur de données RGB. Voici un autre exemple :

```
# let c' = RGB(125,255,1);;
val c' : couleur = RGB (125, 255, 1)

# match c' with
| RGB(x1,_,x3) -> (x1,x3)
| _ -> (-1,-1);;
- : int * int = (125, 1)
```

De même, l'exemple qui suit permet de filtrer une liste d'éléments en apparence différents (le type polymorphe *message* est défini à la page 30) :

```
# let liste = [Send "hello"; Get 5];;
val liste : (int, string) message list = [Send "hello"; Get 5]

# match liste with
| [] -> None
| (Send s)::_ -> Some s
| (Get i)::_ -> Some (string_of_int i);;
- : string option = Some "hello "
```

Dans cet exemple, on utilise aussi le type prédéfini *'a option* présenté à la page 31.

Par ailleurs, on peut aussi naturellement filtrer des types inductifs :

```
# type arbre_int = Vide | F of int | N of int * arbre_int * arbre_int;;
type arbre_int = Vide | F of int | N of int * arbre_int * arbre_int

# let a1 = N(1, F 2, F 3);;
val a1 : arbre_int = N (1, F 2, F 3)

# match a1 with
| Vide -> 0
| F x -> x
| N(x,_,_) -> x;;
- : int = 1
```

Évidemment, le filtrage de ce type de données (récuratif) sera pertinent lorsqu'on les traitera dans le cadre de fonctions récuratives (section 2.6).

Types de base : Il est aussi possible de filtrer les valeurs ayant un type de base ; voici quelques exemples illustrant la manière avec laquelle on peut accéder, par filtrage, à des valeurs entières :

```
# let x = 2;;
val x : int = 2

# match x with
| 1 -> "1"
| 2 -> "2"
| _ -> "autre";;
- : string = "2"
```

Le cas du filtre des valeurs caractères est particulièrement intéressant notamment lorsqu'on combine des motifs (dans cet exemple, on teste si un caractère est une voyelle) :

```
# let c = 'y';
val c : char = 'y'

# match c with
| 'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
| _ -> false;;
- : bool = true
```

Pour les caractères, Ocaml permet aussi de définir des intervalles pour décrire des filtres :

```
# match c with
| '0' .. '9' -> true
| _ -> false;;
- : bool = false

# match c with
| 'a' .. 'z' | 'A' .. 'Z' -> true
| _ -> false;;
- : bool = true
```

Combinaison de types : Il est possible de décrire des motifs qui filtrent des données complexes :

```
# type arbre = F of int * char | N of sous_arbre
and sous_arbre = {v: int * char; fils: arbre list};;
type arbre = F of int * char | N of sous_arbre
and sous_arbre = { v : int * char; fils : arbre list; }

# let a1 = N {v=(1,'1'); fils=[F (2,'2'); F (3,'3'); F (4,'4')]};;
val a1 : arbre = N {v = (1, '1'); fils = [F (2, '2'); F (3, '3'); F (4, '4')]}

# match a1 with
| F (i,_) -> i
| N {v=(i,_); fils=[]} -> i
| N {fils=(F (i,_))::_} -> i
| N {v=(i,_)} -> i;;
- : int = 2
```

Pour cet exemple, si on omet le dernier motif, Ocaml signale un avertissement.

Nommer une valeur filtrée : Il est possible, à l'aide du mot-clé **as**, de nommer une partie d'un motif, facilitant ainsi l'écriture d'un filtre par motifs (l'exemple qui suit considère une paire de deux rationnels représentés chacun par une paire, dont le premier élément représente le numérateur et le deuxième élément représente le dénominateur du rationnel; l'exemple permet de retourner le plus petit des deux rationnels) :

```
# let p_rat = (2,3),(1,2);;
val p_rat : (int * int) * (int * int) = ((2, 3), (1, 2))

# match p_rat with
| (_,0),r2 -> r2
| r1,(_,0) -> r1
| ((n1,d1) as r1), ((n2,d2) as r2) ->
  if (n1 * d2) < (n2 * d1) then r1 else r2;;
- : int * int = (1, 2)
```

Filtrer avec gardes : Il est aussi possible, à l'aide du mot-clé **when**, d'associer des conditions aux motifs utilisés. L'exemple qui suit permet de tester l'égalité de deux nombres rationnels :


```
# let p_rat = (2,3),(4,6);;
val p_rat : (int * int) * (int * int) = ((2, 3), (4, 6))

# match p_rat with
| (_,0),(_,0) -> true
| (_,0),_ -> false
| _,(0) -> false
| (n1,1), (n2,1) when n1 = n2 -> true
| (n1,d1), (n2,d2) when ((n1 * d2) = (n2 * d1)) -> true
| _ -> false;;
- : bool = true
```

Notez cependant qu'en utilisant de telles gardes, OCaml ne peut plus assurer la «complétude» des motifs définis.

Retour sur les liaisons : La forme générale, tant de la liaison permettant de définir un identificateur global que celle permettant de définir un identificateur local, est en réalité :

```
let m = e
let m = e in e'
```

où m désigne un motif quelconque. Par conséquent, il sera possible de déclarer plusieurs identificateurs en même temps grâce à l'utilisation de motifs :

```
# let (x,y) = (1,2);;
val x : int = 1
val y : int = 2
```

Dans cet exemple, les identificateurs x et y sont désormais liés respectivement aux valeurs 1 et 2.

De la même manière, il sera possible d'accéder aux dénominateurs des deux rationnels contenus dans `p_rat` simplement en utilisant la liaison et le filtrage par motifs suivants :

```
# let (_,d1),(_,d2) = p_rat;;
val d1 : int = 3
val d2 : int = 6
```

Pour terminer, notons qu'il est évidemment possible d'utiliser le filtrage par motifs avec des variants polymorphes (section 2.6.8, page 61).

2.6 Comportements : Fonctions

Les valeurs fonctionnelles sont les entités de base les plus importantes dans les langages de programmation fonctionnelles. Mathématiquement, une fonction f de type $A \rightarrow B$ est une relation entre des valeurs de type A et des valeurs de type B . Plus précisément, chaque élément du domaine de la fonction est associé à au plus un élément de B . Si x appartient à A , alors on peut écrire $f(x)$ pour l'application de f à x . Notons par ailleurs que la notation $f(x)$ est l'unique manière de noter une application fonctionnelle en mathématiques.

En OCaml, la forme la plus générale, et la plus simple, pour définir une fonction est :

```
function x -> e
```

Cette expression définit une fonction anonyme (elle n'est liée à aucun identificateur) qui prend comme unique argument x et qui a comme corps l'expression e . Appliquer cette expression (fonction) à une valeur v revient à évaluer l'expression e (corps de la fonction) en considérant que x (argument de la fonction) est lié à la valeur v . Autrement dit, en utilisant la syntaxe d'OCaml, l'expression «(function $x \rightarrow e$) v » (application de la fonction à v) correspond en quelque sorte à l'évaluation de l'expression «let $x = v$ in e ».

De manière plus générale, l'application d'une expression e à une expression e' , notée $e\ e'$ ou $e(e')$, est évaluée comme suit :

1. l'expression e est (forcément) évaluée en une fonction, soit f ;
2. l'expression e' est évaluée en une valeur v ;
3. on applique la fonction f à la valeur v , c'est-à-dire qu'on évalue le corps de cette fonction en considérant que son paramètre est lié à v .

Voici quelques exemples illustrant l'utilisation de cette forme générale pour la définition de fonctions :

```
# function x -> x + 1;;
- : int -> int = <fun>

# (function x -> x + 1) 5;;
- : int = 6

# let x = 5 in x + 1;;
- : int = 6

# function x -> (x+1,x-1);;
- : int -> int * int = <fun>

# (function x -> (x+1,x-1)) 2;;
- : int * int = (3, 1)
```

Le corps d'une fonction étant une expression quelconque du langage, il peut correspondre à une paire d'expressions, comme c'est illustré dans le dernier exemple (ci-dessus) ; en particulier le corps d'une fonction pourrait correspondre à une autre fonction :

$$\text{function } x \rightarrow \underbrace{\text{function } y \rightarrow e'}_e$$

Voici des exemples illustrant ces propos :

```
# function x -> (function y -> x + y);;
- : int -> int -> int = <fun>

# (function x -> (function y -> x + y)) 2;;
- : int -> int = <fun>

# ((function x -> (function y -> x + y)) 2) 3;;
- : int = 5

# (function x -> function y -> x + y) 2 3;;
- : int = 5
```

Dans le premier exemple, les parenthèses sont utilisées afin de bien délimiter le corps de la première fonction (celle englobante) ayant comme argument x . Le deuxième exemple applique la fonction à la valeur 2. Cette application consiste alors à évaluer le corps de la fonction, soit **function** $y \rightarrow x + y$, en considérant l'argument x lié à la valeur 2. Le résultat de cette application est donc la fonction **function** $y \rightarrow 2 + y$. Le troisième exemple applique le résultat de l'application précédente à la valeur 3, soit «(**function** $y \rightarrow 2 + y$) 3». Cette application résulte naturellement en la valeur 5. Le dernier exemple reprend le même exemple précédent mais en limitant au maximum l'utilisation des parenthèses.

D'une manière plus générale, en ce qui concerne l'application, il faut utiliser l'associativité à gauche :

$$e_1 \ e_2 \ e_3 \ \dots \ e_n \quad \equiv \quad (\dots((e_1 \ e_2) \ e_3) \ \dots) \ e_n$$

Par exemple, l'expression suivante :

$$\underbrace{(\text{function } x \rightarrow \text{function } y \rightarrow \text{function } z \rightarrow x + y + z)}_{e_1} \underbrace{(2 - 1)}_{e_2} \underbrace{(\text{max } 5 \ 6)}_{e_3} \underbrace{(\text{abs } 5)}_{e_4}$$

équivalent à celle-ci :

$$(((\text{function } x \rightarrow \text{function } y \rightarrow \text{function } z \rightarrow x + y + z) \ (2 - 1)) \ (\text{max } 5 \ 6)) \ (\text{abs } 5)$$

c'est-à-dire qu'on applique e_1 à e_2 , puis le résultat de cette application est appliqué à e_3 puis le nouveau résultat est appliqué à e_4 . Écrit en Ocaml :

```
# (function x -> function y -> function z -> x + y + z) (2-1) (max 5 6) (abs 5);;
- : int = 12
```

Quand une fonction, comme celle définie dans l'exemple précédent, prend plusieurs arguments, l'un à la suite de l'autre, on dit alors que c'est une fonction curryfiée (voir section 2.6.5). Cependant, il est possible de faire en sorte qu'une fonction soit directement appliquée à une paire d'éléments :

```
# function x -> let x1 = fst x and x2 = snd x in x1 + x2;;
- : int * int -> int = <fun>

# (function x -> let x1 = fst x and x2 = snd x in x1 + x2) (2,3);;
- : int = 5
```

La fonction prend toujours un seul argument, x , mais du fait que dans le corps de la fonction on utilise, à travers les fonctions `fst` et `snd`, cet argument comme une paire de valeurs, Ocaml détermine alors (comme on peut le constater à travers le type inféré) que l'argument de la fonction ne peut être qu'une paire de valeurs (dans un deuxième temps, il détermine qu'il s'agit d'une paire de valeurs entières).

D'une manière plus générale, lors de l'application d'une fonction à plusieurs valeurs, il s'agit en fait de l'application de cette fonction à un tuple (unique argument de la fonction) dont les éléments correspondent aux paramètres de la fonction.

Par ailleurs, en utilisant le filtrage par motifs (section 2.5), il est possible de définir cette même fonction de la manière suivante :

```
# function x -> let (x1,x2) = x in x1 + x2;;
- : int * int -> int = <fun>

# (function x -> let (x1,x2) = x in x1 + x2) (2,3);;
- : int = 5
```

Dans ce cas, pour que le terme $\langle(x1,x2) = x\rangle$ soit valide, il faut que x soit une paire de valeurs. De plus, ce terme, à travers le filtrage, permet d'extraire le premier et le deuxième élément de x en les liant aux identificateurs $x1$ et $x2$. Cette technique, de filtrage, est valable pour un triplet, un quadruplet, etc. :

```
# function x -> let (x1,x2,x3) = x in x1 + x2 + x3;;
- : int * int * int -> int = <fun>

# (function x -> let (x1,x2,x3) = x in x1 + x2 + x3) (1,2,3);;
- : int = 6
```

Par ailleurs, les deux fonctions définies précédemment auraient pu être définies comme suit :

```
# function (x1,x2) -> x1 + x2;;
- : int * int -> int = <fun>

# (function (x1,x2) -> x1 + x2) (2,3);;
- : int = 5

# function (x1,x2,x3) -> x1 + x2 + x3;;
- : int * int * int -> int = <fun>

# (function (x1,x2,x3) -> x1 + x2 + x3) (1,2,3);;
- : int = 6
```

Dans cette forme-là, la fonction comprend toujours un unique argument, soit «(x1,x2)» (resp. «(x1,x2,x3)»), mais ce dernier doit correspondre à (doit filtrer) une paire (resp. un triplet) de valeurs. En effet, dans le langage OCaml, une paire (resp. un triplet), et d’une manière plus générale un tuple (section 2.3.1), est une valeur de première classe (une valeur considérée au même titre que les autres valeurs du langage) :

```
# let p = (2,3);;
val p : int * int = (2, 3)

# (function (x1,x2) -> x1 + x2) p;;
- : int = 5

# let t = (1,2,3);;
val t : int * int * int = (1, 2, 3)

# (function (x1,x2,x3) -> x1 + x2 + x3) t;;
- : int = 6
```

En résumé, et en réalité, la forme générale de définition de fonction :

function *m* -> *e*

où *m* représente un motif. On comprend alors pourquoi il est possible de définir une fonction ayant comme unique argument un «tuple» ; en réalité, l’argument d’une telle fonction est un motif ne pouvant filtrer que des tuples d’une certaine taille. En reprenant l’exemple précédent :

```
# function (x1,x2,x3) -> x1 + x2 + x3;;
- : int * int * int -> int = <fun>
```

on définit, en réalité, une fonction anonyme qui prend comme argument toute valeur qui filtre le motif (x1,x2,x3), soit tout triplet de valeurs (entières, puisque dans le corps de cette fonction, les identificateurs x1, x2 et x3 sont utilisés d’une manière qui impose qu’ils soient des entiers).

Ainsi, avec cette forme générale de définition de fonction, il est possible de définir :

- des fonctions de manière curryfiée (traitant leur argument un à la suite de l’autre) ;
- des fonctions prenant comme argument, grâce aux motifs, des tuples de valeurs (retrouvant ainsi la forme standard des fonctions admettant «plusieurs arguments»).

Aussi, une fonction étant une valeur de première classe, il est possible de l’utiliser :

- comme argument d’une autre fonction :

```
# function g -> function x -> if x mod 2 = 0 then g x else g (x+1);
- : (int -> 'a) -> int -> 'a = <fun>

# (function g -> function x -> if x mod 2 = 0 then g x else g (x+1))
  string_of_int 5;;
- : string = "6"

# (function g -> function x -> if x mod 2 = 0 then g x else g (x+1))
  string_of_int 8;;
- : string = "8"
```

Dans cet exemple, comme le précise clairement le type inféré ($int \rightarrow 'a$), l'argument g est une fonction.

— comme résultat d'une autre fonction :

```
# function x -> let f = function y -> y + 1 in f;;
- : 'a -> int -> int = <fun>

# (function x -> let f = function y -> y + 1 in f) "toto";;
- : int -> int = <fun>

# (function x -> let f = function y -> y + 1 in f) "toto" 1;;
- : int = 2
```

Dans cet exemple, on déclare un identificateur local f lié à la fonction «**function** $y \rightarrow y + 1$ » et on retourne comme résultat la valeur de f . Le résultat de la fonction est clairement une fonction. Le type de la fonction ne l'illustre pas clairement car au niveau des types de fonctions, il faut toujours appliquer l'associativité à droite. Par conséquent, le type « $'a \rightarrow int \rightarrow int$ » correspond implicitement au type « $'a \rightarrow (int \rightarrow int)$ », version où on voit clairement que le résultat de la fonction est une autre fonction ($int \rightarrow int$).

D'une manière plus générale, on a :

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots \rightarrow t_n \quad \equiv \quad t_1 \rightarrow (t_2 \rightarrow (t_3 \rightarrow (\dots \rightarrow t_n) \dots))$$

— comme identificateur local à une autre fonction (c'est-à-dire définie localement dans une autre fonction) :

```
# function x -> let f = function y -> y + 1 in f x;;
- : int -> int = <fun>

# (function x -> let f = function y -> y + 1 in f x) 2;;
- : int = 3

# f;;
-
Error: Unbound value f
```

Le dernier exemple montre clairement que l'identificateur f est local à la fonction dans lequel il est défini.

Les deux derniers exemples illustrent un point intéressant : en liant la définition d'une fonction à un identificateur, elle peut alors être appliquée, autant de fois que désiré, en n'utilisant que cet identificateur (désormais son identifiant). Autrement dit :

```
# let inc = function x -> x + 1;;
val inc : int -> int = <fun>

# inc 4;;
- : int = 5
```

On retrouve, en quelque sorte, la notation utilisée en mathématiques :

$$\begin{aligned} inc &: int \rightarrow int \\ x &\mapsto x + 1 \end{aligned}$$

Par ailleurs, OCaml offre une forme syntaxique permettant d'abrégier ce type d'expression :

```
# let inc x = x + 1;;
val inc : int -> int = <fun>

# inc 4;;
- : int = 5
```

D'une manière plus générale, on a :

$$\text{let } f \text{ } m_1 \dots m_n = e \quad \equiv \quad \text{let } f = \text{function } m_1 \text{ -> } \dots \text{function } m_n \text{ -> } e$$

Une autre abréviation permet de simplifier la définition de fonctions anonymes admettant plusieurs paramètres de manière curryfiée :

$$\text{fun } m_1 \dots m_n \text{ -> } e \quad \equiv \quad \text{function } m_1 \text{ -> } \dots \text{function } m_n \text{ -> } e$$

En résumé :

1. Quand on veut définir une fonction anonyme, on a deux possibilités :
 - (a) Si la fonction requiert qu'un seul argument (qu'il soit un tuple ou pas), on peut alors utiliser la forme : **function** $m \text{ -> } e$
 - (b) Si la fonction requiert plus d'un argument, en forme curryfiée, alors il est plus adéquat d'utiliser cette forme : **fun** $m_1 \dots m_n \text{ -> } e$

L'utilisation de **function** demeure possible mais moins intéressante :

$$\text{function } m_1 \text{ -> } \dots \text{function } m_n \text{ -> } e$$

La différence entre **fun** et **function** se situe donc au niveau du nombre de paramètres (successifs, en forme curryfiée) qu'ils peuvent admettre.

2. Quand on veut définir une fonction ayant un nom (associé à un identificateur), la forme suivante est appropriée : **let** $f \text{ } m = e$

Bien sûr, cette forme supporte la définition de fonctions admettant plusieurs arguments de manière curryfiée :

$$\text{let } f \text{ } m_1 \dots m_n = e$$

Et évidemment, pour ce type de fonctions, l'utilisation de **function** ou de **fun** est possible :

$$\text{let } f = \text{fun } m_1 \dots m_n = e$$

$$\text{let } f = \text{function } m_1 \text{ -> } \dots \text{function } m_n \text{ -> } e$$

Dans le reste de la matière, lorsqu'il sera nécessaire de définir une fonction anonyme, le mot-clé **fun** sera utilisé, autrement ce sera **let**. Cependant, ces constructions syntaxiques seront toujours considérées comme des abréviations (on parle d'ailleurs de *sucre syntaxique*) puisque le mot-clé **function** suffit pour définir toutes les formes de fonctions désirées. D'ailleurs c'est ce qu'on retrouve dans le λ -calcul, calcul dans lequel n'est considérée qu'une forme de définition de fonction : $\lambda x.e$. En OCaml, cette forme correspond clairement à **function** $x \text{ -> } e$. Avec cette forme syntaxique, $\lambda x.e$, il sera possible de considérer toute sorte de fonctions. Aussi, en y associant la construction syntaxique **let** $x = e \text{ in } e'$, il sera alors possible de lier une fonction à un identificateur pour en simplifier l'utilisation.

Pour terminer, voici la version correspondante en λ -calcul de quelques fonctions définies dans la présente section :

- $(\lambda x.x + 1) 5$
- **let** $x = 5 \text{ in } x + 1$

```

—  $\lambda x.(x+1, x-1)$ 
—  $\lambda x.\lambda y.x + y$ 
—  $(\lambda x.\lambda y.\lambda z.x + y + z) (2-1) (\max 5 6) (\text{abs } 5)$ 
—  $\lambda g.\lambda x.\text{if } x \bmod 2 = 0 \text{ then } g\ x \text{ else } g\ (x+1)$ 
—  $\lambda x.\text{let } f = \lambda y.y + 1 \text{ in } f$ 
—  $\lambda x.\text{let } f = \lambda y.y + 1 \text{ in } f\ x$ 

```

2.6.1 Fonctions récursives

Pour la définition de fonctions récursives, il faut utiliser la construction syntaxique **let rec** :

```

# let rec fact n =
  if n <= 1 then 1 else n * fact (n-1);;
val fact : int -> int = <fun>

# fact 4;;
- : int = 24

```

2.6.2 Fonctions polymorphes

Une fonction est dite polymorphe si et seulement si l'un de ses arguments est de type variable, autrement dit la fonction agit sur des arguments de types différents. Un type polymorphe doit donc comporter des variables de type. Le type d'une fonction polymorphe est toujours polymorphe, et la collection des types pour lesquels elle est définie est l'ensemble infini déterminé par les instances du type polymorphe.

L'exemple typique de fonction polymorphe est la fonction identité qui peut être appliquée à n'importe quelle valeur :

```

# let id x = x;;
val id : 'a -> 'a = <fun>

# id 4;;
- : int = 4

# id "toto";;
- : string = "toto"

# id (+);;
- : int -> int -> int = <fun>
#

```

2.6.3 Filtrage par motifs

Dans le corps d'une fonction, il est évidemment possible d'utiliser le filtrage afin de déconstruire, par exemple, une valeur passée en argument de la fonction. Pour illustrer ces propos, prenons l'exemple d'une fonction qui teste si une liste est nulle :

```

# let est_vide l = match l with
| [] -> true
| _ -> false;;
val est_vide : 'a list -> bool = <fun>

# est_vide [];;
- : bool = true

# est_vide [1;2;3];;
- : bool = false

```

La fonction `est_vide` est définie pour deux cas possibles : Si l'argument est une liste égale à [], alors la fonction retourne `true`. Sinon, l'argument filtre forcément le motif `_`, elle retourne `false`, ce qui est conforme à l'intuition.

Voici d'autres exemples de définitions de fonctions par filtrage :

```
# let rec member e l = match l with
| [] -> false
| e'::r -> (e = e') || (member e r);;
val member : 'a -> 'a list -> bool = <fun>

# member 5 [1;2;3;4;5;6];;
- : bool = true
```

```
# let rec maMap f l = match l with
| [] -> []
| e::r -> (f e)::(maMap f r);;
val maMap : ('a -> 'b) -> 'a list -> 'b list = <fun>

# maMap succ [1;2;3;4];;
- : int list = [2; 3; 4; 5]
```

Ce dernier exemple définit une fonction permet d'appliquer un traitement (argument `f`) à tous les éléments d'une liste passée en argument, et de retourner la liste des résultats de ces applications.

Par ailleurs, plusieurs des exemples présentés dans la section 2.5 (page 37) peuvent produire des fonctions utiles. À titre d'exemples :

```
# let estVoyelle c = match c with
| 'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
| _ -> false;;
val estVoyelle : char -> bool = <fun>

# let estNum c = match c with
| '0' .. '9' -> true
| _ -> false;;
val estNum : char -> bool = <fun>

# let estAlpha c = match c with
| 'a' .. 'z' | 'A' .. 'Z' -> true
| _ -> false;;
val estAlpha : char -> bool = <fun>
```

```
# let egalRat r = match r with
| (_,0),(_,0) -> true
| (_,0),_ -> false
| _,(_,0) -> false
| (n1,1), (n2,1) when n1 = n2 -> true
| (n1,d1), (n2,d2) when ((n1 * d2) = (n2 * d1)) -> true
| _ -> false;;
val egalRat : (int * int) * (int * int) -> bool = <fun>
```

Filtrage sans la construction `match-with`

Soit la définition du type «complexe» permettant de représenter les nombres complexes :

```
# type complex = C of float * float;;
type complex = C of float * float;;

# let z = C(4.,3.);;
val z : complex = C (4., 3.)
```


On peut alors définir, par filtrage, une fonction «norm» permettant de normaliser un nombre complexe :

```
# let norm z = match z with (C (re,im)) -> sqrt (re ** 2. +. im ** 2.);;
  val norm : complex -> float = <fun>

# norm z;;
- : float = 5.
```

Cependant, Ocaml permet de définir cette fonction sans utiliser les mots-clés **match-with** :

```
# let norm' (C (re,im)) = sqrt (re ** 2. +. im ** 2.);;
  val norm' : complex -> float = <fun>
```

L'argument de la fonction «norm» est un motif que doit filtrer tout argument effectif de cette fonction lors de son application. Dans l'exemple qui suit, l'argument, étant de type *complex*, filtre naturellement le motif de la fonction «norm'» :

```
# norm' (C(4.,3.));;
- : float = 5.
```

Par ailleurs, on ne peut définir, à l'aide des mots-clés **let** et **fun**, des fonctions *directement* par filtrage lorsque le type de données sur lequel s'effectue le filtrage comprend plus d'un constructeur :

```
# type value = Int of int | Float of float;;

# let inc (Int i) = Int (i + 1)
    | inc (Float f) = Float (f +. 1.);;
  -
  Error: Syntax error

# fun (Int i) -> Int (i + 1)
    | (Float f) -> Float (f +. 1.);;
  -
  Error: Syntax error
```

Par contre, le mot-clé **function** le permet :

```
# function (Int i) -> Int (i + 1)
    | (Float f) -> Float (f +. 1.);;
- : value -> value = <fun>
```

En résumé, lorsqu'un type de données comprend seulement un constructeur, il est possible de définir une fonction directement par filtrage (sans utiliser les mots-clés **match-with**) en utilisant les mots-clés **let**, **fun** et **function**. Lorsque le type de données comprend plus d'un constructeur, les seules options disponibles sont le mot-clé **function** ou les mots-clés **match-with**.

Remarque : On pourrait se questionner sur l'intérêt d'avoir des types de données définis à l'aide d'un seul constructeur, comme c'est le cas pour le type *complex* défini précédemment. En fait, il aurait été possible de définir ce type d'une manière plus simple :

```
type complex = float * float
```

Cependant, au niveau typage (et notamment inférence de types), il n'est alors plus possible de distinguer de simples paires de nombres réels des nombres complexes :

```
# (4.,3.);;
- : float * float = (4., 3.)

# let norm (re,im) = sqrt (re ** 2. +. im ** 2.);;
val norm : float * float -> float = <fun>
```

Dans ce cas, la signature de la fonction «norm» ne précise pas clairement qu'elle requiert un nombre complexe en argument ! Le seul moyen de le préciser est de passer par le typage explicite :

```
# ((4.,3.) : complex);;
- : complex = (4., 3.)

# let norm' ((re,im) : complex) = sqrt (re ** 2. +. im ** 2.);;
val norm' : complex -> float = <fun>
```

Par contre, en passant par un type inductif, même restreint à un constructeur, on distingue clairement le type *complex* :

```
# type complex = C of float * float;;
type complex = C of float * float;;

# let norm (C (re,im)) = sqrt (re ** 2. +. im ** 2.);;
val norm : complex -> float = <fun>

# C (4.,3.);;
- : complex = C (4., 3.)
```

2.6.4 Fonctions anonymes

OCaml permet de définir des valeurs fonctionnelles c'est-à-dire des fonctions sans noms ou anonymes. En effet, il est parfois nécessaire de passer en argument une fonction sans être contraint de lui donner un nom. Par exemple, grâce à la définition récursive de fonctions, au filtrage de motifs sur les listes, et aux fonctions anonymes, la fonction suivante :

```
# let rec member p liste =
  match liste with
  | [] -> false
  | e::r -> (p e) || (member p r);;
val member : ('a -> bool) -> 'a list -> bool = <fun>
```

permet de rechercher un élément dans une liste polymorphe grâce à un prédicat (fonction retournant un booléen) passé en argument. Ainsi :

```
# member (fun x -> x = 5) [1;2;3;4;5];;
- : bool = true

# member (fun x -> x = 'd') ['a';'b';'c'];;
- : bool = false

# member (fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : bool = true
```

Les deux premiers exemples permettent de chercher un éléments précis (respectivement 5 et 'd') dans une liste donnée (respectivement d'entiers et de caractères); le troisième exemple permet de chercher un entier pair dans une liste d'entiers donnée.

2.6.5 Curryfication

En programmation fonctionnelle, la notion de curryfication consiste à transformer la définition d'une fonction et ce de manière à garder la même sémantique. Cette transformation consiste à éliminer les types produits (*), i.e. le type des tuples, et à les remplacer par des types fonctions (\rightarrow). Son effet est qu'une fonction qui prenait un tuple en argument, prendrait désormais la première composante du tuple et rendrait une fonction ayant pour argument la deuxième composante du tuple et ainsi de suite. Par exemple :

```
# let curry f x y = f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let uncurry g (x,y) = g x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

La fonction `curry` prend en argument une fonction ayant comme argument une paire de valeurs, et retourne la forme curryfiée de cette fonction. Par exemple, l'opérateur `(+)`, dont le type est `int -> int -> int`. Pour pouvoir obtenir une version définie sur une paire d'entiers, il est possible de le décorryfier à l'aide de la fonction `uncurry` :

```
# (+);;
- : int -> int -> int = <fun>

# let add = uncurry (+);;
val add : int * int -> int = <fun>

# add (4,5);;
- : int = 9

# (+) 4 5;;
- : int = 9
```

De même, il est possible de retrouver la forme curryfiée de la fonction `add` en lui appliquant la fonction `curry` :

```
# let add_curry = curry add;;
val add_curry : int -> int -> int = <fun>

# add_curry 8 9;;
- : int = 17
```

L'avantage d'une forme curryfiée est de pouvoir évaluer partiellement les arguments d'une fonction. Par exemple, la fonction curryfiée `member`, définie dans la section précédente, peut être appliquée à un seul argument retournant ainsi une nouvelle fonction (permettant de rechercher dans une liste d'entiers un nombre pair) :

```
# let member_pair = member (fun x -> x mod 2 = 0);;
val member_pair : int list -> bool = <fun>

# member_pair [1;2;3;4];;
- : bool = true
```

2.6.6 Typage explicite

Comme mentionné au début du présent chapitre, il est possible de typer explicitement une fonction (comme on le ferait dans la plupart des langages de programmation) ; dans un tel cas, OCaml va inférer le type de la fonction en considérant les types explicites que vous aurez mentionnés :

```
# let incremente (x : int) : int = x + 1;;
val incremente : int -> int = <fun>

# let premier ((x : int), y) : int = x;;
val premier : int * 'a -> int = <fun>
```

Le premier exemple correspond à une définition de fonction plus standard (relativement à d'autres langages) : on y précise le type de l'argument, ainsi que celui du résultat de la fonction.

Dans le deuxième exemple, on définit notre version de la fonction «fst» qui retourne le premier élément d'une paire d'éléments; cette fonction, comme on le sait, est doublement polymorphe puisqu'il y a aucune contrainte sur le type du premier élément de la paire, ni sur celui du deuxième. Cependant, comme on peut le constater dans notre version de la fonction, le type généré est seulement polymorphe sur le deuxième élément de la paire en argument.

Le même scénario est constaté dans cette version de la fonction «snd» :

```
# let second (x , y) : int = y;;
val second : 'a * int -> int = <fun>
```

Notons, comme mentionné en début de chapitre, qu'OCaml peut traiter un code, sans aucune mention de type (le tout sera automatiquement inféré), comme il peut considérer du code explicitement typé à la constante près!

```
# let incremente (x : int) : int =
  let (y : int) = (x : int) in
  (((+): int -> int -> int) (y : int) (1: int) : int);;
val incremente : int -> int = <fun>
```

Clairement, on privilégiera l'inférence automatique de types. Aussi, on évitera ainsi de rendre monomorphe, par typage explicite, un code alors que dans les faits, il serait polymorphe.

2.6.7 Fonctions d'ordre supérieur

Nous avons déjà établi que les fonctions dans OCaml sont considérées comme des valeurs. Elles ont les mêmes droits et privilèges que toute autre valeur du langage. En particulier, cela signifie que les fonctions peuvent être passées comme des arguments à d'autres fonctions et les applications peuvent s'évaluer en fonctions. Les fonctions qui traitent des fonctions de cette manière sont appelées fonctions d'ordre supérieur.

Dans cette section, nous présentons un ensemble de fonctions d'ordre supérieur définies (dans le module List) sur les listes. Par la suite, nous présentons l'utilisation de telles fonctions dans des structures de données inductifs.

Listes et fonctions d'ordre supérieur

La fonction `exists` teste s'il existe un élément d'une liste donnée qui satisfait un prédicat donné (cette fonction correspond à la fonction «member» définie dans une précédente section). Les exemples qui suivent illustrent l'utilisation de cette fonction :

```
# List.exists (fun (x,y) -> x <= y) [(7,5);(3,4);(8,6)];;
- : bool = true

# List.exists (fun (x,y) -> x = y) [("a","b");("a","c")];;
- : bool = false
```

Les deux exemples testent s'il existe dans la liste passée en argument un élément qui satisfait le prédicat (ou la condition) passé en argument. Dans le premier cas, le prédicat correspond à

l'opérateur `<=`. Il s'agit alors de tester s'il existe une paire dans la liste `[(7,5);(3,4);(8,6)]` telle que l'opérateur `<=` appliqué à cette paire retourne la valeur `true`. Comme la paire `(3,4)` satisfait cette condition, le résultat de la fonction `exists` est `true`. Dans le deuxième cas, il n'existe pas de paire telle que l'opérateur `=` appliqué à celle-ci retourne la valeur `true`. Par conséquent, la fonction `exists` retourne `false`.

À l'instar de la fonction `exists`, les fonctions `map`, `iter` et `fold_right` (`fold_left`) sont d'ordre supérieur puisqu'elles admettent comme premier argument une autre fonction. La première applique une fonction à chaque élément de la liste obtenant ainsi une liste de résultats, et retourne cette liste comme résultat final. La deuxième applique à chaque élément de la liste une fonction donnée, et retourne la valeur `()`. La troisième (resp. `fold_left`) applique tout d'abord une fonction au dernier élément (resp. premier élément) de la liste avec une valeur initiale obtenant ainsi une nouvelle valeur. Puis elle applique cette même fonction à l'avant dernier élément (resp. deuxième élément) avec comme valeur initiale la valeur retournée par la première évaluation, et ainsi de suite. Donc, étant donnée une liste `l` de valeurs `[v1;...;vn]`, nous avons :

- `map f l = [f(v1); f(v2); ...; f(vn)]`
- `iter f l = f(v1); f(v2); ...; f(vn); ()`
- `fold_right f l v0 = f v1 (f v2 ... (f vn-1 (f vn v0)) ...)`
- `fold_left f v0 l = f (... (f (f v0 v1) v2) ... vn-1) vn`

On retrouve la signification de ces fonctions dans leur type respectif :

```
# map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

# iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>

# fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Les exemples qui suivent illustrent l'utilisation des fonctions `map` et `iter` :

```
# map string_of_int [1;2;3];;
- : string list = ["1"; "2"; "3"]

# map string_of_float (map sqrt [4.;16.]);;
- : string list = ["2."; "4."]

# iter print_string ["1";"2";"3"];;
123- : unit = ()
```

Les exemples qui suivent illustrent l'utilisation de la fonction `fold_right` :

```
# fold_right (@) [[1;2];[3;4;5];[6]] [];;
- : int list = [1; 2; 3; 4; 5; 6]

# fold_right (+) [1;2;3;4;5] 0;;
- : int = 15

# fold_right max [2;4;1;8;5;3] min_int;;
- : int = 8

# fold_right min [2;4;1;8;5;3] max_int;;
- : int = 1

# let max_list l = fold_right max l min_int;;
  val max_list : int list -> int = <fun>

# max_list [2;4;1;8;5;3];;
- : int = 8
```

À l'aide de la fonction `fold_right`, et des fonctions `min` et `max`, il est possible de calculer le maximum ou le minimum d'une liste d'entiers. Dans les deux cas, l'évaluateur procède comme suit : en considérant la liste `[2;4;1;8;5;3]`, il applique, en commençant par le dernier élément de la liste, la fonction `max` aux valeurs 3 et `min_int`, obtenant ainsi la valeur 3; puis il applique la fonction `max` aux valeurs 5 et 3, c'est-à-dire à l'avant dernier élément de la liste et au résultat de l'évaluation précédente : il obtient la valeur 5. Et ainsi de suite, en ayant à chaque fois comme deuxième argument de la fonction `max` la plus grande valeur des éléments de la liste déjà traités. Quand il atteint le premier élément de la liste, il applique la fonction `max` aux valeurs 2 et 8 et retourne le résultat de cette évaluation, soit 8, comme résultat final de l'expression `fold_right max [2;4;1;8;5;3] min_int`.

Types inductifs et fonctions d'ordre supérieur

Le type prédéfini `'a list` désignant l'ensemble des listes polymorphes peut être défini comme suit :

```
# type 'a liste = Vide | Cons of 'a * 'a liste;;
type 'a liste = Vide | Cons of 'a * 'a liste

# Cons(1,Vide);;
- : int liste = Cons (1, Vide)

# Cons("a",Cons("b",Vide));;
- : string liste = Cons ("a", Cons ("b", Vide))
```

Cette définition suggère que le type `'a liste` est composé soit de la valeur `Vide`, soit de valeurs construits à partir du constructeur «Cons», c'est-à-dire de valeurs composées d'une valeur de type `'a` et d'une liste de type `'a liste`.

Il est alors possible de définir une fonction qui s'apparente à la fonction prédéfinie «map» illustrée dans la section précédente :

```
# let rec map_liste f l =
  match l with
  | Vide -> Vide
  | Cons(e,r) -> Cons(f e, map_liste f r);;
val map_liste : ('a -> 'b) -> 'a liste -> 'b liste = <fun>

# map_liste string_of_int (Cons(1,Cons(2,Cons(3,Vide))));;
- : string liste = Cons ("1", Cons ("2", Cons ("3", Vide)))
```

De même, un arbre polymorphe peut être défini comme suit :

```
# type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre;;
type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre
```

Par conséquent, un arbre contenant des éléments d'un type quelconque, correspond soit à une feuille contenant un de ces éléments, soit un nœud contenant deux sous-arbres. Il s'agit en fait de la définition d'un arbre binaire polymorphe. Par exemple `feuille (1)` correspond à un `int arbre` (un arbre d'entiers), `Noeud(Noeud(Feuille(1), Feuille (2)), feuille (3))` correspond à un autre arbre d'entiers contenant les valeurs 1, 2 et 3. Voici quelques exemples de définitions d'arbres binaires :

```
# let arb_int = Noeud(Noeud( Feuille 1, Noeud( Feuille 2, Feuille 3)), Feuille 4);;
val arb_int : int arbre =
  Noeud (Noeud (Feuille 1, Noeud (Feuille 2, Feuille 3)), Feuille 4)

# let arb_str = Noeud(Feuille "f1", Feuille "f2");;
val arb_str : string arbre = Noeud (Feuille "f1", Feuille "f2")

# let arb_float = Feuille 1.;;
val arb_float : float arbre = Feuille 1.
```

L'arbre `arb_int` définit un arbre binaire contenant les entiers 1, 2, 3 et 4. L'arbre `arb_str` définit un arbre binaire contenant les chaînes de caractères "f1" et "f2". L'arbre `arb_real` définit un arbre contenant une seule feuille, le réel 1.0.

À partir de la définition de la structure de données `'a arbre`, il est possible de définir des fonctions qui la manipulent. L'exemple qui suit définit deux versions d'un réducteur (communément appelé fonction *reduce* ou *fold*) sur les arbres :

```
# let fold_arb z c =
  let rec f a =
    match a with
    | Feuille v -> z v
    | Noeud(g,d) -> c (f g) (f d)
  in
  f;;
val fold_arb : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>

# let rec fold_arb' z c a =
  match a with
  | Feuille v -> z v
  | Noeud(g,d) -> c (fold_arb' z c g) (fold_arb' z c d);;
val fold_arb' : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>
```

Comme nous allons le constater, l'utilité d'un réducteur est, entre autres, la possibilité de définir d'autres fonctions utiles. Il permet, comme on peut le deviner en regardant sa définition, d'appliquer un certain traitement à un arbre binaire. Il prend deux arguments, `z` et `c`. Comme nous allons le voir, la fonction `z` est appliquée aux feuilles d'un arbre, alors que `c` est appliquée aux nœuds d'un arbre. Par conséquent, la fonction `fold_arb` est une fonction générique qui permet d'appliquer n'importe quel traitement à un arbre binaire. Son rôle est similaire à celui des fonctions `fold_right` et (`fold_left`), sauf que dans ce cas, on l'applique sur des arbres binaires et non des listes. Pourquoi alors définir une telle fonction ? Et bien pour le programmeur, ça lui procure une fonction qu'il pourra utiliser quand il aura à appliquer des traitements sur des arbres binaires. Tout ce que le programmeur aura à faire, est de se dire que dois-je appliquer aux feuilles et que dois-je appliquer aux nœuds ?

Concernant la définition de cette fonction, elle est constituée de la déclaration locale d'une fonction `f`, et le corps de la fonction `fold_arb` correspond en fait à `f`. Par conséquent, quand vous appelez la fonction `fold_arb` en lui spécifiant deux arguments (correspondant à `z` et `c`), elle retourne

comme résultat une fonction, qui correspond en fait à f . Cette fonction, f , s'applique à des arbres binaires, et est définie par filtrage. Si l'arbre qu'on a en argument correspond à *feuille* (v), alors le comportement de la fonction consiste à appliquer z à v . Si l'argument correspond à *Noeud*(g, d), alors il faut appliquer récursivement f à g et à d , qui correspondent aux deux sous-arbres, et à appliquer ensuite c aux résultats de ces applications. D'après l'utilisation de z et c , on peut déjà deviner que z est appliquée à des éléments quelconques, et que c est appliquée à des paires d'éléments. Pour s'en convaincre, il suffit d'analyser le type de `fold_arb` :

```
val fold_arb = fn : ('a -> 'b) -> ('b * 'b -> 'b) -> 'a arbre -> 'b
```

On voit bien que le premier argument de `fold_arb` est une fonction (z), le deuxième est une autre fonction (c) qui prend en argument une paires de valeurs, le troisième argument (correspondant à l'argument de f) est un arbre binaire, et le résultat de la fonction est une valeur polymorphe $'b$.

Supposons par exemple, qu'on applique la fonction `fold_arb` à l'arbre binaire suivant : `Noeud(Noeud(Feuille(1), Feuille(2)), Feuille(3))`. En considérant des fonctions z et c quelconques, on aurait alors comme résultat, le résultat de l'expression suivante :

`c(c(z(1), z(2)), z(3))`

Donc, ça correspond d'une certaine manière à un parcours en profondeur d'abord d'un arbre binaire en appliquant z aux feuilles et c aux nœuds.

Essayons maintenant d'exploiter concrètement cette fonction. À cet effet, nous allons définir à l'aide de ce réducteur trois fonctions permettant de faire des traitements assez usuels sur des arbres binaires :

```
# let hauteur a = fold_arb (fun x -> 1) (fun x y -> (max x y) + 1) a;;
val hauteur : 'a arbre -> int = <fun>

# let nbr_feuilles a = fold_arb (fun x -> 1) (+) a;;
val nbr_feuilles : 'a arbre -> int = <fun>

# let liste_feuilles a = fold_arb (fun x -> [x]) (@) a;;
val liste_feuilles : 'a arbre -> 'a list = <fun>
```

Comme on peut le constater, la définition de chacune de ces fonctions requiert une ligne de code. Le programmeur doit se contenter de fournir le code des fonctions z et c .

Comment obtenir (deviner) le code de ces définitions ?

- Pour la fonction `hauteur`, qui calcule la hauteur d'un arbre, il suffit de se dire que la hauteur d'un arbre correspondant à une feuille est 1 (`fun x -> 1`) et la hauteur d'un arbre contenant 2 sous arbres est la hauteur maximale des 2 sous arbres + 1 (`fun x y -> (max x y) + 1`).
- Pour la fonction `nbr_feuilles`, il s'agit d'un raisonnement similaire : Si l'arbre est une feuille, alors ce nombre vaut 1. Si l'arbre est composé de deux sous-arbres, alors il faut calculer le nombre de feuilles dans chaque sous-arbre, puis les additionner. L'opérateur `+` abrège la fonction `fun x y -> x + y`.
- Pour la fonction `liste_feuilles`, il s'agit du même raisonnement que celui de la fonction précédente.

Étant donné l'arbre `arb_int` défini précédemment, nous avons :

```
# hauteur arb_int;;
- : int = 4

# nbr_feuilles arb_int;;
- : int = 4

# liste_feuilles arb_int;;
- : int list = [1; 2; 3; 4]
```


2.6.8 Variants polymorphes et types ouverts

La définition de fonctions utilisant le filtrage par motifs peut évidemment manipuler des variants polymorphes :

```
# let toInt x = match x with
| 'One -> 1
| 'Two -> 2
| 'Three -> 3
| 'Four -> 4;;
val toInt : [< 'Four | 'One | 'Three | 'Two ] -> int = <fun>

# toInt 'One;;
- : int = 1

# toInt 'Five;;
-----
Error: This expression has type [> 'Five ]
      but an expression was expected of type
      [< 'Four | 'One | 'Three | 'Two ]
      The second variant type does not allow tag(s) 'Five
```

Cet exemple permet de comprendre le sens du symbole < dans le type de la fonction «toInt», c'est-à-dire le type : [< 'Four | 'One | 'Three | 'Two]. Ce type précise que la fonction «toInt» peut être appliquée à une valeur correspond soit à 'One, 'Two, 'Three ou 'Four. Il est donc normal qu'en appliquant la fonction «toInt» à la valeur 'Five, on nous signale une erreur.

Il existe aussi des types de variants polymorphes appelés types ouverts. Voici un exemple :

```
# let toInt ' x = match x with
| 'One -> 1
| 'Two -> 2
| 'Three -> 3
| 'Four -> 4
| _ -> 0;;
val toInt : [> 'Four | 'One | 'Three | 'Two ] -> int = <fun>

# toInt ' 'One;;
- : int = 1

# toInt ' 'Five;;
- : int = 0

# toInt ' ('I 5);;
- : int = 0
```

Dans ce cas, le type [> 'Four | 'One | 'Three | 'Two] est dit ouvert (symbole >) et indique que la fonction peut être appliquée à 'One, 'Two, 'Three, 'Four ou tout autre variant polymorphe (comme 'Five ou 'I 5).

Pour terminer voici un exemple avancé de l'utilisation des variants polymorphes combinée avec l'inférence de types d'OCaml :

```
# let f x = match x with
| 'A -> 'B
| 'C -> 'D
| x -> x;;
val f : ([> 'A | 'B | 'C | 'D ] as 'a) -> 'a = <fun>

# f 'A;;
- : [> 'A | 'B | 'C | 'D ] = 'B
```

Notez le résultat des évaluations dynamique et statique de l'application de «f» à «'A». Voici un autre exemple d'application de cette fonction :

```
# f 'E;;
- : [> 'A / 'B / 'C / 'D / 'E ] = 'E
```

On constate ici que le type du résultat a été étendu par rapport à celui de l'exemple précédent !

2.6.9 Paramètres étiquetés et optionnels

Ocaml offre la possibilité d'étiqueter les paramètres de fonctions (ou de méthodes, dans le cadre orienté objet). En procédant ainsi, il est possible, lors de l'appel d'une fonction (ou de l'invocation d'une méthode), de préciser les arguments dans l'ordre de notre choix.

Pour étiqueter un paramètre, il suffit de le précéder par le symbole «~». Voici un exemple :

```
# let f ~x ~y = x - y;;
val f : x:int -> y:int -> int = <fun>
```

Ocaml indique que le type de la fonction f, à la place d'être simplement `int -> int -> int`, comprend des paramètres étiquetés (`x:int` et `y:int`).

Il sera donc possible d'invoquer f avec les arguments précisés dans le choix voulu :

```
# f 3 5;;
Warning 6 [labels-omitted]: labels x, y were omitted in the application of this function.
- : int = -2

# f 5 3;;
Warning 6 [labels-omitted]: labels x, y were omitted in the application of this function.
- : int = 2

# f ~x:3 ~y:5;;
- : int = -2

# f ~y:5 ~x:3;;
- : int = -2
```

Le dernier exemple montre comment il est possible de spécifier l'argument y avant x. Les deux premiers exemples montrent qu'il est possible d'appeler f de manière standard sans se soucier des étiquettes associées aux paramètres.

Bien sûr, il est toujours possible d'appliquer f partiellement :

```
# let g = f ~y:5;;
val g : x:int -> int = <fun>

# g 3;;
Warning 6 [labels-omitted]: labels x, y were omitted in the application of this function.
- : int = -2

# g ~x:3;;
- : int = -2
```

Notons qu'Ocaml propose plusieurs modules, correspondant à des modules standards, proposant des fonctions ayant des paramètres étiquetés. Par exemple, on retrouve dans le module «ListLabels» (qui comprend les mêmes fonctions que celles définies dans le module «List»), les fonctions suivantes :

```
# ListLabels.map;;
- : f:( 'a -> 'b) -> 'a list -> 'b list = <fun>

# ListLabels.fold_right;;
- : f:( 'a -> 'b -> 'b) -> 'a list -> init:'b -> 'b = <fun>

# ListLabels.filter;;
- : f:( 'a -> bool) -> 'a list -> 'a list = <fun>
```

ce qui permet, par exemple, d'utiliser la fonction `fold_right` en précisant la fonction «f» et la valeur «init», tous deux arguments de `fold_right`, dans l'ordre souhaité :

```
# ListLabels.fold_right ~init:0 ~f:(+) [1;2;3;4;5];;
- : int = 15
```

La page qui suit :

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html>

propose d'autres modules, comme `StringLabels`, offrant le même type de fonctions à paramètres étiquetés.

Par ailleurs, `Ocaml` permet de déclarer des paramètres optionnels. Étant optionnels, il est nécessaire de leur préciser une valeur par défaut :

```
# let incr ?(pas = 1) x = x + pas;;
val incr : ?pas:int -> int -> int = <fun>

# incr 4;;
- : int = 5

# incr ~pas:10 4;;
- : int = 14
```

2.7 Caractéristique impératives

Cette section est consacrée aux propriétés impératives du langage `Ocaml`.

2.7.1 Références

À l'instar des langages de programmation impérative, `OCaml` fournit des moyens pour traiter les références et les affectations. Cependant, à la différence des langages tels que `C`, `OCaml` exclut l'arithmétique des pointeurs.

L'utilisation des références se fait à travers un nouveau type polymorphe nommé `'a ref`, permettant de représenter une référence vers une valeur d'un certain type, ainsi que des fonctions et opérateurs :

- **ref** est une fonction qui alloue de l'espace mémoire dans le tas (*heap* en anglais) pour la valeur passée en argument, et qui retourne la référence de cet emplacement.
- «!» est un opérateur qui s'applique à une référence et qui retourne comme résultat le contenu de l'emplacement donné par la valeur de cette référence.
- «:=» est un opérateur qui permet de modifier le contenu référé par une référence.

Les exemples suivant illustrent leur utilisation :

```
# let x = ref 3;;
val x : int ref = {contents = 3}

# !x;;
- : int = 3

# let z = ref 3;;
val z : int ref = {contents = 3}

# !x = !z;;
- : bool = true

# let t = x + 1;;
Characters 8-9:
  let t = x + 1;;
           ^
Error: This expression has type int ref but is here used with type int
```

La première expression définit un identificateur *x* auquel on associe la référence d’une valeur entière 3. L’expression qui suit accède à la valeur référée par *x*. La quatrième expression teste l’égalité du contenu de deux références *x* et *z*. L’égalité entre deux références n’est pas permise en OCaml. De même, l’arithmétique des références est interdite. La dernière expression illustre ces propos.

Notons que la double indirection est supportée par le langage comme l’illustrent ces exemples :

```
# let y = ref x;;
val y : int ref ref = {contents = {contents = 4}}

# !y;;
- : int ref = {contents = 4}

# !(!y);;
- : int = 4
```

Par ailleurs, voici un exemple illustrant l’utilisation de références de fonctions :

```
# let pf = ref (fun x -> x + 1);;
val pf : (int -> int) ref = {contents = <fun>}

# !pf 4;;
- : int = 5
```

```
# pf := (fun x -> x - 1);;
- : unit = ()

# !pf 4;;
- : int = 3
```

L’exemple suivant illustre l’expressivité du langage OCaml et des langages fonctionnels. Dans cet exemple, on définit deux fonctions qui partagent une même référence *x*. Autrement dit, la référence *x* est à la fois globale, puisque visible dans deux fonctions différentes, *get* et *set*, mais aussi privée puisqu’accessible seulement à ces deux fonctions :

```
# let get, set =
  let x = ref 0 in
    (fun () -> !x), (fun x' -> x:=x'; !x);;
val get : unit -> int = <fun>
val set : int -> int = <fun>

# get();;
- : int = 0

# set 5;;
- : int = 5

# get();;
- : int = 5
```

2.7.2 Structures de contrôle

Certaines structures de contrôle ont un aspect impératif, comme par exemple les boucles. Dans cette section, on complète la syntaxe du langage en présentant ces structures de contrôle.

Séquence : Une séquence d'expressions s'écrit comme suit en OCaml :

begin $e_1; e_2; \dots e_n$ **end**

À la place des mots-clés **begin** et **end**, on peut utiliser des parenthèses :

```
# begin
  print_string "First_line\n";
  print_string "Second_line\n";
  print_string "Third_line\n"
end;;
First_line
Second_line
Third_line
- : unit = ()
```

Boucles for : Deux formes de commande sont disponibles :

for $id = e_1$ **to** e_2 **do** e_3 **done**
for $id = e_1$ **downto** e_2 **do** e_3 **done**

Ces commandes utilisent implicitement des références :

```
# for i = 1 to 10 do
  print_int i;
  print_string "_"
done;
print_newline();;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

```
# for i = 10 downto 1 do
  print_int i;
  print_string "_"
done;
print_newline();;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

Boucles *while* : Voici la syntaxe à respecter :

while e_1 do e_2 done

Cette commande utilise explicitement des références :

```
# let i = ref 1 in
  while !i <= 10 do
    print_int !i;
    print_string "_";
    incr i
  done;
  print_newline();;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

La fonction `incr` est une fonction prédéfinie qui permet d'incrémenter un entier référencé par une référence :

```
# incr;;
- : int ref -> unit = <fun>
```

2.7.3 Enregistrements à champs modifiables

Le langage OCaml offre la possibilité de définir des champs modifiables (*mutable*) dans les enregistrements :

```
# type etudiant = { mat: int; mutable note: float };;
type etudiant = { mat: int; mutable note: float }

# let e1 = { mat = 11111111; note = 87.0 };;
e1 : etudiant = { mat = 11111111; note = 87. }

# e1.note;;
- : float = 87.

# e1.note <- 92.0;;
- : unit = ()

# e1.note;;
- : float = 92.
```

Références revisitées

L'exemple suivant illustre comment les références sont implantées en interne (grâce aux enregistrements à champs modifiables) :

```
# { contents=8 };;
- : int ref = { contents = 8 }
```

Par ailleurs, voici une implantation d'un type `'a _ref` polymorphe, qui simule les références, ainsi que des fonctions et opérateurs associés :

```
# type 'a _ref = {mutable contenu : 'a};;
type 'a _ref = { mutable contenu : 'a; }

# let (!!) x = x.contenu;;
val ( !! ) : 'a _ref -> 'a = <fun>

# let ref' x = {contenu = x};;
val ref' : 'a -> 'a _ref = <fun>

# let (<-->) r x = r.contenu <- x;;
val ( <--> ) : 'a _ref -> 'a -> unit = <fun>

# let x = ref' 4;;
val x : int _ref = {contenu = 4}

# !!x;;
- : int = 4

# x <-- 6;;
- : unit = ()

# !!x;;
- : int = 6
```

On peut constater que les types des différents opérateurs et fonctions définis

```
# ref;;
- : 'a -> 'a _ref = <fun>
# (!!);;
- : 'a _ref -> 'a = <fun>
# (<-->);;
- : 'a _ref -> 'a -> unit = <fun>
```

sont similaires ceux des fonctions et opérateurs prédéfinis dans OCaml :

```
# ref;;
- : 'a -> 'a ref = <fun>
# (!);;
- : 'a ref -> 'a = <fun>
# (:=);;
- : 'a ref -> 'a -> unit = <fun>
```

2.7.4 Tableaux

À l'instar d'une liste, un tableau est une séquence d'objets de même type. Sa forme générale est :

$$[e_1; \dots; e_n]$$

L'évaluation dynamique d'un tableau d'expressions est un tableau de valeurs correspondant à l'évaluation de chaque expression :

$$[v_1; \dots; v_n]$$

L'évaluation statique de cette forme syntaxique est (où t correspond au type des expressions e_i) :

$$t \text{ array}$$

L'expression $[]$ représente le tableau vide. Par ailleurs,

- la «notation pointée» permet d'accéder à un i -ème élément : «arr.(i)».
- il est possible de modifier l'élément d'un tableau : «arr.(i) <- v»
- le module prédéfini Array offre plusieurs fonctions intéressantes : création, concaténation, réducteurs, tri, etc.

Voici quelques exemples d'utilisation des tableaux :

```
# [|]|;;
- : 'a array = [|]|

# let tab1 = [|1;2;3;4;5|];;
  val tab1 : int array = [|1; 2; 3; 4; 5|]

# tab1.(2);;
- : int = 3

# tab1.(4) <- 10;;
- : unit = ()

# tab1;;
- : int array = [|1; 2; 3; 4; 10|]
```

Par ailleurs, les tableaux peuvent contenir tout type de valeurs :

```
# let tab2 = [|(+);(-)|];;
  val tab2 : (int -> int -> int) array = [|<fun>; <fun>|]

# tab2.(0) 3 4;;
- : int = 7

# tab2.(0) <- ( * );;
- : unit = ()

# tab2.(0) 3 4;;
- : int = 12
```

Aussi, le module `Array` offre plusieurs fonctions permettant de créer des tableaux d'éléments :

```
# open Array;;

# let tab1 = make 10 0.;;
  val tab1 : float array = [|0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.|]

# let tab2 = make 3 (make 2 0);;
  val tab2 : int array array = [| [|0; 0|]; [|0; 0|]; [|0; 0|] |]

# tab2.(0).(0) <- 1;;
- : unit = ()

# tab2;;
- : int array array = [| [|1; 0|]; [|1; 0|]; [|1; 0|] |]

# let tab3 = make_matrix 3 2 0;;
  val tab3 : int array array = [| [|0; 0|]; [|0; 0|]; [|0; 0|] |]

# tab3.(1).(1) <- 1;;
- : unit = ()

# tab3;;
- : int array array = [| [|0; 0|]; [|0; 1|]; [|0; 0|] |]
```

La plupart des fonctions d'ordre supérieur définies sur les listes, sont définies pour les tableaux : `map`, `mapi` (`map` avec un index), `fold_left`, `fold_right`, `iter`, `iteri`, `to_list`, `of_list`, `append`, `concat`, `sub`, `sort`, etc.


```
# map string_of_float tab1;;
- : string array =
  [|"0."; "0."; "0."; "0."; "0."; "0."; "0."; "0."; "0."; "0."|]

# mapi;;
- : (int -> 'a -> 'b) -> 'a array -> 'b array = <fun>

# mapi (fun i x -> x +. (float_of_int i)) tab1;;
- : float array = [|0.; 1.; 2.; 3.; 4.; 5.; 6.; 7.; 8.; 9. |]
```

Pour terminer, voici l'implantation, dans un style impératif, de la fonction prédéfinie «mapi» :

```
# let imap f tab =
  match tab with
  | [] -> []
  | _ -> let n = length tab in
    let tab' = make n (f tab.(0)) in
    for i = 1 to n-1 do
      tab'.(i) <- f tab.(i)
    done;
    tab';
  val imap : ('a -> 'b) -> 'a array -> 'b array = <fun>

# imap succ (make 5 0);;
- : int array = [|1; 1; 1; 1; 1|]
```

2.7.5 Exceptions

Supposons que nous voulions définir une fonction `head` qui retourne la tête d'une liste. La tête d'une liste non vide est facile à obtenir, mais qu'en est-il de la tête d'une liste vide, égale à [] ?

```
# let head l = List.hd l;;
val head : 'a list -> 'a = <fun>

# head [1;2;3];;
- : int = 1

# head [];;
Exception: Failure "hd".
```

Comme l'illustrent ces exemples, il apparaît que ce cas provoque une erreur lors de l'exécution. Pour éviter cette erreur, il est possible de retourner une valeur par défaut. Mais cette solution n'est pas souhaitable car elle limiterait l'utilisation d'une telle fonction. En effet, si la tête de la liste contient une valeur égale à la valeur par défaut, alors il est impossible de savoir si la valeur retournée correspond à la valeur par défaut d'une liste vide ou si elle correspond au premier élément d'une liste.

Pour résoudre efficacement ce genre de situations, OCaml possède un mécanisme de traitement d'exceptions. Son but est de fournir les moyens à une fonction de ne pas s'arrêter brutalement et de traiter l'erreur de façon élégante. Dans ce cas, la fonction `head` est définie comme suit :

```
# exception Head;;
exception Head

# let head l =
  match l with
  | [] -> raise Head
  | h::_ -> h;;
  val head : 'a list -> 'a = <fun>

# head [];;
Exception: Head.
```

La première expression, **exception** Head, déclare une exception Head. Puis, la fonction head est définie suivant les deux situations suivantes : si l'argument est une liste vide, alors la fonction lève l'exception Head et ce à l'aide du mot-clé **raise** ; si l'argument est une liste différente de [], soit $h :: _$, alors la fonction retourne le premier élément de cette liste. Dans cet exemple, il n'y a pas d'erreur d'exécution, mais une exception est levée. Notons qu'il existe plusieurs exceptions prédéfinies dans le langage OCaml. Par exemple, l'exception `Division_by_zero` est levée lors de la division d'un entier par la valeur 0 :

```
# 3 / 0;;
Exception : Division_by_zero.
```

De plus, OCaml offre un mécanisme de traitement de ces exceptions. En effet, grâce aux mots-clés **try** et **with**, il est possible de traiter les exceptions. Par exemple, il est possible de récupérer l'exception Head levée par la fonction head pour afficher un message d'erreur comme suit :

```
# let print_head l =
  try
    print_string ((head l) ^ "\n")
  with
    Head -> print_string "Liste_vide!\n";;
val print_head : string list -> unit = <fun>

# print_head ["Hello"; "World"];;
Hello
- : unit = ()

# print_head [];;
Liste_vide!
- : unit = ()
```

Dans cet exemple, la fonction `print_head` affiche le premier élément d'une liste d'entiers. Cette fonction utilise la fonction head pour accéder au premier élément de cette liste. De plus, elle récupère éventuellement l'exception Head levée par la fonction head et affiche un message d'erreur.

La forme générale de l'expression **try—with** est :

$$\text{try } e \text{ with } E \rightarrow e'$$

Cette forme syntaxique est évaluée comme suit : l'expression e est évaluée ; si cette évaluation aboutit à une valeur v , alors cette dernière est retournée comme résultat de l'évaluation de toute l'expression ; sinon, et si cette évaluation provoque la levée d'une exception qui filtre E , alors l'évaluateur retourne la valeur résultante de l'évaluation de l'expression e' , soit v' . Finalement, si l'exception ne filtre pas E alors celle-ci est levée. Deux remarques s'imposent : premièrement, le fait que l'évaluation de cette forme syntaxique puisse retourner soit la valeur v soit la valeur v' implique que les expressions e et e' doivent être de même type ; deuxièmement, à l'aide du motif `"_"`, il est possible de récupérer toutes les exceptions susceptibles d'être levées. Dans l'exemple qui suit, la fonction `print_div_head` affiche le premier élément d'une liste en le divisant préalablement par une valeur passée en argument :

```
# let div_and_print_head l x =
  try
    print_string ((string_of_int ((head l) / x)) ^ "\n")
  with
    | Head -> print_string "Liste_Vide!\n"
    | _ -> print_string "Autre_erreur!\n";;
val div_and_print_head : int list -> int -> unit = <fun>

# div_and_print_head [6;8;9] 2;;
3
- : unit = ()

# div_and_print_head [] 2;;
Liste_Vide!
- : unit = ()
```

```
# div_and_print_head [6;8;9] 0;;
Autre_erreur!
- : unit = ()

# div_and_print_head [] 0;;
Liste_Vide!
- : unit = ()
```

En fait il aurait été judicieux de définir cette fonction comme suit :

```
# let div_and_print_head' l x =
  try
    print_string ((string_of_int ((head l) / x)) ^ "\n")
  with
    | Head -> print_string "Liste_Vide!\n"
    | Division_by_zero -> print_string "Division_par_zéro!\n"
    | _ -> print_string "Autre_erreur!\n";;
val div_and_print_head' : int list -> int -> unit = <fun>

# div_and_print_head' [1;2;3] 0;;
Division par zéro!
- : unit = ()
```

Dans ce cas, l'évaluateur affiche pour chaque exception, un message approprié.

Filtrage par motifs et exceptions

Le dernier exemple de la précédente section aurait pu être défini comme suit :

```
# let div_and_print_head' l x =
  try
    (match l with
     | [] -> print_string "Liste_Vide!\n"
     | e::l -> print_string ((string_of_int (e / x)) ^ "\n"))
  with
    | Division_by_zero -> print_string "Division_par_zéro!\n"
    | _ -> print_string "Autre_erreur!\n";;
val div_and_print_head' : int list -> int -> unit = <fun>

# div_and_print_head' [1;2;3] 0;;
Division par zéro!
- : unit = ()
```

Il est aussi possible de le définir comme suit :

```
# let div_and_print_head' l x =
  match (List.hd l) / x with
  | r -> print_string ((string_of_int r) ^ "\n")
  | exception Division_by_zero -> print_string "Division par zéro!\n"
  | exception Failure _ -> print_string "Liste Vide!\n"
  | exception _ -> print_string "Autre erreur!\n";;
val div_and_print_head' : int list -> int -> unit = <fun>

# div_and_print_head' [1;2;3] 0;;
Division par zéro !
- : unit = ()
```

Ainsi, dans un filtrage par motifs d'une valeur, il est possible de préciser les différents motifs qui peuvent filtrer cette valeur, ainsi que les différents motifs d'exceptions qui peuvent être déclenchées suite à l'évaluation de cette valeur.

2.8 Données + Comportements = Modules

Un module est une collection de déclarations vue comme un tout. En général, il définit un type de données abstrait formé d'un ensemble de types, de constructeurs associés et de fonctions opérant sur ce type abstrait.

Ainsi, il est possible d'organiser un programme en plusieurs entités relativement indépendantes. Cela offre une plus grande flexibilité dans la mise au point de programmes de grande envergure, chaque module pouvant être implémenté séparément. De plus, la spécification du comportement d'un module est plus simple que celle de l'ensemble du programme. La réutilisation ou la mise à jour des modules favorise l'évolutivité des programmes, en offrant aux concepteurs et aux programmeurs des bibliothèques pouvant être réutilisées dans d'autres programmes.

OCaml est lui-même composé d'un ensemble de modules prédéfinis prêts à être exploités pour l'écriture de nouveaux programmes ou de nouveaux modules. Quelques-uns de ces modules ont été implicitement ou explicitement utilisés tout au long de ce chapitre. Pour accéder à l'ensemble des modules définis dans la distribution standard OCaml, il faut consulter la référence qui suit :

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html>

Le reste de la section est consacré à la présentation des différents concepts qui gravitent autour de la notion de modules, à savoir les modules, les types de modules (signatures) et les foncteurs. Chacun d'entre eux fait l'objet d'une section.

2.8.1 Module

Un module est essentiellement un environnement, c'est-à-dire un ensemble de liaisons de noms à leur valeurs qui peuvent être des types, des constantes, des fonctions ou d'autres (sous-)modules.

La forme générale de cette forme syntaxique, délimitée par les mots-clé **struct** et **end**, est appelée déclaration d'encapsulation. Pour pouvoir utiliser une structure définie, il faut la lier à un identificateur. Cette liaison se fait par l'intermédiaire du mot-clé **module**. L'exemple suivant définit un module `s` composé d'une abréviation de type, **type** entier = int, d'une déclaration d'un identificateur, **let** z = 0, et d'une déclaration d'une fonction, **let** succ x = x + 1 :

```
# module S = struct
  type entier = int
  let z = 0
  let succ x = x + 1
end;;
module S : sig type entier = int val z : int val succ : int -> int end
```

OCaml compile cette définition et génère son type, appelé signature, qui fait l'objet de la section suivante. Brièvement, une signature constitue l'interface d'un module. Elle définit, entre autres, les éléments accessibles d'un module.

Pour exploiter un module ainsi défini, deux méthodes sont disponibles :

— En utilisant la notation point de la manière suivante :

```
# S.z;;
- : int = 0

# S.succ S.z;;
- : int = 1

# let add (x : S.entier) (y : S.entier) = x + y;;
val add : S.entier -> S.entier -> int = <fun>
```

Cette façon d'accéder aux valeurs d'un module permet d'utiliser les mêmes noms d'identificateurs dans des modules différents en évitant les conflits.

— En utilisant la fonction **open**, on évite l'utilisation des qualificatifs :

```
# open S;;

# z;;
- : int = 0

# succ z;;
- : int = 1

# let add' (x:entier) (y:entier) = x + y;;
val add' : S.entier -> S.entier -> int = <fun>
```

En utilisant la fonction **open**, toutes les déclarations de ce module deviennent disponibles dans l'environnement globale.

Notons qu'il est possible d'ouvrir localement un module :

```
# let open S in succ z;;
- : int = 1

# z;;
-
Error: Unbound value z
```

Comme l'illustre l'exemple, bien que l'on ait pu utiliser les valeurs définies dans le module S, après la construction **let-in**, ces valeurs ne sont plus accessibles.

Une autre notation est possible :

```
# S.(succ z);;
- : int = 1

# z;;
-
Error: Unbound value z
```

Cette possibilité permettrait, par exemple, d'utiliser des fonctions du module List sans avoir à l'ouvrir :

```
# List.(map (exists (fun x -> x mod 2 = 0)) [[1;3;5];[2;4;6]]);;
- : bool list = [false; true]
```

Par ailleurs, le module S précédemment défini n'est pas usuel. En général, il existe une relation conceptuelle entre les types et les fonctions définies dans un module : par exemple, il est possible de définir un module Pile qui regroupe les définitions nécessaires pour créer et exploiter des piles de données :

```

# module Pile = struct
  type 'a pile = Vide | Empiler of 'a * 'a pile

  exception Pile_vide

  let vide = Vide
  let empiler e p = Empiler(e,p)

  let depiler p =
    match p with
    | Vide -> raise Pile_vide
    | Empiler(_,p') -> p'

  let sommet p =
    match p with
    | Vide -> raise Pile_vide
    | Empiler(e,_) -> e

  let rec membre e p =
    match p with
    | Vide -> false
    | Empiler(e',p') -> (e = e') || (membre e p')
end;;

```

```

module Pile :
sig
  type 'a pile = Vide | Empiler of 'a * 'a pile
  exception Pile_vide
  val vide : 'a pile
  val empiler : 'a -> 'a pile -> 'a pile
  val depiler : 'a pile -> 'a pile
  val sommet : 'a pile -> 'a
  val membre : 'a -> 'a pile -> bool
end

```

Ainsi, ce module comprend :

- la définition d'un type *'a pile* permettant de créer des piles de données polymorphes,
- la déclaration d'une exception,
- et la définition de fonctions permettant d'exploiter ces piles.

À partir de cette définition, il est possible d'ouvrir le module *Pile* et de l'exploiter :

```

# open Pile;;

# let pile_int = empiler 4 (empiler 1 vide);;
  val pile_int : int Pile.pile = Empiler (4, Empiler (1, Vide))

# sommet pile_int;;
- : int = 4

# membre 1 pile_int;;
- : bool = true

```

```
# let pile_str = empiler "c" vide;;
val pile_str : string Pile.pile = Empiler ("c", Vide)

# sommet pile_str;;
- : string = "c"

# let pile_str = empiler "a" pile_str;;
val pile_str : string Pile.pile = Empiler ("a", Empiler ("c", Vide))

# membre "a" pile_str;;
- : bool = true
```

2.8.2 Signatures

Une signature est une structure de données qui donne une spécification à un module. Elle est similaire à une déclaration, mais au lieu de donner une valeur à un identificateur, elle lui associe un type. L'expression délimitée par les mots-clé **sig** et **end** est appelée signature. À l'instar des modules, il est nécessaire de lier la définition d'une signature à un identificateur. Cette liaison se fait à l'aide des mots-clés **module type**.

En termes de génie logiciel, la signature constitue la spécification d'un programme alors que le module correspond à son implémentation. Étant plus abstraite que le module, la signature ne définit pas les corps des fonctions et ne donne, en général, pas de détail aux types déclarés. Par exemple, la signature PILE, définie ci-dessous, déclare une exception, un type et quatre fonctions :

```
# module type PILE = sig
  type 'a pile
  exception Pile_vide
  val vide : 'a pile
  val empiler : 'a -> 'a pile -> 'a pile
  val depiler : 'a pile -> 'a pile
  val membre : 'a -> 'a pile -> bool
end;;
```

```
module type PILE =
  sig
    type 'a pile
    exception Pile_vide
    val vide : 'a pile
    val empiler : 'a -> 'a pile -> 'a pile
    val depiler : 'a pile -> 'a pile
    val membre : 'a -> 'a pile -> bool
  end
```

La déclaration du type *'a pile* ne donne pas de détail sur la structure de ce type (type de données composé avec constructeurs ou type de données composé sans constructeurs, etc.). Celle des fonctions se contente de spécifier le type des arguments ainsi que celui du résultat, donc d'en spécifier la signature. Les signatures servent principalement à décrire des spécifications que certains modules doivent respecter. Pour définir un module conforme à cette signature, il faut que les types des valeurs définies dans ce module soient compatibles à ceux définis dans la signature. Par exemple, à partir du module Pile défini précédemment, il est possible de définir un nouveau module MaPile conforme à la signature PILE :

```
# module MaPile : PILE = Pile;;
module MaPile : PILE

# depiler (empiler 4 vide);;
- : int MaPile.pile = <abstr>

# sommet (empiler 4 vide);;
  sommet (empiler 4 vide);;
  -----
Error: Unbound value sommet
```

Le module `MaPile` définit le type `'a pile` avec deux constructeurs de type, `Vide` et `Empiler`. La définition des fonctions `empiler`, `depiler` et `membre`, ainsi que celle de la constante `vide`, sont conformes, au niveau des types, à ceux précisés dans la signature. Par contre, la fonction `sommet` n'est pas disponible à partir du module `MaPile`. En effet, bien qu'elle soit définie dans le module `Pile`, le fait qu'elle ne soit pas déclarée dans la signature `PILE` implique qu'elle n'est pas exportable (accessible de l'extérieur du module). Elle demeure néanmoins utilisable à l'intérieur du module défini, c'est-à-dire disponible aux autres fonctions du module. Il est fréquent d'écrire un module ou un paquetage concernant une structure de données (piles, files, listes, etc.) dans lequel sont définies des fonctions locales fl_i utilisées par d'autres fonctions f_j . Seulement celles qui sont pertinentes, par exemple les fonctions f_j , sont exportées par l'intermédiaire de leur déclaration dans une signature ou dans une interface (Java, C#, etc.).

Par ailleurs, à une signature peuvent correspondre plusieurs modules. En effet, et par analogie, à une spécification peuvent correspondre plusieurs implémentations. Voici, par exemple, la définition d'un module `Pile_List` qui est implémenté à l'aide du type de données `'a list` :

```
# module Pile_List = struct
  type 'a pile = 'a list
  exception Pile_vide

  let vide = []
  let empiler e p = e::p

  let depiler p = match p with
    | [] -> raise Pile_vide
    | e::p' -> p'

  let rec membre e p = List.exists (fun x -> x = e) p
end;;
```

```
module Pile_List :
sig
  type 'a pile = 'a list
  exception Pile_vide
  val vide : 'a list
  val empiler : 'a -> 'a list -> 'a list
  val depiler : 'a list -> 'a list
  val membre : 'a -> 'a list -> bool
end
```

À l'instar du module `Pile`, il est possible d'exploiter les structures de données et les fonctions définies dans cette structure :


```
# open Pile_List;;

# let pile_list_int = empiler 1 (empiler 4 vide);;
  val pile_list_int : int list = [1; 4]

# membre 4 pile_list_int;;
- : bool = true

# let pile_list_int' = [1;2;3;4];;
  val pile_list_int' : int list = [1; 2; 3; 4]

# membre 3 pile_list_int';;
- : bool = true

# List.hd pile_list_int';;
- : int = 1
```

```
# module MaPileList : PILE = Pile_List;;
  module MaPileList : PILE

# open MaPileList;;

# let p = [1;2;3;4];;
  val p : int list = [1; 2; 3; 4]

# membre 3 p;;
-
  Error: This expression has type int list but is here used with type
         int MaPileList.pile

# let p' = empiler 1 (empiler 2 (empiler 3 (empiler 4 vide)));;
  val p' : int MaPileList.pile = <abstr>

# membre 3 p';;
- : bool = true
```

2.8.3 Foncteurs

Un foncteur est une fonction particulière associant un module à un autre module : c'est une méthode qui peut servir à définir des modules génériques. Prenons l'exemple d'une signature ELEM_ORDRE composée d'un type abstrait *t* et d'une fonction ordre qui compare deux valeurs de type *t* :

```
# module type ELEM_ORDRE = sig
  type t
  val ordre : t -> t -> bool
end;;
module type ELEM_ORDRE = sig type t val ordre : t -> t -> bool end
```

Une fois cette signature définie, il est possible, par exemple, de définir un module Entier conforme à cette signature :

```
# module Entier : ELEM_ORDRE with type t = int = struct
  type t = int
  let ordre = (=)
end;;
module Entier : sig type t = int val ordre : t -> t -> bool end
```

L'ouverture de ce module donne accès au type *t* et à la fonction ordre :

```
# open Entier;;

# ordre 6 7;;
- : bool = false

# ordre 3 3;;
- : bool = true
```

Supposons que nous voulions définir un module `Paire_Entier` de type `ELEM_ORDRE` dans lequel le type `t` désigne une paire d'entiers et la fonction `ordre` compare deux paires de valeurs entières. À cette fin, il est possible de définir un foncteur (ou une fonction) `PAIRE_ORDRE` qui associe à un module de type `ELEM_ORDRE`, un autre module de même type :

```
# module PAIRE_ORDRE (E : ELEM_ORDRE) : ELEM_ORDRE with type t = E.t * E.t =
  struct
    type t = E.t * E.t
    let ordre (x1,y1) (x2,y2) = (E.ordre x1 x2) && (E.ordre y1 y2)
  end;;
module PAIRE_ORDRE :
  functor (E : ELEM_ORDRE) ->
    sig type t = E.t * E.t val ordre : t -> t -> bool end
```

En appliquant ce foncteur à un module de type `ELEM_ORDRE`, tel que `Entier`, nous obtenons un module défini pour des paires de valeurs du type défini dans le module en argument, soit `int` :

```
# module Paire_Entiers = PAIRE_ORDRE(Entier);;
module Paire_Entiers :
  sig type t = Entier.t * Entier.t val ordre : t -> t -> bool end

# open Paire_Entiers;;

# ordre (3,4) (3,4);;
- : bool = true

# ordre (1,2) (2,1);;
- : bool = false
```

En considérant des modules *Booleen*, *Reel* et *Chaine* de type `ELEM_ORDRE` et définis de la même manière que `Entier`, ce foncteur permet de générer automatiquement des modules `Paire_Bool`, `Paire_Reel` et `Paire_Chaine` comme suit :

```
# module Paire_Booleens = PAIRE_ORDRE(Booleen);;
module Paire_Booleens :
  sig type t = Booleen.t * Booleen.t val ordre : t -> t -> bool end

# module Paire_Reels = PAIRE_ORDRE(Reel);;
module Paire_Reels :
  sig type t = Reel.t * Reel.t val ordre : t -> t -> bool end

# module Paire_Chaines = PAIRE_ORDRE(Chaine);;
module Paire_Chaines :
  sig type t = Chaine.t * Chaine.t val ordre : t -> t -> bool end
```

Sans la notion de foncteurs, il aurait fallu définir individuellement les modules `Paire_Bool`, `Paire_Reel` et `Paire_Chaine` en spécifiant pour chacune d'entre eux la valeur du type `t` et le corps de la fonction `ordre`.

Notons de plus, qu'à partir de ce foncteur, il est possible d'en définir d'autres. L'exemple suivant définit un foncteur `PP_O` qui associe à un module de type `ELEM_ORDRE` un autre module de même type en appliquant deux fois le foncteur `PAIRE_ORDRE` à l'argument :

```
# module PP_O (E : ELEM_ORDRE) = PAIRE_ORDRE(PAIRE_ORDRE(E));;
module PP_O :
  functor (E : ELEM_ORDRE) ->
    sig
      type t = PAIRE_ORDRE(E).t * PAIRE_ORDRE(E).t
      val ordre : t -> t -> bool
    end
```

À partir d'un module Entier, il est possible de générer un module PP_Entier manipulant des paires de paires d'entiers :

```
# module PP_Entiers = PP_O(Entier);;
module PP_Entiers :
  sig
    type t = PAIRE_ORDRE(Entier).t * PAIRE_ORDRE(Entier).t
    val ordre : t -> t -> bool
  end

# open PP_Entiers;;

# ordre ((4,5),(6,9)) ((4,5),(6,9));;
- : bool = true

# ordre ((1,2),(3,4)) ((3,4),(1,2));;
- : bool = false
```

Notons que ce module aurait pu être généré en appliquant le foncteur PAIRE_ORDRE au module Paire_Entier :

```
# module PP_Entiers' = PAIRE_ORDRE(Paire_Entiers);;
module PP_Entiers' :
  sig
    type t = Paire_Entiers.t * Paire_Entiers.t
    val ordre : t -> t -> bool
  end

# open PP_Entiers';;

# ordre ((4,5),(6,9)) ((4,5),(6,9));;
- : bool = true
```

Nous terminons cette section par un exemple plus classique et plus complexe : module de pile générique. Grâce à ce module, il est possible de définir des piles de valeurs pouvant être comparées. Les définitions de la signature PILE et du module Pile ont été présentées dans les sections précédentes.

Supposons que l'on veuille créer une pile d'enregistrements, représentant des étudiants, composés des champs matricule, nom et moyenne. Le problème dans ce cas réside dans le fait que la pile ne doit contenir qu'un seul enregistrement par étudiant. Ainsi, la fonction membre ne devrait comparer que le champ matricule de deux enregistrements¹⁰. Avec la définition actuelle du module Pile, il n'est pas possible de comparer deux enregistrements en ne tenant compte que d'un seul champ. En effet, par définition, deux enregistrement e_1 et e_2 sont égaux si leurs champs respectifs le sont aussi :

¹⁰. En terme de bases de données, ce champ représente une clé primaire.

```
# type etudiant = {mat: int; nom: string; mutable moy: float};;
type etudiant = { mat : int; nom : string; mutable moy : float; }

# let e1 = {mat=123; nom="Michel"; moy=80.};;
val e1 : etudiant = {mat = 123; nom = "Michel"; moy = 80.}

# let e2 = {mat=124; nom="Gaston"; moy=95.};;
val e2 : etudiant = {mat = 124; nom = "Gaston"; moy = 95.}

# let pe = empiler e1 (empiler e2 vide);;
val pe : etudiant MaPileList.pile = <abstr>

# membre e1 pe;;
- : bool = true

# membre {e1 with moy = 70.} pe;;
- : bool = false
```

L'idéal serait d'avoir un module Etudiant défini comme suit (la fonction «ordre» compare correctement deux enregistrements de type *etudiant*) :

```
# module Etudiant : ELEM_ORDRE with type t = etudiant =
  struct
    type t = etudiant
    let ordre {mat=m1} {mat=m2} = (m1 = m2)
  end;;
module Etudiant : sig type t = etudiant val ordre : t -> t -> bool end
```

puis de définir une pile de valeurs de type ELEM_ORDRE.

La solution à ce problème est d'utiliser un foncteur qui, étant donné un module de type ELEM_ORDRE en argument, génère une pile de valeurs de type correspondant au type *t* défini dans le module en argument. À cette fin, il faut redéfinir la signature PILE comme suit :

```
# module type PILE_ORDRE = sig
  module E : ELEM_ORDRE
  type pile
  exception Pile_vide
  val vide : pile
  val empiler : E.t -> pile -> pile
  val depiler : pile -> pile
  val sommet : pile -> E.t
  val membre : E.t -> pile -> bool
end;;
module type PILE_ORDRE =
  sig
    module E : ELEM_ORDRE
    type pile
    exception Pile_vide
    val vide : pile
    val empiler : E.t -> pile -> pile
    val depiler : pile -> pile
    val sommet : pile -> E.t
    val membre : E.t -> pile -> bool
  end
```

Le foncteur S_PILE_ORDRE est défini comme suit :

```

# module S_PILE_ORDRE(E' : ELEM_ORDRE) : PILE_ORDRE with module E = E' =
  struct
    module E = E'
    type pile = Vide | Empiler of E.t * pile
    exception Pile_vide

    let vide = Vide
    let empiler e p = Empiler(e,p)

    let depiler p =
      match p with
      | Vide -> raise Pile_vide
      | Empiler(_,p') -> p'

    let sommet p =
      match p with
      | Vide -> raise Pile_vide
      | Empiler(e,_) -> e
    let rec membre e p =
      match p with
      | Vide -> false
      | Empiler(e',p') -> (E.ordre e e') || (membre e p')
    end;;

module S_PILE_ORDRE :
  functor (E' : ELEM_ORDRE) ->
  sig
    module E : sig type t = E'.t val ordre : t -> t -> bool end
    type pile
    exception Pile_vide
    val vide : pile
    val empiler : E.t -> pile -> pile
    val depiler : pile -> pile
    val sommet : pile -> E.t
    val membre : E.t -> pile -> bool
  end

```

Il est alors possible de définir un module Pile_Etudiant comme suit :

```

# module Pile_Etudiants = S_PILE_ORDRE(Etudiant);;
module Pile_Etudiants :
  sig
    module E : sig type t = Etudiant.t val ordre : t -> t -> bool end
    type pile = S_PILE_ORDRE(Etudiant).pile
    exception Pile_vide
    val vide : pile
    val empiler : E.t -> pile -> pile
    val depiler : pile -> pile
    val sommet : pile -> E.t
    val membre : E.t -> pile -> bool
  end

```

puis l'ouvrir et l'exploiter :

```

# open Pile_Etudiants;;

# let pe = empiler e1 (empiler e2 vide);;
  val pe : Pile_Etudiants.pile = <abstr>

# membre e1 pe;;
- : bool = true

# membre {e1 with moy = 70.} pe;;
- : bool = true

```

Notons enfin que ce foncteur permet de générer une multitude de modules comme l'illustrent les exemples suivants :

```
# module Pile_Entiers = S_PILE_ORDRE(Entier);;
module Pile_Entiers :
  sig
    module E : sig type t = Entier.t val ordre : t -> t -> bool end
    type pile = S_PILE_ORDRE(Entier).pile
    exception Pile_vide
    val vide : pile
    ...
  end

# module Pile_PP_Entiers = S_PILE_ORDRE(PP_Entiers);;
module Pile_PP_Entiers :
  sig
    module E : sig type t = PP_Entiers.t val ordre : t -> t -> bool end
    type pile = S_PILE_ORDRE(PP_Entiers).pile
    exception Pile_vide
    val vide : pile
    ...
  end

# module Pile_Paires_Entiers = S_PILE_ORDRE(PAIRE_ORDRE(Entier));;
module Pile_Paires_Entiers :
  sig
    module E :
      sig type t = PAIRE_ORDRE(Entier).t val ordre : t -> t -> bool end
    type pile = S_PILE_ORDRE(PAIRE_ORDRE(Entier)).pile
    exception Pile_vide
    val vide : pile
    ...
  end
```

Le dernier exemple illustre l'application de deux foncteurs à un module donné. Le résultat de ces applications est un module représentant une pile de paires d'entiers.

2.9 Interaction avec l'interpréteur

Plusieurs commandes sont disponibles pour, par exemple, charger un fichier, changer de répertoire, etc. Chaque commande est précédée par le symbole "#". Par exemple, la commande «**#quit**» permet de quitter l'interpréteur :

```
# #quit;;
```

D'autres fonctions sont disponibles :

- **#use** «*nom_fichier*» : Charge le fichier *nom_fichier*.
 - **#load** «*nom_fichier*» : Charge une bibliothèque de code ou un fichier compilé (*bytecode*).
- Par exemple :

```
# #load "graphics.cma";;
```

permet de charger le fichier (*library*) "graphics.cma".

- **#directory** «*+nom_rep*» : Permet d'ajouter un répertoire, *nom_rep*, à la liste des répertoires courants. Cette liste est utilisée pour charger un fichier donné. Par exemple, pour charger la bibliothèque «Thread» permettant de créer des processus légers, il faut procéder comme suit :

```
# #directory "+threads";;
# #load "unix.cma";;
# #load "threads.cma";;
```

En effet, le fichier "threads.cma" se trouve dans le répertoire "threads" du répertoire "lib" de la distribution standard d'OCaml. De plus, pour être utilisé, on note que ce fichier nécessite que le fichier "unix.cma" soit préalablement chargé.

Une fois le fichier chargé, on peut bien sûr ouvrir un module qui y est défini et exploiter ses fonctions. Par exemple, pour créer un processus léger, on a accès à la fonction «create» :

```
# open Thread;;
# create;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

- **#trace** «*nom_fonction*» : Permet de «tracer» la pile d'appels de la fonction passée en argument. Supposons, par exemple, qu'une fonction «fact» soit définie :

```
# fact 4;;
- : int = 24
```

En traçant cette fonction, on peut alors suivre les différents appels sous-jacents à l'appel initial de la fonction :

```
# #trace fact;;
# fact 4;;
fact <— 4
fact <— 3
fact <— 2
fact <— 1
fact —> 1
fact —> 2
fact —> 6
fact —> 24
- : int = 24
```

- **#untrace** «*nom_fonction*» : Permet de ne plus «tracer» la fonction *nom_fonction* :

```
# #untrace fact;;
# fact 4;;
- : int = 24
```

- **#show** «*nom_module*» : Permet d'afficher la liste des valeurs définies dans un module, sans avoir à ouvrir ce module; dans l'exemple qui suit, on demande à l'interpréteur d'afficher la liste des valeurs définies dans le module List :

```
# #show List;;
module List :
  sig
    type 'a t = 'a list = [] | (::) of 'a * 'a list
    val length : 'a list -> int
    val compare_lengths : 'a list -> 'b list -> int
    val compare_length_with : 'a list -> int -> int
    val cons : 'a -> 'a list -> 'a list
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val nth : 'a list -> int -> 'a
    val nth_opt : 'a list -> int -> 'a option
    val rev : 'a list -> 'a list
    val init : int -> (int -> 'a) -> 'a list
    val append : 'a list -> 'a list -> 'a list
    val rev_append : 'a list -> 'a list -> 'a list
    val concat : 'a list list -> 'a list
    val flatten : 'a list list -> 'a list
```

```

val equal : ('a -> 'a -> bool) -> 'a list -> 'a list -> bool
val compare : ('a -> 'a -> int) -> 'a list -> 'a list -> int
val iter : ('a -> unit) -> 'a list -> unit
val iteri : (int -> 'a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val filter_map : ('a -> 'b option) -> 'a list -> 'b list
val concat_map : ('a -> 'b list) -> 'a list -> 'b list
val fold_left_map : ('a -> 'b -> 'a * 'c) -> 'a -> 'b list -> 'a * 'c list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val mem : 'a -> 'a list -> bool
val memq : 'a -> 'a list -> bool
val find : ('a -> bool) -> 'a list -> 'a
val find_opt : ('a -> bool) -> 'a list -> 'a option
val find_map : ('a -> 'b option) -> 'a list -> 'b option
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
val filteri : (int -> 'a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
val partition_map :
  ('a -> ('b, 'c) Either.t) -> 'a list -> 'b list * 'c list
val assoc : 'a -> ('a * 'b) list -> 'b
val assoc_opt : 'a -> ('a * 'b) list -> 'b option
val assq : 'a -> ('a * 'b) list -> 'b
val assq_opt : 'a -> ('a * 'b) list -> 'b option
val mem_assoc : 'a -> ('a * 'b) list -> bool
val mem_assq : 'a -> ('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val sort_uniq : ('a -> 'a -> int) -> 'a list -> 'a list
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
val to_seq : 'a list -> 'a Seq.t
val of_seq : 'a Seq.t -> 'a list
end

```

Notons que le code source (l'implantation) de tous les modules de la librairie standard d'OCaml est disponible à cette adresse GitHub <https://github.com/ocaml/ocaml/tree/trunk/stdlib>; plus précisément, pour le source des fonctions du module List : <https://github.com/ocaml/ocaml/blob/trunk/stdlib/list.ml>.

- **#cd** «*nom_répertoire*» : Permet de changer de répertoire courant.
- **#print_depth** *n* : Permet d'afficher plus en profondeur le contenu des structures de données.
- **#print_length** *n* : Permet d'afficher plus de valeurs contenues dans des structures de données.

D'autres commandes sont disponibles : voir le manuel du langage OCaml, plus précisément à cette adresse <https://v2.ocaml.org/releases/5.0/htmlman/toplevel.html#%3Aoplevel-directives>.

2.10 Conclusion

Ce chapitre a présenté le langage fonctionnel OCaml dont les principales caractéristiques sont :

- un système d'inférences de types garantissant que les programmes n'auront pas d'erreurs de typage à l'exécution ;
- un collecte-miettes permettant d'augmenter la modularité et d'éviter certaines erreurs ;
- un système de modules qui assure la construction de large projet et la réutilisation de code ;
- l'utilisation de fonctions d'ordre supérieur et de polymorphisme, permettant d'augmenter la généricité du code et donc sa réutilisation ;
- un mécanisme de définition de structures de données et de filtrage permettant de définir simplement, et de manière élégante, plusieurs types de structures de données et plusieurs algorithmes.

Exercices

2.1 Écrire une expression OCaml qui calcule la somme des entiers compris entre 1 et 1000.

2.2 Définir des fonctions `digit`, `lower_case`, `upper_case`, et `letter` qui testent, respectivement, si un caractère est un chiffre, une lettre miniscule, une lettre majuscule ou une lettre quelconque.

2.3 Définir une fonction `max3` qui retourne la plus grande valeur d'un triplet de valeurs.

2.4 Définir une fonction `to_bin` qui prend comme argument un entier positif et retourne comme résultat la représentation binaire (comme une chaîne de caractères) de cet entier.

2.5 Déterminer les types des fonctions OCaml suivantes :

- `fun x -> x * x`
- `fun x -> floor x`
- `fun (x,y) => if (x < y) then x else y`

2.6 Déterminer le type de la fonction suivante :

```
let if2 x y z = if x then y else z
```

Quelle est la différence entre cette fonction et le constructeur syntaxique `if`.

2.7 Déterminer les types des expressions suivantes :

- `fun (x,y) -> (y,x)`
- `fun (x,(y,z)) -> ((x,y),z)`
- `["Hello"]`
- `tl [true]`
- `[[[]]]`
- `tl [3, []]`

2.8 Que fait la fonction suivante :

```
let rec fl =
  match l with
  | [] -> []
  | h :: r -> append (f r) [h]
```

Rappelons que la fonction `append` est définie dans ce chapitre.

2.9 À l'aide des fonctions prédéfinies `hd` et `tl`, définir une fonction `second` qui retourne le deuxième élément d'une liste donnée. On suppose que cette liste comprend au moins deux éléments.

2.10 Définir une fonction `del_last` qui supprime le dernier élément d'une liste non vide. Il est possible d'utiliser la fonction prédéfinie `rev`.

2.11 Définir, par filtrage, une fonction récursive `len` qui calcule la taille d'une liste (nombre d'éléments).

2.12 En s'inspirant de la définition de la fonction `len`, définir deux fonctions `sum_list` et `prod_list` qui calculent, respectivement, la somme et le produit des éléments d'une liste donnée.

2.13 Définir une forme récursive de la fonction prédéfinie `rev`.

2.14 Définir une fonction récursive `nbr` qui calcule le nombre d'occurrence d'un élément donné dans une liste.

2.15 Soient deux listes `l1` et `l2` initialement triées par ordre croissant. Définir une fonction `merge` qui calcule à partir de ces deux listes une liste triée par ordre croissant comprenant les éléments des deux listes.

2.16 Définir la fonction `second` en utilisant des filtres.

2.17 Supposons qu'on ait déclaré la variable `x` comme suit :

— `let x = (5,false) ; ;`

Quelles sont les évaluations des expressions suivantes :

— `let (x1,x2) = x`

— `let (5,x2) = x`

— `let (6,x2) = x`

2.18 Supposons que la variable `l` soit liée à la liste `[1;2;3]`. Quelles sont les évaluations des expressions suivantes :

— `let [x;y;z] = l`

— `let (x :: y) = l`

— `let (_ :: y) = l`

— `let (x :: _) = l`

2.19 Supposons que la variable `e` soit liée à l'enregistrement `{nom="Temblay"; age=40}`. Quelles sont les évaluations des expressions suivantes :

— `let {nom=n;age=a} = e`

— `let {nom=n} = e`

— `let {age=a} = e`

2.20 Déterminer les types des fonctions suivantes :

— `let f x = x 1 1`

— `let g x y z = x (y ^ z)`

— `let h x y z = x y z`

— `let t x y = fun z -> x :: y`

2.21 En utilisant les fonctions `sum_list` et `map`, définir une fonction qui calcule le nombre d'éléments d'une liste.

2.22 Définir une fonction `swap_list` qui permute les éléments d'une liste de paires.

2.23 Définir une fonction récursive `append` qui permet de concaténer deux listes passées en argument.

2.24 Soit la signature suivante :

```
module type ORDRE
sig
  type t
  val ordre : t * t -> bool
end
```

Créer une structure qui ordonne des entiers. Cette structure doit être en accord avec la signature `ORDRE`.

Corrigés

2-1 :

```
# 1000 * (1000 + 1) / 2;;
```

2-2 :

```
# let digit c = c >= '0' && c <= '9';;
# let lower_case c = c >= 'a' && c <= 'z';;
# let upper_case c = c >= 'A' && c <= 'Z';;
# let letter c = (lower_case c) || (upper_case c);;
```

2-3 :

```
# let max3 (x,y,z) = max x (max y z);;
```

2-4 :

```
# let rec to_bin n =
  match n with
  | 0 -> "0"
  | 1 -> "1"
  | n -> (to_bin (n/2)) ^ (if n mod 2 = 0 then "0" else "1");;
```

2-5 :

```
# fun x -> x * x;;
- : int -> int = <fun>
# fun x -> floor x;;
- : float -> float = <fun>
# fun (x,y) -> if x < y then x else y;;
- : 'a * 'a -> 'a = <fun>
```

2-6 : Le type de cette fonction est :

```
# let if2 x y z = if x then y else z;;
val if2 : bool -> 'a -> 'a -> 'a = <fun>
```

Pour saisir la différence entre cette fonction et le constructeur syntaxique **if**, prenons les deux exemples suivants :

- `if2 (1=0) (1 / 0) 3`
- `if (1=0) then (1 / 0) else 3`

L'évaluation de la première expression entraîne une erreur d'exécution car l'évaluation du deuxième argument provoque une erreur. Par contre, la deuxième expression s'évalue en la valeur entière 3. En effet, seule la condition (1=0) et l'expression 3 sont évaluées dans ce cas.

2-7 :

```
# fun (x,y) -> (y,x);;
- : 'a * 'b -> 'b * 'a = <fun>
# fun (x,(y,z)) -> ((x,y),z);;
- : 'a * ('b * 'c) -> ('a * 'b) * 'c = <fun>
# [|"Hello"|];;
- : string list list = [|"Hello"|]
# tl [true];;
- : bool list = []
# [[]];;
- : 'a list list = [[]]
# tl [3,[[]]];
- : (int * 'a list) list = []
```

2-8 : Cette fonction se comporte comme la fonction prédéfinie `rev`. Elle inverse l'ordre des éléments d'une liste.

2-9 :

```
# let second l = hd(tl l);;
```

2-10 :

```
# let del_last l = rev(tl(rev l));;
```

2-11 :

```
# let rec len l =
  match l with
  | [] -> 0
  | _::r -> 1 + (len r);;
```

2-12 :

```
# let rec sum_list l =
  match l with
  | [] -> 0
  | x::r -> x + (sum_list r);;

# let rec prod_list l =
  match l with
  | [] -> 1
  | x::r -> x * (prod_list r);;
```

2-13 :

```
# let rec rev' l =
  match l with
  | [] -> []
  | x::r -> (rev' r) @ [x];;
```

2-14 :

```
# let rec nbr e l =
  match l with
  | [] -> 0
  | x::r -> (if e = x then 1 else 0) + (nbr e r);;
```

2-15 :

```
# let rec merge l1 l2 =
  match l1, l2 with
  | [], [] -> []
  | l1, [] -> l1
  | [], l2 -> l2
  | x1::r1, x2::r2 ->
    if x1 <= x2 then x1::(merge r1 l2) else x2::(merge l1 r2);;
```

2-16 :

```
# let second l =
  match l with
  | _::x::_ -> x
  | _ -> failwith "Taille_de_liste_inférieure_à_2";;
```

2-17 :

```
# let (x1,x2) = x;;
val x1 : int = 5
val x2 : bool = false

# let (5,x2) = x;;
(* Affichage d'un warning *)
val x2 : bool = false

# let (6,x2) = x;;
(* Affichage d'une erreur *)
Exception: Match_failure ("", 37, -29).
```

2-18 :

```
# let [x;y;z] = l;;
val x : int = 1
val y : int = 2
val z : int = 3

# let x::y = l;;
val x : int = 1
val y : int list = [2; 3]

# let _::y = l;;
val y : int list = [2; 3]

# let x::_ = l;;
val x : int = 1
```

2-19 :

```
# let {nom=n; age=a} = e;;
val n : string = "Tremblay"
val a : int = 40

# let {nom=n} = e;;
val n : string = "Tremblay"

# let {age=a} = e;;
val a : int = 40
```

2-20 :

```
# let f x = x 1 1;;
val f : (int -> int -> 'a) -> 'a = <fun>

# let g x y z = x (y ^ z);;
val g : (string -> 'a) -> string -> string -> 'a = <fun>

# let h x y z = x y z;;
val h : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let t x y = fun z -> x::y;;
val t : 'a -> 'a list -> 'b -> 'a list = <fun>
```

2-21 :

```
# let nbr2 l = sum_list (map (fun x -> 1) l);;
```

2-22 :

```
# let swap_list l = map (fun (x,y) -> (y,x)) l;;
```

2-23 :

```
# let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | h::r -> h :: (append r l2);;
```

2-24 :

```
# module Int_Ordre : ORDRE = struct
  type t = int
  let ordre = ( < )
end;;
```


Chapitre 3

ML : Notions avancées

Ce chapitre est consacré à la présentation d'autres aspects du langage OCaml ainsi qu'à des techniques de programmation issues du paradigme fonctionnel. Plus précisément, le chapitre est organisé comme suit :

1. Aspects du paradigme fonctionnel : cette section est consacrée à des aspects et techniques issus du paradigme fonctionnel.
2. Aspect déclaratif du paradigme fonctionnel : outre l'aspect déclaratif, nous donnons de l'information concernant les paquetages *LINQ* et *PLINQ* de Microsoft, ainsi que le langage *C#* qui les implante.

3.1 Aspects du paradigme fonctionnel

Cette section décrit un ensemble d'aspects et de techniques liés à la programmation fonctionnelle. Avant de les présenter, nous présentons, dans ce qui suit, quelques fonctions qui seront utiles pour la description des différents exemples présentés dans cette section et le reste du chapitre :

— Constructeurs de listes : On en présente deux :

1. Constructeur de listes contenant les éléments d'un interval $[n,m]$:

```
# let ( -- ) n m =  
  let rec aux k l = match k with  
    | k' when k' < n -> []  
    | k' when k' = n -> n :: l  
    | _ -> aux (k+1) (k :: l)  
  in  
    aux m [];;  
val ( -- ) : int -> int -> int list = <fun>  
  
# 5 -- 15;;  
- : int list = [5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15]  
  
# 1 -- 1;;  
- : int list = [1]  
  
# 1 -- 0;;  
- : int list = []
```

Remarquez la définition locale de la fonction auxiliaire «aux».

Par ailleurs, voici un exemple utilisant cet opérateur ainsi que l'opérateur «pipe» :

```
# (1 — 10) |> filter (fun x -> x mod 2 = 0)
|> map string_of_int
|> iter print_string;;
246810- : unit = ()
```

2. Constructeur de listes comprenant m fois une valeur n :

```
# let ( ^ ) n m =
  let rec aux k l = match k with
    | 0 -> l
    | _ -> aux (k-1) (n::l)
  in
    aux m [];
val ( ^ ) : 'a -> int -> 'a list = <fun>

# 2 ^ 10;;
- : int list = [2; 2; 2; 2; 2; 2; 2; 2; 2; 2]

# 2 ^ 1;;
- : int list = [2]

# 2 ^ 0;;
- : int list = []
```

- Fonction «sleep» : Cette fonction est définie dans le module Unix¹ :

```
# #load "unix.cma";;
# open Unix;;

# sleep;;
- : int -> unit = <fun>

# sleep 5;; (* hibernation pendant 5s avant d'afficher le "résultat" *)
- : unit = ()
```

La fonction permet de simuler l'hibernation du thread courant (l'interpréteur dans ce cas) pendant un laps de temps passé en paramètre de la fonction et spécifié en secondes.

- Fonction «timeRun» : Cette fonction utilise une fonction prédéfinie, «gettimeofday», définie aussi dans le module Unix :

```
# let timeRun f x =
  let time1 = gettimeofday() in
  let r = f x in
  let time2 = gettimeofday() in
  (r, time2 -. time1);;
val timeRun : ('a -> 'b) -> 'a -> 'b * float = <fun>
```

Étant données une fonction et une valeur passées en argument, la fonction «timeRun» retourne le résultat de l'application de la fonction passée en paramètre à la valeur, ainsi que le temps qu'a pris cette application. Par exemple² :

1. Notez qu'il est nécessaire de charger le module compilé "unix.cma".

2. La fonction «d_fact» aurait pu être définie sur une ligne :

```
let rec d_fact n = if n <= 1 then 1 else (sleep 1; n * d_fact(n-1));;
```

```
# let rec d_fact n =
  if n <= 1 then 1
  else
    begin
      sleep 1;
      n * d_fact(n-1)
    end;;
val d_fact : int -> int = <fun>

# let d_add (x,y) = sleep 1; x + y;;
val d_add : int * int -> int = <fun>

# timeRun d_fact 6;;
- : int * float = (720, 5.)

# timeRun d_add (7,8);;
- : int * float = (15, 1.)
```

Remarquez le type polymorphe de la fonction «timeRun». Elle peut être appliquée à n'importe quelle autre fonction et valeur !

```
# timeRun (map d_fact) [1;2;3;4;5];;
- : int list * float = ([1; 2; 6; 24; 120], 10.)
```

Apprécions l'utilité de l'application partielle des fonctions (`map d_fact`) dans le contexte des fonctions curryfiées (`map`).

3.1.1 Représentation des grammaires

Les langages fonctionnels sont très efficaces lorsqu'il s'agit de manipuler des structures de données représentant une grammaire donnée.

Par exemple, voici la grammaire BNF d'une expression que l'on veut représenter :

$$exp ::= n \mid exp + exp \mid - e$$

Voici la représentation en OCaml avec quelques exemples de manipulation d'expressions de ce type :

```
# type exp = Cst of int | Plus of exp * exp | Minus of exp;;
type exp = Cst of int | Plus of exp * exp | Minus of exp

# Plus (Cst 7, Minus (Cst 7));;
- : exp = Plus (Cst 7, Minus (Cst 7))

# let x1 = Cst 5 and x2 = Cst 7;;
val x1 : exp = Cst 5
val x2 : exp = Cst 7

# let x3 = Minus x2 and x4 = Plus (x1,x2);;
val x3 : exp = Minus (Cst 7)
val x4 : exp = Plus (Cst 5, Cst 7)

# let x5 = Plus (x4,x4);;
val x5 : exp = Plus (Plus (Cst 5, Cst 7), Plus (Cst 5, Cst 7))
```

Il est aussi assez facile de définir des fonctions manipulant ce type de données. Par exemple, voici la définition d'une fonction «eval» qui permet d'évaluer une expression de type *exp* :

```
# let rec eval e = match e with
| Cst i -> i
| Plus(e1, e2) -> (eval e1) + (eval e2)
| Minus e -> -(eval e);;
val eval : exp -> int = <fun>

# eval x4;;
- : int = 12

# eval x5;;
- : int = 24
```

De même, il est possible de transformer une expression de type `exp` en une chaîne de caractères :

```
# let rec expToStr e = match e with
| Cst i -> string_of_int i
| Plus(e1, e2) -> (expToStr e1) ^ "+" ^ (expToStr e2)
| Minus e -> "-" ^ (expToStr e);;
val expToStr : exp -> string = <fun>
```

```
# expToStr x4;;
- : string = "5 + 7"

# expToStr x5;;
- : string = "5 + 7 + 5 + 7"
```

Finalement, on peut combiner les deux dernières fonctions pour afficher à l'écran une expression, en format chaîne de caractères, ainsi que son évaluation :

```
# let print_eval e =
  let r = eval e in
  (expToStr e) ^ "=" ^ (string_of_int r);;
val print_eval : exp -> string = <fun>

# print_eval x4;;
- : string = "5 + 7 = 12"

# print_eval x5;;
- : string = "5 + 7 + 5 + 7 = 24"
```

3.1.2 Fermetures (*closures*)

Mathématiquement parlant, une fonction, appliquée à une valeur, retourne toujours le même résultat :

```
# let f x = x + 1;;
val f : int -> int = <fun>

# f 4;;
- : int = 5

# f 4;;
- : int = 5
```

Dans un contexte de programmation impérative, à l'instar du langage C, le résultat d'une fonction peut différer, pour une même valeur en argument, selon l'état de la mémoire au moment de l'appel de la fonction; dans l'exemple qui suit, une même expression «`g(4)`» (application de la fonction «`g`» à la valeur 4) retourne deux valeurs différentes :

```
# let c = ref 5;;
val c : int ref = {contents = 5}

# let g x = x + !c;;
val g : int -> int = <fun>

# g(4);;
- : int = 9

# c := 8;;
- : unit = ()

# g(4);;
- : int = 12
```

Par contre, dans un style purement fonctionnel, le résultat de la fonction ne dépend que de son paramètre et de l'état de l'environnement au moment de sa définition :

```
# let c = 5;;
val c : int = 5

# let h x = x + c;;
val h : int -> int = <fun>

# h 4;;
- : int = 9

# let c = 8;;
val c : int = 8

# h 4;;
- : int = 9
```

On comprend intuitivement que la fonction «h» est encapsulée avec l'environnement dans lequel elle est définie (dans l'exemple précédent, l'environnement comprend l'association entre l'identificateur «c» et la valeur 5) : c'est ce qu'on appelle une fermeture. Le fait que l'identificateur «c» soit redéfini dans l'environnement global n'affecte aucunement le contenu de la fermeture. Ainsi, lorsque la fonction «h» est de nouveau appliquée à une valeur, c'est en considérant son environnement local, encapsulé dans la fermeture qui lui est associée, qu'elle procèdera à son évaluation.

Ainsi, une *fermeture* est un triplet $\langle x, e, M \rangle$ représentant une fonction, ayant comme argument x et comme corps e , et étant définie dans un environnement M ; lorsqu'une fermeture est invoquée, son corps e est évaluée en considérant le paramètre x de la fonction et l'environnement M qui l'accompagne (dans le triplet).

Dans l'exemple qui suit, l'application «ajoute 2» retourne une fermeture $\langle x, n + x, [n \mapsto 2] \rangle$, c'est-à-dire une fonction qui prend comme argument «x», qui retourne «x + n» et qui est définie dans un contexte où «n» vaut 2 :

```
# let ajoute n = let f x = n + x in f;;
val ajoute : int -> int -> int = <fun>

# let ajoute_2 = ajoute 2;;
val ajoute_2 : int -> int = <fun>

# ajoute_2 4;;
- : int = 6
```

3.1.3 Itérateurs

Étant donné un objet représentant une structure de données abstraite (détails d'implantation cachés), objet que l'on qualifie de producteur, et étant donné un code qui utilise cet objet, code

l'on qualifie de consommateur, un itérateur sur cette structure de données offre au consommateur, en absence de détails concrets sur la structure de données, de pouvoir, à la demande, obtenir les différents éléments de cet objet, et itérer, ainsi et indirectement, sur les éléments de la structure de données.

Dans le chapitre précédent, à travers les réducteurs (*fold*, par exemple), nous avons vu un autre type d'interaction entre un tel type de producteur et un tel type de consommateur; cette interaction était réalisée à travers le passage d'une fonction en argument d'une autre fonction. Ainsi, le consommateur précise au producteur quel traitement il aimerait appliquer aux différents éléments qui caractérisent l'objet producteur.

Itérateur sur les listes Supposons que l'on veuille disposer d'une fonction nous retournant, à chaque appel, un élément d'une liste (en suivant un ordre bien précis : du premier au dernier). Une telle fonction pourrait être le résultat d'une fonction principale que l'on invoquerait et à qui on préciserait la liste sur laquelle on aimerait voir l'itérateur appliqué :

```
# let iter_list liste =
  let curseur = ref liste in
  fun () -> match !curseur with
    | [] -> failwith "Fin_de_liste"
    | e::r -> curseur := r; e;;
val iter_list : 'a list -> unit -> 'a = <fun>
```

L'utilisation de cet itérateur sur une liste d'entiers se fait de manière standard :

```
# let it1 = iter_list [1;2;3];;
val it1 : unit -> int = <fun>

# it1 ();;
- : int = 1

# it1 ();;
- : int = 2
```

Son utilisation sur une deuxième liste de *string* est aussi standard et est indépendante de la première : chaque itérateur (fonction) dispose de son propre curseur. Ce dernier est encapsulé dans la fermeture correspondante à chaque itérateur :

```
# let it2 = iter_list ["a";"b";"c"];;
val it1 : unit -> string = <fun>

# it1 ();;
- : string = "a"

# it1 ();;
- : string = "b"
```

Autres versions de l'itérateur sur les listes On pourrait évidemment joindre à la fonction «it» une fonction permettant de tester si la fin de la liste est atteinte :

```
# let iter_list' liste =
  let curseur = ref liste in
  let suivant () = match !curseur with
    | [] -> failwith "Fin_de_liste"
    | e::r -> curseur := r; e in
  let existe_suivant () = !curseur <> [] in
  suivant, existe_suivant;;
val iter_list' : 'a list -> (unit -> 'a) * (unit -> bool) = <fun>
```

Il est alors possible d'itérer sur les éléments de la liste :

```
# let suiv, existe_suiv = iter_list ' ["1";"2";"3"];;
val suiv : unit -> string = <fun>
val existe_suiv : unit -> bool = <fun>

# while existe_suiv() do
  Printf.printf "valeur_=%s\n" (suiv())
done;;
valeur = 1
valeur = 2
valeur = 3
- : unit = ()
```

Par ailleurs, il est possible de définir un type précis permettant de désigner les itérateurs, de même que d'associer les fonctions «suivant» et «existe_suiv» à un itérateur précis :

```
# type 'a list_iterateur = 'a list ref;;
type 'a list_iterateur = 'a list ref

# let iter liste : 'a list_iterateur = ref liste;;
val iter : 'a list -> 'a list_iterateur = <fun>

# let suivant (etat : 'a list_iterateur) = match !etat with
| [] -> failwith "Fin_de_liste"
| e::r -> etat := r; e;;
val suivant : 'a list_iterateur -> 'a = <fun>

# let existe_suivant (etat : 'a list_iterateur) = !etat <> [];;
val existe_suivant : 'a list_iterateur -> bool = <fun>
```

On utilise alors ces fonctions comme suit :

```
# let it = iter [1;2;3];;
val it : int list_iterateur = {contents = [1; 2; 3]}

# while existe_suivant it do
  Printf.printf "valeur_=%d\n" (suivant it)
done;;
valeur = 1
valeur = 2
valeur = 3
- : unit = ()
```

On remarque cependant que le type de l'itérateur est révélé au consommateur, ce qui pourrait engendrer des utilisations non standards de l'itérateur :

```
# it.contents <- [1];;
- : unit = ()

# suivant it;;
- : int = 1
```

La solution à ce problème est de rendre abstrait le type *list_iterateur*, grâce à l'utilisation des modules et des signatures :

```
# module Iter : sig
  type 'a list_iterateur

  val creer : 'a list -> 'a list_iterateur
  val suivant : 'a list_iterateur -> 'a
  val existe_suivant : 'a list_iterateur -> bool
end = struct
  type 'a list_iterateur = 'a list ref

  let creer liste = ref liste

  let suivant etat = match !etat with
    | [] -> failwith "Fin_de_liste"
    | e::r -> etat := r; e

  let existe_suivant etat = !etat <> []
end;;
```

```
module Iter :
  sig
    type 'a list_iterateur
    val creer : 'a list -> 'a list_iterateur
    val suivant : 'a list_iterateur -> 'a
    val existe_suivant : 'a list_iterateur -> bool
  end
```

Il est alors possible d'itérer sur les éléments de la liste :

```
# let it = Iter.creer [1;2];;
val it : int Iter.list_iterateur = <abstr>

# while Iter.existe_suivant it do
  Printf.printf "valeur_=%d\n" (Iter.suivant it)
done;;
valeur = 1
valeur = 2
- : unit = ()
```

3.1.4 Mémoïsation

L'idée de la mémoïsation est d'implanter une fonction qui, à partir de n'importe quelle autre fonction passée en argument, génère une version de celle-ci disposant d'une mémoire cache qui mémorise (*mémoïsation*), pour chaque valeur passée en argument à cette fonction, la valeur résultante de son application³ :

3. Les trois versions proposées sont équivalentes.


```

(* Version 1 *)
# let memoize f =
  let cache = ref [] in
  let memoizedFunc x =
    if mem_assoc x !cache
    then assoc x !cache
    else
      let r = f x in
      cache := (x,r) :: !cache;
      r
  in
  memoizedFunc;;
val memoize : ('a -> 'b) -> 'a -> 'b = <fun>

(* Version 2 *)
# let memoize' f =
  let cache = ref [] in
  fun x ->
    if mem_assoc x !cache
    then assoc x !cache
    else
      let r = f x in
      cache := (x,r) :: !cache;
      r;;
val memoize : ('a -> 'b) -> 'a -> 'b = <fun>

(* Version 3 *)
# let memoize'' f =
  let m = Hashtbl.create 1 in
  fun x ->
    try Hashtbl.find m x
    with Not_found ->
      let r = f x in
      Hashtbl.add m x f_x;
      r;;
val memoize : ('a -> 'b) -> 'a -> 'b = <fun>

```

Voici quelques exemples de la fonction `memoize` sur la fonction `d_fact` (page 94) :

```

(* Tests en utilisant la fonction originale de d_fact *)
# timeRun d_fact 5;;
- : int * float = (120, 4.)

# timeRun d_fact 5;;
- : int * float = (120, 4.)

(* Création d'une version «mémoisée» de d_fact *)
# let m_fact = memoize d_fact;;
val m_fact : int -> int = <fun>

(* Tests en utilisant la version «mémoisée» *)
# timeRun m_fact 5;;
- : int * float = (120, 4.)

# timeRun m_fact 5;;
- : int * float = (120, 0.)

```

Bien sûr, la fonction `memoize` est applicable à n'importe quelle fonction !

```

(* Exemple 1 *)
# let m_add = memoize d_add;;
  val m_add : int * int -> int = <fun>

# timeRun m_add (5,6);;
- : int * float = (11, 1.)

# timeRun m_add (5,6);;
- : int * float = (11, 0.)

(* Exemple 2 *)
# let m_f = memoize (map d_fact);;
  val m_f : int list -> int list = <fun>

# timeRun m_f [1;2;3;4];;
- : int list * float = ([1; 2; 6; 24], 6.)

# timeRun m_f [1;2;3;4];;
- : int list * float = ([1; 2; 6; 24], 0.)

```

Voici un autre exemple de définition de la fonction «factoriel» avec laquelle le fonctionnement de la version mémorisée sera plus explicite :

```

# let rec factorial n =
    Printf.printf "fact(%d);_ " n;
    if n <= 1 then 1 else n * factorial(n-1);;
  val factorial : int -> int = <fun>

# factorial 4;;
fact(4); fact(3); fact(2); fact(1); - : int = 24

# factorial 4;;
fact(4); fact(3); fact(2); fact(1); - : int = 24

```

L'utilisation de la version mémorisée montre clairement le gain en performance que l'on peut avoir :

```

# let factorial' = memoize factorial;;
  val factorial' : int -> int = <fun>

# factorial' 4;;
fact(4); fact(3); fact(2); fact(1); - : int = 24

# factorial' 4;;
- : int = 24

```

Ceci dit, si on veut calculer la factoriel de 5 ou de 6, voici ce qui se passe :

```

# factorial' 5;;
fact(5); fact(4); fact(3); fact(2); fact(1); - : int = 120

# factorial' 6;;
fact(6); fact(5); fact(4); fact(3); fact(2); fact(1); - : int = 720

```

Ce résultat est prévisible puisque ni la factoriel de 5, ni celle de 6 n'ont été auparavant calculées et enregistrées dans la mémoire cache de la version mémorisée de «factorial». Cependant, on remarque que le calcul de la factoriel de 6 utilise celui de la factoriel de 5, lequel utilise celui de la factoriel de 4; cette dernière ayant déjà été calculée et enregistrée dans la mémoire cache, on aurait alors souhaité réutiliser ce résultat lors du calcul de la factoriel de 5 (idem pour le calcul de la factoriel de 6 vis-à-vis du calcul de la factoriel de 5).

Le problème vient du fait que dans la fonction «memoize», l'application de la fonction à mémoriser utilise la version non mémorisée : autrement dit, si on considère l'exemple de la fonction «factorial», lors de l'évaluation de la fonction à mémoriser, dans le code «let r = f x», on utilise la version «factorial» pour le calcul de «r», laquelle version est non mémorisée et ne disposant donc pas de mémoire cache : les appels récursifs successifs ne sont donc pas capturés et enregistrés dans la mémoire cache.

La solution consiste donc à changer la manière avec laquelle la fonction «memoize» est définie ainsi que la version non mémorisée de la fonction à mémoriser. Voici donc la nouvelle version de la fonction «memoize», version adaptée aux fonctions récursives :

```
# let memoize_rec f =
  let cache = ref [] in
  let rec memoizedFunc x =
    if mem_assoc x !cache
    then assoc x !cache
    else
      let r = f memoizedFunc x in
      cache := (x,r) :: !cache;
      r
  in
  memoizedFunc;;
val memoize_rec : ('a -> 'b) -> 'a -> 'b = <fun>
```

Il est aussi nécessaire de définir une version particulière de «factorial» afin qu'elle puisse être utilisée avec la nouvelle version de «memoize» :

```
# let factorial fact n =
  printf "fact(%d);\n" n;
  if n <= 1 then 1 else n * fact(n-1);;
val factorial : (int -> int) -> int -> int = <fun>
```

On peut désormais générer une version mémorisée de cette fonction et apprécier les gains en performance obtenus :

```
# let factorial' = memoize_rec factorial;;
val factorial' : int -> int = <fun>

# factorial' 3;;
fact(3); fact(2); fact(1); - : int = 6

# factorial' 4;;
fact(4); - : int = 24

# factorial' 7;;
fact(7); fact(6); fact(5); - : int = 5040

# factorial' 6;;
- : int = 720
```

Pour terminer, voici un autre exemple classique pour lequel la fonction «memoize_rec» sera d'un grand apport : la fonction fibonacci !

```
# let rec fib n =
  printf "fib(%d);\n" n;
  match n with
  | 0 | 1 -> n
  | _ -> fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>

# fib 3;;
fib(3); fib(1); fib(2); fib(0); fib(1); - : int = 2

# fib 4;;
fib(4); fib(2); fib(0); fib(1); fib(3); fib(1); fib(2); fib(0); fib(1); - : int = 3
```

La version mémorisée est obtenue comme suit :

```
# let fib f n =
  printf "fib(%d);\n" n;
  match n with
  | 0 | 1 -> n
  | _ -> f(n-1) + f(n-2);;
val fib : (int -> int) -> int -> int = <fun>

# let fib' = memoize_rec fib;;
val fib' : int -> int = <fun>
```

Encore une fois, les gains en performance sont appréciables :

```
# fib' 3;;
fib(3); fib(1); fib(2); fib(0); - : int = 2

# fib' 4;;
fib(4); - : int = 3

# fib' 6;;
fib(6); fib(5); - : int = 8
```

Au delà de la mémorisation : À l'instar de la mémorisation, il est possible de définir d'autres types de fonctions qui génèrent une version particulière d'une fonction donnée (ces fonctions sont parfois désignées comme «décorateurs» dans d'autres langages comme Python, par ex.) :

```
# let nbr_appel f =
  let c = ref 0 in
  (fun x -> incr c; f x), (fun () -> !c);;
val nbr_appel : ('a -> 'b) -> ('a -> 'b) * (unit -> int) = <fun>
```

La fonction «nbr_appel» prend pour argument une fonction «f» et retourne une version de celle-ci qui, outre sa fonctionnalité de base, comptabilise le nombre de fois qu'on l'a invoquée; aussi, la fonction «nbr_appel» retourne une deuxième fonction qui permet de connaître la valeur du compteur d'appels.

Voici un exemple d'utilisation dans laquelle nous générons une version de la fonction «d_fact» disposant d'un tel compteur :

```
# let n_fact, counter_f = nbr_appel d_fact;;
val n_fact : int -> int = <fun>
val counter_f : unit -> int = <fun>

# n_fact 5;;
- : int = 120

# n_fact 2;;
- : int = 2

# counter_f();;
- : int = 2
```

Par ailleurs, il est possible de composer ou de chaîner des fonctions «décoratrices». Dans l'exemple qui suit, on utilise l'opérateur de composition «%%» (dont on rappelle la définition) afin de générer une version mémoisée et disposant d'un compteur d'appels de la fonction «d_fact» :

```
# let (%%) f g x = f (g x);;
val ( %% ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let nm_fact, counter_f = (nbr_appel %% memoize) d_fact;;
val nm_fact : int -> int = <fun>
val counter_f : unit -> int = <fun>

# timeRun nm_fact 4;;
- : int * float = (24, 3.0044128894805908)
# timeRun nm_fact 3;;
- : int * float = (6, 2.001251220703125)
# timeRun nm_fact 4;;
- : int * float = (24, 0.)

# counter_f();;
- : int = 3
```

3.1.5 Récursivité terminale

Une fonction est dite récursive terminale (*tail recursive*) si ses appels récursifs se trouvent tous en position terminale d'évaluation (*tail calls*), c'est-à-dire s'ils sont les derniers à être évalués dans les expressions qui les englobent.

Par exemple, la fonction «sum» définie ci-dessous n'est pas récursive terminale puisque l'appel récursif «sum(n-1)» ne correspond pas à la dernière expression évaluée dans la partie **else** du corps de la fonction. La dernière expression à être évaluée est l'addition entre 1 et la valeur résultante de cet appel récursif :

```
# let rec sum n = if n <= 0 then 0 else 1 + sum(n-1);;
val sum : int -> int = <fun>

# sum 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Comme on peut le constater, à cause de la taille limitée de la pile d'exécution, cet exemple provoque une exception de type «*stack overflow*» (débordement de pile). Pour contourner ces limites, la solution consiste à redéfinir la fonction «sum» en version récursivité terminale. Ce type de transformation passe généralement par l'ajout de paramètres additionnels (accumulateurs) et de la définition de fonctions auxiliaires :

```
# let rec sum' n acc =
  if n <= 0 then acc else sum' (n-1) (acc + 1);;
val sum' : int -> int -> int = <fun>

# sum' 1000000 0;;
- : int = 1000000
```

```
# let sum'' n =
  let rec aux k acc =
    if k <= 0 then acc else aux (k-1) (acc + 1)
  in
    aux n 0;;
val sum'' : int -> int = <fun>

# sum'' 1000000;;
- : int = 1000000
```

Comme on peut le constater, dans ces deux versions équivalentes de la fonction «sum» en version récursive terminale, l'exception a disparu et le traitement termine en retournant un résultat correct. La raison de ce succès vient du fait que lorsqu'une fonction récursive est récursive terminale, l'interpréteur ou le compilateur OCaml génère, en *bytecode* ou en binaire, une version itérative de cette fonction. En effet, par définition de la récursivité terminale, étant donné qu'après un appel récursif, aucune autre expression n'est à évaluer, il n'est pas nécessaire d'empiler de nouvelles valeurs dans la pile d'exécution avant cet appel : on peut réutiliser le *frame* courant de la pile ne changeant ainsi pas sa taille.

Pour terminer, voici un exemple plus classique de fonction en récursion terminale :

```
# let fact n =
  let rec aux n' acc =
    if n' <= 1 then acc else aux (n'-1) (acc * n')
  in
    aux n 1;;
val fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

3.1.6 Continuations

Une *continuation* est une fonction qui indique comment un calcul doit se poursuivre. Étant données une expression *e* et une fonction *k*, alors *k(e)* désigne *k* comme une continuation de *e*. On parle surtout de *passage d'arguments par continuation*.

```
# let sqr x = x * x;;
val sqr : int -> int = <fun>

# let f1 x = (sqr x) + 4;;
val f1 : int -> int = <fun>

# let sqr_cont k x = k (sqr x);;
val sqr_cont : (int -> 'a) -> int -> 'a = <fun>

# let f2 = sqr_cont (fun x -> x + 4);;
val f2 : int -> int = <fun>
```

Dans le deuxième exemple, on définit une fonction «sqr_cont» qui prend un argument «k», la continuation, et un argument «x». Cette fonction calcule le carré de «x» puis applique au résultat la continuation. Le dernier exemple illustre l'utilisation de cette fonction avec une continuation qui consiste à ajouter à l'argument la valeur 4.

Une première utilisation intéressante des continuations est la gestion des fonctions partielles, plus précisément, la gestion de leurs cas exceptionnels.

Voici la définition d'une fonction «sqrt» qui calcule la racine carrée d'un nombre réel passé en argument et qui lève une exception dans le cas où l'argument est inférieur à zéro :

```
# let sqrt' x =
  if x < 0.0 then failwith("Negative_argument")
  else sqrt x;;
val sqrt' : float -> float = <fun>
```

La définition de cette fonction peut être généralisée en utilisant le passage de continuations :

```
# let sqrt_cont kfail ksuccess x =
  if x < 0.0 then kfail x
  else ksuccess (sqrt x);;
val sqrt_cont : (float -> 'a) -> (float -> 'a) -> float -> 'a = <fun>
```

On précise donc, sous forme de continuations passées en argument, comment cette fonction devrait poursuivre (*continuer*) son traitement dans le cas où une situation exceptionnelle intervient (*kfail*) ou dans le cas où tout se déroule correctement (*ksuccess*).

À partir de cette fonction, il est alors possible de choisir différents traitements associés à la gestion des cas exceptionnels :

- en précisant que la fonction retourne la constante `None` dans le cas exceptionnel et `Some(r)` pour signifier que le résultat est r^4 ;
- en levant une exception dans le cas exceptionnel ;
- en retournant un nombre complexe !

```
# sqrt_cont (fun x -> None)
              (fun x -> Some x)
              (-1.0);;
- : float option = None

# sqrt_cont (fun x -> failwith("Negative_argument"))
              (fun x -> x)
              (-1.0);;
Exception: Failure "Negative_argument".

# type complex = C of float * float;;
type complex = C of float * float

# sqrt_cont (fun x -> C(0.0, sqrt(-.x)))
              (fun x -> C(x,0.0))
              (-1.0);;
- : complex = C (0., 1.)
```

Bien sûr, à partir de la forme générale de la fonction, «sqrt_cont», il est possible d'extraire de nouvelles fonctions traitant ces différents types de gestion de cas exceptionnels :

```
# let sqrt_option = sqrt_cont (fun x -> None) (fun x -> Some x);;
val sqrt_option : float -> float option = <fun>

# let sqrt_except = sqrt_cont (fun x -> failwith("Negative_argument"))
                              (fun x -> x);;
val sqrt_except : float -> float = <fun>

# let sqrt_complex = sqrt_cont (fun x -> C(0.0, sqrt(-.x)))
                              (fun x -> C(x,0.0));;
val sqrt_complex : float -> complex = <fun>
```

4. `None` et `Some` sont des constructeurs d'un type prédéfini en OCaml :
type 'a option = `None` | `Some of 'a`

Mais là où les continuations s'imposent vraiment, est dans le traitement de certains cas particuliers de récursivité terminale. Soit l'exemple suivant qui propose une définition d'un type de données «intTree» ainsi que d'une fonction récursive qui calcule la somme des éléments d'un arbre de type «intTree» :

```
# type intTree = Leaf of int | Node of intTree * intTree;;
type intTree = Leaf of int | Node of intTree * intTree

# let rec sumTree tree =
  match tree with
  | Leaf(n) -> n
  | Node(l,r) -> (sumTree l) + (sumTree r);;
val sumTree : intTree -> int = <fun>
```

Si on applique la fonction «sumTree» à un arbre (hyper-)non-balancée contenant 1000000 fois l'entier 1, on remarque qu'on aura naturellement une exception de type «*stack overflow*» :

```
(* Arbre hyper-non-balancée contenant les entiers de 0 à 1000000 *)
# let imbTree = (1 ^~ 1000000)
  |> List.fold_left (fun st v -> Node(Leaf v, st)) (Leaf 0);;
val imbTree : intTree =
  Node (Leaf 1, Node (Leaf 1, Node (Leaf 1, Node (Leaf 1, ...))))

sumTree imbTree;;
Stack overflow during evaluation (looping recursion?).
```

La solution passe donc par une version en récursion terminale. Cependant, on remarque que la fonction «sumTree» comprend non pas un appel mais deux appels récursifs qui ne sont pas en position terminale d'évaluation. La technique qui consiste à utiliser un accumulateur en argument ne marche plus ; il faut faire autrement.

L'utilisation des continuations permet de résoudre ce type de problèmes :

```
# let rec sumTreeCont tree cont =
  match tree with
  | Leaf(n) -> cont n
  | Node(l,r) -> sumTreeCont l (fun n -> sumTreeCont r (fun m -> cont (n+m)));;
val sumTreeCont : intTree -> (int -> 'a) -> 'a = <fun>

sumTreeCont imbTree (fun x -> Printf.printf "result is : %d\n" x);;
result is : 1000000
- : unit = ()

sumTreeCont imbTree (fun x -> x);;
- : int = 1000000
```

On remarque que la fonction «sumTreeCont» est en récursion terminale.

Pour terminer, voici une version de la fonction «map» utilisant les continuations :

```
# let map' f =
  let rec aux k = function
    | [] -> k []
    | h::t -> aux (fun t -> k(f h::t)) t
  in
  aux (fun x -> x);;
val map' : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map' (( ^ ) "a") ["1"; "2"; "3"];;
- : string list = ["a1"; "a2"; "a3"]
```


On remarque que la fonction auxiliaire est en récursivité terminale, ce qui implique que «map'» l'est aussi.

Par ailleurs, en utilisant un accumulateur, il est possible d'obtenir autrement une version de la fonction «map» en récursivité terminale :

```
# let map' ' f =
  let rec aux acc = function
    | [] -> acc
    | h::t -> aux (acc@[f h]) t
  in
    aux [];;
val map' ' : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map' ' (( ^ ) "a") ["1"; "2"; "3"];;
- : string list = ["a1"; "a2"; "a3"]
```

3.1.7 Évaluations par valeur/par nom/paresseuse

Évaluation par valeur (*evaluation by value*)

Par défaut, OCaml évalue les expressions en utilisant la stratégie (standard) d'évaluation par valeur. Soit une fonction f ayant comme paramètre x et soit E une expression ; f appliquée à E est évaluée comme suit :

- on évalue E pour obtenir une valeur v ;
- on substitue x par v dans le corps C de f , ce qui nous donne un corps C' ;
- on évalue C' .

Voici quelques exemples illustrant ce type d'évaluation :

```
# let rec fact n = if n <= 1 then 1 else n * fact (n-1);;
val fact : int -> int = <fun>

# fst (1 + 2, 3 + 4);;
- : int = 3

# fst (1 + 2, fact 10);;
- : int = 3

# fst ((print_string "First_element\n"; 1 + 2),
      (print_string "Second_element\n"; fact 10));;
Second_element
First_element
- : int = 3
```

Il est clair que dans ce cas, l'évaluation de «fact 10» n'aura servi à rien.

D'autres situations peuvent cependant être plus problématiques :

```
# fst (1, 1/0);;
Exception: Division_by_zero.

# let rec loop x = loop x;;
val loop : 'a -> 'b = <fun>

# fst (1, loop 0);;
(* boucle infinie ! *)
```

Dans le premier cas, une exception est soulevée ; dans le deuxième cas, aucune valeur n'est retournée puisque l'interpréteur est dans un état de boucle infinie ! Notons cependant que dans ces deux cas, nous voulions seulement retourner le premier élément d'une paire, premier élément, qui dans les deux cas, ne pose aucun problème.

Évaluation par nom (*evaluation by need*)

En considérant les mêmes hypothèses mentionnées précédemment, f appliquée à E est évaluée comme suit :

- on substitue x par E dans le corps C de f , ce qui nous donne un corps C' ;
- on évalue C' .

Par conséquent, en considérant les deux exemples précédents, la valeur 1 sera retournée (ni «1/0», ni «loop 0» ne seront évaluées).

Cependant, en considérant une fonction abs' définie comme suit :

```
# let abs' x = if x >= 0 then x else -x;;
val abs' : int -> int = <fun>
```

L'évaluation de cette fonction à «fact 6» résulterait en l'évaluation de l'expression suivante :

```
(* if (fact 6) >= 0 then (fact 6) else ~(fact 6) *)
```

On remarque donc que l'expression «fact 6» sera évaluée à 2 reprises !

Évaluation paresseuse (*lazy evaluation*)

Il s'agit du même principe que l'évaluation par nom sauf qu'on utilise un mécanisme (cache) pour préserver les évaluations multiples. Donc, dans le corps *C'* où apparaissent plusieurs occurrences de *E*, dès que celle-ci est évaluée une première fois pour donner une valeur *v*, elle ne le sera plus pour les autres occurrences (on réutilisera simplement sa valeur, c'est-à-dire *v*).

Pour illustrer le fonctionnement de la stratégie paresseuse, essayons tout d'abord d'implanter un mécanisme qui simule l'évaluation par nom en utilisant les fonctions ; en effet, ces derniers ont la propriété que leur corps comprend une expression qu'on dit congelée (*frozen expression*) puisqu'elle n'est évaluée que lors d'un appel à la dite fonction.

```
# let f () = 1 / 0;;
val f : unit -> int = <fun>
(* L'expression 1/0 est dite "congelée" puisqu'elle n'est pas évaluée *)

(* C'est seulement à l'application de la fonction qu'elle le sera *)
# f();;
Exception: Division_by_zero.
```

Par conséquent, si on souhaite implanter une version de la fonction *fst* qui prendrait comme argument des expressions congelées, on pourrait le faire comme suit⁵ :

```
# let fst' ((x : unit -> 'a),(y : unit -> 'b)) = x();;
val fst' : (unit -> 'a) * (unit -> 'b) -> 'a = <fun>

# fst' ((fun () -> 1),(fun () -> 1/0));;
- : int = 1

# fst' ((fun () -> 1),(fun () -> loop 0));;
- : int = 1
```

Voici une implantation plus aboutie dans laquelle on utilise un type algébrique pour bien identifier les expressions congelées dans les types inférés par OCaml (la fonction «force» permet de décongeler une expression congelée) :

```
# type 'a frozen = Frozen of (unit -> 'a);;
type 'a frozen = Frozen of (unit -> 'a)

# let force (Frozen e) = e();;
val force : 'a frozen -> 'a = <fun>

# let e = Frozen (fun () -> 1/0);;
val e : int frozen = Frozen <fun>

# force e;;
Exception: Division_by_zero.

# fst (1, Frozen (fun () -> 1/0));;
- : int = 1
```

5. Dans cet exemple, le premier argument de *fst'* aurait pu être passé sous forme non congelée.

L'exemple qui suit montre que le problème de l'évaluation multiple d'une expression (caractéristique de l'évaluation par nom) demeure, et motive donc de recourir à une autre stratégie d'évaluation : l'évaluation paresseuse.

```
# let inc froz_exp n = (force froz_exp) + n;;
val inc : int frozen -> int -> int = <fun>

# let mult_eval froz_exp =
  if (force froz_exp) > 0 then inc froz_exp 1
  else inc froz_exp (-1);;
val mult_eval : int frozen -> int = <fun>

# let x1 = Frozen (fun () -> (print_string "EVAL\n"; 3 + 4));;
val x1 : int frozen = Frozen <fun>

# mult_eval x1;;
EVAL
EVAL
- : int = 8

# mult_eval x1;;
EVAL
EVAL
- : int = 8
```

Ceci peut donc être optimisé grâce à la paresse ! À cette fin, on définit un nouveau type de données qui comprend, en plus de l'expression congelée, une valeur correspondant à l'évaluation de l'expression congelée, une fois décongelée. Étant donné que l'état de ce type de données pourra changer, on utilise naturellement les références.

```
# type 'a lazyaux =
  | Immediate of 'a
  | Frozen of (unit -> 'a);;
type 'a lazyaux = Immediate of 'a / Frozen of (unit -> 'a)

# type 'a lazyval = LazyVal of 'a lazyaux ref;;
type 'a lazyval = LazyVal of 'a lazyaux ref

# let force (LazyVal e) = match !e with
  | Immediate x -> x
  | Frozen f -> let v = f() in e := Immediate v; v;;
val force : 'a lazyval -> 'a = <fun>
```

On remarque donc que la décongélation multiple d'une expression congelée n'entraînera pas son évaluation multiple :

```
# let x1 = LazyVal (ref (Frozen (fun () -> (print_string "EVAL\n"; 3 + 4))));;
val x1 : int lazyval = LazyVal {contents = Frozen <fun>}

# force x1;;
EVAL
- : int = 7

# force x1;;
- : int = 7
```

L'exemple précédent, utilisant la fonction «mul_eval», s'évalue alors comme suit :

```
# let x2 = LazyVal (ref (Frozen (fun () -> (print_string "EVAL\n"; 3 + 4))));
val x2 : int lazyval = LazyVal {contents = Frozen <fun>}

# let inc froz_exp n = (force froz_exp) + n;;
val inc : int lazyval -> int -> int = <fun>

# let mult_eval froz_exp =
  if (force froz_exp) > 0 then inc froz_exp 1
  else inc froz_exp (-1);;
val mult_eval : int lazyval -> int = <fun>

# mult_eval x2;;
EVAL
- : int = 8

# mult_eval x2;;
- : int = 8
```

Pour terminer, notons qu'OCaml offre un module, `Lazy`, qui comprend plusieurs ressources pré-définies pour la paresse! `lazy` est une fonction qui déroge à la stratégie d'évaluation par valeur, utilisée par OCaml : l'application de cette fonction à un argument n'évalue pas ce dernier mais produit une valeur de type `'a lazy_t` (indiquant que le résultat de cette application est une expression congelée); ainsi :

```
# open Lazy;;

# let x1 = lazy (print_endline "EVAL"; sleep 5; 3 + 4);;
val x1 : int lazy_t = <lazy>
```

n'affichera rien à l'écran et retournera une valeur de type `int lazy_t`. La fonction `force`, définie dans `Lazy`, permet de forcer l'évaluation d'une expression congelée :

```
# force x1;;
EVAL
- : int = 7
```

La décongélation de l'expression `x1` a pour effet d'afficher `EVAL` à l'écran, d'hiberner 5 sec., d'évaluer l'expression `3 + 4` et de retourner son résultat, soit 7.

Si on évalue de nouveau :

```
# force x1;;
- : int = 7
```

l'interpréteur n'affichera plus `EVAL` à l'écran, ni hibernera pendant un laps de temps, ni évaluera `3 + 4`, mais retournera directement la valeur 7 (ainsi on a l'effet voulu recherché par la paresse).

3.1.8 Listes paresseuses ou flots (*stream*)

Supposons que l'on veuille appliquer à une liste de 10000 entiers un certain traitement, par exemple élever au carré ses éléments, puis filtrer le résultat obtenu pour ne retenir que les nombres inférieurs à 50000 pour finalement rechercher parmi les résultants s'il en existe un qui est multiple de 7^6 :

6. Remarquez la forme déclarative du code proposé par rapport à l'énoncé de l'exemple.

```
# (1 -- 10000) |> map (fun x -> x * x)
                |> filter (fun x -> x <= 50000)
                |> exists (fun x -> x mod 7 = 0);;
- : bool = true
```

La réponse est immédiate : **true**. Pour connaître le nombre d'éléments que la fonction **filter** a produit, on peut procéder comme suit :

```
# (1 -- 10000) |> map (fun x -> x * x)
                |> filter (fun x -> x <= 50000)
                |> length;;
- : int = 223
```

Et si on veut connaître les éléments qu'a traitée la fonction **exists**, on peut ajouter, avant le test « $x \bmod 7 = 0$ », un affichage de la valeur courante traitée par cette fonction :

```
# (1 -- 10000) |> map (fun x -> x * x)
                |> filter (fun x -> x <= 50000)
                |> exists (fun x -> printf "%d" x; x mod 7 = 0);;
-1-4-9-16-25-36-49- : bool = true
```

Si on résume :

- on a parcouru, avec la fonction **map**, une liste de 10000 entiers pour en calculer le carré et produire une nouvelle liste de 10000 résultats;
- on a parcouru, avec la fonction **filter**, une liste de 10000 entiers pour en retenir 223 qui sont supérieurs à 50000;
- on a parcouru, avec la fonction **exists**, 7 éléments pour en découvrir un qui est multiple de 7!

On constate clairement qu'il y a plusieurs calculs intermédiaires qui n'ont pas du tout servi (gaspillage au niveau mémoire et temps de calcul) : Pourquoi parcourir 10000 entiers, à deux reprises (**map** puis **filter**), quand au final, on n'en utilisera qu'une infime partie (7)? Aussi, pourquoi générer une liste de 10000 entiers (carré des premiers) quand on n'en aurait nécessité que 7?

Même si ce type de traitement (exemple précédent) n'est pas très significatif, il demeure que dans la réalité (requêtes aux bases de données, etc.), il arrive souvent que l'on applique plusieurs traitements inutiles sur des listes d'envergure quand en définitif seule une partie des éléments obtenus sera effectivement traitée : un paresseux ne ferait jamais ça! Effectivement, la paresse peut curieusement apporter une solution à de telles situations.

L'idée est de définir un type de données *'a t* (dans un module *Stm*) représentant des listes de valeurs dont les éléments sont congelés (on ne les évalue qu'au besoin) :

```
# type 'a t =
  | Empty
  | Cons of 'a lazy_t * 'a t lazy_t;;
type 'a t = Empty | Cons of 'a lazy_t * 'a t lazy_t
```

Aussi, pour exploiter ce type de données, on peut créer une fonction assez générale et générique qui crée un flot borné (*stm*) de données congelées à partir d'une fonction *f*, d'une valeur *x* et d'une valeur *n*. L'idée est de créer un flot de *n* données $[x; f(x); f(f(x)); \dots; f(\dots(f(x))\dots)]$:

```
# let rec build_n f x n =
  if n <= 0 then Empty
  else Cons (lazy x, lazy (build_n f (f x) (n-1)));;
val build_n : ('a -> 'a) -> 'a -> int -> 'a t = <fun>
```

Voici deux exemples de flots :

```
# let s1 = Stm.build_n succ 0 1000000000;;
val s1 : int Stm.t = Stm.Cons (lazy 0, <lazy>)

# let s2 = Stm.build_n (fun x -> print_endline x; x ^ x) "a" 1000000000;;
val s2 : string Stm.t = Stm.Cons (lazy "a", <lazy>)
```

Les éléments des deux flots sont congelés, c'est-à-dire qu'aucun élément n'a été réellement créé (la fonction `print_endline` n'est effectivement pas appliquée) : une chance puisque ces flots admettent un milliard d'éléments chacun.

Créons deux fonctions qui permettent d'accéder à certaines parties d'un flot, décongelant ainsi les éléments concernés par cet accès :

```
# let rec nth s n = match s with
| Empty -> failwith "Stm.nth"
| Cons (x, r) ->
    if n = 0 then force x
    else nth (force r) (n-1);;
val nth : 'a Stm.t -> int -> 'a = <fun>

# let rec peak n s = match s with
| Empty -> []
| Cons (x, r) ->
    if n <= 0 then []
    else (force x)::(peak (n-1) (force r));;
val peak : int -> 'a Stm.t -> 'a list = <fun>
```

La première fonction permet d'accéder au (n+1)-ième élément d'un flot ; la deuxième permet de retourner les n premiers éléments d'un flot. Voici un exemple de leur utilisation :

```
# Stm.nth s1 4;;
- : int = 4

# Stm.peak 10 s1;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]

# Stm.nth s2 3;;
a
aa
aaaa
- : string = "aaaaaaa"

# Stm.peak 5 s2;;
aaaaaaaa
aaaaaaaaaaaaaaaa
- : string list = ["a"; "aa"; "aaa"; "aaaaaaa"; "aaaaaaaaaaaaaaaa"]
```

L'effet de bord implanté dans la fonction permettant de construire les éléments du flot `s2` montre clairement ce qui se passe lors de l'utilisation de ces deux fonctions :

- L'expression «`Stm.nth s2 3`» permet d'accéder au quatrième élément de `s2`, décongelant au passage les trois premiers.
- L'expression «`Stm.peak 5 s2`» permet de retourner les cinq premiers éléments de `s2` ; seul le cinquième sera décongelé puisque les quatre premiers l'ont déjà été. La paresse fonctionne parfaitement !

Pour terminer, voici un exemple de fonction créant un flot infini ! En effet, étant congelé, on peut sans risque considérer qu'un flot admet une infinité de valeurs :

```
# let rec build f x =
  Cons (lazy x, lazy (build f (f x)));;
val build : ('a -> 'a) -> 'a -> 'a Stm.t = <fun>
```

Dans ce cas, le code suivant :

```
# let pairs = Stm.build (fun x -> x + 2) 0;;
val pairs : int Stm.t = Stm.Cons (lazy 0, <lazy>)
```

crée l'ensemble infini des nombres entiers pairs! Si on veut accéder aux 10 premiers éléments pairs, il n'y a rien de plus simple :

```
# Stm.peak 10 pairs;;
- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
```

Par ailleurs, on peut définir des fonctionnelles qui prennent des flots en entrée et y appliquent un traitement à tous leurs éléments. Bien sûr, le traitement est congelé et ne sera effectif qu'à l'utilisation des éléments produits. Par exemple, voici une implantation de la fonction «map» pour les flots :

```
# let rec map f s = match s with
| Empty -> Empty
| Cons (x,r) -> Cons(lazy (f (force x)), lazy (map f (force r)))
val map : ('a -> 'b) -> 'a Stm.t -> 'b Stm.t = <fun>
```

Comme le précise sa signature, cette fonction prend une fonction en entrée, puis un flot, et retourne un nouveau flot en résultat. Voici un exemple d'utilisation de cette fonction (appliquée aux flots bornés) :

```
# let s1' = Stm.map (fun x -> x * x) s1;;
val s1' : int Stm.t = Stm.Cons (<lazy>, <lazy>)

# let s2' = Stm.map (fun x -> String.length x) s2;;
val s2' : int Stm.t = Stm.Cons (<lazy>, <lazy>)

# Stm.peak 10 s1';;
- : int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]

# Stm.peak 6 s2';;
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
- : int list = [1; 2; 4; 8; 16; 32]
```

Le dernier exemple montre clairement (affichage du 6ème élément) que seul le sixième élément de s2 sera décongelé lors de la décongélation du traitement passé en argument de la fonction «Stm.map». Aussi, seuls les six premiers éléments de s2' sont effectivement décongelés.

Par ailleurs, notons que la fonction «Stm.map» n'est pas limitée aux flots bornés : on peut l'appliquer à des flots infinis!

```
# let impairs = Stm.map succ pairs;;
val impairs : int Stm.t = Stm.Cons (<lazy>, <lazy>)

# Stm.peak 10 impairs;;
- : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19]
```


Par ailleurs, reprenons l'exemple initial (page 114) qui consiste à calculer le carré de 10000 entiers, à filtrer une partie des résultats obtenus puis à rechercher s'il existe un multiple de 7. À cette fin, il est nécessaire de définir des versions des fonctions **filter** et **exists** adaptées aux flots (ainsi que l'opérateur « \rightarrow » adapté aux flots) :

```
# let rec filter p s = match s with
| Empty -> Empty
| Cons (x, xs) ->
    if p (force x) then Cons(x, lazy (filter p (force xs)))
    else filter p (force xs);;
val filter : ('a -> bool) -> 'a Stm.t -> 'a Stm.t = <fun>

# let rec exists p s = match s with
| Empty -> false
| Cons (x, r) -> p (force x) || exists p (force r);;
val exists : ('a -> bool) -> 'a Stm.t -> bool = <fun>

# let (---) n m =
    build_n succ n (m-n+1);;
val (---) : int -> int -> int Stm.t = <fun>
```

L'exemple devient alors :

```
# Stm.(1 --- 10000) |> Stm.map (fun x -> printf "m(%d)\n" x; x * x)
|> Stm.filter (fun x -> printf "f(%d)\n" x; x <= 50000)
|> Stm.exists (fun x -> printf "e(%d)\n" x; x mod 7 = 0);;
m(1) f(1) e(1) m(2) f(4) e(4) m(3) f(9) e(9) m(4) f(16) e(16) m(5) f(25) e(25)
m(6) f(36) e(36) m(7) f(49) e(49) - : bool = true
```

Clairement, seuls les traitements, respectivement les données, nécessaires ont été calculés, respectivement créés.

Pour terminer, notons que les projets *LINQ* et *PLINQ* de Microsoft utilisent abondamment les notions de calculs et de listes congelés. La motivation d'une telle utilisation est très simple : n'effectuer les traitements que lorsque nécessaire, c'est-à-dire lorsqu'on accède à des valeurs précises du flot résultant de toute une requête *LINQ* et *PLINQ*. Cet accès se fait en général à l'aide d'un itérateur, par exemple l'instruction **foreach** du langage **C#**, et à l'aide du mot-clé **yield** utilisé dans l'implantation de ce type d'itérateurs (voir page 119).

3.2 Programmation déclarative : LINQ/PLINQ

Les exemples qui suivent permettent de considérer le langage OCaml comme un langage déclaratif ou un langage de requête (à la *SQL*) :

```
# let select = List.map
and where = List.filter;;
val select : ('a -> 'b) -> 'a list -> 'b list = <fun>
val where : ('a -> bool) -> 'a list -> 'a list = <fun>

# let list = 1--9;;
val list : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]

# let result =
    list
    |> where (fun x -> x mod 2 = 0)
    |> select (fun x -> x * x);;
val result : int list = [4; 16; 36; 64]
```

LINQ Les concepteurs du langage **C#** ont ajouté ce style de requête à *la SQL* à même la syntaxe du langage. Autrement dit, et c'est en partie l'idée derrière le projet *LINQ*, il est possible d'écrire la requête précédente de cette manière :

```
(* version C# *)
var result =
  from x in list
  where x % 2 == 0
  select x * x;
```

Remarquez que le type de `result` n'est pas précisé. Le compilateur infèrera pour nous son type. Aussi, en réalité, le compilateur considère cette requête (ce code) comme le code suivant :

```
(* version C# *)
var result =
  list
  .Where(x => x % 2 == 0)
  .Select(x => x * x);
```

J'ai volontairement écrit ce code sur plusieurs lignes pour bien faire ressortir la quasi-équivalence au niveau syntaxique entre la version **C#** et la version **OCaml**. Sur une ligne, ce code correspond alors à :

```
(* version C# *)
var result = list.Where(x => x % 2 == 0).Select(x => x * x);
```

Comment lire ce code ? En réalité, en **C#**, les listes, les tableaux, etc. correspondent à des classes qui implantent l'interface `IEnumerable<T>` (interface générique (`T`) qui représente des collections d'éléments). Aussi, dans les paquetages *LINQ*, on a simplement «ajouté» des méthodes de classe («static») : `Where`, `Select`, `Groupby`, ... Donc, dans le code précédent, on invoque simplement, et successivement, la méthode `Where` et la méthode `Select`. On remarque que les paramètres passés en argument correspondent à des fonctions anonymes. La syntaxe utilisée par **C#** est

arg => corps

Celle de **OCaml**, qui date de près de 30 ans, est

fun *arg* -> *corps*

Donc, à la place d'invoquer des méthodes de classe avec la notation «.» et de passer des fonctions anonymes en argument, dans les versions récentes de **C#**, on a ajouté dans la syntaxe de ce dernier un ensemble de constructions syntaxiques qui permettent d'écrire des expressions qui ressemblent au langage *SQL*, en faisant abstraction de la notion de méthodes de classe et de fonctions anonymes. Ce qui nous donne au final, des exemples de code de la forme :

```
(* C# *)
string[] words =
  { "hello", "wonderful", "linq", "beautiful", "world" };

var shortWords =
  from word in words
  where word.Length <= 5
  select word;
```

à la place d'une version plus explicite, comme suit :

```
(* C# *)
var shortWords = words.Where(word => word.Length <= 5).Select(word => word);
```

Le code OCaml correspondant, serait :

```
(* OCaml *)
# let words =
  [ "hello"; "wonderful"; "linq"; "beautiful"; "world" ];;
val words : string list = ["hello"; "wonderful"; "linq"; "beautiful"; "world"]

# let shortWords =
  words
  |> where (fun word -> String.length word <= 5)
  |> select (fun word -> word);;
val shortWords : string list = ["hello"; "linq"; "world"]
```

Ceci dit, reparlons d'évaluation différée ou d'évaluation paresseuse ! En me référant à l'exemple précédent, version (C#), l'idée est qu'après l'exécution du code suivant :

```
(* C# *)
var shortWords =
  from word in words
  where word.Length <= 5
  select word;
```

la variable `shortWords` ne contient pas une liste résultante de la requête (dans ce cas, il s'agirait de la liste contenant les éléments suivants : "hello", "linq", "world"). C'est seulement lorsqu'on utilisera un itérateur pour consulter les éléments de `shortWords` que ces éléments seront réellement calculés (le traitement de la requête sera réellement appliqué à la liste `words`). Donc, ça prend le type de code suivant pour activer les traitements issus de la requête :

```
(* C# *)
foreach (var word in shortWords)
  Console.WriteLine(word);
```

En OCaml :

```
iter (fun word -> print_string (word ^ "\n")) shortWords;
```

PLINQ Maintenant, la beauté de la chose (de la conception de *LINQ* et *PLINQ*) est que le passage vers le parallèle est très simple. Voici les exemples précédents (C#) en «mode» *PLINQ* :

```
(* C# *)
var result =
  list.AsParallel()
  .Where(x => x % 2 == 0)
  .Select(x => x * x);

var shortWords =
  words.AsParallel()
  .Where(word => word.Length <= 5)
  .Select(word => word);
```

Donc, tout ce qu'on a à faire est d'ajouter la mention «*asParallel*» à la collection sur laquelle porte la requête. C'est tout !? Oui, c'est tout : *asParallel* transforme la collection en question, list ou words en me référant aux exemples, en classes implantant *IParallelEnumerate*. On se trouve alors à utiliser d'autres versions des méthodes de classe *Where*, *Select*, etc. Ces versions sont des versions spécialement conçues pour tirer profit du parallélisme. En fait, il est conseillé de toujours mettre *asParallel* dans les requêtes *LINQ/PLINQ* car si votre ordinateur est monoprocesseur, la requête sera alors exécutée comme en mode *LINQ* et si votre machine est multi-coeurs, multi-threads, etc. alors votre requête profitera au maximum de ces ressources. Il n'est pas difficile de constater que la méthode *Where*, ayant une fonction en argument, et ayant à l'appliquer sur tous les éléments d'une collection, peut facilement «distribuer» son traitement sur différentes parties de la collection (même remarque concernant la méthode *Select*). Ceci est vrai d'autant plus que la fonction appliquée aux éléments de la liste n'a pas d'effet de bord (ce qui est toujours le cas en programmation fonctionnelle) ; autrement dit, l'ordre d'exécution n'est plus important.

Pour terminer, voici quelques exemples plus étoffés :

— Exemple 1 :

```
(* C# *)
string[] words =
    { "hello", "wonderful", "linq", "beautiful", "world" };

var groups =
    from word in words
    orderby word ascending
    group word by word.Length into lengthGroups
    orderby lengthGroups.Key descending
    select new { Length=lengthGroups.Key, Words=lengthGroups };

foreach (var group in groups) {
    Console.WriteLine("Words_of_{}_length_{} + group.Length);

    foreach (string word in group.Words)
        Console.WriteLine("{} + word);
}
```

Dans cet exemple, on peut constater l'utilisation étendue de la syntaxe à la *SQL* de *LINQ*. Le seul point nouveau, outre cette utilisation avancée de *LINQ*, est l'«expression» :

```
select new { Length=lengthGroups.Key, Words=lengthGroups }
```

Comme on peut le constater, on peut retourner une collection d'objets créés à la volée. Nous n'avons pas à définir une classe particulière pour représenter les objets utilisés, possiblement, dans le résultat de chaque requête *LINQ* ! Ici, on a la liberté de préciser que nous voulons retourner en résultat une collection d'objets admettant 2 «variables d'instances», *Length* et *Words*, avec un contenu issu de la requête. Si vous avez saisi ce que je viens de mentionner, vous comprendrez alors pourquoi il a fallu ajouter l'inférence de types dans *C#*, à travers la construction syntaxique «*var id = e*» ; en effet, le programmeur ne pourrait spécifier, en me référant à l'exemple précédent, le type de la variable *groups* puisque cette dernière est en réalité composée d'objets dont on ignore la classe dont ils seraient instances.

— Exemple 2 :

```
(* C# *)
var contacts =
    from customer in db.Customers
    where customer.Name.StartsWith("A") && customer.Orders.Count > 0
    orderby customer.Name
    select new { customer.Name, customer.Phone };

var xml =
    new XElement("contacts",
        from contact in contacts
        select new XElement("contact",
            new XAttribute("name", contact.Name),
            new XAttribute("phone", contact.Phone)
        )
    );
```

L'intérêt de cet exemple est double puisqu'on fait une requête sur une table d'une base de données (db.Customers), puis on utilise *LINQ* pour créer un document XML à partir du résultat de la première requête.

Donc, parmi les avantages de *LINQ*, on a :

- Notation uniforme pour interroger différentes sources de données (bases de données, collections en mémoire, documents XML, etc.).
- Par rapport au vrai *SQL*, on bénéficie dans l'écriture de nos requêtes de toutes les classes prédéfinies dans .NET, de toutes les fonctions et opérateurs définis dans *C#*, etc.
- Finalement, tout le traitement est en réalité congelé : on ne fait que manipuler des flots de données (page 113); seuls les éléments qui seront effectivement utilisés seront décongelés.

3.2.1 C# : un langage fonctionnel ?

Si on reprend l'exemple suivant :

```
var contacts =
    from customer in db.Customers
    where customer.Name.StartsWith("A") && customer.Orders.Count > 0
    orderby customer.Name
    select new { customer.Name, customer.Phone };
```

on peut remarquer tout ce qu'il a fallu ajouter à *C#* pour arriver à *LINQ* et *PLINQ* :

- Inférence de types (je viens de le motiver dans la page précédente).
- Fonctions anonymes (qu'on appelle des lambda-expressions) pour avoir la possibilité d'utiliser des fonctions en argument ou en résultat.

C#	OCaml
<code>x => x + 1</code>	<code>fun x -> x + 1</code>
<code>(int x) => x + 1</code>	<code>fun (x:int) -> x + 1</code>
<code>(x, y) => x * y</code>	<code>fun (x,y) -> x * y</code>
<code>() => 1</code>	<code>fun () -> 1</code>
<code>() => Console.WriteLine()</code>	<code>fun () -> print_string "\n"</code>

- Ce que *C#* appelle les types anonymes. Dans cet exemple, il s'agit de créer des objets à la volée dont le type est anonyme (la classe dont sont instances ces objets n'est pas connue par le programmeur). En effet, dans l'expression

```
new { customer.Name, customer.Phone }
```

on voit clairement que le mot-clé «new» n'est pas appliquée à une classe particulière.

En OCaml, il est très simple de générer une telle paire de valeurs !

```
( customer.Name, customer.Phone )
```

En ce qui concerne l'exemple de la variable `groups` (voir ci-dessus) :

```
new { Length=lengthGroups.Key, Words=lengthGroups }
```

En OCaml, on n'a juste qu'à retourner un enregistrement !

```
{ Length=lengthGroups.Key, Words=lengthGroups }
```

C'est donc quasiment la même syntaxe !

3.2.2 C# : un langage définitivement fonctionnel !

Pour parler plus technique à propos de C#, voici la définition de la méthode de classe `Where` :

```
(* C# *)
public static IEnumerable<A> Where<A>(this IEnumerable<A> source,
                                     Func<A, Boolean> predicate)
{
    foreach (A element in source)
    {
        if (predicate(element))
            yield return element;
    }
}
```

On voit clairement que C# a fait beaucoup de progrès au niveau de l'utilisation de la généricité (comparativement à la notion de template héritée du C++). La méthode est générique (type polymorphe `A` pour `alpha`) ; elle prend comme argument une collection de valeurs de type `A` et un prédicat (fonction) de type `A -> Boolean` (`alpha -> bool`) ; la méthode retourne une collection de valeurs de même type `A` (à ce niveau, on peut faire le parallèle avec le type de la fonction `filter`: `('a -> bool) -> 'a list -> 'a list`).

Aussi, on remarque l'utilisation du mot-clé «**yield**» ; c'est lui qui est la cause de l'évaluation paresseuse des requêtes *LINQ*. Prenez un bon livre sur C# (2.0 et +) et lisez les sections concernant les itérateurs et l'utilisation de ce mot-clé.

Donc, encore une fois, la définition de fonctions génériques ayant comme arguments d'autres fonctions, possiblement génériques, est possible en C#.

Pour terminer, voici la définition de «memoize» (page 100) en C# !

```
(* C# *)
Func<A, B> Memoize<A, B>(Func<A, B> func) {
    var cache = new Dictionary<A, B>();
    return x => {
        B value;
        if (cache.TryGetValue(x, out value))
            return value;
        else {
            value = func(x);
            cache.Add(x, value);
            return value;
        }
    };
}
```

Remarquez le typage explicite de la fonction polymorphe (mais on a effectivement que la fonction a le type `A -> B` (`alpha -> beta`) et que le résultat sera une fonction du même type).

Remarquez aussi qu'on retourne comme résultat une fonction (`return x => ...`) créée à la volée comme nous l'avons fait en OCaml (`fun x -> ...`).

Deuxième partie

Programmation orientée objet

Chapitre 4

Fondements de la programmation orientée objet

Les principes de la programmation orientée objet sont nés de deux tendances relativement anciennes :

- D’une part, la difficulté d’exploiter des programmes de taille importante a imposé la notion de programmation structurée. Dans cette approche, l’analyse du problème privilégie les traitements (décomposition de l’application en sous-programmes) et néglige l’aspect données.
- D’autre part la manipulation de volumes d’informations importants a mis en évidence la nécessité de structurer et de regrouper en entités les ensembles de données partageant les mêmes caractéristiques. Ceci a donné naissance à la programmation dirigée par les données (les données orientent l’analyse et influencent la structure des programmes).

L’usage de l’une ou de l’autre de ces deux approches dans la conception de grands systèmes s’est vite révélé largement problématique. Une décomposition orientée purement programme (procédure) ne permet pas de prendre assez en compte les complexités inhérentes aux structures de données et vice-versa. Les grands systèmes nécessitent une décomposition à la fois selon les données et les sous-programmes.

4.1 Objet

L’objet est l’entité élémentaire du paradigme orienté objet. C’est une entité autonome capable d’appliquer un certain nombre de fonctions sur des données qui lui sont propres. Les objets sont simplement des données encapsulées avec leur comportements. Il intègrent les aspects statiques (données) et dynamiques (procédures ou méthodes). Cela illustre un concept fondamental de la programmation orientée objet qui est l’encapsulation.

4.2 Encapsulation

Le principe de l’encapsulation consiste à décrire le comportement d’un objet par l’interface qu’il présente au reste du système. Cette interface se résume à un ensemble de méthodes par lesquelles il est possible d’agir sur l’objet en question. L’idée est donc de protéger l’information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet.

Par ailleurs, cette interface est totalement indépendante de la représentation interne de l’objet. On peut donc changer l’implémentation des méthodes sans changer le comportement extérieur de l’objet. On sépare ainsi la spécification du comportement d’un objet, de l’implémentation pratique de ces spécifications.

4.3 Classe

La classe est un modèle qui décrit la structure statique (variables) et le comportement dynamique (méthodes) communs à un ensemble d'objets créés à partir de ce modèle. Les variables représentent la composante statique (ce sont les données décrivant les caractéristiques communes des objets de la classes). Les méthodes constituent la composante dynamique : ce sont les procédures décrivant les comportements communs.

Notons que dans plusieurs langages orientés objets, à l'aide de modificateurs d'accès, on a la possibilité de violer partiellement ou totalement la propriété d'encapsulation ; ainsi, il est possible d'accéder aux informations décrivant l'état d'un objet :

- *public* : les attributs publics sont accessibles à tous ;
- *protégé* : les attributs protégés sont accessibles seulement à la classe elle-même et aux classes dérivées ;
- *privé* : les attributs privés sont accessibles seulement par la classe elle-même.

4.4 Instanciation

La classe est l'entité conceptuelle qui décrit les objets. Sa définition sert en quelque sorte de moule pour construire ses représentants physiques dits instances. Une instance est donc un objet créé conformément au modèle décrit par sa classe. Une instance possède donc les mêmes variables et les mêmes méthodes que les autres instances de sa classe. Toutefois, les valeurs de ses variables caractérisent l'état propre de cette instance.

4.5 Héritage

Le concept d'héritage est fondamental dans le modèle orienté objet. Il permet la définition d'une classe par une spécialisation d'une autre classe. Cette spécialisation constitue des affinements de la classe spécialisée et ce via des ajouts d'autres variables et d'autres méthodes. La classe définie est dite sous-classe de la classe spécialisée dite super-classe.

Une sous-classe hérite de l'interface d'accès de sa super-classe ainsi que de la définition de l'implémentation de l'état de ses instances. Une sous-classe peut ajouter ou surcharger (overloading) des méthodes de l'interface qu'elle a héritée. Dans la plupart des langages orientés objet, les classes sont organisées hiérarchiquement en une arborescence. La racine de cette arborescence correspond à la classe générale (pré-définie) à partir de laquelle toutes les autres classes sont issues. La structure hiérarchique des classes et des sous-classes constitue l'arbre d'héritage. Il est possible, pour certains langages orientés objet, pour une classe d'hériter de plusieurs autres classes. C'est ce que l'on appelle communément l'héritage multiple. L'héritage multiple permet de fusionner dans une classe les propriétés de plusieurs classes existantes du système.

4.6 Transmission de message

Le mécanisme de transmission de message assure l'interaction entre les objets par des envois de messages. L'adressage d'une requête (message) à un objet provoque l'activation d'une méthode de son destinataire. Un envoi de message mentionne toujours :

- Le receveur du message.
- Le nom de la méthode à exécuter.
- Les arguments de la méthode à exécuter.

Les langages orientés objet supportent en général deux modes de transmission de message :

- Transmission avec retour : Après exécution de la méthode avec les arguments passés, le résultat est retourné à l'expéditeur du message. On parle alors de transmission avec retour.

- Transmission avec continuation : Dans ce mode de transmission, le résultat est envoyé à un autre objet dit continuation.

À la réception d'un message, l'objet récepteur recherche le sélecteur (nom de la méthode) dans un dictionnaire de méthodes, puis procède à l'activation de la méthode associée. Si le sélecteur ne figure pas dans le dictionnaire des méthodes du récepteur, la recherche est alors aiguillée vers les dictionnaires des classes supérieures en remontant l'arbre d'héritage. Ainsi un programme vu sous l'angle programmation orientée objets, est considéré comme un ensemble d'objets qui interagissent entre eux par envois de messages.

4.7 Exemple

Ci-dessous, nous présentons une classe qui modélise une pile dans une syntaxe d'un noyau orienté objet avec un sucre syntaxique proche du langage SmallTalk. Dans cet exemple, un objet pile est modélisée a deux variables d'instances : **elements** (un tableau contenant les éléments de la pile) et **top** (un entier qui pointe vers l'indice de l'élément du tableau **elements** qui est sensé être le sommet de la pile). La classe contient aussi des méthodes d'instances pour dépiler, empiler et initialiser la pile.

```

Class Stack
  Instance variables
    elements
    top
  instance methods
    pop
      | temp |
      temp := elements at: top.
      top top - 1.
      ^temp
    push: anElement
      top := top + 1.
      elements at: top put: anElement
    initialize
      elements := Array new: 99.
      top := 0

| s |
s Stack new.
s initialize.
s push: 'SmallTalk'.
s push: 80.
```

4.8 Langages orientés objet

Le premier langage ayant proposé une technique orienté objet fût Simula en mariant les deux approches données et procédures. La dernière version de Simula 67 reprend et clarifie les concepts de classe et d'objet introduits dans la première version qui a été développé comme une extension du langage Algol 60. Une classe encapsule une structure de données et un ensemble de procédures permettant de les manipuler. C'est une entité générique dont les représentants, les objets, sont créés dynamiquement. Cette philosophie remettait en cause la séparation qui existait entre données et programmes. Elle fût exploitée plus tard dans la conception des langages à types abstraits d'où sont issues Clu et Ada. A l'époque, ces concepts étaient surtout exploités pour la simulation ; leurs

intérêt pour la programmation des systèmes avait été peu perçu. C'est seulement en 1972 qu'est apparue la première version du langage SmallTalk qui reprenait ces idées. SmallTalk a généralisé la notion d'objet qui devient plus tard l'entité unique de son univers (modèle uniforme). SmallTalk 72 introduit l'interaction via des messages. La notion d'héritage par hiérarchie de classes apparaît dans SmallTalk 76. SmallTalk 80 a défini une norme et a uniformisé le modèle en perfectionnant la définition de la classe par l'introduction de la notion de méta-classe. Depuis, la vogue des langages de manipulation d'objet s'est répandue et de très nombreux outils faisant explicitement référence aux mêmes concepts sont disponibles ou apparaissent actuellement. Sans vouloir être exhaustif, on peut citer un certain nombre : C++, Objective-C, Eiffel, Java, Objective Caml, Python, Ruby, Scala, etc.

Simula : Le langage Simula (O.J Dahl) fût le premier langage à introduire les notions de classe et d'objet. Il a été développé comme une extension du langage Algol 60 à partir de 1963 au Norwegian Computer Center (Oslo) ; et a été conçu pour le développement des systèmes de simulation. La seconde version de Simula 67, formalise et généralise les nouveaux concepts introduits par la précédente.

SmallTalk : Le langage SmallTalk (Alain Kay) est le père spirituel des langages orientés objets modernes et le digne héritier de Simula et Lisp. SmallTalk est plus qu'un langage de programmation, il est également synonyme d'environnement de programmation.

C++ : Le langage C++ (B. Stroustrup) ajoute au langage C les notions de classes hiérarchisées et d'héritage. Presque entièrement compatible avec C, il est par ailleurs très influencé par Simula 67 et Algol 68. Dans C++, l'entité fondamentale est la classe plutôt que l'objet. La possibilité d'organiser les classes hiérarchiquement en bénéficiant d'un mécanisme d'héritage permet de programmer de manière orientée objet dans une très large mesure.

4.9 Évaluation

Modularité :

La modularité est assurée par définition même du modèle à objet qui consiste à concevoir et à réaliser une application en termes d'objets interagissant entre eux par des envois de messages. Ceci est mis en oeuvre par un découpage de l'application en modules (classes) et en entités plus fines (objets). Chaque nouvelle définition d'une classe peut être réalisée d'une manière indépendante.

Ainsi la programmation est maintenant le fait de concevoir ou de construire de nouvelles classes ou des composants logiciels qui sont purement des extensions de l'ensemble des composants du système.

Incrémentalité :

Elle relève de la nature même des langages objets. Ces derniers se prêtent donc très bien à la mise au point progressive de maquettes (prototypes) par petites touches successives de programmation. Cette faculté d'incrémentalité est d'autant plus accrue par la dynamique des associations (message-méthodes).

Réutilisation :

Un autre avantage des langages objets est la réutilisation qui fait qu'une application complexe peut être construite à partir des composants logiciels disponibles "en rayon". Par exemple les capacités génériques développées pour une première application, peuvent être réutilisées pour la mise en oeuvre d'une autre application.

Factorisation et économie du texte source :

La structure d'héritage permet une factorisation de la partie commune de la définition de plusieurs classes. Cette partie commune étant implémentée par une classe qui est la super-classe de ces dernières. Cette factorisation est intéressante dans la mesure où elle induit une économie du texte source. Ceci constitue plutôt un cheminement ascendant de la conception.

4.10 Conclusion

Les concepts fondamentaux, de la programmation par objets se résument aux principes de décomposition hiérarchique, d'encapsulation et de relation d'héritage. La décomposition hiérarchique permet le découpage d'une application en un ensemble de parties fonctionnellement indépendantes et à définir leur interaction. Les langages à objets présentent un net progrès par rapport aux langages modulaires qui permettent la définition des modules mais pas leur création dynamique. L'encapsulation spécifie que les objets sont simplement des données encapsulées avec les méthodes correspondantes à leurs manipulations. Le comportement de l'objet est alors décrit par l'interface extérieure qu'il présente au reste du système. Cette interface se résume à un ensemble de message auxquels l'objet est apte à répondre, et est indépendante de la structure interne de l'objet. Ceci procure donc une protection des données de l'objet en interdisant tout accès "sauvage". L'accès est coordonné par l'emploi de messages de l'interface. De plus, l'encapsulation procure à ces langages (à objets) une souplesse supérieure à celle des langages modulaires grâce à la possibilité d'association dynamique "message-méthode" et permet la co-existence de réalisations multiples d'une même interface. La relation d'héritage permet la définition d'une classe d'objets par une spécialisation d'une autre classe (affinement). La classe ainsi définie est alors une sous-classe de la classe spécialisée et hérite de l'interface d'accès de sa super-classe. D'autres concepts moins fondamentaux pour le modèle mais néanmoins très riches ont été introduits dans la programmation par objets telle que la généricité qui permet de paramétrer un module, un type, ou une classe avec un autre type.

Chapitre 5

Objective Caml

Ce chapitre présente le langage **Objective Caml** qui est une extension du langage **Caml** présenté dans le chapitre 2. L'ajout du concept orienté objet apporte une plus grande expressivité au langage. De plus, étant défini à partir de **Caml**, il hérite des principales caractéristiques de ce dernier, à savoir :

- langage fonctionnel et fortement typé ;
- fonctions d'ordre supérieurs ;
- polymorphisme ;
- définition de nouveaux types ainsi que de types abstraits ;
- notion de modules.

Objective Caml offre en plus la plupart des caractéristiques d'un langage orienté objet, parmi lesquelles :

- polymorphisme au niveau des classes paramétrées ;
- héritage multiple et sous-typage ;
- encapsulation ;
- traitement des méthodes binaires.

Par ailleurs, **Objective Caml** reste compatible avec **Caml**, c'est-à-dire que toute expression **Caml** garde la même évaluation statique et dynamique en **Objective Caml**.

Le reste du chapitre est organisé comme suit : La première section présente la définition et l'interaction avec les objets en **OCaml**. Par la suite, nous présentons les classes et leurs caractéristiques (héritage, etc.). Puis, nous présentons quelques notions avancées du langage **OCaml** avant de conclure.

5.1 Objets

Contrairement à plusieurs langages de programmation orientés objet, **OCaml** permet de définir des objets sans faire référence à la notion de classe. Ces objets sont appelés «objets immédiats». La définition d'un objet consiste à encadrer un ensemble de définitions (variables d'instances et méthodes) par les mots-clés **object** et **end** :

- Les variables d'instances sont privées par défaut. Elles sont définies par le mot-clé **val**, ou **val mutable** lorsqu'elles sont modifiables.
- Les méthodes sont définies par le mot-clé **method**. Notez qu'il est possible de définir des méthodes privées par le mot-clé **method private**.

Bien sûr, **OCaml** infère automatiquement le type d'un objet. Ce type correspond à l'interface de cet objet, c'est-à-dire à l'ensemble des signatures de ces méthodes (non privées). Ce type a la syntaxe suivante (où τ_i est la signature de la méthode m_i) :

$$\langle m_1 : \tau_1, \dots, m_n : \tau_n \rangle$$

Voici un premier exemple de définition d'un objet immédiat :

```
# object
  val mutable x = 0
  method getx = x
  method move d = x <- x + d
end;;
- : < move : int -> unit; getx : int > = <obj>
```

Cet objet est composée d'une variable d'instance x (valeur entière) et de deux méthodes : «getx» et «move». Le mot-clé **mutable** permet de déclarer des variables d'instances dont le contenu peut être modifié physiquement par les méthodes. Cette modification se fait à l'aide de l'opérateur «<-» comme l'illustre le corps de la méthode «move». La variable d'instance constitue la partie privée de l'objet : elle n'est pas accessible de l'extérieur de l'objet. Alors que les méthodes désignent la partie publique ou l'interface de l'objet.

Il est aussi possible de définir des méthodes privées qui n'apparaîtront pas dans le type inféré de l'objet :

```
# object val x = 0 method private name = "poo" method getx = x end;;
- : < getx : int > = <obj>
```

Voici deux exemples de définitions d'objets immédiats qui admettent une interface vide :

```
# object val x = 0 method private name = "poo" end;;
- : < > = <obj>

# object end;;
- : < > = <obj>
```

5.1.1 Encapsulation

Comme mentionné précédemment, pour typer un objet, Objective Caml ne considère que les types des méthodes de l'objet. Ceci découle de la propriété d'encapsulation issue du paradigme orienté objet. Cette propriété consiste à interdire l'accès aux variables d'instances à partir de l'extérieur. Pour accéder à ces variables, l'objet offre des méthodes telles que «getx». Cette propriété assure aux programmes une modularité importante. En effet, grâce à elle, il est possible de changer l'implémentation d'un objet (choix des structures de données) sans affecter le reste du système.

5.1.2 Valeurs de première classe

En OCaml, comme pour la plupart des langages de programmation orientés objets, les objets sont de première classe. L'accès à un objet (ses méthodes publiques) se fait par l'intermédiaire de l'opérateur «#» :

$$\text{objet}\#m_i$$

Il est bien sûr possible de lier un objet à un identificateur à l'aide du mot-clé **let** :

```
# let o1 = object val x = 0 method getx = x end;;
val o1 : < getx : int > = <obj>

# o1#getx;;
- : int = 0

# o1#x;;
This expression has type < getx : int >
It has no method x
```


Étant de première classe, il est possible de créer des listes d'objets, de les passer comme arguments, ou de les retourner comme résultats de fonctions :

```
# [o1; o1];;
- : < getx : int > list = [<obj>; <obj>]

# fst;;
- : 'a * 'b -> 'a = <fun>

# fst (o1, o1);;
- : < getx : int > = <obj>
```

Voici un exemple de fonction qui retourne comme résultat des objets créés à la volée (objets immédiats) :

```
# let minmax x y =
  if x < y
  then object method min = x method max = y end
  else object method min = y method max = x end;;
val minmax : 'a -> 'a -> < max : 'a; min : 'a > = <fun>
```

On remarque que l'inférence du type le plus général ('a) est étendue naturellement à l'aspect orienté objet d'OCaml.

5.1.3 Propriétés fonctionnelles

Les variables d'instances et les méthodes sont définies par des expressions du langage. Par conséquent, il n'y a aucune limite quant aux types d'expressions que l'on peut utiliser dans ce contexte (on récupère toute la puissance expressive de l'aspect fonctionnel du langage OCaml) :

```
# let o = object
  val mutable incr = fun x -> x + 1
  method setIncr incr' = incr <- incr'
  method doIt = fun x -> incr x
end;;
- : < doIt : int -> int; setIncr : (int -> int) -> unit > = <obj>

# o#doIt (-5);;
- : int = -4

# o#setIncr (fun x -> abs(x) + 1);;
- : unit = ()

# o#doIt (-5);;
- : int = 6
```

Par ailleurs, en combinant les aspects fonctionnels et objets du langage, il est possible de définir une fonction qui retourne des objets ayant le même type (même interface) :

```
# let new_point init = object
    val mutable x = init
    method getx = x
    method move d = x <- x + d
end;;
new_point : int -> < getx : int; move : int -> int > = <fun>

# let p1 = new_point 4;;
val p1 : < getx : int; move : int -> unit > = <obj>

# let p2 = new_point 7;;
val p2 : < getx : int; move : int -> unit > = <obj>

# (p1#getx , p2#getx );;
- : int * int = (4, 7)
```

5.1.4 Références mutuelles

Comme pour la plupart des langages de programmation orientés objets, Il est possible aux méthodes d'un objet de s'invoquer mutuellement à travers une «référence» vers l'objet lui-même :

```
# let p1 =
  object(self)
    val mutable x = 0
    method move_one = self#move 1
    method move d = x <- x + d
  end;;
val p1 : < move : int -> unit; move_one : unit > = <obj>

# let p1' =
  object(moi)
    val mutable x = 0
    method move_one = moi#move 1
    method move d = x <- x + d
  end;;
val p1' : < move : int -> unit; move_one : unit > = <obj>
```

Cependant, comme décrit dans l'exemple précédent, il n'y a pas de variable prédéfinie dans le langage, variable `self`, pour référer l'objet courant. Le programmeur peut utiliser le nom de son choix (ex : `moi`).

5.1.5 Typage explicite

À l'aide du mot-clé **type**, il est possible de définir des abréviations pour les types objets. On peut donc typer explicitement une variable liée à un objet :

```
# type point_t = < getx: int; move: int -> unit >;
type point_t = < getx: int; move: int -> unit >

# let p1 : point_t =
  object
    val mutable x = 0
    method getx = x
    method move d = x <- x + d
  end;;
val p1 : point_t = <obj>
```

Ce typage explicite permet, par exemple, d'implanter l'objet «point» différemment tout en assurant que son type (interface) soit identique à celui défini dans l'exemple précédent :

```
# let p2 = object
    val mutable a = 0
    val mutable x = 0
    method getx = x
    method move d = x <- x + d; a <- a + 1
end;;
val p2 : < move : int -> unit; getx : int > = <obj>
```

On peut alors considérer l'objet p2 de même type que p1 :

```
# let p3 : point_t = p2;;
val p3 : point_t = <obj>

# [p1; p2; p3];;
- : point_t list = [<obj>; <obj>; <obj>]
```

Cependant, il n'est pas possible d'utiliser le typage explicite dans un contexte où les objets ne sont pas de même type (même interface) :

```
# type reduced_point_t = < getx : int >;
type reduced_point_t = < getx : int >

# let p3' : reduced_point_t = p2;;
This expression has type < getx : int; move : int -> unit >
but is here used with type reduced_point_t
Only the first object type has a method move
```

Pour terminer, l'utilisation du typage explicite permet la définition de la fonction «new_point» (section 5.1.3) de manière à se rapprocher davantage de la notion de constructeurs d'objets de même type (classes) :

```
# let new_point' init : point_t = object
    val mutable x = init
    method getx = x
    method move d = x <- x + d
end;;

new_point' : int -> point_t = <fun>

# let p1 = new_point' 4;;
val p1 : point_t = <obj>

# let p2 = new_point' 7;;
val p2 : point_t = <obj>

# (p1#getx, p2#getx);;
- : int * int = (4, 7)
```

5.1.6 Transtypage

Il est possible de «forcer» le type d'un objet à être d'un autre type plus général (interface restreinte). La syntaxe à utiliser est comme suit :

```
# let p3' = (p2 :> reduced_point_t);;
val p3' : reduced_point_t = <obj>

# p3'#getx;;
- : int = 7

# p3'#move 1;;
This expression has type reduced_point_t
It has no method move

# p2#move 1;;
- : unit = ()
```

Notons qu'on peut définir des fonctions qui effectuent le transtypage :

```
# [(p1 :> reduced_point_t); p3'];;
- : reduced_point_t list = [<obj>; <obj>]

# let reduce_point (p : point_t) = (p :> reduced_point_t);;
val reduce_point : point_t -> reduced_point_t = <fun>

# [reduce_point p1; p3'];;
- : reduced_point_t list = [<obj>; <obj>]
```

Cependant, on ne peut transtyper vers un type moins général (*downcasting*) :

```
# let p4 = (p3' :> point_t);;
This expression cannot be coerced to type
point_t = <getx : int; move : int -> unit>;
it has type reduced_point_t but is here used with type
<getx : int; move : int -> unit; ..>
Only the second object type has a method move
```

5.1.7 Types ouverts (..)

Objective Caml offre la possibilité de définir des fonctions polymorphes applicables à des objets dont le type est dit «ouvert». Comme les types polymorphes α du λ -calcul, un type ouvert est un type polymorphe étendu aux objets et qui peut être instancié par plusieurs types (interfaces) d'objets différents. Les exemples qui suivent définissent trois fonctions applicables à des objets :

```
# let get o = o#getx + 1;;
val get : <getx : int; ::> -> int = <fun>

# let set o = o#setx;;
val set : <set_x : 'a; ::> -> 'a = <fun>

# let incr' o = set o (get o);;
val incr' : <getx : int; setx : int -> 'a; ::> -> 'a = <fun>
```

La première fonction est applicable à tout objet o ayant au moins une méthode «getx» de type `int`. Cette fonction retourne le successeur de la valeur retournée par l'invocation de cette méthode. De même, la deuxième fonction s'applique à tout objet possédant au moins une méthode nommée «setx» dont le type est `'a`. Le résultat de cette fonction est une valeur de type `'a`. La dernière fonction permet d'incrémenter la valeur entière associée à un objet possédant au moins deux méthodes, «getx» et «setx», de types respectifs `int` et `int -> 'a`.

Prenons l'exemple d'une classe définie comme suit :

```
# let o = object
  val mutable x = 0
  method getx = x
  method setx v = x <- v
  method isZero = x = 0
end;;
val o : < getx : int; isZero : bool; setx : int -> unit > = <obj>

# o#getx;;
- : int = 0

# incr ' o';;
- : unit = ()

# o#getx;;
- : int = 1
```

Voici un autre exemple utilisant un objet défini de manière différente mais admettant les deux méthodes «getx» et «setx» :

```
# let o' = object
  val mutable x = 1
  val mutable y = 1
  method getx = x
  method gety = y
  method setx v = x <- v; x
  method sety v = y <- v; y
end;;
val o' : <getx: int; gety: int; setx: int -> int; sety: int -> int> = <obj>

# o'#getx;;
- : int = 1

# incr ' o';;
- : int = 2

# o'#getx;;
- : int = 2
```

5.1.8 Interaction avec les aspects fonctionnels

L'objet «p1» de type «point_t», défini dans la section 5.1.5, utilise une variable d'instance «x» modifiable. Il est possible de définir cet objet de manière tout à fait fonctionnel, c'est-à-dire sans effet de bord dû à l'utilisation de variables modifiables :

```
# let p1' = object
  val x = 0
  method getx = x
  method move d = x + d
end;;
val p1' : < getx : int; move : int -> int > = <obj>
```

Cependant, dans cette version, la méthode «move» n'a aucun effet sur l'état de l'objet puisqu'elle ne peut changer la valeur de sa variable d'instance «x» (non modifiable).

Il existe une construction syntaxique qui permet de résoudre ce «problème» en retournant une copie de l'objet courant avec la possibilité de préciser de nouvelles valeurs pour ses variables d'instances :

$$\{< var_1 = v_1, \dots, var_n = v_n >\}$$

L'objet «p1» pourrait donc être défini de cette manière :

```
# let pf = object val x = 1
  method getx = x
  method move d = {< x = x + d >}
end;;
val pf : < getx : int; move : int -> 'a > as 'a = <obj>

# pf#getx;;
- : int = 1

# (pf#move 3)#getx;;
- : int = 4

# pf#getx;;
- : int = 1

# let pf' = pf#move 5;;
- val pf' : < getx : int; move : int -> 'a > as 'a = <obj>
```

Dans cette version, chaque appel de la méthode «move» retourne un nouvel objet, créé à partir de l'objet courant, dont la variable d'instance «x» vaut «x + d», ce qui correspond au résultat recherché.

Notons que le type de l'objet «pf» introduit un opérateur de liaison de type **as**. Dans cet exemple, l'opérateur **as** lie la variable de type 'a au type < getx : int; move : int -> 'a >. Il permet de spécifier que le résultat de la méthode «move» et l'objet lui-même ont le même type.

Par ailleurs, les types polymorphes ('a, 'b, etc.), utilisés dans la programmation fonctionnelle, sont étendus aux objets. Ainsi la fonction identité «id», définie en introduction de ce chapitre, peut être appliquée à n'importe quelles valeurs y compris les objets. Voici quelques exemples :

```
# let p1 = object
  val mutable x = 0
  method getx = x
  method move d = x <- x + d
end;;
val p1 : < getx : int; move : int -> unit > = <obj>

# let id x = x;;
val id : 'a -> 'a = <fun>

# id p1;;
- : < getx : int; move : int -> unit > = <obj>

# let p1' = id p1;;
val p1' : < getx : int; move : int -> unit > = <obj>

# p1'#move 8;;
- : unit = ()

# p1'#getx;;
- : int = 8

# p1#getx;;
- : int = 8
```

Comme suggéré dans cet exemple, «p1» et «p1'» référencent le même objet. La fonction «id» ne retourne pas une copie de l'objet. Pour ce faire, il aurait fallu créer une fonction «idCopy» qui retourne une copie de l'objet courant. À cette fin, il faut que l'objet admette une méthode qui retourne une copie de lui-même :

```
# let o1 = object
    val mutable x = 0
    method getx = x
    method move d = x <- x + d
    method copy = {< >}
end;;

val o1 : < copy : 'a; getx : int; move : int -> unit > as 'a = <obj>

# let idCopy x = x#copy;;
val id : < copy : 'a; .. > -> 'a = <fun>

# let o1' = idCopy o1;;
val o1' : < copy : 'a; getx : int; move : int -> unit > as 'a = <obj>

# o1'#move 8;;
- : unit = ()

# o1'#getx;;
- : int = 8

# o1#getx;;
- : int = 0
```

Notons qu'OCaml offre une fonction «copy», définie dans le module «Oo», permettant de retourner une copie de l'objet en paramètre. Par conséquent, il aurait été possible d'utiliser cette fonction dans la définition de la fonction «idCopy» sans imposer aux objets en arguments de disposer de méthode «copy» :

```
# let p1 = object
    val mutable x = 0
    method getx = x
    method move d = x <- x + d
end;;

val p1 : < getx : int; move : int -> unit > = <obj>

# Oo.copy;;
- : (< .. > as 'a) -> 'a = <fun>

# let id' x = Oo.copy x;;
val id' : (< .. > as 'a) -> 'a = <fun>

# let p1' = id' p1;;
val p1' : < getx : int; move : int -> unit > = <obj>

# p1'#move 8;;
- : unit = ()

# p1'#getx;;
- : int = 8

# p1#getx;;
- : int = 0
```

Le type de la fonction «Oo.copy» indique que l'argument de cette fonction et son résultat ont le même type, le type ouvert `< .. >`. Ce type représente l'ensemble des objets, sans aucune restriction sur le nombre, les noms et les signatures des méthodes. En effet, si on considère le type moins général `< getx : int; .. >`, il représente l'ensemble des objets possédant au moins une méthode «getx» qui retourne une valeur entière.

Pour terminer, vu que la fonction «id'» correspond en fait à la fonction «Oo.copy», il aurait été possible d'utiliser cette dernière à la place de la première :

```
# let p1'' = Oo.copy p1;;
val p1'' : < getx : int; move : int -> unit > = <obj>
```

5.1.9 Objets immédiats : Conclusion

Dans cette section, nous avons présenté un aspect original d'OCaml qui permet de créer et de manipuler des objets immédiats sans faire référence à la notion de classe. Les types ouverts permettent d'étendre le polymorphisme aux objets. La combinaison de l'aspect fonctionnel et orienté objet offre une expressivité non négligeable au langage.

La section qui suit présente les classes en OCaml, classes permettant de créer des objets de même type.

5.2 Classes

5.2.1 Classes = Constructeurs d'objets

L'exemple qui suit définit une classe «point» pour la représentation des points linéaires :

```
# class point =
  object
    val mutable x = 1
    method getx = x
    method move d = x <- x + d
  end;;
class point :
  object val mutable x : int method getx : int
  method move : int -> unit end
```

OCaml infère automatiquement le type de la classe! Même si les variables d'instances apparaissent dans le type inféré, ils ne font pas partie des types des objets construits à partir (instances) de cette classe.

La syntaxe complète de la déclaration d'une classe est plus complexe que celle présentée dans cet exemple. Elle sera complétée tout au long des sections qui suivent.

Pour créer une instance d'une classe, il faut utiliser le mot-clé standard **new**. Une fois l'instance (l'objet) créé, l'invocation de ses méthodes se fait comme illustrée dans la section précédente consacrée aux objets immédiats :

```
# let p = new point;;
val p : point = <obj>

# p#getx;;
- : int = 1

# p#move 3;;
- : unit = ()

# p#getx;;
- : int = 4

# p#x;;
This expression has no method x
```

Le dernier exemple illustre le fait qu'il n'est pas possible d'accéder à une variable d'instance à partir d'un objet instancié.

5.2.2 Types de classes, Types d'objets et Types ouverts

Pour chaque classe définie, outre le type de la classe qu'il infère, OCaml définit un type qui porte le même nom que celui de la classe et qui permet de typer leurs instances. Pour l'exemple précédent (classe «point»), OCaml définit une abréviation de type comme suit :

```
type point = <getx: int; move: int -> unit>;;
```

Aussi, OCaml définit un type ouvert qui porte le même nom que celui de la classe mais précédé par le symbole `#`. Pour l'exemple précédent, OCaml définit un type `#point` correspondant à la version ouverte du type `point`, i.e :

```
type #point = <getx: int; move: int -> unit, ..>;;
```

5.2.3 Classe paramétrée

Il est possible d'attribuer un (ou plusieurs) paramètre(s) à une classe. L'exemple qui suit utilise une λ -abstraction pour atteindre cet objectif :

```
# class point = fun x_init ->
  object
    val mutable x = x_init
    method getx = x
    method move d = x <- x + d
  end;;
class point : int ->
  object val mutable x : int method getx : int
        method move : int -> unit end

# new point;;
- : int -> point = <fun>

# let p = new point 7;;
val p : point = <obj>
```

Le type de la classe «point» précise qu'il s'agit d'une fonction qui prend comme argument un entier et qui retourne comme résultat un objet disposant d'une variable d'instance «x» et de deux méthodes «getx» et «move».

Notons qu'il existe une forme plus simple pour définir une telle classe paramétrée :

```
# class point x_init =
  object
    val mutable x = x_init
    method getx = x
    method move d = x <- x + d
  end;;
class point : int ->
  object val mutable x : int method getx : int
        method move : int -> unit end

# let cf = new point;;
cf : int -> point = <fun>

# [cf; cf];;
- : (int -> point) list = [<fun>; <fun>]

# (fst (cf, cf)) 7;;
- : point = <obj>
```

La valeur initiale de la variable d'instance «x» est obtenue grâce au paramètre «x_init» de la classe. D'autres langages utilisent la notion de constructeur d'objet qui correspond en fait à une méthode particulière de la classe invoquée automatiquement à l'instanciation d'un nouvel objet. Cette méthode contient le code nécessaire pour initialiser les variables d'instances.

Par ailleurs, même si une classe, en OCaml, et dans tout langage de programmation orienté objet, n'est pas une valeur de première classe, notons que lorsque l'on invoque **new** pour créer une instance d'une classe paramétrée mais en omettant de lui fournir son paramètre effectif, le résultat est une fonction qui prend comme paramètre une valeur correspondant à ce paramètre effectif (omis) et qui retourne une instance de la classe initialisée avec ce paramètre. Comme toute fonction en OCaml est une valeur de première classe, il est donc possible, en OCaml, de manipuler indirectement les classes comme des valeurs de première classe.

Pour terminer, il est possible de définir une classe en utilisant la définition d'une autre classe (ce n'est pas de l'héritage) :

```
# class point' x_init = point ((x_init / 10) * 10);;
class point' : int -> point
```

Cet exemple définit une nouvelle classe «point'» à partir d'une classe existante «point». La classe «point'» prend comme argument un entier et retourne des objets de même type que ceux instanciés à partir de la classe «point». La classe «point'» est une classe à part entière qui bénéficie de toutes les propriétés que peut avoir une classe (héritage, etc.). Par exemple, les types *point'* et *#point'*, qui désignent respectivement le type des instances de la classe «point'» et la version ouverte de ce type, sont définis par OCaml (comme pour toute classe définie) et peuvent être utilisés :

```
# let f (x : point') = x;;
val f : point' -> point' = <fun>

# let g (x : #point') = x;;
val g : (#point' as 'a) -> 'a = <fun>
```

Par ailleurs, il est aussi possible de définir une fonction «new_point» à partir de la classe «point» :

```
# let new_point x_init = new point ((x_init / 10) * 10);;
val new_point : int -> point = <fun>
```

Dans ce cas, «new_point» est une fonction, bénéficiant ainsi de toutes les possibilités offertes par une valeur de première classe.

5.2.4 Déclaration de variables locales

En combinant la programmation fonctionnelle et la programmation orientée objet d'OCaml, il est possible de déclarer des variables locales dans la définition d'une classe :

```
# class point' x_init =
  let origin = (x_init / 10) * 10 in
  object
    val mutable x = x_init
    method getx = x
    method get_origin = x - origin
    method move d = x <- x + d
  end;;
class point' : int ->
  object val mutable x : int method getx : int
        method get_origin : int method move : int -> unit end
```

Les déclarations de variables locales, combinées à la notion de fermeture des langages fonctionnels (voir sémantique dynamique du λ -calcul), permettent de définir simplement¹ des variables de classes (variables partagées par toutes les instances d'une classe donnée) :

```
# class point1 =
  let n = ref 0 in
  fun x_init ->
    object
      val mutable x = x_init
      method getx = x
      method getn = !n
      method move d = x <- x + d; n := !n + 1
    end;;
class point1 : int ->
  object val mutable x : int method getx : int
        method getn : int method move : int -> unit end

# let p1 = new point1 7;;
  val p1 : point1 = <obj>

# p1#move 4;;
- : unit = ()

# let p2 = new point1 5;;
  val p2 : point1 = <obj>

# p2#getn;;
- : int = 1

# p2#move 5;;
- : unit = ()

# p1#getn;;
- : int = 2
```

5.2.5 Référence à l'objet courant et méthodes privées

Comme dans tout langage de programmation orienté objet, il est possible de référer les méthodes de l'objet courant à l'aide d'une variable spéciale. Cependant, OCaml se démarque des autres langages en laissant au programmeur le choix du nom de la variable en question :

1. Il n'est pas nécessaire de définir une construction syntaxique (*static*) propre aux variables de classes.

```
# class printable_point x_init =
  object(moi)
    val mutable x = x_init
    method getx = x
    method move d = x <- x + d
    method print = print_int moi#getx
  end;;
class printable_point : int ->
  object val mutable x : int method getx : int
    method move : int -> unit method print : unit end

# let p = new printable_point 7;;
val p : printable_point = <obj>

# p#print;;
7- : unit = ()
```

À l'instar des objets immédiats, il est possible de définir des méthodes privées :

```
# class restricted_point x_init =
  object(self)
    val mutable x = x_init
    method getx = x
    method private move d = x <- x + d
    method jump = self#move 1
  end;;
class restricted_point : int ->
  object val mutable x : int method getx : int
    method jump : unit method getx : int
    method private move : int -> unit end

# let p = new restricted_point 0;;
val p : restricted_point = <obj>

# p#move 10;;
This expression has type restricted_point
It has no method move
```

5.2.6 Initialisation d'objets

OCaml offre la possibilité d'initialiser un objet, une fois créé. Cette initialisation se fait à l'aide du mot-clé **initializer** :

```
# class printable_point x_init =
  let origin = (x_init / 10) * 10 in
  object(s)
    val mutable x = origin
    method getx = x
    method move d = x <- x + d
    method print = print_int s#getx
    initializer print_string "Nouveau_point_";
      s#print; print_newline()
  end;;
class printable_point : int ->
  object val mutable x : int method getx : int
    method move : int -> unit method print : unit end

# let p = new printable_point 17;;
Nouveau point 17
val p : printable_point = <obj>
```

Combinée aux déclarations de variables de classes, il est possible, par exemple, de compter le nombre d'instances créées pour une classe donnée :

```
# class point1 ' =
  let n = ref 0 in
  fun x_init ->
    object
      val mutable x = x_init
      method getx = x
      method getn = !n
      method move d = x <- x + d
      initializer
        n := !n + 1;
        if ( !n) = 1
        then print_string "1er objet <point> défini\n"
        else print_string ((string_of_int ( !n))
                           "ème objet <point> défini\n")
    end;;
class point1 ' : int ->
  object val mutable x : int method getx : int
        method getn : int method move : int -> unit end

# new point1 ' 5;;
1er objet <point> défini
- : point1 ' = <obj>

# new point1 ' 7;;
2ème objet <point> défini
- : point1 ' = <obj>
```

5.2.7 Héritage

Une autre caractéristique fondamentale de l'orienté objet qui est supportée par Objective Caml est l'héritage. Grâce à cette caractéristique, une classe peut être définie à partir d'autres classes, héritant ainsi de leurs variables d'instances et de leurs méthodes. La classe définie s'appelle classe fille et les classe héritées s'appellent classes mère ou super-classes. Il est possible dans la classe fille de redéfinir des variables d'instances ou des méthodes qui ont déjà été définies dans les classes mère. Deux types d'héritage sont supportés :

- Héritage simple,
- Héritage multiple.

Héritage simple

L'exemple qui suit définit la classe «colored_point» à partir de la classe «point» (définie dans la section 5.2.3) :

```
# class colored_point x (c : string) =
  object (self)
    inherit point x
    val c = c
    method color = c
  end;;
class colored_point : int -> string ->
  object val c : string val mutable x : int
        method color : string method getx : int
        method move : int -> unit end
```

En héritant de la classe «point», il n'est pas nécessaire de définir la variable d'instance «x» ou les méthodes «getx» et «move» puisqu'elles sont héritées. Par contre, la variable d'instance «c»

et la méthode «color» sont propres aux points colorés. Par conséquent elles doivent être définies dans la classe «colored_point». Le type résultant de la définition de cette classe spécifie bien que celle-ci comprend deux variables d'instances «x» et «c», ainsi que trois méthodes «getx», «move» et «color». Tout objet instancié à partir de cette classe peut invoquer l'une de ces méthodes :

```
# let p' = new colored_point 5 "rouge";;
val p' : colored_point = <obj>

# p'#getx, p'#color;;
- : int * string = (5, "rouge")
```

Cet exemple illustre le cas d'un héritage simple avec ajout de nouvelles méthodes et de nouvelles variables d'instances. Par ailleurs, il est possible dans la classe fille de redéfinir des méthodes de la classe mère, comme l'illustre la définition de la classe «reduced_point» :

```
# class reduced_point x r0 =
  object(self)
    inherit point x as parent
    val r = r0
    method reduction = r
    method move d = parent#move (d * self#reduction)
  end;;
class reduced_point : int -> int ->
object val r : int val mutable x : int
  method getx : int method reduction : int
  method move : int -> unit end
```

Comme dans l'exemple précédent, la variable d'instance «x» et la méthode «getx» sont héritées. Par contre, la méthode «move» est redéfinie pour traiter le cas particulier des points réduits. Le corps de cette méthode fait appel à deux variables particulières : «self» et «parent».

Comme mentionné précédemment, la variable «self» est définie en la liant à l'objet courant (plus précisément au mot-clé **object**). Elle permet d'invoquer, dans le corps d'une méthode, les autres méthodes de la classe. Dans cet exemple, la méthode «move» fait appel à la méthode «réduction» de la même classe; tandis que la deuxième variable «parent» est liée à la classe «point» héritée (grâce aussi à l'opérateur de liaison **as**). Elle permet d'invoquer des méthodes appartenant à une classe mère et redéfinies dans la classe fille. Dans cet exemple, la méthode «move», définie dans la classe fille, fait appel à la version de cette même méthode définie dans la classe mère liée à la variable «parent».

Les exemples qui suivent illustrent l'effet de ces variables :

```
# let pr = new reduced_point 2;;
val pr : reduced_point = <obj>

# pr#getx;;
- : int = 0

# pr#reduction;;
- : int = 2

# pr#move 5;;
- : unit = ()

# pr#getx;;
- : int = 10
```

En fait, la variable «self» est liée à l'objet qui invoque la méthode dans laquelle elle est définie («move», dans ce cas); Supposons qu'une classe «reduced_point2» soit définie à partir de la classe «reduced_point» comme suit :

```
# class reduced_point2 r0 =
  object(self)
    inherit reduced_point r0
    method reduction = r * 2
  end;;
class reduced_point2 : int ->
  object val r : int val mutable x : int
    method getx : int method reduction : int
    method move : int -> unit end
```

Dans ce cas, pour un objet instance de cette nouvelle classe, l'expression `self#reduction` fait appel à la méthode «reduction» définie dans la classe «reduced_point2» et non celle définie dans la classe «reduced_point». Les exemples qui suivent illustrent ces propos :

```
# let pr2 = new reduced_point2 2;;
val pr2 : reduced_point2 = <obj>

# pr2#getx;;
- : int = 0

# pr2#reduction;;
- : int = 4

# pr2#move 5;;
- : unit = ()

# pr2#getx;;
- : int = 20
```

La valeur de la variable d'instance «v» étant égale à 2, l'invocation de la méthode «reduction» retourne la valeur 4, c'est-à-dire le double de cette variable. L'invocation de la méthode «move» avec le paramètre 5 s'évalue comme suit : Tout d'abord, l'expression `self#reduction` est invoquée. Dans ce cas, l'évaluateur invoque la méthode «reduction» appartenant à l'objet `pr2`, c'est-à-dire celle définie dans la classe «reduced_point2». Le résultat de cette invocation est la valeur entière 4. Puis cette valeur est multipliée par la valeur du paramètre formel «d» valant 5. Le résultat est la valeur 20. La dernière étape consiste à invoquer la méthode «move» héritée de la classe mère (ou «grand-mère») «point». En fait, on invoque la classe «move» définie dans la classe mère liée à la variable «parent», dans ce cas «point». Le résultat de cette invocation modifie la valeur de la variable d'instance «x» qui vaut désormais 20.

Héritage multiple

Objective Caml supporte aussi l'héritage multiple, c'est-à-dire qu'il est possible pour une classe d'être définie à partir de plusieurs autres classes. Dans ce cas, pour désigner une méthode d'une super-classe particulière, il est possible de créer autant de variables liées (telles que «parent») que de classes héritées. Pour décrire un exemple de classe héritant de plusieurs classes, nous définissons une classe «printable_point» comme suit :

```
# class printable_point x0 =
  object(self)
    inherit point x0
    method print = print_int self#getx
  end;;
class printable_point : int ->
  object val mutable x : int method getx : int
    method move : int -> unit
    method print : : unit end
```

Les exemples suivant illustrent le comportement d'un objet instance de cette classe :

```
# let pi = new printable_point 5;;
val pi : printable_point = <obj>

# pi#getx;;
- : int = 5

# pi#print;;
5- : unit = ()

# pi#move 4;;
- : unit = ()

# pi#print;;
9- : unit = ()
```

À partir des classes «printable_point» et «colored_point», nous définissons la classe «printable_colored_point» comme suit :

```
# class printable_colored_point x0 s0 =
  object(self)
    inherit colored_point x0 s0
    inherit printable_point x0 as super
    method print = print_string "(Position_=_" ;
                  super#print ;
                  print_string ",_Couleur_=_" ;
                  print_string (self#color) ;
                  print_string ")\n"
  end;;
class printable_colored_point : int -> string ->
  object val c : string val mutable x : int method getx : int
    method move : int -> unit method color : string
    method print : unit end
```

Dans cette classe, la méthode «print» est redéfinie pour afficher en conséquence des points colorés imprimables. Les autres méthodes sont héritées des classes «colored_point» et «printable_point». Les exemples suivant illustrent le comportement d'un objet instance de cette classe :

```
# let pci = new printable_colored_point 5 "Rouge";;
val pci : printable_colored_point = <obj>

# pci#getx;;
- : int = 5

# pci#move 5;;
- : unit = ()

# pci#color;;
- : string = "Rouge"

# pci#print;;
(Position = 10, Couleur = Rouge)
- : unit = ()
```

La méthode «print» utilise celle portant le même nom et définie dans la classe mère «printable_point» pour afficher la position d'un point.

L'ordre des classes héritées est important. En cas de conflits entre méthodes héritées (mêmes méthodes définies dans les deux super-classes), celles qui sont définies en dernier seront retenues.

Par exemple, si à l'instar de la classe «`printable_point`», la classe «`colored_point`» définissait une méthode «`print`», celle qui est définie dans la classe «`printable_point`» sera héritée.

Cependant, comme l'illustre l'exemple suivant, dans un contexte d'héritage multiple, il est possible d'invoquer une méthode particulière appartenant à une classe héritée. Il suffit de spécifier cette classe à l'aide des variables liées définies par l'opérateur de liaison **as**. Pour présenter cette classe, nous définissons une nouvelle classe «`colored_point2`» possédant une méthode «`print`» :

```
# class colored_point2 x0 c0 =
  object(self)
    inherit colored_point x0 c0
    method print = print_string (self#color)
  end;;
class colored_point2 : int -> string ->
  object val c : string val mutable x : int method getx : int
    method move : int -> unit method color : string
    method print : unit end
```

La définition de la classe «`printable_colored_point2`» est :

```
# class printable_colored_point2 x0 s0 =
  object(self)
    inherit colored_point2 x0 s0 as super_c
    inherit printable_point x0 as super_i
    method print = print_string "(Position_=";
      super_i#print;
      print_string ",_Couleur_=";
      super_c#print;
      print_string ")\n"
  end;;
class printable_colored_point2 : int -> string ->
  object val c : string val mutable x : int method getx : int
    method move : int -> unit method color : string
    method print : unit end
```

L'invocation de la méthode «`print`» produit le même résultat :

```
# let pci2 = new printable_colored_point2 5 "Rouge";;
val pci2 : printable_colored_point2 = <obj>

# pci2#print;;
(Position = 5, Couleur = Rouge)
- : unit = ()
```

Deux remarques s'imposent :

- Le type d'une méthode ou d'une variable d'instance redéfinie ne doit pas changer dans la classe fille. Par exemple, dans la classe «`reduced_point`» qui hérite de la classe «`point`», la méthode «`move`» est redéfinie. Elle garde cependant le même type qu'elle avait dans la classe mère, à savoir `int -> unit`.
- Le type attribué à la variable **self** est le même que celui attribué à une classe fille. La section 5.3.2 qui suit présente un exemple illustrant ces propos.

Ces propriétés assurent au langage le support des méthodes binaires et des méthodes qui retournent **self**.

5.2.8 Objets immédiats et héritage

Pour terminer, mentionnons qu'il est possible, lors de la définition d'un objet immédiat, d'utiliser l'héritage pour bénéficier des variables d'instances et des méthodes qui y sont définies :

```
# let p = object(self)
    inherit point 1
    method print = print_int self#getx
end;;
val p : <getx: int; print: unit; move: int -> unit> = <obj>

# let new_p x0 = object(self)
    inherit point x0
    method print = print_int self#getx
end;;
val new_p: int -> <getx: int; print: unit; move: int -> unit> = <fun>
```

5.3 Aspects orientés objets du langage : Notions avancées

5.3.1 Classes et méthodes virtuelles

Une classe est déclarée virtuelle dès qu'une de ses méthodes l'est aussi (c'est-à-dire que la méthode n'est pas définie; on ne spécifie que sa signature); dès lors, il n'est pas possible d'instancier une telle classe :

```
# class virtual abstract_point x_init =
    object(moi)
        val mutable x = x_init
        method virtual getx : int
        method get_origin = moi#getx - x_init
        method virtual move : int -> unit
    end;;
class virtual abstract_point : int ->
    object val mutable x : int method get_origin : int
        virtual method getx : int
        virtual method move : int -> unit end
```

Voici une classe qui sous-classe la classe «abstract_point» en donnant une définition aux méthodes virtuelles «getx» et «move» :

```
# class point x_init =
    object
        inherit abstract_point x_init
        method getx = x
        method move d = x <- x + d
    end;;
class point : int ->
    object val mutable x : int method get_origin : int
        method getx : int method move : int -> unit end
```

Cette classe peut être instanciée.

5.3.2 Méthodes binaires

Objective Caml supporte les déclarations de méthodes binaires c'est-à-dire des méthodes qui prennent en argument un objet de même type que l'objet courant (*self*). Par exemple, une classe «point_m», qui admet une méthode «distance» calculant la distance entre deux points, pourrait être définie comme suit :

```
# class point_m x0 =
  object(_ : 'a) (* ou (self : 'a) *)
    inherit point x0
    method distance (p : 'a) =
      let x1 = abs x and x2 = abs (p#getx) in
      (max x1 x2) - (min x1 x2)
    end;;
class point_m : int ->
  object ('a) val mutable x : int method getx : int
    method move : int -> unit
    method distance : 'a -> int end
```

Ce qui est particulier dans cette déclaration, c'est la variable de type 'a. En effet, comme la méthode «distance» prend en argument un autre objet instance de la classe «point_m», il est nécessaire que le type de cet argument soit le même que celui de la variable **self**. Le type d'un objet de cette classe est attachée au mot-clé **object**, plus précisément au nom de l'objet courant (**self**, moi, etc.) qui, dans ce cas, est omis (**_**) car non utilisé.

Cet exemple introduit de nouveau les types d'objets récursifs. En effet, l'abréviation engendrée par cette déclaration est :

```
type point_m = <getx : int; move : int -> unit; distance : 'a -> int> as 'a;;
```

L'invocation de la méthode «distance» se fait comme suit :

```
# let pm = new point_m 5;;
val pm : point_m = <obj>

# pm#getx;;
- : int = 5

# let pm' = new point_m 9;;
val pm' : point_m = <obj>

# pm'#getx;;
- : int = 9

# pm#distance pm';;
- : int = 4

# pm'#distance pm;;
- : int = 4
```

Soit la classe «colored_point_m», sous-classe de «point_m», et définie comme suit :

```
# class colored_point_m x0 (s0 : string) =
  object(_ : 'a)
    inherit point_m x0
    inherit colored_point x0 s0
  end;;
class colored_point_m : int -> string ->
  object ('a) val c : string val mutable x : int
    method getx : int method move : int -> unit
    method distance : 'a -> int method color : string end
```

La particularité de cette classe est que la méthode «distance» garde le même type que celui défini dans la classe mère ('a -> int), et celui de **self** est le même que celui de la classe mère. Ceci est conforme au résultat souhaité pour les méthodes binaires. Ainsi, la méthode «distance» est spécialisée en conséquence et peut être appliquée à des objets instances de la classe «colored_point_m» :

```
# let pcm = new colored_point_m 5 "Vert";;
val pcm : colored_point_m = <obj>

# pcm#getx;;
- : int = 5

# let pcm' = new colored_point_m 9 "Rouge";;
val pcm' : colored_point_m = <obj>

# pcm'#getx;;
- : int = 9

# pcm#distance pcm';;
- : int = 4
```

5.3.3 Méthodes retournant self

L'exemple suivant illustre la définition d'une méthode retournant **self** à travers l'utilisation du constructeur d'objet $\{<\dots>\}$:

```
# class duplicate =
  object
    method copy = {< >}
  end;;
class duplicate : object ('a) method copy : 'a end
```

En invoquant la méthode «copy», celle-ci retourne une copie de l'objet en cours. Par exemple :

```
# let d = new duplicate ();;
val d : duplicate = <obj>

# let d' = d#copy;;
val d' : duplicate = <obj>
```

À l'instar de la méthode «distance» de la section précédente, cette méthode admet un type conforme à celui attribué à l'objet **self**. De plus, si cette méthode est héritée, elle s'adapte en conséquence pour copier les objets de la nouvelle classe (classe fille). La classe suivante est définie à partir des classes «point» et «duplicate» :

```
# class duplicate_point x0 =
  object
    inherit duplicate ()
    inherit point x0
  end;;
class duplicate_point : int ->
  object ('a) val mutable x : int method copy : 'a
    method move : int -> unit method getx : int end
```

Un objet instance de cette classe peut invoquer la méthode «copy» pour retourner une copie de lui-même :

```
# let pd = new duplicate_point 6;;
val pd : duplicate_point = <obj>

# pd#getX;;
- : int = 6

# let pd' = pd#copy;;
val pd' : duplicate_point = <obj>

# pd'#getX;;
- : int = 6
```

Dans le type de la classe «duplicate», la variable de type 'a est liée au type de la variable `self`, c'est-à-dire au type de l'objet en cours. Dans le cas de la classe «duplicate», elle désigne n'importe quel objet ayant un type qui filtre le type :

$$\text{rec } \alpha . \langle \text{copy} : \alpha; \dots \rangle$$

en d'autres termes, un objet qui possède une méthode «copy» dont le résultat est un autre objet de même type. Cet objet peut posséder d'autres méthodes : elles sont représentées par les trois points de suspensions.

Notons que les deux types ci-dessous sont différents puisque le premier oblige le résultat de la fonction à être de même type que l'objet en argument :

$$\begin{aligned} \text{Type1} &: \{\langle \dots \rangle\} \text{ as } \alpha \rightarrow \alpha \\ \text{Type2} &: \{\langle \dots \rangle\} \rightarrow \{\langle \dots \rangle\} \end{aligned}$$

Par contre, le deuxième désigne une fonction qui prend comme argument un objet quelconque et qui retourne en résultat un autre objet n'ayant pas forcément le même type que l'argument.

5.3.4 Retour aux aspects fonctionnels

Les 2 exemples qui suivent illustrent la différence entre une copie de l'objet courant et la création d'une instance de la classe courante :

```

# class fonctionnal_point y =
  object
    val x = y
    method getx = x
    method move d = {< x = x + d >}
  end;;
class fonctionnal_point : int ->
  object ('a) val x : int method getx : int
    method move : int -> 'a end

# let p = new fonctionnal_point 7;;
val p : fonctionnal_point = <obj>

# p#getx;;
- : int = 7

# (p#move 3)#getx;;
- : int = 10

# p#getx;;
- : int = 7

# class bad_fonctionnal_point y =
  object
    val x = y
    method getx = x
    method move d = new bad_fonctionnal_point (x+d)
  end;;
class bad_fonctionnal_point : int ->
  object val x : int method getx : int
    method move : int -> bad_fonctionnal_point end

```

Il est clair que dans le premier cas, une classe «C» qui hériterait de la classe «fonctionnal_point» disposera d'une méthode «move» qui retournera une copie de l'objet courant, c'est-à-dire un objet instance de la classe «C»; Par contre, une classe «C'» qui hériterait de la classe «bad_fonctionnal_point» disposera d'une méthode «move» qui retournera toujours une instance de la classe «fonctionnal_point». À la fin de présent chapitre (page 158), un exercice corrigé motive davantage cette différence.

5.3.5 Polymorphisme et sous-typage

Le polymorphisme est réalisé grâce à la notion de sous-typage. Le principe de base associé au sous-typage est la substitution. Si A est un sous-type de B , alors une expression de type A peut être utilisée (sans erreur de typage) dans un contexte qui requiert une expression de type B . Cette relation est notée \leq .

Objective Caml utilise un sous-typage particulier puisqu'il est nécessaire de le spécifier syntaxiquement. L'exemple qui suit définit une liste constituée d'un objet de type «colored_point» et d'un autre de type «reduced_point». L'opérateur «:>» contraint, d'une manière explicite, ces deux objets à être de type «point», obtenant ainsi une liste homogène d'objets de même type :

```

# let points = [(new reduced_point 2 : reduced_point :> point);
  (new colored_point 1 "bleu" : colored_point :> point)];;
val points : point list = [<obj>; <obj>]

```

Quand cela est possible, il n'est pas nécessaire de spécifier le type des objets à transtyper :

```

# let points = [(new reduced_point 2 :> point);
  (new colored_point 1 "bleu" :> point)];;
val points : point list = [<obj>; <obj>]

```

5.3.6 Classe polymorphe (générique)

Une classe est générique dès que l'un de ses éléments (variables d'instances ou méthodes) l'est aussi. Il est alors nécessaire de préciser explicitement les variables de types en jeu :

```
# class ['a] ref x_init =
  object
    val mutable x = (x_init : 'a)
    method get = x
    method set y = x <- y
  end;;
class ['a] ref : 'a ->
  object val mutable x : 'a method get : 'a
    method set : 'a -> unit end

# let r = new ref 1 in r#set 2; (r#get);;
- : int = 2
```

Voici un autre exemple où le système d'inférence de types d'OCaml raffine le type polymorphe 'a utilisé par le programmeur, en spécifiant que ce dernier doit correspondre à un type ouvert d'objets admettant au moins une méthode «move» ayant comme signature `int -> unit` :

```
# class ['a] circle (c : 'a) =
  object
    val mutable center = c
    method center = center
    method set_center c = center <- c
    method move = (center#move : int -> unit)
  end;;
class ['a] circle : 'a ->
  object constraint 'a = < move : int -> unit; .. >
    val mutable center : 'a method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit end
```

5.3.7 Type d'une classe

Il est possible de définir un type de classe en précisant les types de ces variables d'instances et les signatures de ses méthodes :

```
# class type point_t =
  object
    val x : int
    method getx : int
    method move : int -> unit
  end;;
class type point_t =
  object val x : int method getx : int method move : int -> unit end

# class point : point_t =
  object
    val mutable x = 0
    method getx = x
    method move d = x <- x + d
  end;;
class point : point_t
```

La classe «point», étant déclarée de type «point_t», a l'obligation de définir les variables d'instances et les méthodes présents dans le type de classe «point_t», en respectant leur type.

À l'instar des langages C# et Java, on peut considérer une hiérarchie entre les types de classes (ou interfaces) :

```
# class type ['a] point_ext_t =
  object
    inherit point_t
    method reset : 'a -> unit
  end;;
class type ['a] point_ext_t =
  object val x: int method getx: int method move: int -> unit
    method reset: 'a -> unit end
```

5.4 Aspects orientés objets et modules

Il est possible de combiner l'utilisation des modules avec celle des classes. Dans l'exemple qui suit, nous définissons une signature «POINT» qui requiert la définition d'une classe «point» ayant comme argument un entier, ayant comme variables d'instances, une variable modifiable «x» de type entier et ayant comme méthodes, la méthode «move» de type `int -> unit` et la méthode «getx» de type `int` :

```
# module type POINT =
  sig
    class point : int ->
      object
        val mutable x : int
        method getx : int
        method move : int -> unit
      end
  end;;
(* OCaml répond avec un type identique *)
```

L'exemple qui suit définit un module «Point» qui respecte la signature «POINT» :

```
# module Point : POINT =
  struct
    class point x_init =
      object
        val mutable x = x_init
        method getx = x
        method move d = x <- x + d
      end
    end;;
  module Point : POINT
```

Par la suite, il est possible de bénéficier des définitions (classe «point») qui se trouvent dans le module «Point» :

```
# let p = new Point.point 5;;
val p : Point.point = <obj>

# p#getx;;
- : int = 5

# p#move 4;;
- : unit = ()

# open Point;;
# let p' = new point 10;;
val p' : Point.point = <obj>
```

Dans la pratique, un module comprend plus d'une classe ainsi que, par exemple, des définitions de types inductifs, des déclarations d'exceptions, etc.

5.5 Conclusion

L'aspect prédominant dans le langage **Objective Caml** est la simplicité d'intégration des constructions orientées objet avec ceux de **Caml**. De plus, il reste compatible avec les expressions **Caml** existantes, puisqu'à l'instar de **Caml**, l'algorithme d'inférence de types est basé sur l'unification du premier ordre (fonction *mgu* utilisée par la fonction *Infer* étudiée dans la matière consacrée à la sémantique statique du λ -calcul).

La seule difficulté en **Objective Caml** est la nécessité pour l'utilisateur de spécifier des contraintes sur les types afin de les forcer à être d'un autre type. Théoriquement, ces contraintes peuvent être implicites. Il faut alors utiliser les inférences de types basées sur les contraintes.

En résumé, **Objective Caml** traite la plupart des caractéristiques orientées objet :

- Héritage multiple, avec la possibilité de lier chaque classe héritée à une variable (grâce à l'opérateur de liaison **as**). Cette variable permet d'accéder à une méthode en particulier d'une super-classe.
- Le sous-typage est assuré grâce aux contraintes sur les types. En effet, **Objective Caml** permet de gérer des listes d'objets hétérogènes forçant leur types à être similaires (type d'une super-classe commune).
- Encapsulation qui assure l'accès seulement aux méthodes, les données étant privées.

Notons qu'**Objective Caml** offre d'autres fonctionnalités originales, dont :

- objets immédiats ;
- polymorphisme au niveau des objets et des types ouverts ;
- traitement des fonctions retournant **self** ;
- traitement des méthodes binaires ;
- inférence de contraintes sur les types ;
- etc.

Exercices

5.1 Expliquer la différence entre les deux classes suivantes et préciser leur type respectif :

```
class c1 = object(self) method c = Oo.copy self end
class c2 = object(self) method c = new c2 end
```

5.2 Définir une classe «backup» qui comprend 2 méthodes «save» et «restore» qui permettent, pour n'importe quelle sous-classe qui en hérite, de respectivement sauvegarder l'état de l'objet courant et de restaurer son état. Il va de soi qu'il faudra définir une variable d'instance dans laquelle on sauvegardera l'état d'un objet ; on peut utiliser *Oo.copy* ; on peut aussi utiliser le type «'a option» pour distinguer l'état où on dispose d'une copie d'un objet de l'état initial où l'on ne dispose de rien.

5.3 Déterminer le type de l'expression suivante :

```
fun x -> fun y -> if x#leq y then x else y
```

Corrigés

7-1 : Le type de chaque classe permet de bien les différencier :

```
class c1 : object ('a) method c : 'a end
class c2 : object method c : c2 end
```

La différence se situe au niveau de l'héritage et plus précisément au niveau de leurs sous-classes. Soient les classes «sc1» et «sc2» définies comme suit :

```
class sc1 = object inherit c1 method m = () end
class sc2 = object inherit c2 method m = () end
```

Si on crée des instances de ces classes et on invoque la méthode «c» comme suit :

```
((new sc1)#c : sc1);;
((new sc2)#c : sc2);;
```

Dans le premier cas, tout se passe correctement car la méthode retourne une copie de l'objet courant et donc un objet de type sc1. Par contre, dans le deuxième cas, on retourne un objet instance de c2 que l'on ne peut transtyper vers sc2.

7-2 :

```
class backup =
  object(self)
    val mutable backup = None
    method save = backup <- Some (Oo.copy self)
    method restore = match backup with None -> self | Some x -> x
  end;;
```

7-3 : (< leq : 'a -> bool; .. > as 'a) -> 'a -> 'a

Troisième partie

Programmation parallèle et
concurrente

Chapitre 6

Introduction

6.1 Description du parallélisme

Dans la vie courante, les systèmes séquentiels sont peu nombreux. Par contre, la plupart des systèmes sont de nature parallèle et distribuée. Nombreux sont les chercheurs qui se sont penchés sur l'étude du parallélisme. Par conséquent, un large éventail de langages, logiques, modèles et calculs ont été élaborés afin de mieux décrire, étudier et comprendre les problèmes inhérents aux systèmes concurrents et parallèles tels que le non-déterminisme, la communication, et la composition de comportements.

6.2 Distinction entre programmation concurrente, parallèle et distribuée

Les réseaux comprennent un ensemble d'ordinateurs ; chaque ordinateur peut comprendre un ensemble de processeurs ; et chaque processeur comprend désormais plusieurs cœurs. Toutes ces unités de calculs fonctionnent de manière quasi-indépendantes les unes par rapport aux autres, et partagent des ressources (comme la mémoire, par exemple).

Dans ce contexte, étant donnés différents processus à exécuter, on distingue les trois formes de programmation suivantes :

1. La programmation concurrente qui est caractérisée par l'entrelacement entre les exécutions des différents processus ; généralement, tous ces processus partagent une même unité de calcul (par exemple, un cœur), pour leur exécution. À tout moment, grâce à un ordonnanceur (*scheduler*), seul un processus dispose du cœur pour réaliser une partie de son traitement ; cependant, en tant qu'utilisateur externe au système, on a l'impression que les différents processus s'exécutent en même temps, en parallèle, alors qu'ils s'exécutent en réalité de manière concurrente.
2. La programmation parallèle qui est caractérisée par le fait que tous les processus sont exécutés en même temps, chacun disposant de sa propre unité de calcul ; si le nombre de processus à exécuter dépasse le nombre d'unités de calcul disponibles, une partie des processus s'exécutera de manière parallèle, alors qu'une autre devra se partager certaines unités de calcul. La programmation concurrente et parallèle se trouvent alors entremêlées.
3. La programmation distribuée qui est caractérisée par le fait que les différentes unités de calcul peuvent se situer sur un réseau, voire sur plusieurs réseaux d'ordinateurs. Ce type de programmation implique que la communication inter-processus est limitée.

6.3 Langages de programmation concurrente et parallèle

Un grand nombre de langages de programmation offrent aujourd’hui des primitives permettant de programmer les systèmes concurrents et parallèles (ou distribués). Nous pouvons distinguer deux catégories de langages selon le mode de communication entre processus :

- Communication par mémoire partagée.
- Communication par passage de messages.

Examinons de plus près chacun de ces critères.

Communication par mémoire partagée

Dans cette catégorie, les processus utilisent des variables partagées à des fins de communication. Le problème crucial dans cette approche est de prévenir toute interférence entre processus. Ce problème dit de sections critiques a été amplement traité dans la littérature et un éventail de solutions ont été proposées. Celles-ci consistent en des mécanismes de synchronisation de contrôle qui permettent aux processus d’entrer et de sortir de manière saine dans leurs sections critiques.

Communication par passage de messages

Dans cette catégorie de langages, la communication et la synchronisation sont réalisées au moyen de deux primitives de base qui sont l’envoi et la réception d’un message. Divers modes de passage de messages peuvent être répertoriés selon la manière dont l’émetteur et le récepteur sont désignés. La méthode la plus simple consiste à utiliser l’identificateur d’un processus afin de lui adresser un message. Une manière plus générale de procéder consiste à introduire des ports de communication dits également canaux. Un processus peut alors utiliser un canal auquel il a accès afin de recevoir ou d’envoyer un message. La création des canaux peut être statique (canaux alloués avant l’exécution du programme) ou dynamique (canaux créés à la volée, pendant l’exécution du programme). Nous pouvons distinguer trois méthodes d’émission de messages : émission bloquante (passage de messages synchrone), émission non bloquante (passage de messages asynchrone) et l’émission-réception (passage de messages par requête-réponse). Nous parlons d’un passage de messages uni-directionnel dans les deux premiers cas, et de bi-directionnel dans le dernier cas.

- Le passage asynchrone de messages est caractérisé par une émission non bloquante et par des messages tamponnés (bufférisés). Très fréquent dans les systèmes distribués, il est également adopté dans les langages d’acteurs comme Erlang.
- Le passage synchrone de messages est caractérisé par une émission et une réception bloquantes. Une telle communication est généralement subdivisée en deux parties. La première partie consiste en une synchronisation sur deux actions complémentaires (émission-réception ou vice-versa) i.e. rendez-vous. Ce dernier est généralement suivi, dans un deuxième temps, par un passage de valeur. Ce mode de passage est présent dans les langages tels que Concurrent ML, Ocaml et Go.
- Le passage de messages par requête-réponse est connu dans la littérature anglophone sous le nom de «*Remote Procedure Call (RPC)*». Une communication dans ce mode s’effectue par un appel et un retour de procédure. L’appel consiste en un message requête adressé par le client au serveur. Le retour de procédure correspond alors à la réponse du serveur. Notons que ce mode de communication peut être implémenté au moyen d’un passage synchrone ou asynchrone de messages. Ce mode de communication est adopté dans des langages tels que Concurrent C.

Chapitre 7

OCaml parallèle et concurrent

7.1 Introduction

L'idée d'unifier les paradigmes de programmation fonctionnelle (impérative, générique et modulaire), orienté objets et concurrente a suscité un grand intérêt aussi bien chez les praticiens que chez les théoriciens de l'informatique. Auparavant, ces paradigmes ont fait l'objet d'une grande investigation. Ainsi, plusieurs langages, modèles et calculs ont été proposés. Cependant, nous réalisons que, le plus souvent, chacun de ces paradigmes a été étudié de manière indépendante.

Dans ce chapitre, nous poursuivons la présentation du langage OCaml qui intègre ces différents paradigmes. Plus précisément, dans ce chapitre, nous présentons les concepts liés au langage Concurrent ML (CML) à travers leur implantation en OCaml (principaux modules : *Thread*, *Mutex*, *Condition* et *Event*). Il convient de signaler qu'à l'origine, CML a été implanté en Standard ML.

Le choix de CML est motivé par le fait qu'il combine à la fois concision syntaxique et expressivité sémantique. En effet, CML est un langage possédant un nombre restreint de constructions syntaxiques. Il peut être perçu comme un λ -calcul augmenté par des extensions impératives et concurrentes.

Ce langage est fondé sur l'idée de plonger une algèbre de processus dans un langage fonctionnel et impératif, voire orienté objets (OCaml). CML offre des opérations de synchronisation et de communication : la communication s'effectue par passage synchrone de messages à travers des canaux typés ; la création de fils d'exécution et de canaux est dynamique ; les fonctions (définissant les fil d'exécution) et les canaux sont mobiles.

Dans le cœur de tout formalisme de programmation de systèmes concurrents, il y a la notion d'interaction entre entités autonomes (ou agents). Cette interaction se manifeste à travers la communication et la synchronisation. La présence de plusieurs fils d'exécution s'exécutant en parallèle engendre une complexité due à l'explosion combinatoire des différents scénarios d'exécution. Une méthode est donc nécessaire pour gérer cette complexité. L'approche utilisée dans CML est l'introduction d'un nouveau type abstrait de valeurs, appelées événements, et une action de synchronisation. Les événements représentent des communications potentielles ou latentes : ils n'exécutent leur action de communication que lorsqu'ils sont activés au moyen d'une primitive de synchronisation.

Ce chapitre est composé de quatre parties. La première partie présente les concepts et les caractéristiques du langage ainsi que le paradigme sous-jacent. La deuxième partie présente le langage à travers des exemples implantés en utilisant les modules *Thread*, *Mutex*, *Condition* et *Event*. Seules les constructions fondamentales du langage sont présentées dans le but de saisir l'essence de CML. La troisième partie illustre la combinaison de la programmation modulaire et de la programmation concurrente. La quatrième partie illustre, quant à elle, la combinaison des différents paradigmes de programmation.

7.2 Concepts et caractéristiques

Concurrent ML est un système de programmation concurrente. Il constitue une extension de ML en y ajoutant des primitives pour la création et la gestion de fils d'exécution (ou processus légers). Ainsi, un programme CML est composé d'un ensemble de fils d'exécution. Chaque fil d'exécution est une évaluation séquentielle d'une expression ML qui n'a pas forcément de fin. Par exemple, pour concevoir un serveur, il est souvent utile de définir une fonction qui boucle indéfiniment.

L'évaluation d'un fil d'exécution peut impliquer une communication avec d'autres fils d'exécution. Cette communication est réalisée en envoyant ou en recevant un message sur un canal de communication. Le transfert de message est synchrone c'est-à-dire que, par exemple, une réception d'un message reste bloquée jusqu'à ce qu'une transmission du message sur le même canal soit effectuée. Ce mode de transmission est aussi connu sous l'appellation de communication par rendez-vous.

Ces échanges de messages forment les bases de la communication et de la synchronisation en CML. Ce modèle est étendu par des opérations d'ordre supérieur :

- Création dynamique de fils d'exécution : les fils d'exécution peuvent être définis et activés durant l'exécution du programme.
- Création dynamique de canaux : de même, des canaux peuvent être définis et utilisés en cours d'exécution.
- Mobilité des «fils d'exécution» (des fonctions qui les définissent) et des canaux : cette propriété assure une très grande expressivité au langage. En effet, CML permet l'envoi ou la réception de canaux ou de fonctions sur des canaux. Par exemple, un fil d'exécution P_1 peut envoyer, sur un canal c , à un fil d'exécution P_2 , un autre canal c' . Ce dernier est désormais disponible aux deux fils d'exécution pour un échange de messages privés par exemple.

Mais la particularité du langage CML se situe au niveau de l'abstraction de la communication. À l'instar des langages de programmation fonctionnelles, impératives et orienté objets, dans lesquels sont définies des mécanismes d'abstraction au niveau des procédures, des types abstraits ou des interfaces, CML introduit un nouveau mécanisme d'abstraction au niveau de la communication : les événements. Ces derniers correspondent à des communications ou des synchronisations latentes. Pour résumer, les principales caractéristiques du langage sont :

- Fournit un modèle concurrent de haut niveau avec création dynamique de fils d'exécution, de canaux typés et avec une communication de type rendez-vous.
- Procure des abstractions de communication, appelées événements.
- Fournit un système d'entrées-sorties intégré et adapté aux environnements concurrents.
- Est efficace dans la création de fils d'exécution et dans la transmission de messages.
- Est portable car les fonctions sont implantées en OCaml modulo quelques extensions.

Ces caractéristiques sont détaillées dans la section suivante à travers notamment des exemples.

7.3 Langage

7.3.1 Abréviations et fonctions utiles

Nous cette partie des notes de cours, nous utilisons plusieurs fonctions utiles, certaines ayant déjà été présentées dans les chapitres précédents (pour celles-ci, il s'agirait donc d'un rappel) :

```
(* Abréviations utiles *)
module U = Unix
module D = Domain
module T = Domainslib.Task
module E = Event
module R = Random

module P = Printf
module A = Array
module L = List
module S = String
```

```

(* Fonctions utiles *)
let timeRun f x =
  let time1 = U.gettimeofday() in
  let r = f x in
  let time2 = U.gettimeofday() in
  (r,time2 -. time1)

let explode s =
  L.init (S.length s) (S.get s)

let implode l =
  S.init (L.length l) (L.nth l)

let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)

let rec fact n = if n <= 1 then 1 else n * (fact(n-1))

let randomArray n i =
  let _ = Random.self_init () in
  Array.init n (fun _ -> Random.int i)

let randomList n i =
  let _ = R.self_init () in
  L.init n (fun _ -> R.int i)

let compare_fct f1 f2 i =
  let res1,t1 = timeRun f1 i in
  let res2,t2 = timeRun f2 i in
  res1=res2, t1, t2

let pr1 s = print_string s; print_newline(); flush stdout

let pr2 s =
  print_string
    (Printf.sprintf "<%.3f_sec;_id=%d>_" (Sys.time()) (Domain.self() :> int));
  pr1 s

```

7.3.2 Apport de CML

Dans le contexte du langage OCaml, CML se présente sous la forme de deux principaux modules : *Thread* et *Event*. Ces modules doivent être ouverts (à l’aide de la commande «open») afin d’utiliser les différentes primitives présentées dans ce chapitre. Une autre solution serait de préfixer toutes ces primitives par le nom du module. Par exemple :

```

# Thread.create
- : ('a -> 'b) -> 'a -> Thread.t = <fun>

# Event.send
'a channel -> 'a -> unit event = <fun>

```

Notez aussi que pour pouvoir utiliser ces modules, il faut au préalable charger les paquetages «Unix» et «Threads» :

```

# #thread;;
/home/user/.opam/5.1.0/lib/ocaml/threads: added to search path
/home/user/.opam/5.1.0/lib/ocaml/unix: added to search path
/home/user/.opam/5.1.0/lib/ocaml/unix/unix.cma: loaded
/home/user/.opam/5.1.0/lib/ocaml/threads/threads.cma: loaded

```

Une fois ces paquetages chargés, pour exploiter les fonctions définies dans les modules «Thread» et «Event», il est préférable de les ouvrir :

```
# open Thread;;
# open Event;;
```

Dans cette section, nous passons en revue les principales primitives de concurrence de CML ainsi que leur type.

7.3.3 Fil d'exécution

Un programme CML est une collection de fils d'exécution dits «*threads*» (ou processus légers) qui s'exécutent en parallèle et qui utilisent des canaux pour se synchroniser et communiquer. Un nouveau fil d'exécution est créé au moyen de la primitive `create`, définie dans le module `Thread` :

```
# Thread.create;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

Comme l'illustre son type, cette fonction prend en argument une fonction polymorphe de type «*a -> b*», une valeur de type «*a*» et retourne une valeur de type «*Thread.t*». Le corps de la fonction passée en argument comprend l'expression ML exécutée en arrière-plan (en parallèle avec le reste du programme). Cette expression sera évaluée lorsque la fonction `create` appliquera son premier argument, la fonction, à son deuxième argument, la valeur de type «*a*». Le tout, lorsqu'il termine, retourne une valeur de type «*b*».

L'exemple qui suit illustre l'utilisation de cette fonction¹ :

```
# create (fun () -> pr1 "Hello World") ();;
- : Thread.t = <abstr>
Hello World
```

Dans cet exemple, la fonction `create` est appliquée à une fonction qui affiche la chaîne de caractères "Hello World". Dans cet exemple, le rôle de la fonction `create` est d'appliquer à la fonction passée en argument, la valeur `()` passée en argument de la fonction aussi. Cette application permet de «lancer» en arrière-plan la fonction.

En général, un fil d'exécution a un comportement plus complexe que celui présenté précédemment et son traitement comprend une boucle qui est parfois sans fin (comportement propre aux programmes que l'on nomme serveurs). Voici un exemple de code, un peu plus réaliste qu'un simple affichage d'un "Hello World", qui motive le besoin d'un paradigme propre à la mise en concurrence, ou en parallèle, de calculs :

```
# let f () =
  delay 10.; pr1 "<f>_a_fini_son_traitement_lourd";;
val f : unit -> unit = <fun>
# let g () =
  pr1 "<g>_termine_rapidement_et_retourne_une_valeur"; 10 * 10;;
val g : unit -> int = <fun>

# let _ = f () in
  let v = g () in
    v;;
(* 10 secondes plus tard *)
<f> a fini son traitement lourd
<g> termine rapidement et retourne une valeur
- : int = 100
```

1. Dans le reste du chapitre, on suppose que les modules `List` et `Printf` sont ouverts (`open`).

Dans cet exemple, on définit deux fonctions : «f» qui requiert 10 secondes pour réaliser son traitement (considéré comme «lourd»); une fonction «g» qui retourne rapidement un résultat (la valeur 100); cet exemple illustre que l'exécution de la fonction «g» ne peut se faire que lorsque la fonction «f» termine son traitement; c'est le sens même de la séquentialité des opérations.

La solution au problème illustré par l'exemple précédent passe par la création de fils d'exécution; plus précisément par l'exécution de la fonction «f» en arrière-plan (ou, en «parallèle») avec le reste du code, soit celui qui comprend l'exécution de la fonction «g» :

```
# let _ = create f () in
  let v = g () in
    v;;
<g> termine rapidement et retourne une valeur
- : int = 100
(* 10 secondes plus tard *)
<f> a fini son traitement lourd
```

On remarque clairement que la fonction «g» termine rapidement son traitement en retournant la valeur 100. «f», quant à elle, réalise son traitement en arrière-plan, et termine en affichant un message (notez qu'il aura fallu entrer une expression quelconque pour voir, dans l'interpréteur, le message affiché par «f»).

Par ailleurs, un fil d'exécution peut se terminer suite à une exception non capturée ou suite à un suicide au moyen de l'exception `Exit` :

```
# create (fun () -> raise Exit; pr1 "Hello_World") ();;
- : Thread.t = <abstr>
```

Aussi, pour simplifier la présentation des exemples présentés dans ce chapitre, nous définissons la fonction suivante :

```
# let spawn f v = ignore (create f v);;
val spawn : ('a -> 'b) -> 'a -> unit = <fun>
```

La fonction «ignore» est utilisée afin de forcer le type d'une expression quelconque à être de type `unit`; par conséquent, la fonction `spawn` permet d'avoir une version de la fonction `create` qui retourne toujours une valeur de type `unit`. Ceci est utile lorsqu'on veut créer plusieurs fils d'exécution dans une séquence d'expressions. Dans un tel contexte, OCaml requiert que, hormis la dernière, toutes les expressions formant cette séquence retournent une valeur de type `unit`. L'utilisation de la fonction `spawn` évite donc l'affichage de messages d'avertissement.

Mon premier programme CML

Voici un premier exemple de programme CML :

```
# pr1 "Debut";
  spawn (fun () -> pr1 "Deb1"; pr1 "Fin1") ();;
  spawn (fun () -> pr1 "Deb2"; pr1 "Fin2") ();;
  pr1 "Fin";;
Debut
Fin
- : unit = ()
Deb1
Fin1
Deb2
Fin2
```

L'exécution de ce programme se déroule comme suit :

- La première étape consiste à évaluer l'expression «`pr1 "Debut\n"`». Le résultat de cette évaluation correspond à l'affichage de la chaîne de caractères "Debut" suivie d'un retour chariot.
- Par la suite, la première expression `spawn` est évaluée. Elle consiste à mettre en arrière-plan une fonction en argument.
- L'évaluation de la deuxième expression `spawn` commence dès que la première fonction est mise en arrière-plan. L'évaluation de cette expression est similaire à la première. Elle consiste à mettre en arrière-plan une fonction passée en argument.
- Puis au même moment (en parallèle), la quatrième expression est évaluée (affichage du mot "Fin").

En résumé, l'évaluation de cette expression peut produire plusieurs résultats (affichages) possibles. Ce qui reste constant est l'affichage de la première chaîne, soit "Debut". Par la suite, l'évaluation du corps des deux fonctions passées en argument des deux fonctions `spawn` ainsi que celle du dernier affichage se font d'une manière concurrente². Ainsi, ces évaluations peuvent produire l'affichage de la chaîne "Fin" suivie de celui de "Deb1", suivie de celui de "Deb2", et ainsi de suite.

On constate cependant que l'exécution du programme produit un résultat similaire à une exécution séquentielle. Ce qui est en cause est la rapidité avec laquelle s'exécutent les fils d'exécution (très simples) présentés dans cet exemple. Une solution consiste alors à ralentir leur exécution en les faisant berner un laps de temps :

```
# pr1 "Debut" ;
  spawn (fun () -> pr1 "Deb1"; delay 1.; pr1 "Fin1") ();
  spawn (fun () -> pr1 "Deb2"; delay 1.; pr1 "Fin2") ();
  delay 1.;
  pr1 "Fin" ;;
Debut
Deb1
Deb2
Fin1
Fin2
Fin
- : unit = ()
```

Dans cette exécution «réussie», nous utilisons la fonction prédéfinie «`delay`» pour faire hiberner un fil d'exécution pendant un laps de temps; ceci offre à l'ordonnanceur la possibilité d'activer d'autres fils d'exécution en attente d'exécution. Notons aussi que l'exécution répétée de cet exemple pourra produire différents comportements (ordre d'affichage différent d'une exécution à l'autre), ce qui est propre à la programmation concurrente.

Autres fonctions définies dans le module *Thread*

Outre les fonctions `create`, `exit` et `delay`, fonctions présentées dans les sections précédentes, le module *Thread* comprend d'autres fonctions utiles pour la programmation concurrente, parmi lesquelles nous retrouvons :

- la fonction `self` retourne l'identifiant du fil d'exécution courant (celui dans lequel la fonction est invoquée); de même, à partir de son identifiant, la fonction `id` retourne un entier identifiant de manière unique le fil d'exécution en question;
- la fonction `join` suspend l'exécution du fil d'exécution courant jusqu'à la terminaison du fil d'exécution dont l'identifiant est passé en argument; la fonction `kill` termine l'exécution d'un fil d'exécution dont l'identifiant est passé en argument (notez que dans plusieurs distributions d'OCaml, cette fonction n'est pas implantée).

2. Plus exactement, de manière *entrelacée*.

Récupération du résultat d'une fil d'exécution

Considérons un des exemples présentés dans la section précédente, et imaginons que l'on veuille récupérer, en plus de la valeur v de la fonction «g», une valeur v' retournée par la fonction «f»; une fonction «h» serait alors appliquée à ces deux valeurs :

```
# let f () =
  delay 10.; pri "<f>a_fini_son_traitement_lourd"; 5 * 5;;
val f : unit -> int = <fun>
# let g () =
  pri "<g>termine_rapidement_et_retourne_une_valeur"; 10 * 10;;
val g : unit -> int = <fun>
# let h v1 v2 = v1 + v2;;
val h : int -> int -> int = <fun>

# let _ = create f () in
  let v = g () in
    v;; (* h v v': comment récupérer le résultat de f? *)
<g> termine rapidement et retourne une valeur

- : int = 100
(* 10 secondes plus tard *)
# <f> a fini son traitement lourd
```

Comme on peut le constater, seule la valeur v est disponible. Comment alors récupérer la valeur résultante de «f»? Plusieurs solutions sont possibles; en voici quelques-unes.

Solution 1 : utilisation d'une variable partagée En terme d'OCaml, on utilise une référence vers une valeur; aussi, étant donné que l'on voudra distinguer l'état où la référence comprend la valeur résultante de «f», de l'état où elle ne le comprendrait pas, on utilise une référence vers une valeur de type *'a option* :

```
# let r = ref None in
  let _ = create (fun () -> r := Some (f ())) () in
  let v = g () in
  match !r with
  | Some v' -> h v v'
  | None -> (* ??? *) 0;;
<g> termine rapidement et retourne une valeur

- : int = 0
(* 10 secondes plus tard *)
# <f> a fini son traitement lourd
```

Clairement, c'est la section «None», du filtrage par motifs, qui est exécutée et la valeur 0 est retournée comme résultat de toute l'expression (à la place de la valeur attendue «10 * 10 + 5 * 5», soit 125).

Solution 2 : utilisation d'une fonction récursive Dans ce cas, on définit une fonction qui s'appelle récursivement tant que la valeur retournée par «f» n'est pas disponible :

```
# let r = ref None in
  let _ = create (fun () -> r := Some (f ())) () in
  let v = g () in
  let rec wait () =
    match !r with
    | Some v' -> v'
    | None -> wait ()
  in
  let v' = wait () in
  h v v';;
<g> termine rapidement et retourne une valeur

(* 10 secondes plus tard *)
<f> a fini son traitement lourd
- : int = 125
```

Cette solution est correcte puisque le bon résultat est retourné. Cependant, on identifie au moins deux inconvénients à cette méthode :

1. une consommation abusive du processeur qui s'occupe d'exécuter, sans relâche, la fonction récursive «wait» jusqu'à l'obtention du résultat de «f» ;
2. ce code dépend de l'*atomicité* de l'expression «Some (f ())» ; si cette expression s'exécute comme un bloc (de manière *atomique*), tout le code est correct et on obtient le résultat souhaité ; si par contre, l'expression n'est pas atomique, c'est-à-dire que, par exemple, le stockage dans la référence «r» du constructeur «Some» et de son argument, «f ()», peut se dérouler en deux étapes distinctes, il y aurait alors une possibilité que seul le constructeur soit stocké dans la référence «r», que l'ordonnanceur donne la main au fil d'exécution qui se charge de l'exécution de la fonction «wait», que cette dernière constate, par filtrage par motifs, que le contenu de «r» est de type «Some v'» (même si «v'» n'est toujours pas présente), et que le résultat «v'» retournée par «wait» soit indéfini.

Le deuxième inconvénient implique la nécessité de disposer d'un mécanisme qui assure que la fonction «f» a terminé son exécution.

Solution 3 : utilisation de la fonction join Cette solution utilise adéquatement la fonction `join`, définie dans le module «Thread», et résout correctement le problème posé :

```
# let r = ref None in
  let t = create (fun () -> r := Some (f ())) () in
  let v = g () in
  join t;
  match !r with
  | Some v' -> h v v'
  | None -> failwith "Cas_impossible!";;
<g> termine rapidement et retourne une valeur

(* 10 secondes plus tard *)
<f> a fini son traitement lourd
- : int = 125
```

À partir de cette solution, il est possible d'introduire une abstraction via une fonction qu'on nommera «future» (qui existe dans la littérature, et qui se retrouve dans la plupart des langages récents) :


```
# module Future :
sig
  type 'a future
  val future : ('a->'b) -> 'a -> 'b future
  val force : 'a future -> 'a
end =
struct
  type 'a future = {t_id : Thread.t ; v : 'a option ref}

  let future f x =
    let r = ref None in
    let t = create (fun () -> r := Some(f x)) () in
    {t_id=t ; v=r}

  let force f =
    join f.t_id;
    match !(f.v) with
    | Some v -> v
    | None -> failwith "Cas_impossible!"
end;;
module Future :
sig
  type 'a future
  val future : ('a -> 'b) -> 'a -> 'b future
  val force : 'a future -> 'a
end
```

Cette solution offre au programmeur une solution efficace et élégante pour lancer un calcul en parallèle (d'un autre calcul) et récupérer ultérieurement son résultat :

```
# open Future;;

# let x = future f () in
  let v = g () in
  let v' = force x in
  h v v';;
<g> termine rapidement et retourne une valeur
(* 10 secondes plus tard *)
<f> a fini son traitement lourd
- : int = 125
```

7.3.4 Version 5.0 et plus d'OCaml

Depuis la version 5.0, OCaml bénéficie de nouveaux modules et paquetages offrant le vrai parallélisme (via, entre autres, le module standard «Domain» et le paquetage «Domainslib») ainsi que des gestionnaires d'effets (via, entre autres, le module «Effect» et le paquetage «eio») qui sont une généralisation des gestionnaires d'exceptions permettant l'expression de flux de contrôle non-locaux tels que les exceptions résumables ou les fils d'exécution (*threads*) légers (programmation concurrente).

Ressources du cours antérieurs à hiver 2024 Tous les exemples issus des examens antérieurs à l'hiver 2024, sont basés sur une version d'OCaml antérieure à 5.0 qui ne comporte ni de vrai parallélisme, ni de gestionnaires d'effets. Elles utilisent plutôt et principalement le module «Thread». Ce module offre la possibilité de définir des fils d'exécution légers qui s'exécutent dans un même fil d'exécution système (on parle alors d'entrelacement et on se situe dans la programmation concurrente); par conséquent, on ne dispose de vrai parallélisme, sauf dans les 2 cas suivants :

1. on fait appel à «Thread.delay» pour faire hiberner le fil d'exécution courant pendant un certain nombre de secondes;

2. on fait appel à des fonctions systèmes, comme par exemple l'utilisation du réseau, qui résultent en une exécution asynchrone, ce qui libère le fil d'exécution courant et donne la chance à un autre fil d'exécution d'être exécutée.

Dans ces 2 cas de figure, l'ordonnanceur interrompt le fil d'exécution courant (le temps que l'hibernation finisse ou que l'appel de fonctions systèmes retourne avec un résultat) et «donne la main» à un autre fil d'exécution.

La principale fonction du module «Thread» permettant de lancer «en arrière plan» un nouveau fil d'exécution est :

```
# Thread.create
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

À partir de cette fonction, nous avons créé une autre fonction («maison»), «spawn», permettant d'ignorer le résultat de la fonction «create» et donc de l'utiliser en séquence pour lancer successivement plusieurs fils d'exécutions évitant ainsi le message d'avertissement (*warning*) pour des expressions en séquence qui ne retournent «unit» :

```
# let spawn f v = ignore (create f v);;
val spawn : ('a -> 'b) -> 'a -> unit = <fun>
```

Par conséquent, dans les exemples tirés des examens antérieurs (à H-24), on utilise essentiellement «spawn» et parfois «create». Par ailleurs, en plus de «create», on retrouve une autre fonction pertinente dans le module «Thread» :

```
# Thread.join;;
- : Thread.t -> unit = <fun>
```

Cette fonction prend en argument un fil d'exécution léger et attend jusqu'à sa terminaison. Malheureusement, cette fonction retourne «unit» même si la fonction liée au fil d'exécution léger retourne un certain résultat (différent de «unit»). Pour surmonter cette limite, nous avons aussi créé un module «Future» permettant de créer des «futures» qui correspondent à des fils d'exécution légers créés avec «create» mais à partir desquels on a la possibilité de récupérer le résultat de la fonction qui lui est liée (fonction «force» qui bloque tant que le fil d'exécution léger n'a terminé son exécution et retourne par la suite la valeur de la fonction qui lui est liée).

Notes de cours, acétates et code fournis à l'hiver 2024 Bien que le module «Thread» n'offre de vrai parallélisme, toute la matière, les exemples et le code fournis demeurent corrects et fonctionnels et illustrent parfaitement les aspects inhérents à la programmation concurrente et parallèle. Par conséquent, toutes les ressources du cours demeurent cohérentes, notamment par rapport aux énoncés et corrigés des examens antérieurs à l'hiver 2024.

Par ailleurs, il est possible de modifier très légèrement chaque exemple de code fourni afin de bénéficier des apports de la version 5 et celles qui ont suivi d'Ocaml. Dans cette perspective, quand cela sera jugé pertinent, j'indiquerai dans les ressources de cours, dans une partie de texte encadrée et identifiée par «**OCAML 5**», des exemples utilisant les fonctionnalités offertes par ces versions d'Ocaml. Voici un exemple d'un tel texte :

OCAML 5

Voici un exemple simple de création d'un nouveau *domaine* (ou fil d'exécution) avec la fonction «spawn» du module «Domain» :

```
# Domain.spawn (fun () -> print_endline "Hello_World!");;
Hello_World!
- : unit Domain.t = <abstr>
```

Le domaine créé affiche un message à l'écran et termine son exécution.

Principales nouveautés à partir de la version 5 d’Ocaml Depuis la version 5.0, on retrouve essentiellement la fonction suivante définie dans le module «Domain» :

```
# Domain.spawn;;
- : (unit -> 'a) -> 'a Domain.t = <fun>
```

qui, à la différence de «Thread.create» (et donc de la fonction maison «spawn» qu’on s’est définie), permet de créer un fil d’exécution système qui s’exécute en parallèle (si la machine hôte comprend plusieurs coeurs) avec les autres fils d’exécutions créés avec cette même fonction. Aussi, à la différence de «Thread.create» qui retourne un type abstrait «Thread.t» représentant un fil d’exécution léger, «Domain.spawn» retourne un type «'a Domain.t», «'a» correspondant au type de retour de la fonction lancée en arrière plan ; ce qui veut dire qu’on peut récupérer le résultat de cette fonction lorsqu’elle termine son calcul (alors qu’avec «Thread.create», nous étions contraints de définir un module «Future» nous permettant de récupérer le résultat de la fonction lancée en arrière plan).

Par ailleurs, s’agissant de fils d’exécution système, «Domain.spawn» est limitée à la création de 128 fils d’exécution, en tenant compte du fil d’exécution principal (celui du «main»). Pour surmonter cette limite, un paquetage à code ouvert «Domainslib» est disponible (il existe d’autres paquetages aussi pertinents) et permet de créer une «infinité» de fils d’exécution légers qui, à la différence de «Thread.create», s’exécuteront en parallèle grâce à un groupe (ou «pool») de fils d’exécutions système (créés avec «Domain.spawn») qui se partagent ces fils d’exécution légers. Les 2 principales fonctions utilisées dans ce paquetage sont :

```
#require "domainslib";;

# Domainslib.Task.async;;
- : pool -> 'a task -> 'a promise = <fun>
# Domainslib.Task.await;;
- : pool -> 'a promise -> 'a = <fun>
```

La première, «async», prend en argument un «pool» et une tâche de type «'a task» (qui correspond à une fonction de type «unit -> 'a») et retourne une «promesse» correspondant à un fil d’exécution léger qui retourne une valeur de type «'a». La deuxième, «await», permet d’attendre la fin de l’exécution d’une «promesse» et retourne sa valeur (lorsque le fil d’exécution léger liée à cette «promesse» termine son calcul) ; elle prend aussi en argument un *pool*.

Par conséquent, lorsqu’il s’agit de créer un nombre restreint de fils d’exécution, que chacun d’eux admet un traitement assez lourd en tâches et en temps de calcul, il est préférable de créer de tels fils à partir de la fonction «Domain.spawn». Par contre, lorsque le nombre de fils d’exécution à créer est très important et qu’il surpasse nettement le seuil de 128 fils d’exécution actifs en même temps, il est préférable d’utiliser les fonctions fournies par le paquetage «Domainslib», fonctions qui auront à créer autant de fils d’exécution système que nécessaire (grâce à la fonction Domain.spawn).

En résumé, voici un tableau récapitulatif des principales différences entre la version 5.0 d’Ocaml (et celles qui suivent) et les versions antérieures à 5.0, au sujet des fonctions permettant de créer des fils d’exécution et de se synchroniser avec leur terminaison :

Ocaml < 5.0	Ocaml ≥ 5.0
(* Exclusivement fil d’exécution léger *) Thread.create spawn (fonction «maison») Thread.join + module «Future»	(* Fil d’exécution système - limité à 128 fils d’exécution *) Domain.spawn (* remplace «Thread.create» et «spawn» maison *) Domain.join (* remplace Thread.join et module «Future» *) (* Fil d’exécution léger - «aucune limite»*) Domainslib.Task.async Domainslib.Task.await

Au sujet du module «Domain» Au même titre que les modules «String», «List», etc., ce module est standard et est disponible dès le lancement de l'interpréteur, ou à la compilation d'un projet Ocaml. On peut donc accéder à ses fonctions soit avec la notation pointée, soit en utilisant «open» pour ouvrir ce module.

Comme souligné dans la section précédente, le module «Domain» fournit une fonction «spawn» et une fonction «join» permettant de faire abstraction des fonctions «Thread.create», «spawn» créée à partir de «Thread.create» et «Thread.join». Cependant :

1. contrairement à «Thread.create» qui requiert 2 arguments (une fonction f de type « $a \rightarrow b$ » et une valeur v de type « a » ; la fonction lancée en arrière plan correspondant à « $f v$ »), «Domain.spawn» ne requiert qu'un seul argument f de type « $unit \rightarrow a$ » ; puisqu'elle impose que la fonction passée en argument prenne comme argument «unit», elle lance en arrière plan « $f ()$ » ;
2. contrairement à «Thread.create», «Domain.spawn» crée un fil d'exécution système et non un fil d'exécution léger ; l'avantage est qu'on dispose du vrai parallélisme ; l'inconvénient est que la création d'un tel fil d'exécution système implique une charge non négligeable en ressources mémoire et temps de calcul ; de plus, s'agissant de fils d'exécution système, on est limité à 128 fils d'exécution s'exécutant en même temps à un instant t (ce nombre inclut le fil d'exécution principal).

Par ailleurs, voici les principales fonctions fournies par le module «Domain» :

```
val spawn : (unit -> 'a) -> 'a t
val join : 'a t -> 'a

val get_id : 'a t -> id
val self : unit -> id

val is_main_domain : unit -> bool
val before_first_spawn : (unit -> unit) -> unit

val at_exit : (unit -> unit) -> unit

val cpu_relax : unit -> unit

val recommended_domain_count : unit -> int
```

Outre les 2 premières fonctions déjà discutée, voici quelques informations au sujet des autres fonctions :

- Les fonctions «get_id» et «self» permettent respectivement de retourner l'identifiant unique d'un fil d'exécution passé en argument et l'identifiant unique du fil d'exécution courant ; dans les 2 cas de figure, «id» étant un type privé, on doit utiliser le transtypage pour forcer son type à être de type «int» et pouvoir l'utiliser (ou l'afficher) :

```
# let open Printf in
let open Domain in
let t1 = spawn (fun () -> pr1 "T1 lancé en arrière plan") in
sprintf "id(Tmain)=%d, et id(T1)=%d" (self():>int) (get_id t1 :> int);;
T1 lancé en arrière plan
- : string = "id(Tmain) = 0, et id(T1) = 10"
```

- La fonction «is_main_domain» permet, comme son nom l'indique, de tester si le fil d'exécution courant correspond au principal fil d'exécution du programme («main»). La fonction «before_first_spawn» permet d'enregistrer une fonction passée en argument qui sera invoquée avant que le premier domaine ne soit créé par le programme ; il est possible d'utiliser cette fonction pour enregistrer plusieurs fonctions successivement : ces fonctions seront lancées en suivant le protocole «premier arrivé premier servi» (*First-in first-out* (FIFO)) ; une exception «Invalid_argument» est soulevée si lors de l'appel de la fonction «before_first_spawn», au moins un domaine est déjà créé.

- La fonction «at_exit» permet d’enregistrer une fonction passée en argument qui sera invoquée à la terminaison du domaine dans lequel la fonction est invoquée; il est possible d’utiliser cette fonction pour enregistrer plusieurs fonctions successivement : ces fonctions seront lancées en suivant le protocole «dernier arrivé premier servi» (*Last-in first-out* (LIFO)).
- La fonction «cpu_relax» permet à un domaine de retourner explicitement la main à l’ordonnonneur, donnant une opportunité à d’autres domaines de s’exécuter.
- La fonction «recommended_domain_count», comme son nom le suggère, retourne le nombre de domaines maximum recommandé pour être exécutés en même temps (incluant les domaines déjà en exécution); plus précisément, on utilisera cette fonction pour déterminer le nombre de coeurs de calculs disponibles dans notre machine, en visant de ne pas dépasser ce nombre lors de la création des domaines, s’assurant ainsi de bénéficier un maximum du parallélisme. Voici un exemple simple d’appel à cette fonction :

```
# Domain.recommended_domain_count();
- : int = 16
```

La fonction nous indique donc que le maximum de domaines (ou fils d’exécution) s’exécutant à un instant t ne devrait idéalement (recommandation) dépasser les 16 domaines, y compris le domaine principal qui est créé dès le lancement d’un programme, et qui sera donc toujours actif jusqu’à la fin du programme.

Au sujet du module «Atomic» Bien qu’introduit dès la version 4.12 d’Ocaml, ce module offre des fonctions fort utiles dans un contexte de vrai parallélisme. Les principales fonctions sont :

```
val make : 'a -> 'a t
val get : 'a t -> 'a
val set : 'a t -> 'a -> unit

val exchange : 'a t -> 'a -> 'a
val compare_and_set : 'a t -> 'a -> 'a -> bool
val fetch_and_add : int t -> int -> int

val incr : int t -> unit
val decr : int t -> unit
```

La fonction «make» permet de créer une valeur référence *atomique*, c’est-à-dire un espace mémoire mutable où on pourra stocker une valeur qui pourra être modifiée; chaque opération d’accès en lecture ou en écriture à cet espace mémoire (d’un type abstrait) est une opération atomique, dans le sens où elle ne peut être interrompue par l’ordonnonneur.

La fonction «exchange» agit comme «set» mais retourne en résultat la nouvelle valeur de la référence. La fonction «compare_and_set» permet de mettre à jour la valeur d’une référence seulement si cette dernière a comme valeur courante une valeur passée en argument (2^e argument de la fonction); elle retourne «true» si la mise à jour a réussi; «false», sinon.

La fonction «fetch_and_add» incrémente de n , de manière atomique, une valeur référence atomique comprenant un entier; elle retourne la valeur courante de la référence avant l’incrément. En particulier, les fonctions «incr» et «decr» permettent de mettre à jour une telle référence comprenant un entier en l’incrémentant de 1 ou en le décrémentant de 1. Ces fonctions impliquent en théorie plusieurs opérations assembleur distinctes («MOV» de la valeur dans un registre; mise à jour de la valeur; «MOV» de la nouvelle valeur calculée dans la référence); mais le fait qu’elles soient atomiques impliquent qu’il ne pourra y avoir d’interruption machine entre l’une de ces 3 opérations assembleur (en fait, les processeurs récents fournissent des instructions machine qui permettent de faire ce type de calcul de manière atomique).

Pour illustrer l’intérêt d’utiliser ces fonctions, considérons l’exemple séquentielle suivant :

```

let num_domains = Domain.recommended_domain_count () - 1;;
val num_domains : int = 15

# let test0 n =
  let cpt = ref 0 in
  let f () =
    for _ = 1 to n do incr cpt done
  in
  for i = 1 to num_domains do f() done;
  pr1 ( sprintf "Test0(Seq)_ _cpt_=%d" (!cpt) );
  val test1 : int -> unit = <fun>

# test0 1000000;;
Test0(Seq) - counter = 15000000
- : unit = ()

```

«test0» appelle la fonction «f» 15 fois et à chaque appel, cette fonction incrémente de 1 n fois le compteur «cpt»; comme n vaut 1 million, le résultat retournée est bien 15 millions.

Maintenant, considérons une version parallélisée de cette fonction grâce aux fonction «Domain.spawn» et «Domain.join» :

```

# let test1 n =
  let cpt = ref 0 in
  let f () =
    for _ = 1 to n do incr cpt done
  in
  List.iter D.join (List.init num_domains @@ fun _ -> D.spawn f);
  pr1 ( sprintf "Test1(ParBrut)_ _cpt_=%d" (!cpt) );
  val test1 : int -> unit = <fun>

# test1 1000000;;
Test1(ParBrut) - cpt = 2537951
- : unit = ()

```

Dans cet exemple, on crée une liste de 15 domaines chacun lançant en arrière plan un appel à la fonction «f»; comme tous ces appels se font en parallèle est que l'accès à la variable «cpt» se fait de manière non atomique, le résultat retourné n'est pas celui attendu; on a donc choisi le parallélisme en vue de chercher de la performance, on se retrouve avec un programme qui ne retourne le bon résultat!

Finalement, considérons une version utilisant une référence atomique :

```

# let test2 n =
  let cpt = Atomic.make 0 in
  let f () =
    for _ = 1 to n do Atomic.incr cpt done
  in
  List.iter D.join (List.init num_domains @@ fun _ -> D.spawn f);
  pr1 ( sprintf "Test2(ParAtom)_ _cpt_=%d" (Atomic.get cpt) );
  val test2 : int -> unit = <fun>

# test2 1000000;;
Test2(ParAtom) - cpt = 15000000
- : unit = ()

```

Dans cet exemple, on retrouve le bon résultat malgré l'exécution en parallèle de 15 domaines qui font chacun appel à «f»; comme l'accès à l'incrément de la variable «cpt» se fait désormais de manière atomique, il n'y plus d'incohérence dans le résultat calculé.

Au sujet du module «Domainlib.Task» du paquetage «Domainlib» Contrairement au module «Domain», le module «Domainlib» requiert l'installation du paquetage «Domainlib» via

la commande «*opam*». Une fois installé, au niveau de l'interpréteur, on peut charger les modules qui se trouvent dans ce paquetage à travers la commande «*#require*», puis exploiter les fonctions qui se trouvent dans le module «*Domainslib.Task*» en l'ouvrant avec la commande «*open*» : bénéficiant ainsi d'un maximum de parallélisme.

```
# #require "domainslib";;
...
/home/user/.opam/5.1.0/lib/domainslib/domainslib.cma: loaded

# open Domainslib.Task;;
# async;;
- : pool -> 'a task -> 'a promise = <fun>
```

Par ailleurs, voici les principales fonctions fournies par ce module :

```
type 'a task = unit -> 'a
type 'a promise
type pool

val setup_pool : ?name:string -> num_domains:int -> unit -> pool
val teardown_pool : pool -> unit
val get_num_domains : pool -> int

val async : pool -> 'a task -> 'a promise
val await : pool -> 'a promise -> 'a
val run : pool -> 'a task -> 'a

val parallel_for : ?chunk_size:int -> start:int -> finish:int ->
    body:(int -> unit) -> pool -> unit
val parallel_find : ?chunk_size:int -> start:int -> finish:int ->
    body:(int -> 'a option) -> pool -> 'a option
val parallel_for_reduce : ?chunk_size:int -> start:int -> finish:int ->
    body:(int -> 'a) -> pool -> ('a -> 'a -> 'a) -> 'a -> 'a
val parallel_scan : pool -> ('a -> 'a -> 'a) -> 'a array -> 'a array
```

Un type concret est déclaré permettant de représenter des tâches (*tasks*) correspondant à des fonctions ayant pour argument «*unit*» et retournant un résultat de type «*'a*» ; 2 types abstraits sont aussi déclarés : un pour définir des promesses ayant des valeurs de type «*'a*» ; et un pour représenter des *pool* permettant de regrouper les tâches dans un même bassin afin d'être sélectionnées et exécutées par différents domaines s'exécutant en parallèle.

De plus, on retrouve différentes fonctions qu'on peut classer dans 3 groupes :

1. Le premier comprend les fonctions «*setup_pool*», «*teardown_pool*» et «*get_num_domains*» permettant respectivement de créer un *pool* lié à un nombre de domaines passé en argument, de supprimer un *pool*, et de retourner le nombre de domaines disponibles dans un *pool*. Typiquement, avant le lancement de différents fils d'exécution, on crée un *pool* ; et lorsque toutes les tâches ont été exécutées, on supprime le *pool*.

Naturellement, toutes les autres fonctions disponibles dans ce module prennent comme argument un *pool* préalablement créé.

2. Le deuxième regroupe les 3 fonctions suivantes :

- (a) La fonction «*async*» ajoute une tâche passé en argument dans un *pool* (lui aussi passé en argument) ; elle est exécutée de manière asynchrone avec le reste du programme ; autrement dit, aussitôt que la commande «*async*» est invoquée, une tâche est ajoutée au *pool* et on reprend tout de suite la main (pas de blocage jusqu'à la terminaison de la tâche, d'où l'asynchronisme) ; le résultat retourné instantanément par la fonction est une valeur de type «*'a promesse*» qui nous permettra de récupérer, lorsque voulu, la valeur retournée à la fin de l'exécution de la tâche.

- (b) La fonction «*await*» va de paire avec «*async*» puisqu'elle s'applique à une promesse forcément produite par «*async*» et bloque jusqu'à ce que la tâche qui a initié cette promesse termine et retourne un résultat : ce dernier est retourné comme résultat de «*await*».
- (c) La fonction «*run*» ajoute une tâche passée en argument dans un *pool* (lui aussi passé en argument) qui est exécutée de manière synchrone avec le reste du programme ; autrement dit, la commande «*run*» bloque jusqu'à la terminaison de la tâche et un résultat récupéré est retourné.

Voici un exemple concret faisant intervenir les 5 fonctions «*setup_pool*», «*teardown_pool*», «*async*», «*await*» et «*run*» (qui sont habituellement utilisées ensemble) :

```
# module T = Domainslib.Task;;
# let rec fib_par pool n =
  if n > 25 then
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b
  else
    fib n;;
val fib_par : T.pool -> int -> int = <fun>

# let main n =
  let ncoeurs = Domain.recommended_domain_count() in
  let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
  let res = T.run pool (fun _ -> fib_par pool n) in
  T.teardown_pool pool;
  pr1 (sprintf "fib(%d) = %d\n" n res);;
val main : int -> unit = <fun>

# main 44;;
fib(44) = 1134903170
- : unit = ()
```

Cet exemple présente une implémentation parallèle du calcul du fibonnacci d'un nombre n :

- la fonction «*fib_par*» prend en argument un *pool* et un entier n ; si cet entier est inférieure à une certaine valeur pour laquelle on jugerait que la version séquentielle de la fonction fibonnacci serait plus efficace, alors on utilise cette version pour retourner le résultat ; sinon, on crée 2 promesses «*a*» et «*b*» qui correspondent respectivement, et grâce à la fonction «*asyn*», de calculer de manière asynchrone fibonnacci de $n - 1$ et fibonnacci de $n - 2$ puis d'attendre (de bloquer), grâce à la fonction «*await*» la terminaison de ces 2 tâches ou promesses afin de récupérer leur résultat et de retourner leur somme ;
 - la fonction «*main*» crée un *pool* composé de «*ncoeurs* - 1» domaines («*ncoeurs*» étant le nombre de domaines maximum créés recommandé), puis lance de manière synchrone, via «*run*», la tâche qui consiste à calculer fibonnacci de n (n étant l'argument de «*main*») ; une fois le résultat obtenu, signifiant que cette tâche est terminée et qu'on a récupéré les résultats de toutes les promesses éventuellement créées, on supprime naturellement le *pool*, puisque plus nécessaire, et on affiche le résultat à l'écran.
3. Le troisième regroupe les 4 fonctions suivantes qui doivent être appelées à l'intérieur d'un appel à la fonction «*run*» :
- (a) «*parallel_for*», comme son nom l'indique, est une version parallèle de l'expression «**for-do-done**» d'Ocaml ; elle prend comme arguments «*start*», «*finish*» et «*body*», qui correspondent respectivement à l'indice du début du traitement, l'indice de fin, et le traitement (fonction) à appliquer à chaque fois (pour chaque indice i , i étant l'argument de la fonction «*body*»), et lance en parallèle (via un *pool*) les tâches (appels à fonction «*body*») à

exécuter. La fonction comprend aussi un argument optionnel «*chunk_size*» qui correspond au nombre d'appels de «*body*» pour une tâche donnée. Sa valeur par défaut est égale à « $\max(1, (\text{finish} - \text{start} + 1) / (8 * \text{num_domains}))$ » ; autrement dit, le nombre total d'appels à «*body*» à effectuer et on le divise par 8 fois le nombre de domaines. Supposons, par exemple, que le nombre de domaines vaut 8 et que «*start*» et «*finish*» valeur respectivement 0 et 127 ; alors la valeur par défaut de «*chunk_size*» est « $128 / 8 * 8$ », soit 2. Par conséquent, chaque tâche créée devra faire appel à 2 reprises à la fonction «*body*», ce qui fait qu'on créera en tout « $128 / 2$ », soit 64, tâches que devront exécuter les 8 domaines.

Voici un exemple d'utilisation de cette fonction :

```
# let main n1 n2 =
  let ncoeurs = Domain.recommended_domain_count() in
  let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
  let _ = pr1(sprintf "#domaines=%d\n" (T.get_num_domains pool)) in
  let _ = T.run_pool (fun _ ->
    T.parallel_for ~start:n1 ~finish:n2 ~body:(fun i ->
      if est_premier i then pr1(sprintf "T(%d):%d_" (Domain.self():>int) i)
    ) pool
  ) in
  T.teardown_pool pool;
  print_newline();;
val main : int -> int -> unit = <fun>

# main 100 200;;
#domaines = 16
- : int array * int array =
T(0):199 T(0):163 T(53):173 T(59):167 T(58):193 T(58):191 T(0):157T(55):197
T(0):151 T(50):149 T(0):113 T(0):103 T(0):101 T(58):109 T(0):139 T(58):107
T(0):179 T(54):181 T(0):137 T(54):131 T(0):127
- : unit = ()
```

Dans cet exemple, on itère, de manière parallèle, sur les nombres compris entre 100 et 200 à la recherche de nombres premiers ; chaque tâche qui en trouve un l'affiche à l'écran avec l'identifiant unique du domaine courant («*self*») c'est-à-dire le domaine qui a pris cette tâche disponible dans le *pool* et qu'il a exécutée. La valeur par défaut de l'argument optionnel «*chunk_size*» vaut dans ce cas « $\max(1, 101 / 8 * 16)$ », soit 1. Par conséquent, chaque tâche aura à sa charge un appel à «*body*», pour un total de 101 tâches.

On remarque qu'en tout 7 domaines différents (d'identifiants respectifs 0, 53, 59, 58, 55, 50, 54), donc y compris le domaine principal (qui a toujours l'identifiant 0), ont été utilisés pour réaliser les 101 tâches lancées dans le bassin de tâches (ou *pool*).

Voici un autre exemple qui consiste à calculer, de manière parallèle, les carrés de nombres de trouvant dans un tableau de *n* éléments :

```
# let main n vmax =
  let ncoeurs = Domain.recommended_domain_count() in
  let t = randomArray n vmax in
  let t' = Array.make n 0 in
  let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
  let _ = T.run_pool (fun _ ->
    T.parallel_for ~start:0 ~finish:(n-1)
      ~body:(fun i -> t'.(i) <- t.(i)*t.(i)) pool
  ) in
  T.teardown_pool pool;
  t, t';;
val main : int -> int -> int array * int array = <fun>
```

```
# main 50 100;;
- : int array * int array =
[|33; 93; 17; 16; 37; 36; 25; 18; 46; 19; 28; 98; 60; 93; 1; 76; 19; 74; 54;
 37; 4; 16; 33; 9; 47; 66; 62; 76; 52; 61; 27; 47; 99; 7; 25; 56; 87; 93;
 32; 79; 98; 79; 17; 99; 57; 59; 90; 45; 13; 16|],
[|1089; 8649; 289; 256; 1369; 1296; 625; 324; 2116; 361; 784; 9604; 3600;
 8649; 1; 5776; 361; 5476; 2916; 1369; 16; 256; 1089; 81; 2209; 4356; 3844;
 5776; 2704; 3721; 729; 2209; 9801; 49; 625; 3136; 7569; 8649; 1024; 6241;
 9604; 6241; 289; 9801; 3249; 3481; 8100; 2025; 169; 256|]
```

Dans cet exemple, bien qu'une opération d'affectation d'une cellule du tableau, soit une opération impliquant un effet de bord, est utilisée, le traitement parallèle des tableaux «t» et «t'» demeurent sûr car chaque traitement effectuée par une tâche n'interfère pas avec les traitements effectués par les autres tâches.

De cet exemple, on peut générer une variante parallèle de la fonction «Array.map» :

```
# let pmap f t =
  let n = Array.length t in
  if n < 10 then
    Array.map f t
  else
    let ncoeurs = Domain.recommended_domain_count() in
    let t' = Array.make n (f t.(0)) in
    let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
    let _ = T.run_pool (fun _ ->
      T.parallel_for ~start:1 ~finish:(n - 1)
        ~body:(fun i -> t'.(i) <- f t.(i)) pool
    ) in
    T.teardown_pool pool;
    t';;
  val pmap : ('a -> 'b) -> 'a array -> 'b array = <fun>

# let t1 = randomArray 20 100;;
val t1 : int array =
[|4; 44; 66; 52; 60; 41; 82; 92; 93; 55; 39; 63; 63; 92; 64; 0; 55; 67; 67; 47|]

# pmap string_of_int t1;;
- : string array =
[|"4"; "44"; "66"; "52"; "60"; "41"; "82"; "92"; "93"; "55"; "39"; "63";
 "63"; "92"; "64"; "0"; "55"; "67"; "67"; "47"|]
```

On retrouve la signature attendue d'une telle fonction, soit

(*'a* -> *'b*) -> *'a* array -> *'b* array

sauf que le traitement est totalement parallélisé et distribué entre plusieurs domaines définis par la fonction «parallel_for».

- (b) «parallel_find» s'apparente à la version parallèle de la fonction «exists» (définie sur les liste), mais à la place de retourner un booléen, elle retourne une valeur de type «*'a* option» selon que l'élément a été trouvé («Some v», retournée par un des appels à «body»), ou ne l'a été (tous les appels à «body» retournent «None»). Voici un exemple de code qui cherche, de manière parallèle, un nombre premier dans un tableau donné :

```
# let t1 = randomArray 50 100;;
val t1 : int array =
[|61; 65; 63; 43; 34; 29; 87; 63; 58; 76; 80; 81; 75; 22; 15; 50; 30; 89;
 39; 83; 76; 8; 75; 81; 25; 65; 14; 50; 42; 23; 90; 69; 15; 76; 5; 0; 9;
 41; 80; 89; 43; 74; 87; 52; 16; 70; 34; 11; 17; 3|]

# let pexists p t =
  let num_domains = Domain.recommended_domain_count () - 1 in
  let pool = T.setup_pool ~num_domains ~name:"pool" () in
  let res = T.run_pool (fun _ ->
    T.parallel_find ~start:0 ~finish:(Array.length t - 1)
      ~body:(fun i -> if p t.(i) then Some t.(i) else None
    ) pool
  ) in
  T.teardown_pool pool;
  res;;
  val pexists : ('a -> bool) -> 'a array -> 'a option = <fun>

# pexists est_premier t1;;
- : int option = Some 3
```

Ainsi, la première valeur trouvée est 3 qui correspond bien à un nombre premier. On peut remarquer que si on appelle de nouveau la fonction «pexists» avec le même argument (même tableau), on peut recevoir d'autres valeurs en résultat ; c'est bien la première tâche qui trouve un nombre entier qui arrête tout le traitement parallèle en retournant ce nombre :

```
# pexists est_premier t1;;
- : int option = Some 43
# pexists est_premier t1;;
- : int option = Some 3
# pexists est_premier t1;;
- : int option = Some 29
```

- (c) «parallel_for_reduce» s'apparente à «parallel_for» mais comprend 2 arguments additionnels ; aussi, contrairement à «parallel_for» dans laquelle la fonction «body» retourne «unit» comme résultat, l'argument «body» de «parallel_for_reduce» est une fonction qui retourne un résultat de type «'a» ; ainsi, chaque traitement effectué retourne un résultat qui sera récupéré par le premier argument additionnel de la fonction pour calculer un résultat final par accumulation en considérant une première valeur de l'accumulateur correspondant au deuxième argument additionnel de la fonction, le tout produisant un résultat final comme le ferait la fonction «fold_left» (ou «fold_right») ; la seule différence étant que «fold_left» (ou «fold_right») traite les données dans un sens bien précis (gauche à droite ou inversement), et en séquence, alors que dans «parallel_for_reduce», on fait les traitements dans un ordre aléatoire.

Voici un exemple d'utilisation de cette fonction qui consiste à faire la somme des éléments d'un tableau :

```
# let t1 = randomArray 50 10;;
val t1 : int array =
[[4; 0; 7; 4; 4; 5; 6; 4; 4; 7; 7; 6; 2; 3; 1; 2; 7; 4; 4; 6; 6; 2; 9; 0; 6;
 9; 9; 3; 1; 8; 0; 8; 4; 2; 5; 3; 5; 8; 3; 5; 7; 2; 8; 7; 2; 1; 0; 4; 9; 6]]

# let main t =
  let ncoeurs = Domain.recommended_domain_count() in
  let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
  let res = T.run_pool (fun _ ->
    T.parallel_for_reduce
      ~start:0 ~finish:(Array.length t - 1)
      ~body:(fun i -> t.(i)) pool (+) 0
  ) in
  T.teardown_pool pool;
  res;;
val main : int array -> int = <fun>

# main t1;;
- : int = 229
```

Ce même exemple pourrait être écrit avec la fonction «parallel_for» grâce à l'utilisation d'une valeur atomique «sum» qui sera utilisée et mise à jour par les différentes tâches :

```
# let main' t =
  let ncoeurs = Domain.recommended_domain_count() in
  let pool = T.setup_pool ~num_domains:(ncoeurs - 1) () in
  let sum = Atomic.make 0 in
  let _ = T.run_pool (fun _ ->
    T.parallel_for
      ~start:0 ~finish:(Array.length t - 1)
      ~body:(fun i -> ignore (Atomic.fetch_and_add sum t.(i)))
  ) pool
  in
  T.teardown_pool pool;
  Atomic.get sum;;
val main' : int array -> int = <fun>

# main' t1;;
- : int = 229
```

- (d) «parallel_scan» est la seule fonction, parmi les quatre, qui prend en argument un tableau et retourne en résultat un tableau; elle s'apparente à la version parallèle de la fonction «fold_left» («fold_right») mais à la place de retourner la dernière valeur d'un accumulateur, elle retourne un tableau de toutes les valeurs intermédiaires de l'accumulateur. Voici un exemple qui reprend le même type de calcul que les 2 exemples précédents, soit le calcul de la somme des éléments d'un tableau :

```
# let main'' t =
  let num_domains = Domain.recommended_domain_count () - 1 in
  let pool = T.setup_pool ~num_domains~name:"pool" () in
  let t' = T.run_pool (fun _ -> T.parallel_scan pool (+) t) in
  T.teardown_pool pool;
  t, t';
val main'' : int array -> int array * int array = <fun>

main'' t1;;
- : int array * int array =
  ([4; 0; 7; 4; 4; 5; 6; 4; 4; 7; 7; 6; 2; 3; 1; 2; 7; 4; 4; 6; 6; 2; 9; 0; 6;
    9; 9; 3; 1; 8; 0; 8; 4; 2; 5; 3; 5; 8; 3; 5; 7; 2; 8; 7; 2; 1; 0; 4; 9; 6],
  [4; 4; 11; 15; 19; 24; 30; 34; 38; 45; 52; 58; 60; 63; 64; 66; 73; 77; 81;
    87; 93; 95; 104; 104; 110; 119; 128; 131; 132; 140; 140; 148; 152; 154; 159;
    162; 167; 175; 178; 183; 190; 192; 200; 207; 209; 210; 210; 214; 223; 229])
```

Dans le tableau résultant de ce programme, on retrouve comme dernière valeur du tableau la somme des éléments du tableau «t1» en argument de «main»; et les éléments précédents cette valeur correspondent à toutes les valeurs de l'accumulateur, soit la somme des éléments qui le précèdent; autrement dit, le premier élément de «t'» correspond à celui de «t», soit 4; puis le 2^e élément correspond au 2^e de «t» plus la valeur précédente de l'accumulateur, soit «0 + 4 = 4»; pour le 3^e, on obtient «7 + 4 = 11», et ainsi de suite.

Conclusion Dans le reste des notes de cours, des acétates du cours ainsi que des autres ressources du cours (comme les examens antérieurs et leur corrigé), on utilise les fonctions du module «Thread» avec un module «Future» défini pour rendre plus intéressante la programmation de fils d'exécution. Il sera possible de légèrement changer le code présenté pour bénéficier des nouvelles fonctions disponibles à partir de la version 5 d'OCaml. À cette fin, voici un tableau qui présente les principales modifications à apporter au code, selon le type d'expressions (ou de fonctions utilisées) :

Matière utilisant Ocaml < 5.0	Traduction vers fonctions d'Ocaml ≥ 5.0
let t1 = Thread.create f v in ...	let t1 = Domain.spawn (fun f () -> f v) in ...
spawn f v	ignore (Domain.spawn (fun () -> f v))
let x = future f v in ... force x	let x = Domain.spawn (fun f () -> f v) in ... Domain.join x
let x = future f v in ... force x	let x = Domainlib.Task.async pool (fun () -> f v) in ... Domainlib.Task.await pool x (* en supposant un pool préalablement défini *)

OCAML 5

Voici l'exemple présenté à la page 173 utilisant les fonctions du module «Domain» :

```
# let f () =
  Thread.delay 10.; pr1 "<f>a_fini_son_traitement_lourd"; 5 * 5;;
val f : unit -> int = <fun>
# let g () =
  pr1 "<g>_termine_rapidement_et_retourne_une_valeur"; 10 * 10;;
val g : unit -> int = <fun>
# let h v1 v2 = v1 + v2;;
val h : int -> int -> int = <fun>

# let df = Domain.spawn f in
  let v = g () in
  let v' = Domain.join df in
  h v v';;
<g> termine rapidement et retourne une valeur
(* 10 secondes plus tard *)
<f> a fini son traitement lourd
- : int = 125
```

Comme souligné dans le tableau précédent, la modification est mineure (voir version utilisant le module «Future», page 173).

Voici un autre exemple, présenté à la page 169, qui utilise la fonction «Domain.spawn» :

```
# pr1 "Debut";
  ignore (Domain.spawn (fun () -> pr1 "Deb1"; pr1 "Fin1"));
  ignore (Domain.spawn (fun () -> pr1 "Deb2"; pr1 "Fin2"));
  pr1 "Fin";;
Debut
Deb1
Fin1
Fin
Deb2
Fin2
- : unit = ()
```

Exécuter ce code une deuxième fois peut engendrer un résultat (affichage) différent, ce qui caractérise la programmation parallèle (à la différence de l'utilisation du module «Thread») :

```
# pr1 "Debut";
  ignore (Domain.spawn (fun () -> pr1 "Deb1"; pr1 "Fin1"));
  ignore (Domain.spawn (fun () -> pr1 "Deb2"; pr1 "Fin2"));
  pr1 "Fin";;
Debut
Deb1
Fin1
Deb2
Fin2
Fin
- : unit = ()
```

7.3.5 Verrous (module *Mutex*)

Dans ce qui suit, nous présentons quelques fonctions définies dans le module *Mutex*. Ce module offre les fonctions nécessaires pour gérer l'exclusion mutuelle (protection de l'accès à une «zone»

partagée). Son utilisation est très simple et se résume aux fonctions suivantes (il faut bien sûr ouvrir le module pour utiliser ces fonctions) :

```
# create;;
- : unit -> Mutex.t = <fun>

# lock;;
- : Mutex.t -> unit = <fun>

# try_lock;;
- : Mutex.t -> bool = <fun>

# unlock;;
- : Mutex.t -> unit = <fun>
```

La fonction «create» permet de créer un verrou ; les fonctions «lock» et «unlock» permettent respectivement d'acquérir et de libérer un verrou. Lorsqu'un fil d'exécution acquiert, à l'aide la fonction «lock», le verrou, tous les autres fils d'exécution qui tentent d'acquérir ce verrou doivent attendre (donc bloquent), jusqu'à ce que le premier fil d'exécution le libère, grâce à la fonction «unlock».

Finalement, «try_lock» agit comme la fonction «lock» mais ne suspend pas le fil d'exécution qui utilise cette fonction. La fonction retourne «false» si le verrou est déjà acquis ; elle retourne «true» et acquiert le verrou, dans le cas contraire.

Voici un exemple d'utilisation de ces fonctions :

```
# let f () =
  let verrou = Mutex.create () in
  let rnum = ref 0. in
  let t = create (fun (f_r,v) -> delay (Random.float 3.);
                    Mutex.lock v;
                    f_r := 90.;
                    Mutex.unlock v)
              (rnum,verrou) in
  let t' = create (fun (f_r,v) -> delay (Random.float 3.);
                  Mutex.lock v;
                  f_r := Random.float 10.;
                  Mutex.unlock v)
            (rnum,verrou) in
  join t;
  join t';
  pr1 (sprintf "Valeur: %.2f" !rnum);;
val f : unit -> unit = <fun>

# f();;
Valeur: 1.83
- : unit = ()

# f();;
Valeur: 90.00
- : unit = ()
```

Notons qu'un problème assez courant est d'oublier de libérer le verrou («Mutex.unlock v»). Afin d'éviter ce type de problème (qui a comme conséquence de bloquer éventuellement tout un programme), on fait appel aux fonctionnels (fonctions en argument d'autres fonctions) afin d'abstraire l'utilisation des verrous :

```
# let with_lock (v : Mutex.t)
  (code : unit -> 'b) : 'b =
  Mutex.lock v;
  let res = code () in
  Mutex.unlock v;
  res;;
val with_lock : Mutex.t -> (unit -> 'b) -> 'b = <fun>
```

La fonction «with_lock» s’assure de prendre le verrou, d’exécuter le code critique, avant de libérer le verrou et de retourner le résultat du code exécuté. Le code précédent s’écrit alors comme suit :

```
# let f () =
  let verrou = Mutex.create () in
  let rnum = ref 0. in
  let t = create (fun (f_r,v) -> delay (Random.float 3.);
    with_lock v (fun () -> f_r := 90.))
    (rnum,verrou) in
  let t' = create (fun (f_r,v) -> delay (Random.float 3.);
    with_lock v (fun () -> f_r := Random.float 10.))
    (rnum,verrou) in
  join t;
  join t';
  pr1 (sprintf "Valeur: %.2f" !rnum);;
val f : unit -> unit = <fun>
```

OCAML 5

Depuis la version 5.1, le module «Mutex» comprend une fonction équivalente à celle proposée ci-dessus («with_lock»); il s’agit de la fonction :

```
# Mutex.protect;;
- : Mutex.t -> (unit -> 'a) -> 'a = <fun>
```

Elle a la même signature que «with_lock» et s’utilise exactement de la même manière.

Interblocage Il y a interblocage lorsqu’un fil d’exécution t_1 attend un verrou m_2 possédé par un fil d’exécution t_2 qui lui-même attend un verrou m_1 possédé par T_1 . Voici un exemple :

```
# let m1 = Mutex.create() and m2 = Mutex.create();;
val m1 : Mutex.t = <abstr>
val m2 : Mutex.t = <abstr>

# let t1 () =
  Mutex.lock m1;
  pr1 "T1-<m1>_verrouillé";
  delay 0.1;
  Mutex.lock m2;
  pr1 "T1-<m2>_verrouillé";
  Mutex.unlock m2;
  pr1 "T1-<m2>_déverrouillé";
  Mutex.unlock m1;
  pr1 "T1-<m1>_déverrouillé";;
val t1 : unit -> unit = <fun>
```

```
# let t2 () =
  Mutex.lock m2;
  pr1 "T2-<m2>_verrouillé";
  delay 0.1;
  Mutex.lock m1;
  pr1 "T2-<m1>_verrouillé";
  Mutex.unlock m1;
  pr1 "T2-<m1>_déverrouillé";
  Mutex.unlock m2;
  pr1 "T2-<m2>_déverrouillé";
val t2 : unit -> unit = <fun>

# spawn t1 (); spawn t2 ();
- : unit = ()
```

L'exécution des 2 fils d'exécution se déroule comme suit :

- t_1 ou t_2 débute son exécution; on va supposer que c'est t_1 qui débute (le raisonnement s'appliquerait si c'est t_2 qui aurait débuté);
- il verrouille m_1 , affiche un message et hiberne;
- t_2 débute alors son exécution : il verrouille m_2 , affiche un message et hiberne;
- après son hibernation, t_1 reprend ses activités et tente de verrouiller m_2 ; il bloque car m_2 est verrouillé par t_2 ;
- pendant que t_1 est bloqué, et après son hibernation, t_2 reprend ses activités et tente de verrouiller m_1 ; il bloque car m_1 est verrouillé par t_1 ;
- les 2 fils d'exécution sont tous les deux bloqués puisque chacun des 2 attend que l'autre déverrouille son verrou afin de pouvoir poursuivre : aucun des 2 ne peut donc continuer son exécution : c'est l'interblocage!

Dans la section qui suit, nous présentons les canaux de communication qui permettent aux fils d'exécution de se synchroniser et d'échanger des messages.

7.3.6 Canaux de communication (module *Event*)

Une fois les fils d'exécution créés grâce à la primitive **create** (**spawn**), il est possible de les faire coopérer. Cette coopération se fait par l'intermédiaire de canaux de communication typés. La création d'un canal, permettant de véhiculer des valeurs d'un type donné, se fait au moyen de la primitive **new_channel** :

```
# new_channel ;;
- : unit -> 'a channel = <fun>
```

Cette fonction prend en argument la valeur () et retourne un canal véhiculant des valeurs polymorphes de types «'a». Notons qu'un canal créé à l'aide de cette primitive ne peut être considéré comme pouvant véhiculer des valeurs pouvant appartenir à plusieurs types différents au cours d'une exécution.

Les exemples qui suivent définissent respectivement un canal d'entiers puis un canal de canaux de booléens :

```
# let c_int : int channel = new_channel ();;
val c_int : int channel = <abstr>

# let c_c_bool : bool channel channel = new_channel ();;
val c_c_bool : bool channel channel = <abstr>
```

Pour permettre de s'échanger des valeurs sur les canaux, il est nécessaire d'introduire la notion d'événements.

7.3.7 Évènements

CML introduit un type paramétré «*'a event*» permettant d'utiliser des événements (des abstractions de communication). Plusieurs primitives de base permettant de créer des événements sont disponibles. Dans cette section, nous nous focalisons sur deux d'entre elles :

```
# send;;
- : 'a channel -> 'a -> unit event = <fun>

# receive;;
- : 'a channel -> 'a event = <fun>
```

Ces fonctions créent des événements qui, lorsque synchronisés (activés), effectueront des tentatives de communication consistant respectivement à envoyer et à recevoir une valeur sur un (respectivement à partir d'un) canal de communication; Lors de ces tentatives, elles bloquent si aucun fil d'exécution tente d'effectuer l'opération duale à savoir l'écoute et l'envoi sur ce canal.

Voici un exemple d'utilisation de ces fonctions :

```
# let c = new_channel();;
val c : 'a channel = <abstr>

# send c 99;;
- : unit event = <abstr>

# receive c;;
- : int event = <abstr>
```

Dans cet exemple, l'envoi et l'écoute sur le canal «*c*» n'ont pas eu lieu puisque les fonctions **send** et **receive** n'effectuent aucune communication mais retournent des événements qui correspondent à des communications latentes.

Pour activer une communication latente (un événement), CML offre la primitive **sync** :

```
# sync;;
- : 'a event -> 'a = <fun>
```

Cette fonction synchronise la valeur événement passée en argument. S'il s'agit d'un événement correspondant à une écoute latente sur un canal «*c*» donné, la synchronisation de cet événement active cette écoute, bloque si aucun fil d'exécution n'effectue un envoi de message sur ce même canal, et retourne une valeur reçue, de type «*'a*» dès qu'il y a envoi sur ce canal. De même, s'il s'agit d'un événement correspondant à un envoi latent sur un canal «*c*» donné, la synchronisation de cet événement active cet envoi.

Voici l'exemple précédent repris en utilisant des communications effectives :

```
# spawn (fun () -> sync(send c 99)) ();;
- : unit = ()

# spawn (fun () -> pr1 (sprintf "Valeur_reçu:_%d" (sync(receive c)))) ();;
- : unit = ()
Valeur_reçu: 99
```

Afin de simplifier la présentation des exemples, nous définissons deux autres fonctions utiles :

```
# let send_b c v = sync (send c v);;
val send_b : 'a channel -> 'a -> unit = <fun>

# let recv_b c = sync (receive c);;
val recv_b : 'a channel -> 'a = <fun>
```

L'exemple précédent devient alors :

```
# spawn (fun () -> send_b c 99) ();;
- : unit = ()

# spawn (fun () -> pr1 (sprintf "Valeur_reçu:_%d" (recv_b c))) ();;
- : unit = ()
Valeur_reçu: 99
```

Autres exemples

Dans cette section, nous présentons plusieurs exemples qui utilisent les fonctions de base proposées par CML.

Exemple₁ Commençons par un exemple qui utilise, à la place de la fonction `delay`, des fonctions d'envoi et de réception de canaux :

```
# let ch1, ch2 = new_channel(), new_channel();;
val ch1 : 'a channel = <abstr>
val ch2 : 'a channel = <abstr>

# let t1() = pr1 "Deb<T1>";
            pr1 (sprintf "T1_received_<%s>" (recv_b ch1));
            send_b ch2 "Hello_from_T1";
            pr1 "Fin<T1>"
  and t2() = pr1 "Deb<T2>";
            send_b ch1 "Hello_from_T2";
            pr1 (sprintf "T2_received_<%s>" (recv_b ch2));
            pr1 "Fin<T2>"

  in
    pr1 "Debut";
    spawn t1 ();
    spawn t2 ();
    pr1 "Fin";;
Debut
Fin
- : unit = ()
# Deb<T1>
Deb<T2>
T1 received <Hello from T2>
Fin<T1>
T2 received <Hello from T1>
Fin<T2>
```

Exemple₂ Dans CML, Les fonctions de communication sont par défaut synchrones (ou bloquantes). Comme l'illustre la définition suivante, il est assez simple de définir une version asynchrone de ces fonctions :

```
# let async_send ch v = spawn (fun () -> send_b ch v) ();;
val async_send : 'a channel -> 'a -> unit = <fun>
```

Voici un exemple utilisant la version asynchrone de la fonction `send_b` :

```
# let ch = new_channel();;
val ch : 'a channel = <abstr>
```

```
# let t1 () = pr1 "Deb1";
              async_send ch 1;
              async_send ch 2;
              pr1 "Fin1"

in
  pr1 "Debut";
  spawn t1 ();
  if (recv_b ch) = 1 then pr1 "Then-1" else pr1 "Else-1";
  if (recv_b ch) = 1 then pr1 "Then-2" else pr1 "Else-2";
  pr1 "Fin";;
Debut
Deb1
Fin1
Then-1
Else-2
Fin
- : unit = ()
```

En utilisant la version synchrone de la fonction `send_b`, le résultat est naturellement différent :

```
# let t1 () = pr1 "Deb1";
              send_b ch 1;
              send_b ch 2;
              pr1 "Fin1"

in
  pr1 "Debut";
  spawn t1 ();
  if (recv_b ch) = 1 then pr1 "Then-1" else pr1 "Else-1";
  if (recv_b ch) = 1 then pr1 "Then-2" else pr1 "Else-2";
  pr1 "Fin";;
Debut
Deb1
Then-1
Fin1
Else-2
Fin
- : unit = ()
```

Exemple₃ L'exemple qui suit illustre le fait qu'il est possible d'envoyer sur des canaux des valeurs quelconques, y compris des fonctions (puisque valeurs de première classe dont le type unifie naturellement le type polymorphe «'a'») :

```
# let c = new_channel() and d = new_channel() in
  spawn (fun () -> send_b c (fun x -> send_b d x)) ();
  spawn (fun () -> (recv_b c) 3) ();
  (recv_b d) + 1;;
- : int = 4
```

Serveur echo L'exemple qui suit définit un fil d'exécution «echo» dont la fonction est d'afficher un message reçu sur un canal défini («c80»). Ce fil d'exécution boucle toujours jusqu'à la réception du message "quit" lui indiquant d'arrêter son activité :

```
# let c80 = new_channel();;
val c80 : 'a channel = <abstr>
```

```
# let echo() =
  let rec boucle () =
    let msg = recv_b c80 in
    if msg = "quit" then
      begin
        pr1 "Canal_c80_not_available_for_echo!";
        raise Exit
      end
    else
      begin
        pr1 (sprintf "Message-%s-received!" msg);
        boucle()
      end
    end
  in
  spawn boucle ();
  pr1 "canal_c80_ready_for_echo...";
val echo : unit -> unit = <fun>
```

OCAML 5

Si on veut utiliser cette fonction avec le module «Domain», il suffit de remplacer «spawn boucle ();» par «ignore(Domain.spawn boucle);».

Cette fonction met en arrière plan une fonction «boucle» qui place dans une variable «x» la valeur reçue sur le canal «c80». Si cette valeur est égale à "quit", alors la fonction «boucle» affiche un message et appelle la fonction `exit` pour arrêter le fil d'exécution. Dans le cas où la valeur est différente de cette chaîne de caractères, alors la fonction affiche cette valeur et réitère son traitement.

Voici quelques exemples qui illustrent l'utilisation de ce serveur «echo»; il faut bien sûr, en premier, lancer le serveur :

```
# echo();;
canal c80 ready for echo ...
- : unit = ()

# send_b c80 "Hello";;
- : unit = ()
Message-Hello-received!

# send_b c80 "World";;
- : unit = ()
Message-World-received!

# send_b c80 "quit";;
- : unit = ()
Canal c80 not available for echo!
```

Dans les deux premiers envois, le fil d'exécution «echo» se contente d'afficher le message qu'il reçoit. Par contre, dans le troisième exemple, en recevant le message "quit", ce fil d'exécution arrête son exécution et affiche un message en conséquence.

Notons qu'il aurait été possible de créer des événements sans qu'il y ait pour autant de communication :

```
# let e1 = send c80 "Hello";;
val e1 : unit event = <abstr>

# let e2 = send c80 "World";;
val e2 : unit event = <abstr>

# let e3 = send c80 "quit";;
val e3 : unit event = <abstr>
```

Comme l'illustrent ces exemples, aucune communication n'a lieu avec le serveur «echo»³ (pas d'affichage de messages à l'écran) puisqu'il n'y a eu que création d'événements (communications latentes).

Par ailleurs, les événements, étant des valeurs de première classe, peuvent être utilisés comme n'importe quelle valeur : dans l'exemple qui suit, on les met dans une liste :

```
# let liste = [e1;e2;e3];;
val liste : unit event list = [<abstr>; <abstr>; <abstr>]
```

On peut, par la suite, les envoyer sur un canal de communication et à la réception de la liste, les synchroniser pour rendre effective leur communication latente!

```
# let c = new_channel();;
val c : 'a channel = <abstr>

# spawn (fun () -> iter (fun e -> sync e) (recv_b c)) ();;
- : unit = ()

# send_b c liste;;
- : unit = ()
Message-Hello-received!
Message-World-received!
Canal c80 not available for echo!
```

Protocoles de communication On peut étendre davantage l'exemple précédent : en effet, le fait de pouvoir stocker, dans une liste, un ensemble d'étapes de communication, définies sous forme d'événements, ouvre la voie à l'envoi de protocoles de communication (protocoles cryptographiques, protocoles d'authentification, etc.) sur des canaux de communication. Ceci permettrait, par exemple, de partager entre plusieurs agents (fils d'exécution) les étapes d'un même protocole. Dans la section 7.3.8 (voir, plus précisément, la page 211), nous présentons un exemple illustrant ces faits.

Encryption : Dans l'exemple qui suit, on définit une fonction «encrypte» (assez simple), une fonction «serveurPuissant» paramétrée par un paramètre «c» (canal) et une fonction «clientLeger» :

3. Dans ce cas, on suppose bien sûr qu'il est démarré.

```
# let encrypte m =
  implode (map (fun c -> Char.chr ((int_of_char c) + 5)) (explode m));;
val encrypte : string -> string = <fun>

# let serveurPuissant c =
  let f1() = let f,x,d = recv_b c in send_b d (f x) in
    spawn f1 ();;
val serveurPuissant : (('a -> 'b) * 'a * 'b channel) channel -> unit = <fun>

# let clientLeger () =
  let c,d = new_channel(),new_channel() in
  let f2() = send_b c (encrypte,"mon_texte_secret",d);
    pr1 (sprintf "%s-" (recv_b d))
  in
    spawn f2 ();
    serveurPuissant c;;
val clientLeger : unit -> unit = <fun>
```

La fonction «serveurPuissant» prend en argument un canal de communication «c», définit une fonction «f1» puis lance en arrière-plan cette fonction. La fonction «f1» reçoit, sur le canal «c» (voir le type de ce canal dans la signature de la fonction «serveurPuissant»), un triplet comprenant une fonction polymorphe «f», une valeur «x» et un canal «d». Par la suite, elle envoie sur le canal «d» le résultat de l'application de «f» à «x».

La fonction «clientLeger» définit deux canaux «c» et «d» et une fonction «f2», lance en arrière-plan cette fonction, et invoque la fonction «serveurPuissant» avec comme paramètre le canal «c». La fonction «f2» envoie sur le canal «c», la fonction «encrypte», une chaîne de caractères et le canal «d». Par la suite, la fonction «f2» attend sur le canal «d» un message qu'elle affiche à l'écran.

En résumé, la fonction «clientLeger» lorsqu'exécutée, aura pour effet de préciser à «serveurPuissant» un canal sur lequel «clientLeger» pourra envoyer une valeur, une fonction et un autre canal et recevoir, sur ce dernier, le résultat de l'application de la fonction à la valeur. Autrement dit, «clientLeger» délègue à «serveurPuissant» l'exécution d'une fonction à une valeur.

Voici un exemple d'exécution :

```
# clientLeger();;
- : unit = ()
- rts%yj} yj%xjhwjy -
```

Serveur d'entiers : La fonction «s_int» qui suit retourne un canal «contenant» une série d'entiers naturels (agit comme un serveur d'entiers).

Le paramètre de cette fonction permet de préciser la valeur de départ de cette série. Les écoutes (recv_b) successives sur le canal permettent de retourner progressivement la série d'entiers. Par exemple, «s_int(1)» retourne un canal «contenant» la série d'entiers 1,2,3,4, ...

```
# let s_int x =
  let c = new_channel() in
  let rec loop y = send_b c y; loop(y+1) in
    spawn loop x;
    c;;
val s_int : int -> int channel = <fun>
```

Cette fonction peut être utilisée pour gérer un compteur de manière concurrente :

```
# let n = s_int 1 in
  let pr m v = pr1 (sprintf m v) in
    spawn (fun () -> delay 1.; pr "<T1>_Valeur_obtenue_:_%d" (recv_b n)) ();
    spawn (fun () -> delay 1.; pr "<T2>_Valeur_obtenue_:_%d" (recv_b n)) ();
    delay 1.;
    pr "<T1>_Valeur_obtenue_:_%d" (recv_b n);;
<T1> Valeur obtenue : 1
<T1> Valeur obtenue : 3
- : unit = ()
<T2> Valeur obtenue : 2
```

Grâce à la synchronisation offerte par les fonctions de communication bloquante, `send_b` et `recv_b`, on a la garantie qu'un seul fil d'exécution à la fois pourra accéder à la valeur courante du compteur géré par le serveur d'entiers.

Par ailleurs, le serveur d'entiers «`s_int`» illustre le fait que les canaux de communication peuvent être utilisés pour représenter des flots infinis d'éléments (section 3.1.8, page 113). En effet, grâce aux fils d'exécution, aux canaux de communication et la synchronisation, il est possible de retrouver la notion de «flot infini»; la communication, étant bloquante, simulera parfaitement le rôle de l'évaluation paresseuse (les valeurs seront récupérées au besoin). Voici une version de la fonction «`stream_build`» (ainsi qu'une version de la fonction «`stream_peak`»), présentée à la section 3.1.8, qui retourne en résultat un canal sur lequel on peut récupérer un flot infini d'éléments :

```
# let rec chan_stream_build f x =
  let c = new_channel() in
    let rec loop x = send_b c x; loop (f x) in
      spawn loop x;
    c;;
val chan_stream_build : ('a -> 'a) -> 'a -> 'a channel = <fun>

# let rec chan_stream_peak n c = match n with
| _ when n <= 0 -> []
| _ -> let v = recv_b c in v::(chan_stream_peak (n-1) c);;
val chan_stream_peak : int -> 'a channel -> 'a list = <fun>
```

Il est intéressant de comparer la signature des fonctions «`chan_stream_build`» et «`chan_stream_peak`» avec celle des fonctions «`stream_build`» et «`stream_peak`» : le type *'a stream* est tout simplement remplacé par le type *'a channel*.

Voici un exemple d'utilisation de telles fonctions :

```
# let c_int = chan_stream_build succ 0;;
val c_int : int channel = <abstr>

# chan_stream_peak 10 c_int;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Dans ce cas, la fonction «`chan_stream_build`» agit comme une généralisation de la fonction «`s_int`» : le canal «`c_int`» véhiculera un flot infini d'entiers positifs commençant par 0. Par la suite, on invoque la fonction «`chan_stream_peak`» pour récupérer les 10 premiers entiers.

Fonction «`filter`» : L'exemple qui suit propose une implantation concurrente de la fonction «`filter`». Rappelons que cette fonction consiste à retourner tous les éléments d'une liste qui respecte un prédicat donné :

```
# let filterc p c =
  let c' = new_channel() in
  let rec loop () =
    let v = recv_b c in
    if p v then send_b c' v else ();
    loop()
  in
  spawn loop ();
  c';;
val filterc : ('a -> bool) -> 'a channel -> 'a channel = <fun>
```

À la différence de la version séquentielle, «filter», cette fonction récupère sur un canal les éléments à traiter, et retourne sur un autre canal (canal retourné en résultat de la fonction), les éléments respectant le prédicat. Voici un exemple d'utilisation de cette fonction :

```
# let c = new_channel();;
val c : 'a channel = <abstr>

# let c' = filterc (fun x -> x mod 2 = 0) c;;
val c' : int channel = <abstr>

# spawn (fun () -> iter (send_b c) [1;2;3;4;5;6;7;8;9]) ();;
- : unit = ()

# recv_b c';;
- : int = 2

# recv_b c';;
- : int = 4
```

Dans cet exemple, on définit un canal «c» sur lequel la fonction «filterc» recevra les valeurs à filtrer; par la suite, on invoque cette fonction en lui passant comme argument le canal «c» ainsi qu'un prédicat qui teste si un entier est pair. On récupère le résultat de la fonction, le canal «c'», à partir duquel on pourra récupérer les éléments qui respectent le prédicat, soit les éléments pairs. Puis, on lance en arrière plan un fil d'exécution qui aura pour tâche d'envoyer (à «filterc») sur le canal «c» une liste d'éléments (de 1 à 9). Par la suite, on peut récupérer, un à un, sur le canal «c'» les entiers pairs de cette liste.

Par ailleurs, en utilisant les fonctions «chan_stream_build» et «chan_stream_peak» (page 194), il est possible de peaufiner davantage l'exemple précédent :

```
# let c_int = chan_stream_build succ 0;;
val c_int : int channel = <abstr>

# let c_pairs = filterc (fun x -> x mod 2 = 0) c_int;;
val c_pairs : int channel = <abstr>

# chan_stream_peak 20 c_pairs;;
- : int list =
[0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20; 22; 24; 26; 28; 30; 32; 34; 36; 38]
```

Dans ce cas, l'appel à la fonction «chan_stream_build» retourne un canal «c_int» qui véhicule un flot infini d'entiers positifs commençant par 0. Par la suite, on invoque la fonction «filterc» en lui transmettant, en argument, ce canal. Et à partir du canal récupéré suite à cette invocation, le canal «c_pairs», et grâce à la fonction «chan_stream_peak», on peut aisément récupérer les 20 premiers entiers pairs.

Réducteur concurrent d'arbres binaires L'exemple qui suit propose une implantation concurrente d'un réducteur d'arbres binaires (version concurrente du réducteur présenté à la section 2.6.7, page 59). Rappelons que ce réducteur consiste à appliquer un traitement (fonction «z») aux feuilles d'un arbre donné, et un autre traitement (fonction «c») aux nœuds (il équivaut les fonctions «fold_left» et «fold_right» définies sur les listes).

```

1 # type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre;;
2 type 'a arbre = Feuille of 'a | Noeud of 'a arbre * 'a arbre
3
4 # let c_fold_arb z c =
5   let rec f a = match a with
6     | Feuille x -> z x
7     | Noeud(g,d) ->
8       let canal = new_channel() in
9       let _ = spawn (fun () -> send_b canal (f g)) () in
10      let r1 = f d in
11      let r2 = recv_b canal in
12      c r1 r2
13   in
14   f;;
15 val c_fold_arb : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>

```

Quelques remarques s'imposent au sujet de cette implantation :

1. Comme on peut le constater, le type de la fonction «c_fold_arb» coïncide avec celui de «fold_arb» (version séquentielle).
2. À chaque nœud, le traitement est distribué entre le fil d'exécution courant et un nouveau fil d'exécution créé à la volée : l'un traite le nœud droit de l'arbre pendant que l'autre traite, de manière concurrente, le nœud gauche. Une fois le résultat du traitement du nœud droit obtenu, on synchronise, grâce à la fonction bloquante «recv_b», sur le résultat du fil d'exécution responsable du traitement du côté gauche.

Afin d'illustrer cette fonction à travers des exemples, nous définissons un arbre «a» de hauteur 4 et contenant 8 entiers aléatoires compris entre 0 et 100000 :

```

# Random.self_init ();;
- : unit = ()

# let rec creer_a p =
  if p = 1 then Feuille(Random.int 100000)
  else Noeud(creer_a (p-1), creer_a (p-1));;
val creer_a : int -> int arbre = <fun>

# let a = creer_a 4;;
val a : int arbre =
  Noeud
    (Noeud (Noeud (Feuille 45076, Feuille 2634),
      Noeud (Feuille 31115, Feuille 38722)),
    Noeud (Noeud (Feuille 28594, Feuille 84307),
      Noeud (Feuille 62250, Feuille 85878)))

```

Voici quelques exemples d'utilisation de la fonction «c_fold_arb» :

```

1 # c_fold_arb (fun x -> 1) (+) a;;
2 - : int = 8
3
4 # c_fold_arb (fun x -> [x]) (@) a;;
5 - : int list = [85878; 62250; 84307; 28594; 38722; 31115; 2634; 45076]
6
7 # c_fold_arb (fun x -> if x mod 2 = 0 then 1 else 0) (+) a;;
8 - : int = 6
9
10 # c_fold_arb (fun x -> [x]) (merge compare) a;;
11 - : int list = [2634; 28594; 31115; 38722; 45076; 62250; 84307; 85878]

```

Le premier exemple permet de compter le nombre de feuilles dans l'arbre, soit 8. L'exemple d'après permet de lister les éléments de l'arbre, soit les éléments qui se trouvent dans les feuilles. Le troisième exemple retourne le nombre de valeurs entières paires dans l'arbre. Quant au dernier exemple, il utilise les fonctions «merge» et «compare», définies dans le module «List», afin de lister, de manière triée, les éléments de l'arbre.

Par ailleurs, à l'instar de la version séquentielle de la fonction («fold_arb»), il est possible de définir des fonctions utiles à partir de la fonction «c_fold_arb» :

```

# let hauteur a = c_fold_arb (fun x -> 1) (fun x y -> (max x y) + 1) a;;
  val hauteur : 'a arbre -> int = <fun>

# let nbr_feuilles a = c_fold_arb (fun x -> 1) (+) a;;
  val nbr_feuilles : 'a arbre -> int = <fun>

# let liste_feuilles a = c_fold_arb (fun x -> [x]) (@) a;;
  val liste_feuilles : 'a arbre -> 'a list = <fun>

# hauteur a;;
- : int = 4

# nbr_feuilles a;;
- : int = 8

# liste_feuilles a;;
- : int list = [85878; 62250; 84307; 28594; 38722; 31115; 2634; 45076]

```

Pour terminer, on peut utiliser la fonction «timeRun» afin de comparer le temps d'exécution de la fonction «c_fold_arb» par rapport à sa version séquentielle. À cette fin, on définit une fonction «est_premier» qui permet de tester si un nombre est premier ou pas et qui utilise la fonction «delay» pour simuler un calcul lent :

```

# let est_premier n =
  let ns = int_of_float(sqrt(float(n))) in
  let rec est_premierUtil m =
    (m > ns || (n mod m <> 0 && (est_premierUtil (m+1))))
  in
  delay 0.1; est_premierUtil 2;;
  val est_premier : int -> bool = <fun>

# timeRun est_premier 5;;
- : bool * float = (true, 0.092999935150146484)

# timeRun est_premier 23;;
- : bool * float = (true, 0.1100001335144043)

```

Voici des exemples qui permettent de comparer les deux fonctions «c_fold_arb» et «fold_arb» :

```
# timeRun (c_fold_arb (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (1, 0.092999935150146484)

# timeRun (fold_arb (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (1, 0.79600000381469727)
```

La différence est plus notable lorsqu'on considère des arbres plus importants (par exemple, en considérant un arbre de taille 8 et contenant 256 éléments) :

```
# let a = creer_a 8;;
val a : int arbre =
  Noeud
    (Noeud
      ...))

# timeRun (c_fold_arb (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (18, 0.12400007247924805)

# timeRun (fold_arb (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (18, 12.808000087738037)
```

Ainsi, les deux fonctions retournent le même résultat, à savoir qu'il y a 18 nombres premiers parmi les 256 éléments de l'arbre; sauf que la fonction «c_fold_arb» a requis 0.12 sec. pour effectuer son calcul, alors que la fonction «fold_arb» aura requis 12.8 sec.

Par ailleurs, notons que la fonction «c_fold_arb» peut bénéficier de la notion de «future» (voir section 7.3.3, page 172) pour en simplifier la définition (qui devient plus expressive) :

```
# let c_fold_arb' z c =
  let rec f a = match a with
  | Feuille x -> z x
  | Noeud(g,d) ->
      let x = future f g in
      let r2 = f d in
      let r1 = force x in
      c r1 r2
  in
  f;;
val c_fold_arb' : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>

# timeRun (c_fold_arb' (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (9, 0.1084737777099609)
```

Version plus adéquate de la fonction «c_fold_arb» La précédente version de la fonction ne tient pas compte du nombre élevé de fils d'exécution qui pourraient être créés (en considérant la profondeur de l'arbre passé en argument) relativement à la capacité de la machine hôte, ainsi que du nombre de processeurs (cœurs) disponibles, dans une optique de chercher le maximum de performance.

L'exemple d'implantation qui suit permet de traiter les problèmes soulevés :

```
# let n_cores = Domain.recommended_domain_count();;
val n_cores : int = 16

# let c_fold_arb z c =
  let rec f depth a = match a with
    | _ when 2. ** float_of_int(depth) >= float_of_int(n_cores)
      -> fold_arb z c a
    | Feuille x -> z x
    | Noeud(g,d) ->
      let canal = new_channel() in
      let _ = spawn (fun () -> send_b canal (f (depth+1) g)) () in
      let r1 = f (depth+1) d in
      let r2 = recv_b canal in
      c r1 r2
  in
  f 0;;
val c_fold_arb : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>

# timeRun (c_fold_arb (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (9, 0.20619845390319824)
```

Évidemment, il est possible d'en déduire une version utilisant les «future» :

```
# let c_fold_arb' z c =
  let rec f depth a = match a with
    | _ when 2. ** float_of_int(depth) >= float_of_int(n_cores)
      -> fold_arb z c a
    | Feuille x -> z x
    | Noeud(g,d) ->
      let x = future (f (depth+1)) g in
      let r2 = f (depth+1) d in
      let r1 = force x in
      c r1 r2
  in
  f 0;;
val c_fold_arb' : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>

# timeRun (c_fold_arb' (fun x -> if est_premier x then 1 else 0) (+)) a;;
- : int * float = (9, 0.20913290977478027)
```

OCAML 5

En utilisant les fonctions du module «Domain», on obtient ceci : module «Domain» :

```
# let c_fold_arb' z c =
  let rec f depth a = match a with
    | _ when 2. ** float_of_int(depth) >= float_of_int(n_cores)
      -> fold_arb z c a
    | Feuille x -> z x
    | Noeud(g,d) ->
      let x = Domain.spawn (fun () -> f (depth+1) g) in
      let r2 = f (depth+1) d in
      let r1 = Domain.join x in
      c r1 r2
  in
  f 0;;
val c_fold_arb' : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>
```

On peut aussi utiliser les fonctions du module «Domainlib.Task» ce qui simplifie davantage le code :

```
# let c_fold_arb' z c a =
  let num_domains = Domain.recommended_domain_count () - 1 in
  let pool = T.setup_pool ~num_domains ~name:"pool" () in
  let rec f a = match a with
    | Feuille x -> z x
    | Noeud(g,d) ->
      let a = T.async pool (fun _ -> f g) in
      let b = T.async pool (fun _ -> f d) in
      c (T.await pool a) (T.await pool b)
  in
  let res = T.run pool (fun _ -> f a) in
  T.teardown_pool pool;
  res;;
val c_fold_arb' : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a arbre -> 'b = <fun>
```

7.3.8 Autres fonctions manipulant les événements

CML offre la possibilité de définir des événements à partir d'autres événements par «*wrapping*» (on parle alors de composition d'événements, par analogie à la composition de fonctions). Le *wrapping* est la définition d'actions (fonctions) devant être activées (appelées) suite à des synchronisations sur événements. Cela se fait au moyen de la primitive **wrap** :

```
# wrap;;
- : 'a event -> ('a -> 'b) -> 'b event = <fun>
```

Comme le souligne son type, cette fonction prend, en argument, un événement et une fonction, et retourne, en résultat, un nouvel événement. Par exemple, appliquée à un événement de type «*int event*» et à une fonction de type «*int -> bool*», la fonction **wrap** retourne un autre événement de type «*bool event*». Cet événement, lorsque synchronisé, retournera donc un booléen qui est le résultat de l'application de la fonction à la valeur résultante de la synchronisation du premier événement passé en argument.

L'exemple qui suit définit un nouvel événement en associant une fonction, qui retourne le carré d'un entier, à une réception latente d'une valeur entière sur un canal :

```
# let c = new_channel();;
val c : 'a channel = <abstr>

# let e = wrap (receive c) (fun x -> x * x);;
val e : int event = <abstr>
```

En envoyant un nombre sur le canal «*c*», et en synchronisant cet envoi avec l'événement «*e*», on reçoit le carré du nombre envoyé :

```
# spawn (fun() -> send_b c 5) ();;
- : unit = ()

# sync e;;
- : int = 25
```

Par ailleurs, à partir d'une liste d'événements, il est possible d'en créer un nouveau. Un opérateur de choix (externe) entre événements est fourni. Il s'agit de la primitive **choose** :

```
# choose;;
- : 'a event list -> 'a event = <fun>
```

Lorsque l'événement résultant de la primitive **choose** est synchronisée (activée à l'aide de la primitive **sync**), une tentative d'activation de tous les événements de la liste passée en argument de **choose** est effectuée; le premier événement qui réussit à se synchroniser avec une communication duale (envoi avec réception, réception avec envoi), sera activé et le résultat de cette synchronisation est retourné en résultat. Toutes les autres tentatives d'activations sont annulées.

La version «active» de ce choix se nomme **select**.

```
# select;;
- : 'a event list -> 'a = <fun>
```

Voici les relations qui existent entre les fonctions **sync**, **wrap**, **choose** et **select**⁴ :

```
select = sync o choose
wrap (wrap ev g) f = wrap ev (f o g)
wrap (choose [ev1;ev2]) f = choose [wrap ev1 f;wrap ev2 f]
```

Dans les paragraphes qui suivent, nous présentons plusieurs exemples utilisant des fonctions de composition d'événements.

Serveur «multicast» L'exemple qui suit simule le phénomène de «multicasting» :

```
# let notifyAll (liste, v) =
  iter (fun c -> spawn (fun () -> send_b c v) ()) liste;;
val notifyAll : 'a channel list * 'a -> unit = <fun>
```

La fonction «notifyAll» prend en argument une liste de canaux et une valeur «v». Elle crée, à travers la fonction **iter**, pour chaque élément (canal) de la liste, un fil d'exécution dont le rôle est d'envoyer la valeur «v» sur le canal. Le fait de créer plusieurs fils d'exécution garantit le non-blocage de la fonction notifyAll. En effet, il aurait été possible de définir cette fonction plus simplement :

```
# let notifyAll_1 (liste, v) =
  iter (fun c -> send_b c v) liste;;
val notifyAll_1 : 'a channel list * 'a -> unit = <fun>
```

Cependant, avec cette version, il suffit qu'un des **send_b** bloque (pas de réception correspondante sur le canal, donc pas de synchronisation) pour que toute la fonction notifyAll soit bloquée, et par conséquent, l'appelant de celle-ci le soit à son tour ! Une alternative à la création, nettement gourmande en ressources, de plusieurs fils d'exécution, est l'utilisation d'une version non bloquante de la fonction d'envoi : il s'agit de la fonction **poll** (voir page 213) :

```
# let notifyAll_2 (liste, v) =
  iter (fun c -> ignore(poll(send c v))) liste;;
val notifyAll : 'a channel list * 'a -> unit = <fun>
```

Plus précisément, la fonction **poll** agit comme la fonction **sync** mais n'est pas bloquante; autrement dit, lorsqu'elle est appliquée à un événement, si ce dernier est prêt, elle retourne **Some** de la valeur résultante; sinon, elle retourne la valeur **None**. Étant donné que la fonction **iter** requiert que la fonction qui lui est passée en argument est comme type de retour unit, on utilise la fonction **ignore** pour éviter un message d'avertissement.

4. «o» est l'opérateur de composition de fonctions.

Voici la définition de la fonction «multicast» :

```
# let multicast () =
  let c,d = new_channel(),new_channel() in
  let rec server liste =
    server (select [ wrap (receive c) (fun x -> x::liste);
                      wrap (receive d) (fun y -> notifyAll (liste, y); liste)
                    ])
  in
  spawn server [];
  c,d;;
val multicast : unit -> 'a channel channel * 'a channel = <fun>
```

La fonction «multicast» définit deux canaux, «c» et «d», une fonction récursive «server» ainsi qu'un fil d'exécution, et retourne la paire de canaux. Le fil d'exécution consiste à exécuter en arrière-plan la fonction «server». Cette fonction récursive prend en argument une liste de canaux, initialement vide, et attend, d'une manière non déterministe, qu'une valeur soit reçue soit sur le canal «c», soit sur le canal «d». Dans le cas où une valeur est reçue sur «c», elle l'ajoute à la liste de canaux et boucle récursivement sur elle-même avec cette nouvelle liste. Dans le cas où une valeur est reçue sur le canal «d», elle invoque alors la fonction «notifyAll» avec comme argument la liste des canaux et la valeur reçue : ceci entraînera l'envoi de cette valeur sur tous les canaux présents dans la liste. Une fois cette tâche réalisée, elle boucle récursivement sur elle-même avec la même liste de canaux.

Autrement dit, le serveur offre deux canaux : le premier pour permettre aux clients (autres fils d'exécution interagissant avec le serveur) de s'enregistrer au service (ajout de leurs canaux (privés) dans une liste de canaux), et le deuxième permettant de recevoir une donnée qu'on enverra alors à tous les abonnés du service.

Finalement, voici un code exploitant le service offert, à travers le serveur, par la fonction «multicast» :

```
# let main() =
  let a,b = new_channel(),new_channel() in
  let pr i s = printf "Valeur obtenue par T%d: %s" i s in
  let _ = spawn (fun() -> pr 1 (recv_b a)) () in
  let _ = spawn (fun() -> pr 2 (recv_b b)) () in
  let c, d = multicast() in
  send_b c a;
  send_b c b;
  send_b d "Welcome to our service!";
val main : unit -> unit = <fun>
```

La fonction «main» définit deux canaux, «a» et «b», et deux fils d'exécution «T1» et «T2». Ces deux fils d'exécution ont comme rôle d'écouter respectivement sur les canaux «a» et «b» et d'afficher le message reçu. Par la suite, la fonction «main» invoque la fonction «multicast» et récupère le résultat de cette dernière dans les variables «c» et «d» (le premier canal servira à s'enregistrer au service, tandis que le deuxième servira à envoyer un message à tous les abonnés au service). Puis elle envoie successivement sur le canal «c» les canaux «a» et «b», et envoie sur le canal «d» la chaîne de caractères "Welcome to our service!". Autrement dit, la fonction «main» enregistre les fils d'exécution «T1» et «T2» en envoyant sur le canal «c» leurs canaux respectifs puis envoie sur le canal «d» un message de bienvenue aux abonnés!

```
# main();;
- : unit = ()
Valeur obtenue par T2: Welcome to our service !
Valeur obtenue par T1: Welcome to our service !
```

Cellule «bufferisée» La fonction qui suit représente une cellule «bufferisée», c’est-à-dire une cellule qui peut stocker plusieurs valeurs et retourner ces valeurs en respectant leur ordre d’entrée (FIFO). Elle offre la possibilité de faire communiquer des fils d’exécution de manière asynchrone.

```
# let buffer () =
  let inCh = new_channel () and outCh = new_channel () in
  let rec f (l1,l2) = match (l1,l2) with
    | ([],[]) -> f ([recv_b inCh],[])
    | ((x::r),l2) ->
      select [ wrap (receive inCh) (fun y -> f(l1,y::l2));
               wrap (send outCh x) (fun () -> f(r,l2))
            ]
  | ([],l2) -> f (rev l2,[])
  in
  spawn f ([],[]);
  (inCh,outCh);;
val buffer : unit -> 'a channel * 'a channel = <fun>
```

Cette fonction retourne deux canaux de communication, «inCh» et «outCh», et crée un fil d’exécution correspondant à un serveur dont le rôle est de stocker dans une liste «l2» toutes les valeurs reçues sur le canal «inCh» et de stocker dans une liste «l1» toutes les valeurs à envoyer sur le canal «outCh».

Voici un exemple illustrant l’utilisation de ce «buffer concurrent» :

```
# let c1,c2 = buffer () in
  let t1 = fun () -> send_b c1 1; pr1 "T1_a_transmis_1"; delay 0.1;
                    send_b c1 3; pr1 "T1_a_transmis_3"; delay 0.1;
                    send_b c1 5; pr1 "T1_a_transmis_5"; delay 0.1;
                    pr1 "T1_termine!"
  and t2 = fun () -> send_b c1 2; pr1 "T2_a_transmis_2"; delay 0.1;
                    send_b c1 4; pr1 "T2_a_transmis_4"; delay 0.1;
                    pr1 "T2_termine!"
  and t3 = fun () -> pr1 (sprintf "T3_obtient:%d" (recv_b c2));
                    pr1 (sprintf "T3_obtient:%d" (recv_b c2));
                    pr1 (sprintf "T3_obtient:%d" (recv_b c2));
                    pr1 (sprintf "T3_obtient:%d" (recv_b c2));
                    pr1 (sprintf "T3_obtient:%d" (recv_b c2));
                    pr1 "T3_termine!"
  in
  spawn t1 ();
  spawn t2 ();
  spawn t3 ();;
- : unit = ()
#T1 a transmis 1
T2 a transmis 2
T3 obtient: 1
T3 obtient: 2
T2 a transmis 4
T1 a transmis 3
T3 obtient: 4
T3 obtient: 3
T1 a transmis 5
T3 obtient: 5
T2 termine!
T3 termine!
T1 termine!
```

Dans cet exemple, les deux fils d’exécution «T1» et «T2» communiquent de manière asynchrone avec le fil d’exécution «T3».

Multiplexeur L'exemple qui suit crée ce qu'on appelle un multiplexeur.

```
# let multiplexeur l =
  let outCh = new_channel() in
  let rec f () =
    let l' = map (fun(c,n) -> wrap (receive c) (fun x -> (x,n))) l in
    let v = select l' in
    send_b outCh v;
    f()
  in
  spawn f ();
  outCh;;
val multiplexeur : ('a channel * 'b) list -> ('a * 'b) channel =
```

La fonction «multiplexeur» crée un fil d'exécution qui écoute sur plusieurs canaux à la fois et qui retransmet les données sur un nouveau canal de sortie «outCh». Les canaux d'entrée sont fournis à «multiplexeur» via le paramètre «l» qui contient une liste de couples canal-numéro. Lorsque le fil d'exécution reçoit une donnée sur un des canaux, disons le canal numéro «i», il emballe, dans un couple, la donnée avec «i». Comme la fonction «f» est récursive, le fil d'exécution continue indéfiniment à écouter les canaux d'entrées et à retransmettre sur «outCh» les données reçues.

Le code suivant utilise la fonction «multiplexeur» en créant 3 canaux d'entrée, numérotés de 1 jusqu'à 3, en déclenchant la création du multiplexeur, en émettant 3004 sur le deuxième canal et en recevant la réponse «(3004, 2)» sur le canal de sortie. Le couple en réponse indique que 3004 avait été reçu sur le canal d'entrée numéro 2.

```
# let c1,c2,c3 = new_channel(),new_channel(),new_channel() in
  let c = multiplexeur [(c1,1);(c2,2);(c3,3)] in
  send_b c2 3004;
  recv_b c;;
- : int * int = (3004, 2)
```

Tri L'exemple qui suit présente une version concurrente d'un algorithme de tri (*quicksort*). Cet exemple comprend deux fonctions. La première fonction, «split» divise une liste «l» d'entiers, donnée en argument, en 3 listes «l1», «l2» et «l3», et rend le triplet (l1,l2,l3) comme résultat. Si «l» est vide, «l1», «l2» et «l3» le sont aussi, sinon les éléments de «l» sont comparés à l'élément «e» se trouvant en tête de «l», ceux dont la valeur est strictement inférieure à «e» sont placés dans «l1», ceux dont la valeur est égale à «e» sont placés dans «l2» et finalement ceux dont la valeur est strictement supérieure sont placés dans «l3».

```
# let split l = match l with
| [] -> ([],[],[])
| [x] -> ([],[x],[])
| (e::_) ->
  let (l2,l') = partition (fun x -> x = e) l in
  let (l1,l3) = partition (fun x -> x < e) l' in
  (l1,l2,l3);;
val split : 'a list -> 'a list * 'a list * 'a list = <fun>
```

La deuxième fonction «quicksort_aux» correspond au fameux algorithme de tri *quicksort*. «quicksort_aux» s'applique à une liste «l» et un canal «c». À la fin du traitement, «quicksort_aux» envoie la liste «l» triée sur le canal «c». «quicksort_aux» effectue le tri de «l» comme suit :

- Elle divise la liste «l» par la fonction «split» en trois listes «l1», «l2» et «l3».
- Elle crée deux canaux «c1» et «c2».
- Elle lance en parallèle le tri de «l1» et celui de «l3» et ce en invoquant récursivement et en parallèle quicksort_aux(l1,c1) et quicksort_aux(l3,c2). Évidemment, la liste «l1» triée

sera attendue sur le canal «c1» et «l3» triée sera attendue sur «c2». On concatène «l1» triée à «l2» et à «l3» triée et on envoie le résultat qui représente la liste «l» triée sur «c». La réception des résultats des deux tris s'effectue comme suit. Une attente simultanée (non déterministe) s'effectue sur les deux canaux «c1» et «c2» : Si on reçoit sur «c1» d'abord, on concatène le résultat à gauche de «l2» pour obtenir une liste «l'», et on attend sur «c2». À la réception du résultat, on concatène à droite de «l'» et on envoie le résultat sur «c». Si on reçoit sur «c2» d'abord, on concatène le résultat à droite de «l2» pour obtenir «l'» et on attend sur «c1». À la réception, on concatène la liste reçue à gauche de «l'» et on envoie le résultat sur «c». Ainsi, dans tous les cas, on envoie sur «c» la liste «l» triée.

```
# let rec quicksort_aux (l,c) = match l with
| [] -> spawn (fun () -> send_b c []) ()
| l ->
  let (l1,l2,l3) = split l
  and c1 = new_channel ()
  and c2 = new_channel () in
  let _ = spawn (fun () -> quicksort_aux (l1,c1)) ()
  and _ = spawn (fun () -> quicksort_aux (l3,c2)) () in
  let rec f (s,l') = match s with
  | "1" -> sync (wrap (receive c1) (fun x -> send_b c (x@l')))
  | "2" -> sync (wrap (receive c2) (fun x -> send_b c (l'@x)))
  | "12" -> select [ wrap (receive c1) (fun x -> f("2", x@l'));
                    wrap (receive c2) (fun x -> f("1",l'@x))
                  ]
  | _ -> () (* pour éviter un message "warning" *)
  in
  spawn f ("12", l2);;
val quicksort_aux : 'a list * 'a list channel -> unit = <fun>
```

Pour plus de commodité, on définit la fonction «quicksort» comme suit :

```
# let quicksort l =
  let c = new_channel() in
  let _ = quicksort_aux (l,c) in
  recv_b c;;
val quicksort : 'a list -> 'a list = <fun>
```

Cette fonction s'applique à une liste «l», crée un canal «c», lance le tri de «l» par l'invocation de «quicksort_aux» appliquée à «l» et à «c», reçoit la liste «l» triée sur le canal «c» et rend cette liste triée comme résultat (le type de la fonction abstrait le fait qu'il s'agit d'une version concurrente de l'algorithme).

Voici un test de la fonction «quicksort» :

```
# quicksort [6;3;1;8;2];;
- : int list = [1; 2; 3; 6; 8]
```

Communication par nom L'exemple qui suit propose une solution permettant à des fils d'exécution de communiquer entre eux en connaissant simplement leur nom respectif (ou tout autre identifiant de fil d'exécution). En effet, à la base, CML permet à deux fils d'exécution de communiquer seulement à partir d'un canal de communication qu'ils partagent au préalable (en général, ce canal est déclaré global aux deux fils d'exécution). Avec les fonctions définies ci-dessous, il n'est pas nécessaire de déclarer globalement des canaux de communication.

Cet exemple nécessite d'abord la définition d'un serveur agissant comme un serveur de noms (un serveur DNS : un tel serveur permet d'associer une adresse «IP» à un nom ; dans notre cas, il s'agira d'associer un canal de communication à un nom). Voici le code de ce serveur :

```
# let serveur_noms () =
  let getCh, regCh = new_channel(), new_channel() in
  let rec f l =
    select [ wrap (receive getCh) (fun (t,c) -> (send_b c (assoc t l); f l));
             wrap (receive regCh) (fun (t,c) -> f ((t,c)::l))
            ]
  in
    spawn f [];
    (getCh, regCh);;
val server_noms : unit -> ('a * 'b channel) channel *
                        ('a * 'b) channel = <fun>
```

La fonction «serveur_noms» crée un serveur (fonction local «f») et deux canaux («getCh» et «regCh») permettant de communiquer avec le serveur. Ce dernier gère une liste de paires (nom de fil d'exécution, canal de communication) précisant pour chaque fil d'exécution enregistré auprès du serveur, le canal de communication permettant de communiquer avec lui. La communication avec le serveur se fait suivant le protocole suivant : lorsqu'un fil d'exécution désire s'enregistrer, il envoie sur le canal «regCh» une paire précisant son nom ainsi qu'un canal de communication privé (unique ou frais). Lorsqu'un fil d'exécution désire connaître le canal privé d'un autre fil d'exécution, il envoie sur le canal «getCh» le nom du fil d'exécution en question ainsi qu'un canal sur lequel il désire recevoir la réponse à sa requête, soit le canal privé du fil d'exécution.

La fonction «serveur_noms» retourne en résultat les deux canaux («getCh» et «regCh») permettant d'interagir avec le serveur.

La fonction qui suit, «enregistre» permet à un fil d'exécution de s'enregistrer.

```
# let enregistre ch_serveur nom thread =
  let ch_frais = new_channel() in
  let _ = send_b ch_serveur (nom, ch_frais) in
  fun () -> thread ch_frais;;
val enregistre : ('a * 'b channel) channel ->
                'a -> ('b channel -> 'c) -> unit -> 'c = <fun>
```

Cette fonction prend comme argument le canal d'un serveur (permettant à un fil d'exécution de s'enregistrer), le nom ainsi que le code d'un fil d'exécution qui désire s'enregistrer. Elle a pour rôle d'enregistrer un fil d'exécution auprès du serveur en créant un canal de communication frais (unique) et en envoyant, au serveur, la paire (nom, canal frais) permettant d'enregistrer le fil d'exécution en question. En résultat, cette fonction retourne une version du fil d'exécution passé en argument dans laquelle ce dernier «connaît» désormais son canal privé.

Pour terminer, voici un code qui permet de tester ces deux fonctions en faisant communiquer deux fils d'exécution, «t1» et «t2», à partir de leur nom :

```
# let main(getCh,regCh) =
  let pr i s = pr1 (sprintf "T%d:_Valeur_obtenue_:_%s" i s) in
  let t1 ch1 =
    let _ = spawn (fun () -> pr 1 (recv_b ch1)) () in
    let ch_local = new_channel() in
    let _ = send_b getCh ("T2",ch_local) in
    let ch2 = recv_b ch_local in
    send_b ch2 "Hello_from_T1"
  and t2 ch2 =
    let _ = spawn (fun () -> pr 2 (recv_b ch2)) () in
    let ch_local = new_channel() in
    let _ = send_b getCh ("T1",ch_local) in
    let ch1 = recv_b ch_local in
    send_b ch1 "Hello_from_T2"
  in
  let t1_enr = enregistre regCh "T1" t1
  and t2_enr = enregistre regCh "T2" t2 in
  spawn t1_enr ();
  spawn t2_enr ();
val main : (string * string channel) channel *
            (string * string channel) channel -> unit = <fun>
```

La fonction «main» prend en argument deux canaux permettant respectivement de faire une requête auprès du serveur de noms afin d’obtenir le canal de communication privé d’un fil d’exécution donné, et de s’enregistrer auprès de ce serveur. Cette fonction crée une fonction «pr» qui permet d’abréger l’utilisation de la fonction «pr1». Par la suite, elle crée deux fils d’exécution «t1» et «t2» qu’on va enregistrer (fonction «enregistre») au serveur de noms en précisant respectivement les canaux avec lesquels on peut communiquer avec eux (c’est la fonction «enregistre» qui assigne un canal frais à chaque fil d’exécution lors de leur enregistrement). Finalement, ces deux fils d’exécution, «t1» et «t2», communiquent avec le serveur pour déterminer respectivement le canal de «t2» et de «t1». Une fois les réponses reçues, ils communiquent entre eux un message de bienvenue :

```
# main(server_noms());;
- : unit = ()
T1: Valeur obtenue : Hello from T2
T2: Valeur obtenue : Hello from T1
```

On appelle la fonction «serveur_noms» pour récupérer les canaux à partir desquels on pourra communiquer avec.

Benchmark La fonction qui suit joue le rôle d’un «benchmark» concurrent. Étant donnés une liste de fonctions (de même type) et un argument, elle applique, en parallèle, ces fonctions à cet argument et retourne l’identifiant de la fonction qui a terminé en premier.

```
# let cbenchmark liste x =
  let liste' = map (fun (id,f) -> (id, f, new_channel())) liste in
  iter (fun (id,f,c) -> spawn (fun () -> f x; send_b c id) ()) liste';
  select (map (fun (_,_,c) -> receive c) liste');;
val cbenchmark : ('a * ('b -> 'c)) list -> 'b -> 'a = <fun>
```

Dans l’exemple qui suit, l’identifiant «f2» est retourné puisque le calcul de la factorielle de 2 termine avant ceux des factorielles de 12, 9 et 11.

```
# let rec f n = if n <= 1 then 1 else (delay 0.1; n * (f (n-1)));
val f : int -> int = <fun>

# let factList = [("f12", fun () -> f 12); ("f9", fun () -> f 9);
  ("f2", fun () -> f 2); ("f11", fun () -> f 11)];;
val factList : (string * (unit -> int)) list =
  [("f12", <fun>); ("f9", <fun>); ("f2", <fun>); ("f11", <fun>)]

# let id = cbenchmark factList () in
  pr1 (sprintf "%s_est_la_plus_rapide!" id);;
f2 est la plus rapide!
- : unit = ()
```

Un exemple plus intéressant serait d'évaluer, en parallèle, plusieurs algorithmes de tri de listes d'entiers. Nous pouvons aussi légèrement changer le code de la fonction «cbenchmark» afin qu'elle retourne non pas l'identifiant du fil d'exécution le plus rapide mais plutôt la liste des identifiants de tous les fils d'exécution ainsi que leur temps d'exécution respectif.

Voici donc la nouvelle version de la fonction «cbenchmark» :

```
# let cbenchmark' liste x =
  let liste' = map (fun (id,f) -> (id, f, new_channel())) liste in
  iter (fun (id,f,c) -> spawn (fun () -> let r,t = timeRun f x in
    send_b c (id,t)) ()) liste';
  map (fun (_,_,c) -> sync (receive c)) liste';;
val cbenchmark' : ('a * ('b -> 'c)) list -> 'b -> ('a * float) list = <fun>
```

Voici un exemple d'utilisation de cette fonction (cet exemple suppose l'implantation de plusieurs fonctions de tri de listes : 'a list -> 'a list) :

```
# let sortAlgoList = [ ("selectSort", selectSort); ("insertSort", insertSort);
  ("mergeSort", mergeSort); ("quickSort", quickSort) ];;
val sortAlgoList : (string * ('a list -> 'a list)) list =
  [("selectSort", <fun>); ("insertSort", <fun>);
  ("mergeSort", <fun>); ("quickSort", <fun>)]

# let l = randomList 10000 (1, 200);;
val l : int list = [188; ... ]

# let l' = cbenchmark' sortAlgoList l in
  pr1_string (result2String l');;
[(selectSort = 30.45) (insertSort = 12.13) (mergeSort = 0.21) (quickSort = 0.46)]
- : unit = ()
```

La fonction «result2String» est utilisée pour afficher les éléments d'une liste de paires; la fonction «randomList» retourne une liste de n entiers aléatoires compris dans un intervalle passé en argument.

Bien sûr, il aurait été possible de retourner l'algorithme le plus rapide en utilisant la première version de la fonction «cbenchmark» :

```
# let id = cbenchmark sortAlgoList l in
  pr1 (sprintf "%s_est_la_plus_rapide!" id);;
mergeSort est la plus rapide!
- : unit = ()
```

Combinaisons (C_n^k) Dans ce qui suit, nous proposons une version concurrente d'une fonction permettant de générer les combinaisons (C_n^k) possibles (sans répétitions) de n éléments parmi k éléments d'une liste donnée en paramètre. Voici la version séquentielle de cette fonction :

```

1 # let rec cnm n l = match n,l with
2   | 0,l -> [[]]
3   | _,[] -> failwith "Liste_trop_petite"
4   | 1,l -> map (fun x -> [x]) l
5   | _,_ when n > length l -> failwith "Liste_trop_petite"
6   | _,_ when n = length l -> [l]
7   | _,h::t -> (map (fun x -> (h::x)) (cnm (n-1) t)) @ (cnm n t);;
8 val cnm : int -> 'a list -> 'a list list = <fun>
9
10 # cnm 3 [1;2;3;4;5];;
11 - : int list list =
12   [[1; 2; 3]; [1; 2; 4]; [1; 2; 5]; [1; 3; 4]; [1; 3; 5]; [1; 4; 5]; [2; 3; 4];
13   [2; 3; 5]; [2; 4; 5]; [3; 4; 5]]

```

Le principal traitement réalisé par cette fonction se situe au niveau de la ligne 7 : deux appels récursifs y sont effectués. Étant donnée une liste $h :: t$, le premier appel récursif (l'appel le plus à droite) consiste à calculer les combinaisons C_n^k du reste de la liste (t). Le deuxième appel consiste à calculer les combinaisons C_{n-1}^k du reste de la liste pour, par la suite, ajouter en entête de toutes les listes résultantes le premier élément de la liste (h).

Voici la version concurrente :

```

1 let rec cnm_c n liste c = match n,liste with
2   | 0,_ -> spawn (fun () -> send_b c [[]]) ()
3   | _,[] -> failwith "Liste_trop_petite"
4   | 1,_ -> spawn (fun () -> send_b c (map (fun x -> [x]) liste)) ()
5   | _,_ when n > length liste -> failwith "Liste_trop_petite"
6   | _,_ when n = length liste -> spawn (fun () -> send_b c [liste]) ()
7   | _,h::t ->
8     let c1,c2 = new_channel(),new_channel() in
9     let _ = spawn (fun () -> cnm_c (n-1) t c1) () in
10    let _ = spawn (fun () -> cnm_c n t c2) () in
11    let l1,l2 = select [ wrap (receive c1) (fun l -> (l, recv_b c2));
12                      wrap (receive c2) (fun l -> (recv_b c1, l))
13                      ]
14    in
15    spawn (fun () -> send_b c ((map (fun x -> (h::x)) l1) @ l2)) ();;
16 val cnm_c : int -> 'a list -> 'a list list channel -> unit = <fun>

```

La fonction «`cnm_c`» permet de récupérer sur un canal de communication passé en argument une liste de combinaisons (C_n^k) possibles (sans répétitions) de n éléments (premier paramètre) parmi k éléments d'une liste donnée en paramètre.

L'idée est de calculer, de manière concurrente, deux listes de résultats qu'on concatène par la suite pour obtenir le résultat final. La création des deux fils d'exécution se fait aux lignes 8 à 10 : chaque fil d'exécution dispose d'un canal qui lui est propre et à partir duquel il est possible de récupérer le résultat. Chaque fil d'exécution a comme tâche de calculer une partie de la solution en faisant un appel récursif à la fonction «`cnm_c`». L'obtention des résultats des deux fils d'exécution se fait de manière concurrente grâce à l'utilisation de la fonction `select`. Par la suite le résultat (concaténation des deux listes obtenues, modulo un ajout d'un élément aux différents éléments de la première liste résultante) est envoyé sur le canal passé en argument de la fonction.

L'utilisation de la fonction `spawn` aux lignes 2, 4, 6 et 15 permet de ne pas bloquer le fil d'exécution courant dans sa tentative d'envoi du résultat sur le canal de communication.

Voici une manière avec laquelle il est possible d'exploiter cette fonction :

```
# let main n liste =
  let c = new_channel() in
  let _ = cnm_c n liste c in
  recv_b c;;
val main : int -> 'a list -> 'a list list = <fun>

# main 3 [1;2;3;4;5];;
- : int list list =
[[1; 2; 3]; [1; 2; 4]; [1; 2; 5]; [1; 3; 4]; [1; 3; 5]; [1; 4; 5]; [2; 3; 4];
 [2; 3; 5]; [2; 4; 5]; [3; 4; 5]]
```

Protocoles de communication Rappelons ce qu'est un protocole⁵ : il s'agit tout simplement d'une séquence d'étapes de communication et de calculs. En particulier, un protocole cryptographique se base sur la cryptographie pour assurer certains objectifs de sécurité. En outre, une étape de communication fait intervenir un agent A , émetteur d'un message, un agent B , récepteur d'un message, et un message m . Elle est en général décrite comme suit :

$$A \longrightarrow B : m$$

Étant donné qu'un protocole est une séquence d'étapes de communication, on associe à chacune de ces étapes un numéro i pour bien distinguer l'ordre des étapes :

$$i \ A \longrightarrow B : m$$

Voici l'exemple du protocole de Woo et Lam qui est un protocole d'authentification unidirectionnel dans lequel seul un agent A a besoin de prouver son identité à un agent B , le tout utilisant un serveur S :

$$\begin{aligned} 1 \quad & A \longrightarrow B : A \\ 2 \quad & B \longrightarrow A : N_b \\ 3 \quad & A \longrightarrow B : \{N_b\}_{k_{as}} \\ 4 \quad & B \longrightarrow S : \{A, \{N_b\}_{k_{as}}\}_{k_{bs}} \\ 5 \quad & S \longrightarrow B : \{N_b\}_{k_{bs}} \end{aligned}$$

Pour votre information, N_b est ce qu'on appelle un nonce (une valeur aléatoire fraîche) ; $\{m\}_k$ désigne un message m encrypté par une clé k ; k_{ab} désigne une clé partagée par les agents A et B .

Pour simplifier l'exemple, faisons abstraction des messages transmis et reçus :

$$\begin{aligned} 1 \quad & A \longrightarrow B : m_1 \\ 2 \quad & B \longrightarrow A : m_2 \\ 3 \quad & A \longrightarrow B : m_3 \\ 4 \quad & B \longrightarrow S : m_4 \\ 5 \quad & S \longrightarrow B : m_5 \end{aligned}$$

On pourrait donc envisager de communiquer à trois agents (fils d'exécution) «A», «B» et «S» les étapes de communication qui les concernent. On suppose que chaque agent dispose d'un canal de communication à partir duquel on peut communiquer avec. Voici donc une implantation d'un tel transfert de protocole d'authentification à différents agents :

— Création de 3 canaux nous permettant de communiquer avec chaque agent :

```
# let c_a, c_b, c_s = new_channel(), new_channel(), new_channel();;
val c_a : 'a channel = <abstr>
val c_b : 'a channel = <abstr>
val c_s : 'a channel = <abstr>
```

5. Cet exemple complète celui présenté à la section 7.3.7 (plus précisément, voir à la page 193).

- Création d'une fonction permettant de générer un comportement typique d'un agent (ce comportement consiste à attendre sur son canal la réception du protocole qu'il doit suivre, puis une fois ce dernier reçu, à effectuer les différentes étapes de communication qui le concerne) :

```
# let agent id c =
  let rec serveur () = iter (fun e -> sync e) (recv_b c);
                        serveur ()
  in
    spawn serveur ();;
  val agent : 'a -> unit event list channel -> unit = <fun>
```

Remarquez le type inféré pour cette fonction !

- Création des trois agents en jeu :

```
# agent "A" c_a; agent "B" c_b; agent "S" c_s;;
- : unit = ()
```

À cette étape, chaque agent est prêt à recevoir sa partie du protocole.

- Création de canaux partagés par les agents (étant donné que les agents *A* et *S* ne communiquent pas entre eux, nous ne définissons pas de canal partagé entre eux) :

```
# let c_ab, c_bs = new_channel(), new_channel();;
  val c_ab : 'a channel = <abstr>
  val c_bs : 'a channel = <abstr>
```

- Création de la liste des étapes relative à chaque agent. Pour simplifier, on associe à chaque étape de communication une action qui permet d'afficher à l'écran une trace des étapes de communication effectuées :

- Partie du protocole qui concerne l'agent *A* ⁶ :

```
# let p_a = [ wrap (send c_ab "m1") (fun () -> pr1 "A->B: m1");
              wrap (receive c_ab) (fun m -> pr "A<-B: %s" m);
              wrap (send c_ab "m3") (fun () -> pr1 "A->B: m3")
            ];;
  val p_a : unit event list = [<abstr>; <abstr>; <abstr>]
```

- Partie du protocole qui concerne l'agent *B* :

```
# let p_b = [ wrap (receive c_ab) (fun m -> pr "B<-A: %s" m);
              wrap (send c_ab "m2") (fun () -> pr1 "B->A: m2");
              wrap (receive c_ab) (fun m -> pr "B<-A: %s" m);
              wrap (send c_bs "m4") (fun () -> pr1 "B->S: m4");
              wrap (receive c_bs) (fun m -> pr "B<-S: %s" m)
            ];;
  val p_b : unit event list = [<abstr>; ...; <abstr>]
```

- Partie du protocole qui concerne l'agent *S* :

```
# let p_s = [ wrap (receive c_bs) (fun m -> pr "S<-B: %s" m);
              wrap (send c_bs "m5") (fun () -> pr1 "S->B: m5")
            ];;
  val p_s : unit event list = [<abstr>; <abstr>]
```

- Et finalement, on envoie à chacun sa partie du protocole !

6. `pr` est défini comme suit : `let pr m1 m2 = pr1 (sprintf m1 m2);;`


```
# send_b c_a p_a; send_b c_b p_b; send_b c_s p_s;;
- : unit = ()
A -> B : m1
B <- A : m1
B -> A : m2
A <- B : m2
A -> B : m3
B <- A : m3
B -> S : m4
S <- B : m4
S -> B : m5
B <- S : m5
```

7.3.9 Autres fonctions définies dans le module *Event*

Outre les fonction présentées dans les sections précédentes, le module *Event* comprend d'autres fonctions utiles pour la programmation concurrente :

```
# always;;
- : 'a -> 'a event = <fun>

# wrap_abort;;
- : 'a event -> (unit -> unit) -> 'a event = <fun>

# guard;;
- : (unit -> 'a event) -> 'a event = <fun>

# poll;;
- : 'a event -> 'a option = <fun>
```

La fonction «always» retourne un événement qui est toujours prêt à être synchronisé : il ne bloque jamais. Voici quelques exemples⁷ d'utilisation de cette fonction :

```
# sync (always 1);;
- : int = 1

# select [ wrap (always 1) (fun x -> x);
           wrap (receive c) (fun x -> x)
          ];;
- : int = 1
```

Le deuxième exemple illustre le fait que lorsque la fonction «always» est utilisée dans le contexte d'un «select», elle sera pratiquement toujours «sélectionnée».

La fonction «wrap_abort» prend en argument un événement et une fonction et retourne comme résultat un événement (s'apparente beaucoup à «wrap»). Elle exécute la fonction passée en argument si l'événement n'est pas sélectionné. Voici deux exemples qui illustrent ces faits :

7. Dans ces exemple, on suppose un canal *c* défini.

```
# select [ wrap (always 1) (fun x -> pr1 (sprintf "Val_de_Always:_%d" x));
          wrap_abort (wrap (receive c) (fun x -> ()))
          (fun () -> pr1 "Always,_trop_fort!")
];;
Val de Always: 1
Always, trop fort!
- : unit = ()

# spawn (fun() -> send_b c 5) ();;
- : unit = ()

# select [ wrap_abort (wrap (receive c) (fun x -> pr1 (sprintf "Val:_%d" x)))
          (fun () -> pr1 "Always,_trop_fort!")
];;
Val: 5
- : unit = ()
```

la fonction «guard» permet d’associer un effet de bord à un événement : lorsque ce dernier est synchronisé, on peut réaliser un ensemble de traitement avant que la communication latente ne soit effectuée :

```
# select [ guard (fun() -> pr1 "C'est_Always_qui_gagne!"; always 1);
          wrap (receive c) (fun x -> x)
];;
C'est Always qui gagne!
- : int = 1
```

Finalement, la fonction «poll» agit comme la fonction «sync» mais n’est pas bloquante ; autrement dit, lorsqu’elle est appliquée à un événement, si ce dernier est prêt, elle retourne «Some» de la valeur résultante ; sinon, elle retourne la valeur «None».

Dans les exemples qui suivent, «poll», dans un premier temps, retourne «Some» de la valeur reçue sur le canal «c» (puisque’il y a un fil d’exécution qui était en attente d’envoi sur ce canal). Dans un deuxième temps, «poll» retourne, sans bloquer, la valeur «None» ; donc, même si aucun autre fil d’exécution n’était prêt pour l’envoi d’une valeur sur le canal, «poll» ne bloque pas pour autant.

```
# spawn (fun() -> send_b c 5) ();;
- : unit = ()

# poll (receive c);;
- : int option = Some 5

# poll (receive c);;
- : int option = None
```

7.4 Programmation modulaire et concurrente

L’utilisation de la programmation modulaire (modules et signatures) et de la programmation concurrente offre des possibilités intéressantes. Dans cette section, nous illustrons, à travers plusieurs exemples, cette combinaison de paradigmes.

7.4.1 Abstraction des détails d’implantation

Soit la signature suivante :

```
# module type CELLULE = sig
  type 'a cell

  val cell : 'a -> 'a cell
  val get  : 'a cell -> 'a
  val put  : 'a cell -> 'a -> unit
end;;
module type CELLULE =
  sig
    type 'a cell
    val cell : 'a -> 'a cell
    val get  : 'a cell -> 'a
    val put  : 'a cell -> 'a -> unit
  end
```

La signature CELLULE déclare un type abstrait *'a cell* ainsi que trois fonctions permettant respectivement de créer un élément de type *'a cell*, de retourner la valeur de cet élément et de mettre à jour la valeur de cet élément. Autrement dit, ces fonctions nous permettraient, dépendamment de l'implantation, de créer une cellule en mémoire et de la manipuler (accès et mise à jour de la valeur).

Voici une première implantation, utilisant la programmation impérative, d'un module respectant cette signature :

```
# module Cellule_1 : CELLULE = struct
  type 'a cell = C of 'a ref

  let get (C c) = !c
  let put (C c) x = c := x
  let cell x = C(ref x)
end;;
module Cellule_1 : CELLULE
```

Une cellule est ainsi implantée sous forme de référence vers la valeur stockée; l'utilisation des références offrent la possibilité de mettre à jour la valeur stockée. Voici quelques exemples d'utilisation des fonctions définies dans le module «Cellule_1» :

```
# open Cellule_1;;

# let c = cell 9;;
  val c : int Cellule_1.cell = <abstr>

# get c;;
- : int = 9

# put c 8;;
- : unit = ()

# get c;;
- : int = 8
```

Comme on peut le constater, à travers les réponses de l'interpréteur, l'utilisation des fonctions de ce module ne fournit aucune information à propos de la structure de données utilisée pour implanter le type abstrait *'a cell*. On peut facilement le constater lorsqu'on tente d'utiliser le constructeur «C» :

```
# match cell 9 with C v -> v;;
Characters 18-21:
  match cell 9 with C v -> v;;
                    ^^^
Error: Unbound constructor C
```

Cette abstraction (du type de données) nous offre la possibilité d'implanter un module, respectant la signature `CELLULE`, en utilisant un tout autre paradigme de programmation :

```
# module Cellule_2 : CELLULE = struct
  type 'a request = GET | PUT of 'a
  type 'a enr = {reqCh : 'a request channel; replyCh: 'a channel}
  type 'a cell = CELL of 'a enr

  let get (CELL{reqCh=reqCh;replyCh=replyCh}) = (send_b reqCh GET;recv_b replyCh)
  let put (CELL{reqCh=reqCh}) x = send_b reqCh (PUT x)

  let rec cell x =
    let reqCh,replyCh = new_channel(), new_channel() in
    let rec server x = match recv_b reqCh with
      | GET -> (send_b replyCh x; server x)
      | PUT x' -> server x'
    in
    spawn server x;
    CELL{reqCh = reqCh; replyCh = replyCh}
  end;;
module Cellule_2 : CELLULE
```

Comme on peut le constater, la définition de ce module, bien qu'en apparence plus complexe que le module «Cellule_1», respecte tout de même la signature `CELLULE`. De plus, son utilisation est identique en tout point à l'utilisation du module «Cellule_1» :

```
# open Cellule_2;;

# let c = cell 9;;
val c : int Cellule_2.cell = <abstr>

# get c;;
- : int = 9

# put c 8;;
- : unit = ()

# get c;;
- : int = 8
```

L'abstraction du type de données fonctionne parfaitement puisque du point de vue de l'utilisateur d'un module respectant la signature `CELLULE`, il n'y aucune différence entre les modules «Cellule_1» et «Cellule_2» : il crée une cellule, la met à jour puis retourne sa valeur, sans savoir qu'en arrière plan, un fil d'exécution a été créé et que les différentes fonctions («get» et «put») qu'il utilise pour interagir avec la cellule font intervenir des communications, à travers différents canaux de communication, avec le fil d'exécution !

Revenons sur la définition du module «Cellule_2». Le type abstrait `'a cell` est implanté sous forme d'un enregistrement comprenant 2 champs : «reqCh» et «replyCh». Le premier champ est de type `'a request channel`, donc le type d'un canal pouvant véhiculer des éléments de type `'a request`, c'est-à-dire soit la constante «GET», soit le constructeur «PUT» avec une valeur de type `'a` en argument. On devine que ce champ va nous permettre de communiquer avec un serveur pour, soit lui demander une valeur («GET»), soit lui indiquer de mettre à jour sa valeur («PUT(v)'). Le deuxième champ est de type `'a channel`, donc le type d'un canal pouvant véhiculer des éléments de type `'a`. Ce canal va permettre au serveur, suite à une requête «GET» sur le premier canal, de nous retourner sa valeur. Ceci découle de la définition de la fonction «get» : on envoie sur le canal «reqCh» la requête «GET» et on attend par la suite sur le canal «replyCh» la réponse du serveur. C'est une sorte de protocole entre un client (l'utilisateur de la fonction «get») et le serveur.

La fonction «put» correspond à l'envoi sur le canal «reqCh» de la requête «PUT» avec comme argument la valeur à transmettre au serveur pour qu'il mette à jour la valeur de la cellule.

On l'aura donc compris, le serveur (fil d'exécution lancé en arrière-plan), va avoir comme rôle de maintenir, à travers son paramètre «x», la valeur de la cellule. Il écoute sur le canal «reqCh» et en fonction de ce qu'il reçoit comme message (requête), agit en conséquence : s'il reçoit un «GET», il retourne sur le canal «replyCh» la valeur de la cellule, soit «x», et boucle récursivement avec cette même valeur «x» (la valeur de la cellule ne change effectivement pas suite à une requête «GET», ou, dit autrement, suite à l'appel de la fonction «get») ; s'il reçoit «PUT x», alors il boucle récursivement avec la nouvelle valeur de la cellule, soit «x'».

La définition de la fonction «cell» correspond tout simplement à la définition de 2 canaux, à la définition de la fonction récursive «server», au lancement en arrière plan d'un fil d'exécution qui consiste à exécuter cette fonction «server», puis à la définition d'un enregistrement de type *'a cell* qu'elle retourne en résultat. Le fait que *'a cell* soit déclaré comme un type abstrait dans la signature CELLULE fait en sorte qu'un utilisateur de la fonction «cell», du module «Cellule_2», n'aura aucune information concernant la structure de données utilisée pour implanter ce type abstrait, ni de son éventuelle complexité (enregistrement comprenant 2 champs de types assez complexes, etc.).

Par ailleurs, voici quelques remarques concernant cette utilisation combinée de la programmation modulaire et de la programmation concurrente :

1. Comme mentionné précédemment, transparence totale pour le programmeur.
2. Implantation du prototype client-serveur : chaque itération du serveur («fonction «server»») correspond à une transaction avec un client (soit la fonction «put», soit la fonction «get»).
3. Pour chaque appel de la fonction «get» (pour chaque requête «GET»), le serveur doit répondre (faire un *reply*), ce qui constitue un comportement propre aux serveurs.
4. Même si plusieurs clients essaient d'accéder à la cellule en même temps, il n'y aura qu'une transaction qui sera effectuée à la fois. La synchronisation est assurée par les messages bloquants.
5. Tentative de violation du protocole : en supposant qu'un client fasse un «`recv_b reqCh`» sans avoir envoyé une requête «GET» :
 - Ceci ne peut arriver car le type *'a cell* est abstrait ; par conséquent, les canaux sont abstraits (cachés à l'utilisateur).
 - la signature CELLULE ne permet à un utilisateur d'interagir avec une cellule qu'à travers les fonctions «get» et «put».

Pour terminer, voici une autre implantation de la signature CELLULE qui s'apparente à celle de «Cellule_2» modulo quelques différences :

- Les canaux «getCh» et «putCh» permettent respectivement de recevoir la valeur de la cellule et de mettre à jour sa valeur.
- Le serveur utilise la primitive **select** pour simultanément tenter d'envoyer la valeur de la cellule sur le canal «getCh» et tenter de recevoir sur le canal «putCh» la nouvelle valeur de la cellule.

```
# module Cellule_3 : CELLULE = struct
  type 'a enr = {getCh : 'a channel; putCh: 'a channel}
  type 'a cell = CELL of 'a enr

  let get (CELL{getCh=getCh}) = recv_b getCh
  let put (CELL{putCh=putCh}) x = send_b putCh x

  let rec cell x =
    let getCh,putCh = new_channel(),new_channel() in
    let rec server x =
      select [ wrap (send getCh x) (fun () -> server x);
               wrap (receive putCh) (fun x -> server x)
            ]
    in
    spawn server x;
    CELL{getCh = getCh; putCh = putCh}
  end;;
module Cellule_3 : CELLULE
```

Encore une fois, du point de vue d'un utilisateur de ce module, il n'y a aucune différence par rapport aux 2 autres implantations :

```
# open Cellule_3;;

# let c = cell 9;;
val c : int Cellule_3.cell = <abstr>

# put c 11;;
- : unit = ()

# get c;;
- : int = 11
```

7.4.2 Définition de nouveaux modules utiles

Il est bien sûr possible d'enrichir les modules existants (*Thread*, *Event*, etc.) par la définition de nouveaux modules. Par exemple, soit la signature suivante :

```
# module type DIR_CHAN = sig
  type 'a in_chan and 'a out_chan

  val new_channel : unit -> ('a in_chan * 'a out_chan)

  val recv_b : 'a in_chan -> 'a
  val send_b : 'a out_chan -> 'a -> unit

  val receive : 'a in_chan -> 'a event
  val send : 'a out_chan -> 'a -> unit event
end;;
module type DIR_CHAN =
  sig
    type 'a in_chan
    and 'a out_chan
    val new_channel : unit -> 'a in_chan * 'a out_chan
    val recv_b : 'a in_chan -> 'a
    val send_b : 'a out_chan -> 'a -> unit
    val receive : 'a in_chan -> 'a Event.event
    val send : 'a out_chan -> 'a -> unit Event.event
  end
```

Le rôle de cette signature est d'offrir au programmeur des modules qui permettent d'utiliser des canaux unidirectionnels : canaux dans lesquels on peut soit envoyer, soit recevoir mais pas les

deux à la fois. On remarque que les noms des fonctions déclarées sont identiques aux noms définis dans le module *Event*. Voici une implantation d'un module respectant cette signature :

```
# module DirChan : DIR_CHAN = struct
  type 'a in_chan = IN of 'a Event.channel
  type 'a out_chan = OUT of 'a Event.channel

  let new_channel() =
    let ch = Event.new_channel() in
    (IN ch, OUT ch)

  let recv_b (IN ch) = Event.sync(Event.receive ch)
  let send_b (OUT ch) x = Event.sync(Event.send ch x)

  let receive (IN ch) = Event.receive ch
  let send (OUT ch) x = Event.send ch x
end;;
module DirChan : DIR_CHAN
```

Les constructeurs «IN» et «OUT» permettent de différencier les 2 types de canaux qui permettent respectivement de recevoir ou d'envoyer une valeur. La fonction `new_channel` est définie pour retourner une paire de canaux : le premier, on l'utilise pour recevoir une valeur tandis que le deuxième est utilisé pour envoyer une valeur. Les fonctions `send`, `receive`, `send_b` et `recv_b` sont définies dans ce sens là.

Voici un exemple d'utilisation de ce module :

```
# let (c1, c2) = DirChan.new_channel();;
val c1 : 'a DirChan.in_chan = <abstr>
val c2 : 'a DirChan.out_chan = <abstr>

# spawn (fun () -> DirChan.send_b c2 1) ();;
- : unit = ()

# (DirChan.recv_b c1) + 1;;
- : int = 2
```

Dans cet exemple, on utilise la notation point pour ne pas perdre (écraser) les versions standards (module *Event*) des fonctions définies dans le module «DirChan». Notons cependant que depuis la version 3.12 d'O'Caml, il est possible d'ouvrir un module dans un contexte local à une expression :

```
# let open DirChan in
  let (c1, c2) = new_channel() in
  spawn (fun () -> send_b c2 1) ();;
  (recv_b c1) + 1;;
- : int = 2
```

7.5 Combinaison de différents paradigmes

Il est aussi possible d'utiliser la programmation modulaire avec la programmation concurrente et orientée objets (la programmation impérative et fonctionnelle étant présentes de facto). Cette combinaison offre alors plusieurs niveaux d'abstractions : un premier au niveau des signatures et modules ; le deuxième au niveau de l'abstraction offerte naturellement par le paradigme objets.

Voici un exemple :

```
# module type S = sig

  class ['a] registered : unit -> object
    val mutable l : 'a list
    method notifyAll : ('a -> unit) -> unit
    method register : 'a -> unit
  end

end;;
module type S =
  sig
    class ['a] registered : unit -> object
      val mutable l : 'a list
      method notifyAll : ('a -> unit) -> unit
      method register : 'a -> unit
    end
  end
end
```

La signature *S* déclare une classe générique *registered* qui doit comprendre une variable d'instance *l* ainsi que 2 méthodes : *notifyAll* et *register*. Cette classe permettrait donc d'enregistrer, par l'intermédiaire de la méthode *register*, un ensemble de valeurs (de type 'a), et de notifier, à l'aide d'une fonction *f* passée en argument de la méthode *notifyAll*, tous les éléments (objets/fils d'exécution/etc.) enregistrés.

Dans un contexte où seul un fil d'exécution utiliserait une instance de cette classe, il n'est pas nécessaire d'utiliser la programmation concurrente pour implanter un module respectant cette signature :

```
# module M : S = struct

  class ['a] registered = object
    val mutable l : 'a list = []

    method register v = l <- v :: l
    method notifyAll f = iter f l
  end

end;;
module M : S
```

Ce module permet, via la méthode *register*, de s'enregistrer à un service. Lorsque la méthode *notifyAll* est invoquée, tous les abonnés au service sont notifiés. Voici un exemple d'utilisation de la classe *registered* :

```
1 # open M;;
2
3 # let reg = new registered;;
4 val reg : '_a M.registered = <obj>
5
6 # let f id = object method notify m = pr1 (id ^ ":" ^ m) end;;
7 val f : string -> < notify : string -> unit > = <fun>
8
9 # iter (fun id -> reg#register (f id)) ["o1"; "o2"; "o3"];;
10 - : unit = ()
11
12 # reg#notifyAll (fun o -> o#notify "Ok!");;
13 o3: Ok!
14 o2: Ok!
15 o1: Ok!
16 - : unit = ()
```


L'expression, à la ligne 9, a pour effet d'enregistrer trois objets (de les ajouter donc à la liste l de la classe *registered*) au service représenté par l'objet *reg*. En effet, l'application de la fonction f à chaque élément de la liste ["o1"; "o2"; "o3"] a pour effet (voir définition de la fonction f) de créer un objet ayant une méthode *notify*.

À la ligne 12, on invoque la méthode *notifyAll* de l'objet *reg* avec pour argument une fonction qui prend comme argument un élément o et qui invoque une méthode *notify* de l'élément o (donc l'objet o) avec le paramètre "Ok!". Si on se réfère à la définition de la méthode *notifyAll* dans la classe *registered*, on s'aperçoit que l'expression à la ligne 12 a pour effet d'invoquer la méthode *notify* de tous les objets présents dans la liste l , soit les trois objets enregistrés précédemment par l'intermédiaire de la méthode *register*.

Dans un contexte concurrent, il est nécessaire de gérer convenablement l'enregistrement simultané à la liste l des différents abonnés à travers l'invocation de la méthode *register*; à cette fin, on utilise les fonctionnalités du module *Mutex* afin de verrouiller l'accès à cette liste :

```

1  # module M : S = struct
2
3      class ['a] registered () =
4          let m = Mutex.create() in
5          object
6              val mutable l : 'a list = []
7
8              method register v =
9                  with_lock m (fun () -> l <- v::l)
10
11             method notifyAll f =
12                 with_lock m (fun () -> List.iter f l)
13         end
14     end;;
15 module M : S
16 
```

Étant donnée que la variable m est définie localement à la classe *registered*, elle n'a aucun impact sur la signature de la classe (pas d'ajout de variables d'instance); ainsi, le module M est toujours respectueux de la même signature S .

Voici une solution qui fait intervenir des fils d'exécution et qui utilisent des canaux de communication :

```

1  # let reg = new registered;;
2  val reg : '_a registered = <obj>
3
4  # let f id =
5      let c = new_channel () in
6      reg#register c;
7      pri (sprintf "Process_%s_:_%s" id (recv_b c));;
8  val f : string -> unit = <fun>
9
10 # iter (spawn f) ["T1"; "T2"; "T3"];;
11 - : unit = ()
12
13 # reg#notifyAll (fun c -> send_b c "Ok");;
14 - : unit = ()
15 Process T3 : Ok
16 Process T1 : Ok
17 Process T2 : Ok

```

Dans ce cas, La fonction f définit un canal de communication et s'enregistre auprès du service *reg* en lui précisant comme argument le canal c nouvellement créé. Ceci a pour effet, dans l'objet *reg*, instance de la classe *registered*, d'ajouter ce canal à la liste l (le tout étant réalisé via un verrou

afin d'éviter une modification simultanée de cette liste). Par la suite, la fonction f attend, en mode bloquant, une valeur sur ce canal c et, à la réception, affiche cette valeur à l'écran. On crée alors trois fils d'exécution, T1, T2 et T3, qui vont tous les trois s'enregistrer auprès du service et par conséquent bloquer, en écoute, sur leurs canaux respectifs. L'invocation de la méthode *notifyAll*, avec comme argument une fonction qui envoie sur un canal c , argument de la fonction, le message "Ok", a pour effet de débloquent les trois fils d'exécution. En effet, comme on peut le constater dans la classe *registered*, la méthode *notifyAll* applique la fonction précisée en argument à tous les éléments de la liste l , c'est-à-dire aux trois canaux enregistrés par les trois fils d'exécution, ce qui a pour effet de débloquent ces derniers.

7.5.1 PFX : un cadre pour la programmation parallèle en .Net

Dans les dernières versions du cadre .Net (*.Net framework*), Microsoft a introduit un cadre pour la programmation parallèle : *PFX*. Ce cadre comporte plusieurs paquetages permettant de faciliter la programmation parallèle et concurrente :

- *PLINQ (Parallel LINQ)*. Comme nous l'avons mentionné à la section 3.2 (page 119), ce module permet de traiter de manière parallèle des requêtes *LINQ*.
- Bibliothèque *TPL (Task Parallel Library)* qui comporte une classe nommée *Parallel* qui admet, entre autres, et sous forme de méthodes, des versions parallèles des commandes *For* et *ForEach* ; elle admet aussi, à travers le module (*namespace*) *System.Threading.Tasks*, plusieurs classes permettant de gérer ce qu'on appelle des tâches (*tasks*) : classe *Future*, etc.
- Des classes pour un accès concurrent et sûr à des collections de valeurs : *ConcurrentStack*, *ConcurrentQueue*, *ConcurrentBag* et *ConcurrentDictionary*.
- D'autres modules offrant des ressources utiles pour la programmation parallèle et concurrente.

Dans cette section, avec l'utilisation des modules *Thread* et *Event* d'O'Cam1, je propose une implantation pour un ensemble de classes et de méthodes issues du paquetage *TPL* de .Net . Dans ce dernier, nous retrouvons, entre autres, une classe *Future* permettant de lancer en arrière-plan un calcul et de récupérer ultérieurement (dans le futur !) son résultat (une implantation, plus limitée et fonctionnelle, a été présentée à la section 7.3.3, page 172). Aussi, la classe *Future* utilise une autre classe, *countDownEvent*, qui permet de synchroniser plusieurs calculs (*threads*) concurrents.

Le module que l'on vise à implanter doit respecter la signature suivante :

```
module type TPL = sig
  (* Classe countDownEvent *****)
  type request

  class countDownEvent : int -> object
    val s : request channel
    method add : unit
    method signal : unit
    method wait : unit
  end
  (* Classe future *****)
  class ['a, 'b] future : ('a -> 'b) -> 'a -> object
    val mutable v : 'b option
    val mutable continuations : 'b channel list
    val mutable whenCompleted : (unit -> unit) list
    val latch : countDownEvent
    method get : 'b
    method isCompleted : bool
    method addWhenCompleted : (unit -> unit) -> unit
    method continueWith : ('b -> 'b) -> unit
  end
end
```

```
(* Fonctions *****)
val waitAny: < addWhenCompleted : (unit -> unit) -> unit; .. > array -> int

val pFor: int -> int -> int -> (int -> 'a) -> unit

end
```

Ainsi, on requiert la définition de :

- une classe, nommée *countDownEvent*, qui utilise un type abstrait *request* et qui utilise aussi un objet instance de la classe *registered*, même s'il n'apparaît pas dans sa signature.
- une classe générique *future* qui utilise la classe *countDownEvent*, pour la définition de la variable d'instance *latch*, et qui admet des méthodes d'ordre supérieur : *addWhenCompleted* et *continueWith*.
- une fonction, *waitAny*, qui admet comme paramètre un tableau d'objets ayant un type ouvert (admettent au moins une méthode *addWhenCompleted* qui, en passant, est d'ordre supérieur, et qui a une signature bien précise).
- une fonction polymorphe concurrente et d'ordre supérieur : *pFor*.

On peut apprécier la combinaison des différents paradigmes : dans cette signature, on retrouve le paradigme fonctionnel à travers le type inductif *request* et les fonctions et méthodes d'ordre supérieur (fonctions en paramètre d'autres fonctions ou méthodes); le paradigme concurrent à travers l'utilisation explicite de canaux de communication; le paradigme objet à travers les deux classes; et la programmation modulaire à travers la spécification de cette signature.

Classe *countDownEvent*

La classe *countDownEvent*, issue du paquetage *TPL* de Microsoft, offre la possibilité d'attendre un signal (qui correspondra, typiquement, à la terminaison d'un fil d'exécution) de *n* fils d'exécution (*n* étant un paramètre de la classe utilisé lors de la création d'instances de cette classe); ceci peut être très utile en programmation concurrente⁸.

Voici un exemple en C# directement tiré de la littérature :

```
( C# *)
CountdownEvent countEvent = new CountdownEvent(4);

Parallel.Invoke( () => { ...; countEvent.Signal(); },
                 () => { ...; countEvent.Signal(); },
                 () => { ...; countEvent.Signal(); },
                 () => { ...; countEvent.Signal(); }
               );

countEvent.Wait();
```

À la dernière instruction du code, il y a blocage jusqu'à la réception d'un signal transmis par chacun des quatre *threads*. Voici «ce même exemple» écrit en O'Caml, grâce à la classe que nous proposons d'implanter⁹ :

8. Contrairement à la fonction *join*, qui permet de se synchroniser avec la terminaison d'un fil d'exécution donné, la classe *countDownEvent*, via sa méthode *wait*, permet d'attendre un signal d'un fil d'exécution; ce signal pourrait être transmis bien avant la fin du traitement du fil d'exécution.

9. En passant, on peut remarquer la facilité avec laquelle il est possible d'encoder le comportement d'API définies dans des paquetages; ainsi, «Parallel.Invoke», définie dans le paquetage *TPL*, pourrait être définie comme suit :

```
let parallelInvoke l = iter (fun f -> spawn f ()) l
```

```
# let countEvent = new countDownEvent 4;;
val countEvent : Parallel.countDownEvent = <obj>

iter (fun f -> spawn f ())
[ (fun () -> delay 10.; pr1 "Thread_T1"; countEvent#signal);
  (fun () -> delay 20.; pr1 "Thread_T2"; countEvent#signal);
  (fun () -> delay 15.; pr1 "Thread_T3"; countEvent#signal);
  (fun () -> delay 5.; pr1 "Thread_T4"; countEvent#signal)
];;
- : unit = ()

# countEvent#wait;;
Thread T4
Thread T1
Thread T3
Thread T2
- : unit = ()
```

Il aurait bien sûr été possible d'avoir deux *threads* en attente des quatre autres :

```
# let countEvent = new countDownEvent 4;;
val countEvent : Parallel.countDownEvent = <obj>

# iter (fun f -> spawn f ())
[ (fun () -> delay 10.; pr1 "Thread_T1"; countEvent#signal);
  (fun () -> delay 15.; pr1 "Thread_T2"; countEvent#signal);
  (fun () -> delay 20.; pr1 "Thread_T3"; countEvent#signal);
  (fun () -> delay 5.; pr1 "Thread_T4"; countEvent#signal)
];;
- : unit = ()

# spawn (fun () -> countEvent#wait; pr1 "Ok1") ();;
- : unit = ()

# spawn (fun () -> countEvent#wait; pr1 "Ok2") ();;
- : unit = ()

Thread T4 (* affiché après 5 sec *)
Thread T1 (* affiché après 10 sec *)
Thread T2 (* affiché après 15 sec *)
Thread T3 (* affiché après 20 sec *)
Ok2
Ok1
```

Voici les caractéristiques de la classe *countDownEvent* :

- Elle admet comme argument un entier précisant le nombre initial de fils d'exécution (*threads*) considérés dont on veut attendre un signal.
- Elle utilise un objet instance de la classe *registered* afin d'enregistrer les fils d'exécution qui sont en attente de la terminaison des fils d'exécution mentionnés dans le point précédent.
- Elle admet une variable d'instance *s* qui correspond à un canal de communication.
- Elle admet trois méthodes :
 1. *signal* : envoie, à travers le canal *s*, un message au serveur (voir sa description ci-dessous) lui indiquant qu'un des fils d'exécutions a possiblement¹⁰ terminé son exécution (le serveur doit donc décrémenter son compteur).
 2. *add* : envoie, à travers le même canal *s*, un message au serveur lui indiquant qu'il faut considérer un nouveau *thread* parmi ceux déjà considérés (le serveur doit donc incrémenter son compteur).

10. Habituellement, un *thread* invoque cette méthode pour signaler sa terminaison ; cependant, il pourrait invoquer cette méthode ailleurs dans son code.

3. *wait* : cette méthode enregistre, à travers la méthode de l'objet instance de *register*, un canal frais, puis bloque sur ce canal jusqu'à la réception d'une valeur. Ainsi, tout utilisateur «externe» (pouvant être un *thread*) qui invoque cette méthode bloquera jusqu'à ce que cette dernière reçoive un message sur son canal.

Comme mentionné précédemment, cette classe admet aussi une fonction qui agit comme serveur (cette fonction est donc lancée en arrière plan dès la création d'un objet instance de cette classe). Ce serveur admet un compteur qui, initialement, vaut la valeur passée en argument de la classe. Il agit comme suit : si son compteur (paramètre) vaut zéro, il notifie (*notifyAll*) alors tous les fils d'exécution qui se sont enregistrés à travers la méthode *wait* ; autrement il écoute sur le canal *s* en vue de mettre à jour son compteur.

Voici l'implantation de cette classe :

```

1  type request = Incr | Decr
2
3  class countDownEvent count =
4    let reg = new registered in
5    object(this)
6
7      val s = new_channel()
8
9      method signal = send_b s Decr
10
11     method add = send_b s Incr
12
13     method wait =
14       let c = new_channel() in
15       reg#register c;
16       recv_b c
17
18   initializer
19     let rec server x =
20       if x <= 0 then
21         reg#notifyAll (fun c -> ig(poll(send c ())))
22       else
23         match recv_b s with
24         | Incr -> server (x+1)
25         | Decr -> server (x-1)
26     in
27     spawn server count
28
29 end

```

Comme on peut le constater :

- à la création de l'objet (**initializer**), on crée un petit serveur qu'on lance aussitôt en arrière plan avec comme paramètre initial la valeur *count* argument de la classe. Ce serveur a un comportement assez basique : à chaque itération (récursion¹¹), il teste si son paramètre, qui va jouer le rôle d'un compteur, est inférieur ou égal à zéro : si ce n'est pas le cas, il bloque en écoute sur un canal *s*, qui lui est réservé (ce canal est défini une fois pour toute et est associé à la variable d'instance, non modifiable, *s*). S'il reçoit la valeur «Incr» (de type *request*) il boucle en incrémentant son compteur ; s'il reçoit la valeur «Decr», il boucle en décrémentant son compteur.

Si la valeur de son compteur est inférieure ou égale à zéro, il invoque la méthode *notifyAll*, de l'objet instance de la classe *registered*, en envoyant sur tous les canaux *c* enregistrés dans la liste *l* de cet objet) la valeur «()». Grâce à l'utilisation de la fonction «poll», ces envois sont effectifs mais ne sont pas bloquants¹².

11. On peut constater que la fonction récursive *server* est en mode récursion terminale.

12. Il aurait été possible d'y aller aussi avec un envoi asynchrone (voir fonction *async_send* définie à la page 190).

- les méthodes *add* et *signal* permettent à des fils d'exécution interagissant avec un objet instance de cette classe de respectivement signaler à l'objet qu'il y a un autre fil d'exécution qui doit être ajouté à la liste des fils d'exécution dont on attend la terminaison éventuelle et signaler à l'objet la terminaison d'un de ces fils d'exécution; ces deux méthodes consistent à interagir avec le serveur en envoyant soit la valeur «Incr», soit la valeur «Decr».
- finalement, la méthode *wait* est utilisée par tout fil d'exécution qui désire attendre un signal de tous les autres fils d'exécution ayant invoqués la méthode *add*. Comme on peut le constater, chaque fil d'exécution qui invoque la méthode *wait* définit un canal privé *c* qu'il enregistre aussitôt via la méthode *register*, de l'objet «reg», pour par la suite bloquer (et donc attendre) en écoute sur ce canal. Ce blocage persiste tant que la méthode *notifyAll* ne sera pas invoquée c'est-à-dire tant que le compteur du serveur ne sera pas inférieur ou égal à zéro, et donc tant que tous les autres fils d'exécution n'auront pas invoqué la méthode *signal*.

À l'instar de la première version de la classe *registered*, cette version de la classe *countDownEvent* n'est pas adaptée à une utilisation concurrente d'une de ses instances. Plus précisément, à la ligne 21 de sa définition, au niveau du corps du serveur, lorsque ce dernier notifie les différents fils d'exécution en attente (qui auront invoqué *wait*), elle termine son traitement (le serveur ne boucle plus; il termine). Tous les fils d'exécution qui invoqueront une des méthode l'objet instance de cette classe bloqueront puisque le serveur ne traite plus de requête (i.e n'écoute plus sur le canal *s*).

Pour remédier à ce problème, on introduit une variable *alive* qui aura comme rôle de déterminer l'état du serveur (de l'objet); aussi, on utilise un verrou pour protéger cette variable d'un accès simultané. Voici une nouvelle implantation de cette classe :

```

1  class countDownEvent count =
2
3  let reg = new registered () in
4  let alive = ref true in
5  let m = Mutex.create () in
6
7  object(this)
8    val s = new_channel()
9
10   method signal = if with_lock m (fun () -> !alive) then send_b s Decr
11   method add = if with_lock m (fun () -> !alive) then send_b s Incr
12
13   method wait =
14     if with_lock m (fun () -> !alive) then
15       let c = new_channel() in
16       reg#register c;
17       recv_b c
18
19   initializer
20     if count < 1 then
21       with_lock m (fun () -> alive := false)
22     else
23       let rec server x =
24         if x = 0 then
25           begin
26             reg#notifyAll (fun c -> ignore(poll(send c ()))));
27             with_lock m (fun () -> alive := false)
28           end
29         else
30           match recv_b s with
31             | Incr -> server (x+1)
32             | Decr -> server (x-1)
33       in
34       spawn server count
35   end

```

Voici un exemple d'utilisation de cette nouvelle version de la classe :

```
# let cde = new countDownEvent 2;;
val cde : countDownEvent = <obj>

# let t1 () = delay 20.; pr1 "T1_termine"; cde#signal;;
val t1 : unit -> unit = <fun>
# let t3 () = cde#add; delay 40.; pr1 "T3_termine"; cde#signal;;
val t3 : unit -> unit = <fun>
# let t2 () = delay 30.; pr1 "T2_termine"; cde#signal;;
val t2 : unit -> unit = <fun>
# let t4 () = cde#add; delay 25.; pr1 "T4_termine"; cde#signal;;
val t4 : unit -> unit = <fun>
# let t5 () = cde#wait; pr1 "T5_termine, après wait";
val t5 : unit -> unit = <fun>
# let t6 () = delay 20.; cde#wait; pr1 "T6_termine, après delay 20. et wait";
val t6 : unit -> unit = <fun>
# let t7 () = delay 60.; cde#wait;
    pr1 "T7_termine plus tard, aucun effet du wait";
val t7 : unit -> unit = <fun>
# let t8 () = delay 50.; cde#add;
    cde#signal;
    pr1 "T8_termine plus tard, aucun effet de son add ni signal";
val t8 : unit -> unit = <fun>

# List.iter (fun f -> spawn f ()) [t1;t2;t3;t4;t5;t6;t7;t8];;
- : unit = ()

(* Affichage débute après 20 sec. *)
T1 termine
T4 termine
T2 termine
T3 termine
T6 termine, après delay 20. et wait
T5 termine, après wait
T8 termine plus tard, aucun effet de son add ni signal
T7 termine plus tard, aucun effet du wait
```

Classe *future*

La classe *future* permet de procéder à un calcul en arrière plan et à offrir la possibilité de récupérer le résultat de ce calcul plus tard (grâce à l'invocation d'une méthode *get*). Ainsi, elle prend en argument un calcul sous forme d'une fonction de type «'a -> 'b» (ce qui explique que la classe est générique), ainsi qu'une valeur «x» sur laquelle la fonction sera appliquée.

Par ailleurs, la classe offre plusieurs autres possibilités. Il est possible d'ajouter un calcul qui sera exécuté suite à la terminaison du calcul présent (celui initialement passé en argument de la classe). Ceci est réalisé grâce à la méthode *continueWith* qui prend comme argument une fonction de type «'b -> 'b» (elle récupère donc le résultat de la première fonction et effectue un nouveau calcul pour finalement retourner son propre résultat). Il est possible de spécifier en cascade plusieurs calculs. Voici un exemple, directement tiré de la littérature (C#), illustrant ces propos :

```
(* C# *)
Future<int> f = Future.Create(() => 5)
    .ContinueWith(a => a.Value - 1)
    .ContinueWith(b => b.Value - 1);

Console.WriteLine(f.Value);
```

Voici ce «même exemple» écrit à l'aide d'O'Caml (et de la classe *future*)¹³ :

13. Notez que dans l'implantation proposée de la classe *future*, et contrairement à celle définie dans le paquetage

```
# let f = new future (fun () -> delay 20.; 5) ();;
      f#continueWith (fun a -> a - 1);;
      f#continueWith (fun b -> b - 1);;
val f : int Parallel.future = <obj>
# - : unit = ()
# - : unit = ()

# pr1 (sprintf "%d" (f#get));;
3
- : unit = ()
```

Dans cet exemple, l'utilisation de la fonction *delay* est nécessaire car, autrement, le (premier) calcul aura déjà terminé son exécution, rendant ainsi les deux appels de la méthode *continueWith* incohérents (inutiles). D'ailleurs, dans cette éventualité, cette méthode soulève une exception :

```
# let f = new future (fun () -> 5) ();;
val f : int Parallel.future = <obj>

# pr1 (sprintf "%d" (f#get));;
5
- : unit = ()

# f#continueWith (fun a -> a - 1);;
Exception: Failure "future is already completed".
```

Il est possible aussi d'ajouter des traitements qui seront automatiquement exécutés lorsque la série de calculs (calcul initial plus ceux ajoutés par l'entremise de *continueWith*) termine. Ceci est réalisé grâce à la méthode *addWhenCompleted* :

```
# let f = new future (fun () -> delay 20.; 5) ();;
      f#continueWith (fun a -> a - 1);;
      f#addWhenCompleted (fun () -> pr1 "Completed1");;
      f#addWhenCompleted (fun () -> pr1 "Completed2");;
val f : int Parallel.future = <obj>
# - : unit = ()
# - : unit = ()
# - : unit = ()

# pr1 (sprintf "%d" (f#get));;
Completed1
4
- : unit = ()
Completed2
```

Finalement, il est possible à plusieurs *threads* de requérir, de manière concurrente, la valeur d'un *future* :

TPL de Microsoft, la fonction passée en argument de la méthode *continueWith* prend comme argument la valeur résultat du précédent *thread* et procède à son propre calcul.


```
# let f = new future (fun () -> delay 20.; 5) ();;
val f : int Parallel.future = <obj>

# spawn (fun () -> pri (sprintf "T1->%d" (f#get))) ();;
- : unit = ()

# spawn (fun () -> pri (sprintf "T2->%d" (f#get))) ();;
- : unit = ()

# spawn (fun () -> pri (sprintf "T3->%d" (f#get))) ();;
- : unit = ()
T2 -> 5
T1 -> 5
T3 -> 5
```

Dans l'implantation proposée, on limite l'utilisation des méthodes *continueWith* et *addWhenCompleted* à un seul et même *thread* (il ne peut y avoir, comme pour la méthode *get*, d'utilisation simultanée de ces méthodes par différents *thread*). On pourrait évidemment étendre la définition de cette classe pour pouvoir offrir plus de flexibilité dans un contexte d'accès concurrent à un objet instance de cette classe.

Voici les caractéristiques de cette classe :

— Elle admet quatre variables d'instances :

1. *v* : contient, une fois calculé, le résultat (s'il y a plusieurs calculs, il s'agit bien sûr du calcul résultant de la séquence en cascade de tous les calculs); à la création de l'objet, cette variable est initialisée à *None*.
2. *continuations* : une liste, initialement vide, de tous les canaux nécessaires pour récupérer les résultats des différents calculs (en cascade) en jeu. Chaque *thread*, impliqué dans un calcul donné, utilise cette liste pour préciser le canal à partir duquel on peut récupérer son résultat; il l'utilise aussi pour récupérer le résultat de son (calcul) prédécesseur.
3. *whenCompleted* : une liste, initialement vide, de tous les traitements à exécuter quand le résultat final est calculé.
4. *latch* : un objet instance de *countDownEvent* (avec comme valeur initial (de l'objet) 1) permettant de se synchroniser avec la fin de tous les calculs (initialement, il n'y a qu'un seul calcul, celui passé en argument de la classe *future*).

— Elle admet quatre méthodes :

1. *get* : si le résultat a déjà été calculé, elle le retourne. Sinon, elle s'enregistre pour l'obtention du résultat (les différents *threads* invoquant cette méthode sont donc en mode bloqué jusqu'à l'obtention du résultat).
2. *isCompleted* : teste si le résultat est disponible.
3. *addWhenCompleted* : assez triviale.
4. *continueWith* : cette méthode prend en argument une fonction (un nouveau calcul) et soulève une exception si le calcul, de l'objet courant (instance de la classe *future*), est déjà complété. Dans le cas contraire, elle crée un canal frais, l'ajoute à la liste *continuations*, met à jour la variable d'instance *latch* et crée un nouveau *thread* qui aura pour rôle de réaliser, à partir du résultat du calcul précédent, le nouveau calcul et à envoyer son résultat (possiblement, au calcul suivant) sur le canal nouvellement créé.

Voici la définition de la classe *future* :

```

class ['a] future (f : 'a -> 'b) (x : 'a) = let reg = new registered in

object(this)
  val mutable v = None
  val mutable continuations = []
  val mutable whenCompleted = []
  val latch = new countDownEvent 1

  method get =
    match v with
    | None ->
      let c = new_channel() in
      reg#register c;
      recv_b c
    | Some x -> x

  method isCompleted = match v with None -> false | _ -> true

  method addWhenCompleted h =
    match v with
    | None -> whenCompleted <- whenCompleted @ [h]
    | _ -> h()

  method continueWith g =
    match v with
    | None ->
      let c = hd continuations in
      let d = new_channel() in
      latch#add;
      continuations <- d::continuations;
      spawn (fun () -> let r = g (recv_b c) in
        latch#signal;
        send_b d r
      ) ()
    | _ -> failwith "future_is_already_completed"

  initializer
    let c = new_channel() in
    continuations <- [c];
    spawn (fun () -> let r = f x in latch#signal; send_b c r) ();
    spawn (fun () ->
      latch#wait;
      let v' = recv_b (hd continuations) in
      v <- Some v';
      reg#notifyAll (fun d -> ig(poll(send d v')));
      iter (fun h -> h()) whenCompleted
    ) ()
end

```

Comme on peut le constater, à la création d'un objet instance de cette classe, il faut créer un canal frais, l'ajouter dans *continuations*, et créer deux *threads* : l'un pour le calcul de la fonction passée en argument de la classe et l'envoi de son résultat sur le canal nouvellement créé; l'autre pour l'attente de la fin de l'exécution de tous les calculs, suivie de l'envoi (notification) de la valeur à tous les *threads* ayant invoqué la méthode *get*, suivie de l'exécution des traitements se trouvant dans *whenCompleted*.

À noter aussi que cette classe utilise un objet instance de la classe *registered* bien que ce dernier n'apparaît pas dans la signature de la classe *future*. En effet, étant utilisé à travers une variable déclarée localement dans le corps de la classe, il ne figure donc pas dans le type inféré de cette classe.

Fonction *waitAny*

La fonction *waitAny* permet d'attendre la fin de l'exécution d'un «*future*» parmi plusieurs. L'implantation de la fonction permet d'avoir un exemple concret de l'utilisation de la méthode *addWhenCompleted*. Cette fonction prend comme argument un tableau de «*future*» (en fait, la fonction n'est pas restreinte aux «*future*» ; elle s'applique à tout objet qui admet la méthode *addWhenCompleted*), crée un canal frais, ajoute, à l'aide de *addWhenCompleted*, aux différents «*future*» une fonction qui consiste à envoyer, sur le canal frais, l'indice du «*future*» en question dans le tableau, puis écoute sur ce canal. Bien sûr, le premier «*future*» qui aura terminé son calcul enverra son indice sur le canal, lequel indice sera retourné comme résultat de la fonction *waitAny*.

Voici un exemple tiré de la littérature (C#) :

```
(* C# *)
Future<int>[] calculations =
    new Future<int>[] {
        Future.Create(() => 5),
        Future.Create(() => 6),
        Future.Create(() => 7)
    };

int calcIndex = Task.WaitAny(calculations);
Console.WriteLine(calculations[calcIndex].Value);
```

Voici le «même code» écrit en O'Caml :

```
# let calculations = [| new future (fun () -> delay 60.; 5) ();
                        new future (fun () -> delay 10.; 6) ();
                        new future (fun () -> delay 45.; 7) ()
                      |];;
val calculations : int Parallel.future array = [|<obj>; <obj>; <obj>|]

# let calcIndex = waitAny calculations;;
val calcIndex : int = 1

# printf "Valeur_:%d!" calculations.(calcIndex)#get;;
Valeur : 6!
- : unit = ()
```

Finalement, voici le code de la fonction *waitAny* :

```
let waitAny t =
  let c = new_channel() in
  Array.iteri (fun i f -> f#addWhenCompleted (fun () -> ig(poll(send c i)))) t;
  recv_b c
```

Notons qu'il est possible de reprendre l'exemple du «benchmark», vu à la section 7.3.8 (page 208), mais en utilisant cette fois-ci la fonction *waitAny* :

```
# #use "sort.ml";;
# let l = randomList 100000 1000;;

# let sortAlgo = [| new future (fun () -> mergeSort l) ();
                  new future (fun () -> quickSort l) ()
                |];;
val sortAlgo : int list Parallel.future array = [|<obj>; <obj>|]

# let index = waitAny sortAlgo;;
val index : int = 0

# sortAlgo.(index)#get;;
...
```

Ainsi, on indique que la fonction *mergeSort* est la plus rapide.

Voici ce même exemple, mais décrit de manière plus précise :

```
# let l = randomList 100000 1000;;
...

# let trace (nom, tri) l =
  delay 20.;
  pr2 (nom ^ "_commence");
  let res, t = timeRun tri l in
  pr2 (sprintf "%s_termine_en_%f_sec" nom t);
  res;;
val trace : string * ('a -> 'b) -> 'a -> 'b = <fun>

# let t_paires = [| ("ocamlSort", List.sort compare);
                  ("mergeSort", mergeSort);
                  ("quickSort", quickSort)
                |];;
val t_paires : (string * ('a list -> 'a list)) array =
|/("ocamlSort", <fun>); ("mergeSort", <fun>); ("quickSort", <fun>)/|
```

```
# let sortAlgo = Array.map (fun paire -> new future (trace paire) l)
                          t_paires;;
val sortAlgo : (int list, int list) Tpl.future array =
|/<obj>; <obj>; <obj>/|

# let index = waitAny sortAlgo in
  fst (t_paires.(index)), sortAlgo.(index)#get;;
<3405.45800> ocamlSort commence
<3405.67800> mergeSort commence
<3406.02500> ocamlSort termine en 0.219856 sec
- : string * int list =
("ocamlSort",
 [...])
<3406.03300> quickSort commence
<3406.59900> mergeSort termine en 0.347080 sec
<3407.49700> quickSort termine en 1.463360 sec
```

Remarque L'exemple, tiré de la littérature, présenté précédemment, est en réalité défini comme suit :

```
(* C# *)
Future<int>[] calculations =
  new Future<int>[] {
    Future.Create(() => 5),
    Future.Create(() => 6),
    Future.Create(() => 7)
  };

int calcIndex = Task.WaitAny(calculations);
Array.ForEach(calculations, c => c.Cancel());
Console.WriteLine(calculations[calcIndex].Value);
```

La seule différence avec la précédente version est l'ajout d'une instruction avec laquelle on invoque une méthode *Cancel* pour annuler les calculs effectués dans les différents «*future*». La version en OCaml aurait été :

```
# let calculations = [| new future (fun () -> delay 60.; 5) ();
                        new future (fun () -> delay 10.; 6) ();
                        new future (fun () -> delay 45.; 7) ()
                      |];;
val calculations : int Parallel.future array = [|<obj>; <obj>; <obj>|/]

# let calcIndex = waitAny calculations;;
val calcIndex : int = 1

# Array.iter (fun c -> c#cancel) calculations;;
- : unit = ()

# pr1 (sprintf "Valeur_:%d!" calculations.(calcIndex)#get);;
Valeur : 6!
- : unit = ()
```

Par conséquent, il aurait fallu que la classe *future* admette une méthode *cancel* permettant d'arrêter (*kill*) l'exécution de tous les fils d'exécution impliqués dans les différents calculs (en cascade). Dans cette optique, la liste *continuations* aurait admis, en plus des canaux, les identifiants (de type *Thread.t*) des *threads* associés à ces canaux (ça aurait été une liste de paires); la méthode *cancel*, lorsqu'invoquée, aurait tout simplement parcouru la liste *continuations* et arrêter (*kill*), à partir du deuxième élément de chaque paire, tous les *threads* en exécution. Malheureusement, la fonction *Thread.kill* n'est pas implantée pour des raisons assez longues à expliquer. Pour «contourner» ce problème, une solution aurait été d'ajouter, à la liste *continuations*, un troisième élément, devenant ainsi une liste de triplets, qui correspond à un canal avec lequel on peut communiquer avec chaque *thread* pour l'arrêter (la méthode *cancel* aurait alors envoyer une valeur quelconque, soit `()`, sur ce canal pour signaler à chaque *thread* qu'il doit s'arrêter). Chaque *thread* aurait alors eu comme comportement de lancer un *thread* fils qui s'occuperait exclusivement du calcul, et d'écouter par la suite sur 2 canaux : le premier lui permettrait de récupérer le résultat de son *thread* fils afin de l'envoyer sur son canal (premier élément du triplet); le deuxième lui permettrait de recevoir sur le troisième élément du triplet, un message lui indiquant qu'il doit terminer son exécution. Il aurait pu alors afficher un message à l'écran, puis appeler la fonction *exit*. Idéalement, entre ces deux dernières instructions, il aurait fallu qu'il arrête (*kill*) son *thread* fils, mais encore une fois, la fonction *kill* n'est pas implantée.

Le gabarit associé à un tel comportement aurait été :

```
spawn(fun () ->
  let t = create (fun () -> ... processus fils qui réalise le calcul ...) () in
  select [ wrap (receive c') (fun x ->... on reçoit le résultat du "fils" ...);
          wrap (receive k) (fun () ->... on reçoit un message de cancel ...
                                on affiche alors un message à l'écran;
                                idéalement, on aurait fait "kill_t";
                                raise Exit)
        ];
) ();
```

Fonction *pFor*

Cette fonction est la forme concurrente de la construction syntaxique **for**. Elle a comme signature :

```
val pFor: int -> int -> int -> (int -> 'a) -> unit
```

Le premier paramètre précise le nombre de fils d'exécution que l'on veut exécuter en parallèle pour effectuer le calcul; les deux arguments suivants précisent la borne inférieure et la borne supérieure de l'itération à effectuer; le quatrième argument est une fonction qui correspond au corps (traitement à effectuer) de l'itération.

Voici un exemple d'utilisation de cette fonction :

```
# pFor 4 1 12 (fun i -> pr2 (sprintf "T%d->%d" (id(self())) i));;
<233.91200> T7 -> 1
<233.91200> T8 -> 4
<233.91300> T10 -> 10
<233.91300> T7 -> 2
<233.91300> T8 -> 5
<233.91300> T10 -> 11
<233.91300> T7 -> 3
<233.91300> T8 -> 6
<233.91300> T10 -> 12
<233.91300> T9 -> 7
<233.91300> T9 -> 8
<233.91300> T9 -> 9
- : unit = ()
```

Dans cet exemple, quatre fils d'exécution ont été créés pour se partager les douze étapes de l'itération ; comme on peut le constater, la charge est partagée de manière équitable entre les quatre fils d'exécution : chacun d'entre eux applique le traitement à trois éléments du tableau.

Voici le code de la fonction *pFor* :

```
let pFor p lo hi body =
  let n = hi - lo + 1 in
  if p <= 1 || n <= 1 then
    for i = lo to hi do body i done
  else
    let p' = if p > n then n else p in
    let div, res = n / p', n mod p' in
    let chunk = if res = 0 then div else if p' = 2 then div + 1 else n / (p' - 1) in
    let latch = new CountdownEvent p' in
    for i = 0 to (p' - 1) do
      spawn (fun () ->
        let _start = lo + (chunk * i) in
        let _end = if i = p' - 1 then hi else _start + chunk - 1 in
        for j = _start to _end do body j done;
        latch#signal
      ) ()
    done;
    latch#wait
```

Même s'il n'était pas nécessaire de le considérer, cette implantation vérifie d'abord si le nombre de fils d'exécution que l'on veut exécuter en parallèle (premier paramètre de la fonction) n'est pas limité à un, et que le nombre de traitements à effectuer n'est pas limité (un), auxquels cas on peut tout simplement utiliser la construction syntaxique **for** séquentielle.

Dans le cas contraire, un autre test est effectué afin de limiter le nombre de fils d'exécution à créer et à exécuter de manière concurrente : si le premier paramètre de la fonction a une valeur supérieure au nombre de traitements à effectuer, soit *n*, alors *n* fils d'exécution suffisent.

Par la suite, on réalise différents calculs pour déterminer le nombre de traitements que chaque fil d'exécution devra effectuer. Puis, on procède à la création effective et à l'exécution de *p'* fils d'exécution, chacun s'occupant d'un nombre bien précis de traitements : chaque fil d'exécution applique alors le traitement *body* autant de fois que nécessaire et signale aussitôt sa terminaison. Finalement, à l'aide de la méthode *wait*, la fonction *pFor* bloque jusqu'à la terminaison de tous les fils d'exécution, soit la terminaison de tous les traitements.

Voici un autre exemple dans lequel la charge est partagée de manière inéquitable car le nombre de traitements à effectuer n'est pas divisible par le nombre de fils d'exécution à considérer.

```
# pFor 2 1 5 (fun i -> pr2 (sprintf "T%d->%d" (id(self())) i));;
<242.26900> T12 -> 1
<242.27000> T12 -> 2
<242.27000> T12 -> 3
<242.27000> T13 -> 4
<242.27000> T13 -> 5
- : unit = ()
```

Finalement, dans l'exemple qui suit, bien qu'il est spécifié que l'on désire créer 200 fils d'exécution pour réaliser les calculs, seuls trois fils d'exécution seront créés car le nombre de traitements à effectuer n'excède pas cette valeur :

```
# pFor 200 1 3 (fun i -> pr2 (sprintf "T%d->%d" (id(self())) i));;
<518.41000> T15 -> 1
<518.41000> T17 -> 3
<518.41000> T16 -> 2
- : unit = ()
```

7.5.2 Exploitation du module *Tpl*

Dans cette section, nous présentons différents exemples d'utilisations des classes et fonctions définies dans le module *Tpl*. On commence d'abord par l'implantation d'une fonction que l'on a déjà traitée plus tôt dans ce chapitre : *cmap*.

Fonction *cmap*

Voici une version de la fonction «*cmap*» («*map*» concurrent) qui utilise la classe *countDownEvent* :

```
# let cmap f l =
  let n = length l in
  let latch = new countDownEvent n in
  let l' = map (fun x -> ref [], x) l in
  iter (fun (rx, v) -> spawn (fun () -> rx := [f(v)]; latch#signal) ()) l';
  latch#wait;
  map (fun (rx, _) -> hd(!rx)) l';
val cmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Dans le contexte de cette implantation, l'objet «*latch*», instance de la classe «*countDownEvent*», va nous permettre d'attendre la fin des traitements (application concurrente de «*f*» aux différents éléments de la liste «*l*») avant de retourner le résultat. Sans l'expression «*latch#wait*», la fonction pourrait provoquer une erreur à l'exécution.

Fonction *cmap* rechargée La fonction *pFor*, définie dans le module *Tpl*, offre une manière plus aboutie et élégante d'implanter la fonction *cmap*. Dans cette version, on vise la signature suivante :

```
cmap : int -> 'a array -> ('a -> 'b) -> 'b array
```

qui utilise des tableaux; elle admet aussi un paramètre permettant de spécifier le nombre de *threads* à considérer.

Voici l'implantation proposée de cette fonction *cmap* :

```

let cmap' n t f = match Array.length t with
| 0 -> [| |]
| m ->
  let init = f (t.(0)) in
  let t' = Array.make m init in
  pFor n 1 (m-1) (fun i -> t'.(i) <- f (t.(i)));
  t'

```

Voici quelques exemples illustrant l'utilisation de cette fonction :

```

# cmap' 4 [|1;2;3;4;5;6;7;8;9|] succ;;
- : int array = [|2; 3; 4; 5; 6; 7; 8; 9; 10|]

# cmap' 4 [|1;2;3;4;5;6;7;8;9|]
  (fun e -> pr2 (sprintf "T%d->%d" (id(self())) e); succ e);;
<1171.53200> T0 -> 1
<1171.53300> T35 -> 4
<1171.53400> T34 -> 2
<1171.53400> T36 -> 6
<1171.53400> T35 -> 5
<1171.53400> T34 -> 3
<1171.53400> T36 -> 7
<1171.53400> T37 -> 8
<1171.53400> T37 -> 9
- : int array = [|2; 3; 4; 5; 6; 7; 8; 9; 10|]

# cmap' 3 [|1;2;3;4;5;6;7;8;9|] string_of_int;;
- : string array = [|"1";"2";"3";"4";"5";"6";"7";"8";"9"|]

```

Fonction *partitionc*

Dans cette section, on propose une version concurrente de la fonction *partition*. Pour ce faire, on définit tout d'abord une fonction «b» comme suit :

```

type 'a msg = M of 'a | GET

let b (c,d) =
  let rec f l =
    match recv_b c with
    | M x -> f (l@[x])
    | GET -> send_b d l
  in f []

```

Son type est *'a msg channel * 'a list channel -> unit*. Elle définit un serveur «f» qui écoute sur un canal «c» : soit il reçoit une valeur «x», auquel cas il l'ajoute à la fin de sa liste (passée en argument de la fonction); soit il reçoit, sur le même canal, un message «GET», auquel cas, il envoie sa liste sur un canal «d».

La fonction *partitionc* est définie comme suit ¹⁴ :

14. Rappel : «nth l i» retourne le $(i+1)^{ème}$ élément de la liste «l». Exemple : nth [6;8;5;2] 1 = 8.


```

let partitionc x y =
  let c1,c2,d1,d2 = new_channel(),new_channel(),new_channel(),new_channel() in
  let l = split y (nbCoeurs()) in
  let latch = new CountdownEvent (length l) in
  let f (i,j) = for k = i to j do
    let v = nth y k in
    if x v then send_b c1 (M v) else send_b c2 (M v)
  done;
  latch#signal
in
  iter (fun (c,d) -> spawn b (c,d)) [(c1,d1);(c2,d2)];
  iter (fun (i,j) -> spawn f (i,j)) l;
  latch#wait;
  send_b c1 GET; send_b c2 GET;
  recv_b d1,recv_b d2

```

La fonction suppose l'existence d'une fonction «nbCoeurs» qui retourne le nombre de *cœurs* disponibles dans une machine donnée, et d'une fonction «split» qui, à partir d'une liste et d'un entier, représentant le nombre de *cœurs* disponibles, retourne une liste de paires d'entiers (indices) ; on divise en quelque sorte la liste en différentes parties¹⁵.

Notons qu'à la différence de la version originale, cette version ne retourne pas forcément des listes avec des éléments dans le bon ordre (relativement à la liste initiale). Par exemple :

```

partitionc (fun x -> x mod 2 = 0) [1;2;3;4;5;6;7;8;9;10;11;12];;
- : int list * int list = ([4; 6; 10; 12; 2; 8], [1; 5; 11; 3; 7; 9])

List.partition (fun x -> x mod 2 = 0) [1;2;3;4;5;6;7;8;9;10;11;12];;
- : int list * int list = ([2; 4; 6; 8; 10; 12], [1; 3; 5; 7; 9; 11])

```

Cette fonction fonctionne comme suit :

- elle lance en arrière plan 2 fils d'exécution qui vont progressivement collecter les valeurs de la liste (passée en argument) respectant le prédicat (passé lui aussi en argument de la fonction) et celles qui ne le respectent pas ;
- puis elle lance en arrière l'équivalent de «n» («n» = *nbCoeurs*) fils d'exécution qui vont traiter, de manière concurrente, différentes zones de la liste (ils communiquent les résultats de leur traitement aux 2 premiers fils d'exécution) ;
- puis elle bloque jusqu'à ce que tous les éléments de la liste soient traités ;
- finalement, elle récupère, de la part des 2 premiers fils d'exécution, les listes résultantes du traitement de «partition».

Traitement concurrent d'arbres

Dans cette section, on considère plusieurs exemples faisant intervenir le module *Tpl* pour le traitement efficace (concurrent) des éléments d'un arbre¹⁶. Considérons la fonction *timeRun*, présentée en début de chapitre, ainsi que la fonction *est_premier* dont on rappelle la définition :

```

let est_premier n =
  let ns = int_of_float(sqrt(float(n))) in
  let rec est_premierUtil m =
    (m > ns || (n mod m < 0 && (est_premierUtil (m+1))))
  in
  delay 0.1; est_premierUtil 2;;

```

On a inséré volontairement un délai d'hibernation pour simuler un calcul important :

15. Exemple : `split [1;2;3;4;5;6;7;8;9;10;11;12;13] 4 = [(0, 2); (3, 5); (6, 8); (9, 12)]`.

16. Notez que dans cette section, on reprend autrement le traitement concurrent des arbres binaires relativement à ce qui a été présenté à la section 7.3.7, page 196, concernant le réducteur concurrent d'arbres binaires.

```
# timeRun est_premier 7;;
- : bool * float = (true, 0.092999935150146484)
```

Maintenant, considérons le type de données suivant permettant de représenter des arbres binaires ayant à leurs feuilles des entiers :

```
# type intTree = Leaf of int | Node of intTree * intTree;;
type intTree = Leaf of int | Node of intTree * intTree
```

ainsi que la fonction suivante permettant de générer un arbre binaire d'une profondeur donnée et ayant à ces feuilles des nombres aléatoires compris entre 1 et 100000 :

```
# Random.self_init ();;
- : unit = ()

# let rec tree(depth) =
  if depth = 0 then Leaf(Random.int 100000)
  else Node(tree(depth-1), tree(depth-1));;
val tree : int -> intTree = <fun>
```

Voici la définition d'un arbre comprenant 16 (2^4) entiers :

```
# let t = tree(4);;
val t : intTree =
  Node
    (Node
      (Node (Node (Leaf 15679, Leaf 15476), Node (Leaf 44380, Leaf 62226)),
        Node (Node (Leaf 46002, Leaf 44182), Node (Leaf 21977, Leaf 18200))),
      Node (Node (Node (Leaf 87222, Leaf 8011), Node (Leaf 35417, Leaf 13362)),
        Node (Node (Leaf 84713, Leaf 91447), Node (Leaf 2014, Leaf 93678)))))
```

La définition qui suit compte de manière séquentielle le nombre de nombres premiers que comprend l'arbre t :

```
# let rec count tree = match tree with
| Leaf n when est_premier n -> 1
| Leaf _ -> 0
| Node(left, right) ->
  let nr = count right in
  let nl = count left in
  nl + nr;;
val count : intTree -> int = <fun>

# timeRun count t;;
- : int * float = (4, 1.61314296722412109)
```

Ainsi, on apprend que l'arbre t comprend 4 nombres premiers et que le traitement a requis 1.61 sec. Remarquez que dans le cas où l'arbre correspond à un nœud, «Noeud(left,right)», on aurait pu, à la place des trois lignes de code associées à ce cas, associer ce code : «(count right) + (count left)». Cependant, dans le but de comparer de manière plus pédagogique cette fonction avec sa version concurrente, j'ai décidé d'y aller avec ces trois lignes de code.

Voici une définition de la fonction *count* dans laquelle on utilise la classe *future* pour optimiser le temps de calcul :

```
# let rec pcount tree = match tree with
| Leaf n when est_premier n -> 1
| Leaf _ -> 0
| Node(left, right) ->
  let f = new future (fun () -> pcount left) () in
  let nr = pcount right in
  let nl = f#get in
  nl + nr;;
val pcount : intTree -> int = <fun>

# timeRun pcount t;;
- : int * float = (4, 0.104544878005981445)
```

Comme on peut le constater, le gain en performance est pas mal significatif ! La seule différence avec la version séquentielle se situe au niveau du traitement des nœuds d'un arbre : à chaque nœud, on crée un fil d'exécution, à travers la classe *future*, qui va s'exécuter en parallèle avec le reste du traitement. Ce fil d'exécution s'occupe du traitement du sous-arbre gauche tandis que le reste du calcul s'occupe du sous-arbre droit. Lorsque ce dernier calcul termine, on n'a juste qu'à attendre, si nécessaire, à travers la méthode *get*, le calcul du *future*, récupérer son résultat et l'additionner à celui obtenu du côté droit de l'arbre et retourner la somme en résultat !

Le gain en performance est très significatif car à chaque nœud de l'arbre, on crée un fil d'exécution qui s'exécute avec le reste du calcul, soit le fil d'exécution courant ; par conséquent, à chaque nœud de l'arbre, on dispose en quelque sorte de deux fils d'exécution qui s'exécute en parallèle. Comme l'arbre a une hauteur égale à 4, par conséquent, il dispose de 15 nœuds ($2^4 - 1$) et puisqu'à chaque nœud, le fil d'exécution courant en crée un nouveau, par conséquent on dispose en tout de 15 fils d'exécution à l'œuvre ! Pour s'en convaincre, il suffit d'ajouter un effet de bord dans la fonction afin d'afficher l'identifiant du fil d'exécution courant : on peut alors constater qu'il y a 8 fils d'exécution différents en jeu :

```
# let rec pcount' tree =
  let _ = pr1 (sprintf "T%d_" (id(self()))) in
  match tree with
  | Leaf n when est_premier n -> 1
  | Leaf _ -> 0
  | Node(left, right) ->
    let f = new future (fun () -> pcount' left) () in
    let nr = pcount' right in
    let nl = f#get in
    nl + nr;;
val pcount' : intTree -> int = <fun>

# timeRun pcount' t;;
T0    T0    T0    T0    T53    T53    T56    T50
T50    T50    T59    T62    T62    T65    T47    T47
T47    T68    T74    T74    T77    T71    T71    T71
T83    T83    T86    T89
- : int * float = (4, 0.102667093276977539)
```

Par ailleurs, notons que pour ce type de code, il aurait été possible d'omettre l'utilisation de la classe *future*. La création, de manière explicite d'un fil d'exécution, associée à l'utilisation explicite d'un canal de communication, font parfaitement l'affaire :

```
# let rec pcount '' tree = match tree with
| Leaf n when est_premier n -> 1
| Leaf _ -> 0
| Node(left, right) ->
  let c = new_channel() in
  let _ = spawn (fun () -> send_b c (pcount '' left)) in
  let nr = pcount '' right in
  let nl = recv_b c in
  nl + nr;;
val pcount '' : intTree -> int = <fun>

# timeRun pcount '' t;;
- : int * float = (4, 0.102667093276977539)
```

On remarque même un gain de performance appréciable par rapport aux précédentes versions. Ceci est dû à l'*overhead* créé par la classe *future*. Par contre, si on aurait eu à écrire un code dans lequel plusieurs fils d'exécution auraient à atteindre la fin de plusieurs autres fils d'exécution, la classe *future* s'imposerait alors car autrement, le code aurait été plus fastidieux, même si plus performant.

Vers une version plus générique Dans ce qui suit, nous nous fixons comme objectif de rendre plus générique les traitements présentés précédemment. Plus précisément, on considère un type de données *arbre* générique dans lequel on n'est pas limité à n'utiliser que des entiers :

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

On considère alors une nouvelle version de la fonction *count* qui, contrairement à la version originale, prend en plus de l'argument *tree*, un argument *p* précisant la condition à tester au niveau de chaque feuille et un argument *o* qui admet des méthodes permettant de représenter la notion de somme de deux résultats, la valeur *un* et la valeur *zéro*¹⁷ :

```
# let gcount p o tree =
  let rec aux tree = match tree with
  | Leaf n when p n -> o#un n
  | Leaf n -> o#zero n
  | Node(left, right) -> o#plus (aux left) (aux right)
  in
  aux tree;;
val gcount : ('a -> bool) ->
< plus : 'b -> 'b -> 'b; un : 'a -> 'b; zero : 'a -> 'b; .. > ->
'a tree -> 'b = <fun>
```

On peut apprécier le type générique inféré de cette fonction.

Par ailleurs, on considère aussi une nouvelle version de la fonction, *gpcount*, version générique et concurrente :

17. Étant donné que la fonction *gcount* admet plusieurs paramètres qui ne changent pas de valeur d'une récursion à l'autre, on définit une fonction *aux* qui évite alors de passer ces arguments d'un appel récursif à l'autre; de plus, à la place des trois lignes de code qu'on a attribué au cas «Node(left,right)» dans la version originale de la fonction, on considère la version abrégée de ce code.

```
# let n_cores = 4;;
val n_cores : int = 4

# let gpcount p o tree =
  let rec aux tree depth = match tree with
  | _ when 2. ** float_of_int(depth) >= float_of_int(n_cores) ->
    gcount p o tree
  | Leaf n when p n -> o#un n
  | Leaf n -> o#zero n
  | Node(left, right) ->
    let f = new future (fun () -> aux left (depth+1)) () in
    let nc = aux right (depth+1) in
    o#plus (f#get) nc
  in
  aux tree 0;;
val gpcount : ('a -> bool) ->
< plus : 'b -> 'b -> 'b; un : 'a -> 'b; zero : 'a -> 'b; .. > ->
'a tree -> 'b = <fun>
```

Dans cette version concurrente, on considère une variable globale qui indique le nombre de cœurs disponibles sur la machine (d'habitude, cette information est accessible via une API système). Si on se situe à une profondeur de l'arbre dans laquelle on sait que l'on a déjà créé autant de fils d'exécution que de cœurs disponibles, alors on peut arrêter la création de ces fils d'exécution et, pour le reste de l'arbre, procéder au comptage de manière séquentielle.

Grâce à l'évaluation partielle, et grâce au fait que les fonctions sont définies de manière curryfiée, il est très aisé de retrouver les versions précédentes des fonctions *count* et *pcount* :

```
# let count = gcount est_premier (object method un = fun _ -> 1
                                method zero = fun _ -> 0
                                method plus = (+) end);;

val count : int tree -> int = <fun>

# timeRun count t;;
- : int * float = (4, 1.61199784278869629)

# let pcount = gpcount est_premier (object method un = fun _ -> 1
                                method zero = fun _ -> 0
                                method plus = (+) end);;

val pcount : int tree -> int = <fun>

# timeRun pcount t;;
- : int * float = (4, 0.404525995254516602)
```

On peut supposer plus de cœur à l'ouvrage et constater les gain en performance :

```
# let n_cores = 8;;
val n_cores : int = 8

...

# timeRun pcount t;;
- : int * float = (4, 0.209215879440307617)
```

Pour que le précédent code fonctionne, il faut au préalable réévaluer la fonction *tree* afin qu'elle tienne compte de la nouvelle définition du type *arbre* (qui devient générique) et réévaluer la valeur *t* pour qu'elle tienne compte de la nouvelle version de la fonction *tree*.

Dans le même ordre d'esprit que les exemples précédents, on peut, à partir des fonctions génériques *gcount* et *gpcount*, extraire différents types de fonctions :

```

# let isEven n = delay 0.1; n mod 2 = 0;;
val isEven : int -> bool = <fun>

# let countObject = object method un = fun _ -> 1
                           method zero = fun _ -> 0
                           method plus = (+) end;;
val countObject : < plus : int -> int -> int; un : 'a -> int;
                  zero : 'b -> int > = <obj>

# let listObject = object method un = fun x -> [x]
                           method zero = fun _ -> []
                           method plus = (@) end;;
val listObject : < plus : 'a list -> 'a list -> 'a list; un : 'b -> 'b list;
                  zero : 'c -> 'd list > = <obj>

# let countPrime = gcount est_premier countObject;;
val countPrime : int tree -> int = <fun>
# let listPrime = gcount est_premier listObject;;
val listPrime : int tree -> int list = <fun>

# let countEven = gcount isEven countObject;;
val countEven : int tree -> int = <fun>
# let listEven = gcount isEven listObject;;
val listEven : int tree -> int list = <fun>

# let pcountPrime = gpcount est_premier countObject;;
val pcountPrime : int tree -> int = <fun>
# let plistPrime = gpcount est_premier listObject;;
val plistPrime : int tree -> int list = <fun>

# let pcountEven = gpcount isEven countObject;;
val pcountEven : int tree -> int = <fun>
# let plistEven = gpcount isEven listObject;;
val plistEven : int tree -> int list = <fun>

```

Par la suite, il est possible de tester ces nouvelles fonctions définies simplement par évaluation partielle des fonctions génériques :

```

# timeRun countPrime t;;
- : int * float = (4, 1.61339807510375977)

# timeRun listPrime t;;
- : int list * float = ([20341; 9377; 92593; 26921], 1.61220216751098633)

# timeRun countEven t;;
- : int * float = (5, 1.61548089981079102)

# timeRun listEven t;;
- : int list * float =
([76418; 18552; 88568; 65412; 20166], 1.61017513275146484)

# timeRun pcountPrime t;;
- : int * float = (4, 0.205718040466308594)

# timeRun plistPrime t;;
- : int list * float = ([20341; 9377; 92593; 26921], 0.204905986785888672)

# timeRun pcountEven t;;
- : int * float = (5, 0.203832149505615234)

# timeRun plistEven t;;
- : int list * float =
([76418; 18552; 88568; 65412; 20166], 0.208015918731689453)

```

Vers une version générique plus optimisée ! ? En profitant davantage des aspects fonctionnels du langage O'Caml, on pourrait envisager des fonctions plus performantes que celles déjà présentées. Comment ? En mémorisant par exemple la fonction *est_premier* qui, on le sait, requiert beaucoup de temps de calcul.

En premier, on rappelle la définition de la fonction *memoize* :

```
# let memoize f =
  let cache = ref [] in
  fun x ->
    if mem_assoc x !cache then
      assoc x !cache
    else
      let r = f x in
      cache := (x,r) :: !cache;
      r;;
val memoize : ('a -> 'b) -> 'a -> 'b = <fun>
```

Par la suite, on mémorise la fonction *est_premier* et on génère de nouveau la fonction *countPrime* :

```
# let m_est_premier = memoize est_premier;;
val m_est_premier : int -> bool = <fun>

# let countPrime' = gcount m_est_premier countObject;;
val countPrime' : int tree -> int = <fun>

# timeRun countPrime' t;;
- : int * float = (4, 1.61157798767089844)
```

On ne remarque pas de gain de performance; par contre, si on évalue de nouveau la dernière expression, les gains sont clairement là! En effet, dans cette nouvelle évaluation, toutes les valeurs à calculer sont déjà présentes dans le cache de la fonction *m_est_premier*.

```
# timeRun countPrime' t;;
- : int * float = (4, 8.17775726318359375e-05)
```

De plus, si on définit de nouveau la fonction *listPrime* avec la version mémorisée de *est_premier* (dont le cache est désormais rempli de paires (*valeur*, *résultat*), grâce à la notion de «fermeture»), on remarque aussitôt le gain de performance (0.sec) :

```
# let listPrime' = gcount m_est_premier listObject;;
val listPrime' : int tree -> int list = <fun>

# timeRun listPrime' t;;
- : int list * float = ([20341; 9377; 92593; 26921], 9.2983245849609375e-05)
```

La version concurrente ne semble plus surpasser cette version «mémorisée» de la fonction séquentielle *listPrime*. Cependant, dans un contexte où les calculs prennent réellement beaucoup de temps, on percevrait rapidement des gains en performance.

Exercices

7.1 *Que fait la fonction suivante : `let rec f i = (send_b c i; f(i+1))`*

7.2 *En utilisant la définition précédente, dites ce que fait l'expression suivante :*

`spawn (fun () -> f(0)) ()`

7.3 *Définir une fonction serveur qui lit sur un canal reqCh les requêtes de fils d'exécution clients, et qui répond sur le canal repCh les réponses correspondant à ces requêtes.*

7.4 *Étant donnée la fonction serveur définie dans l'exercice précédent, définir une fonction client qui envoie une requête au serveur puis traite la réponse retournée.*

Corrigés

10-1 :

Cette fonction envoie sur un canal *c* une série d'entiers.

10-2 :

Cette expression permet d'énumérer dans l'ordre tous les entiers naturels.

10-3 :

```
let rec serveur () =  
  let requete = recv_b reqCh in  
  send_b repCh (traitementServeur requete);  
  serveur();;
```

10-4 :

```
let client r =  
  send_b reqCh r;  
  traitementClient (recv_b repCh);;
```


Quatrième partie

Annexe

Chapitre 8

Smalltalk

Dans les années 70, à Xerox Palo Alto Research Center (Park), Alan KAY proposa un modèle de programmation qui s'avèrera être le précurseur du premier langage orienté objet. L'objectif initial du projet était de concevoir un outil susceptible d'être utilisé aussi bien pour enseigner la programmation que pour développer des activités éducatives. La vocation de ce projet s'est étendue à l'écriture d'interfaces personne-machines évoluées, de compilateurs, de systèmes d'exploitation, etc. Ces travaux aboutirent à la première version du langage : **Smalltalk 72**.

En 1976, D.H. INGALLS présenta le système **Smalltalk 76** comme un système simple et performant. Ce système se démarque de son prédécesseur par sa faculté d'unifier les classes et les contextes en tant qu'objets. Dans cette optique toute entité **Smalltalk** est un objet. Ces recherches se sont développées et se sont orientées jusqu'au point de devenir plus conceptuelle qu'éducatives.

En 1980, GOLDBERG proposa le langage **Smalltalk 80**. Ce dernier définit une norme et apporte une généralisation et une unification des classes et méta-classes : Chaque classe étant désormais associée à une méta-classe qui lui est propre. Depuis, plusieurs mises en oeuvre du modèle **Smalltalk 80** ont été réalisées sur différentes plate-formes (Unix, Dos, Windows, Mac, etc.). Ces modèles ont abouti à des langages tels que : **Smalltalk 80**, **Objective Smalltalk**, **Smalltalk V**, **GNU Smalltalk**, **Smalltalk X**, **Smalltalk.NET**, etc.

Toutes ces versions ont cependant un point commun. Elles proposent presque toutes un environnement de développement comportant :

- Un environnement interactif multi-fenêtres.
- Un interprète du langage.
- Un éditeur de texte.
- Un ensemble d'outils symboliques.
- Un débogueur symbolique.
- Un système de gestion de fichiers.
- Un système de gestion de processus.
- Un système de gestion de la mémoire.
- Un environnement de développement graphique.
- Un environnement de développement d'interfaces personne-machine.

En fait, le langage **Smalltalk** tire ses origines de plusieurs autres langages :

- Simula pour les notions de classes et d'héritage.
- Lisp pour l'interactivité et l'applicativité.
- Algol pour l'aspect impératif.
- Planner et Plasma pour la notion de messages.

Cette diversité procure une très grande expressivité au langage. Parmi les applications pouvant être développées avec **Smalltalk**, nous citons :

- Interfaces personne-machine.
- Applications graphiques.

- Applications interactives.
- Applications d'intelligence artificielle.
- Applications de simulation.

Aussi, Smalltalk excelle en matière de prototypage (élaboration de maquettes).

8.1 Environnement de développement

Cette section décrit l'environnement de développement Squeak¹ (version 3.9) qui implante le langage Smalltalk 80 avec quelques ajouts. Cet environnement est multi-fenêtres et indépendant de la plate-forme d'accueil. Squeak tourne aussi bien sous Unix, Mac ou Windows. Il offre des classes qui masquent les différences de ces systèmes en uniformisant leur utilisation. De même, toutes les références systèmes classiques, comme par exemple l'accès aux fichiers, sont indépendants de la plate-forme.

En fait, Squeak constitue une machine virtuelle. L'état de cette machine est défini dans une image (en général, un fichier avec une extension « `.image` »). Cette technique permet le portage des applications construites sous Squeak. En effet, il suffit de réécrire l'exécutable pour une machine différente pour pouvoir relancer la machine virtuelle Squeak. Remarquons que le code source de Squeak est libre, nous pouvons donc compiler la machine virtuelle pour une plate-forme non supportée si nécessaire.

8.1.1 Installation

Pour programmer en Smalltalk, il faut disposer d'une machine Squeak². Celle-ci, comme la plupart des environnements de développement Smalltalk, est composée de quatre éléments essentiels :

- Le fichier exécutable **Squeak**. Ce fichier constitue la machine virtuelle.
- Le fichier image qui décrit l'état de la machine virtuelle. Le fichier qui accompagne l'environnement est **Squeak3.9-final-7067.image**.
- Le code source des classes de Squeak désigné par le fichier **SqueakV39.sources**.
- Le fichier **Squeak3.9-final-7067.changes** qui décrit le journal (« log ») de tout ce qui s'est produit (ajout/modification de classes, méthodes, ...) dans votre application Squeak depuis sa création. Ce journal associé au fichier image permet, entre autres, de conserver et de retrouver les versions antérieures des classes et méthodes développées.

Pour lancer Squeak, il faut lancer le fichier exécutable. Par défaut, Squeak utilise le fichier image qui est dans son répertoire courant. S'il y a plus d'un fichier image dans ce répertoire, Squeak demandera lequel utiliser.

8.1.2 Environnement

Une fois Squeak lancé, trois fenêtres et deux onglets déroulants apparaissent (figure 8.1) :

- Trois fenêtres intitulées *Workspace* : Ces fenêtres donnent de l'information sur la version courante du système, son historique et le projet Squeak en général. Après avoir pris connaissance de ces informations, ces fenêtres peuvent être fermées (en utilisant le bouton de fermeture dans le coin supérieur gauche des fenêtres).
- L'espace gris situé entre les fenêtres est l'objet monde de Squeak (« World »).
- L'onglet déroulant *Squeak* permet d'effectuer des opérations et d'avoir de l'information sur la machine virtuelle Squeak. Cet onglet permet, entre autres, de configurer Squeak selon nos besoins et de sauvegarder l'état courant de la machine virtuelle.
- L'onglet déroulant *Tools* est le guide principal de navigation dans Squeak. Il permet de créer rapidement des fenêtres (outils) très utiles pour la programmation en Smalltalk. Pour

1. Veuillez noter que le contenu de ce chapitre, notamment les figures, a été adapté à l'environnement Squeak par M. Claude Bolduc, étudiant au doctorat à l'Université Laval.

2. Squeak est disponible à l'adresse internet : <http://www.squeak.org/> .

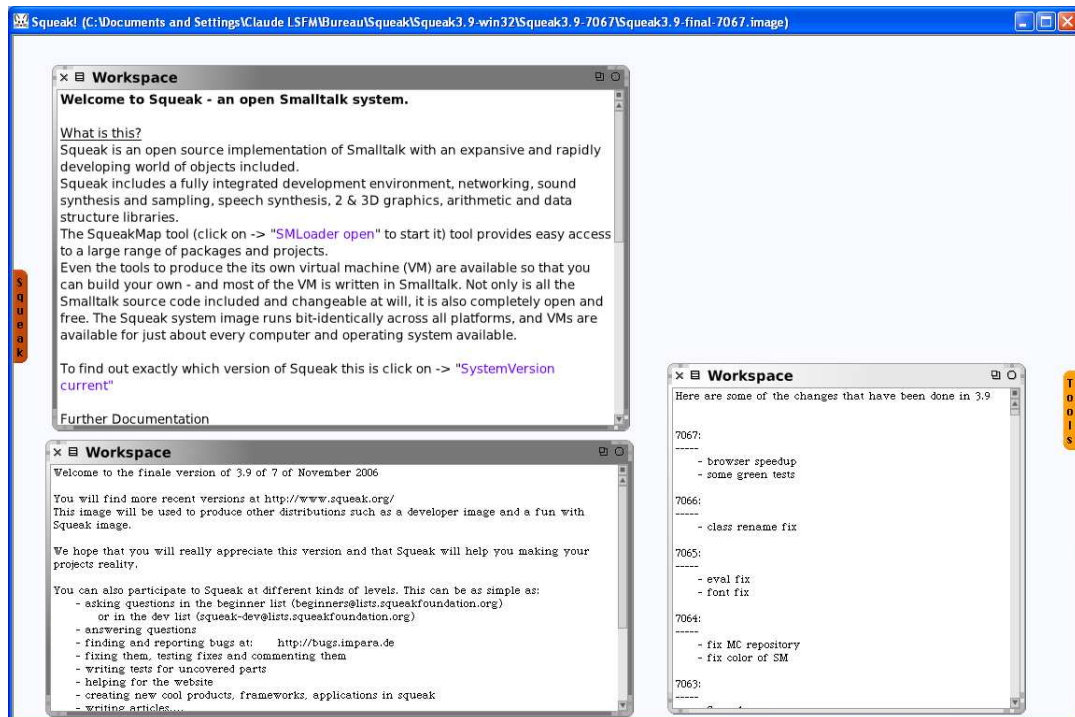


FIGURE 8.1 – Fenêtres au démarrage de Squeak

cela, il suffit de glisser-déplacer une icône de l'onglet directement sur le monde. Cet onglet comporte les éléments suivants :

- *Browser* : Crée une fenêtre *System Browser* qui permet de parcourir l'ensemble des classes, méthodes et du code source de Squeak. Elle permet donc de rechercher de l'information sur le système. Aussi, cette fenêtre est le principal interface pour ajouter des classes et des méthodes au système Squeak.
- *Transcript* : Crée une fenêtre *Transcript* qui permet de présenter de l'information textuelle surtout dans le cadre de journal (« log »).
- *Workspace* : Crée une fenêtre *Workspace* qui est une fenêtre de travail utilisée pour éditer et évaluer des expressions Smalltalk. Normalement, les expressions Smalltalk dans cette fenêtre ont une durée de vie très limitée (rarement plus de quelques heures). Nous nous en servons pour tester rapidement des bouts de code.
- *Change Sorter* : Crée une fenêtre *Changes go to ...* qui permet de comparer deux fichiers portant l'extension *.changes*.
- *Method Finder* : Crée une fenêtre *Selector Browser* qui permet de rechercher toutes les classes de Squeak qui implémentent une méthode désirée. Il est possible d'utiliser juste un nom partiel d'une méthode (exemple : `error` pour trouver `criticalError`).
- *Message Names* : Crée une fenêtre *Message names containing ...* qui est très semblable à la fenêtre *Selector Browser* précédemment décrite excepté que cette nouvelle fenêtre permet de visualiser aussi directement le code source de la méthode.
- *Preferences* : Crée une fenêtre *Preferences* qui permet de définir plusieurs préférences pour notre système. Par exemple, nous pouvons définir l'option de demander une validation avant de fermer une fenêtre quelconque.
- *Recent* : Crée une fenêtre *Recent submissions* qui affiche les messages qui ont été envoyés (méthodes appelées) en ordre du dernier message envoyé au premier message.

- *Processes* : Crée une fenêtre *Process Browser* qui affiche et permet de gérer les processus actifs de l'environnement Squeak.
- *Annotations* : Crée une fenêtre *Annotations* qui permet de spécifier les annotations (à la manière de glisser-déplacer) à afficher dans une fenêtre d'annotation.
- *Packages* : Crée une fenêtre *Package Browser* qui est très semblable à la fenêtre *System Browser* excepté que cette nouvelle fenêtre affiche aussi les paquetages contenant les classes du système.
- *Change Set* : Crée une fenêtre *Changes go to ...* qui permet de manipuler un fichier portant l'extension `.changes`.
- *File List* : Crée une fenêtre qui permet de naviguer dans les fichiers sur le disque dur ou sur un site FTP. En particulier, cette fenêtre permet d'installer des classes *Smalltalk* stockées sur le disque.
- *SUnit Runner* : Crée une fenêtre *Test Runner* qui permet d'effectuer et de gérer les tests unitaires sur le système en cours.

8.1.3 Premiers pas

L'interaction avec le système se fait principalement à l'aide de la souris et ses trois boutons. Ces derniers n'ont pas toujours la même signification (dépend de la configuration et de la plate-forme d'accueil) :

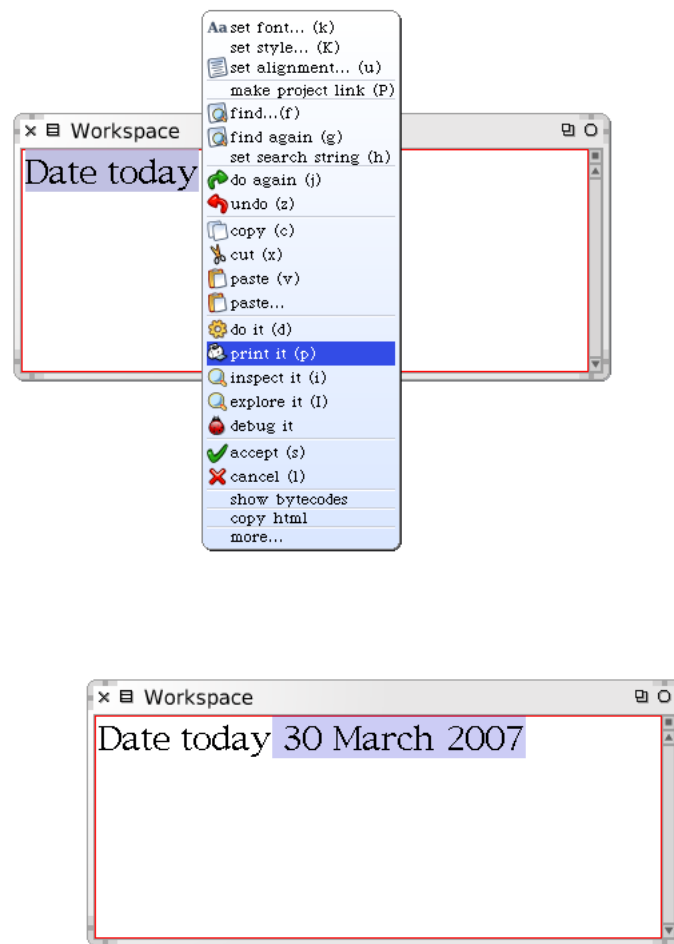
- Le bouton gauche est utilisé pour sélectionner un objet, une position ou une portion de texte à évaluer. Il y a une exception : Sur le « monde », ce bouton appelle un menu contextuel.
- Le bouton du milieu (bouton droit sous Windows) appelle un menu local offrant des actions à réaliser sur la sélection. Bien que ce menu soit contextuel (différent suivant la nature de l'objet sur lequel il est appliqué), il comprend un ensemble d'actions uniformes à tous les contextes. Par exemple, les actions permettant de copier/coller du texte sont disponibles dans presque tous les contextes.
- Le bouton droit (bouton du milieu sous Windows) appelle un menu global (appelé *halo*) à la fenêtre concernée, pour redimensionner, fermer, quitter, etc.

Évaluation

Une session *Smalltalk* ordinaire consiste à ouvrir une fenêtre *Workspace* et à évaluer des expressions. Cette évaluation se fait à l'aide de la souris. Une fois l'expression saisie, il faut la sélectionner avec le bouton gauche puis, à l'aide du bouton ouvrant le menu contextuel, choisir une des actions proposées pour l'évaluation de cette expression :

- L'action `do it` évalue l'expression sans en afficher le résultat. Ce dernier n'est visible qu'à travers des effets de bord provoqués, par exemple, en envoyant le message `show:`, suivie d'une chaîne de caractères à afficher, à l'objet *Transcript*.
- L'action `print it` envoie au résultat de l'expression un message d'impression. Ceci a pour effet d'afficher ce résultat dans la fenêtre dans laquelle a eu lieu l'évaluation. La figure 8.2 illustre l'utilisation de cette action. Celle-ci est appliquée à l'expression `Date today` et retourne comme résultat l'objet ayant la valeur `30 March 2007`. Dans cet exemple, l'expression utilisée est particulière. En effet, elle illustre l'envoi d'un message, `today`, à la classe `Date`. Une particularité de *Smalltalk* est que les classes sont aussi des objets pouvant recevoir des messages. Dans ce cas, le message envoyé permet d'instancier la classe `Date` pour obtenir un objet dont la valeur initiale correspond à la date courante.
- L'action `inspect it` envoie un message d'inspection au résultat de l'évaluation de l'expression. Ceci a pour effet d'ouvrir un inspecteur sur l'objet résultant de l'évaluation de cette expression. Tous les objets peuvent répondre à ce message. En plus d'inspecter l'objet, l'inspecteur permet d'éditer l'objet pour y changer éventuellement son contenu ou son état.

La figure 8.3 décrit l'action d'inspection du tableau `#(5 2 4 3 1)`. Cette action affiche une fenêtre dont le titre correspond à la classe de laquelle est instanciée l'objet inspecté (dans ce cas, la

FIGURE 8.2 – Évaluation à l'aide de l'action `print it`.

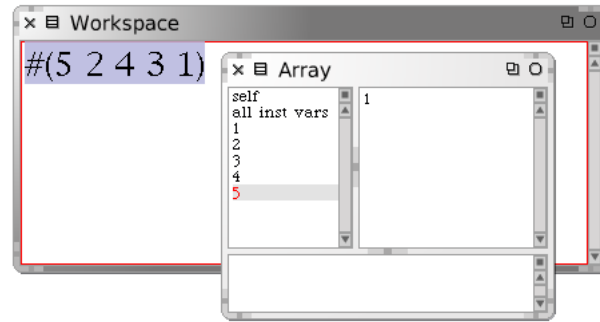


FIGURE 8.3 – Inspection d'un objet.

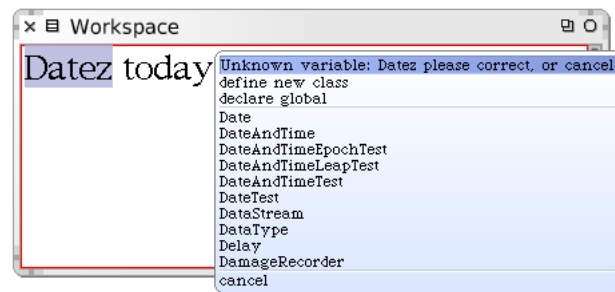


FIGURE 8.4 – Utilisation d'une référence inconnue.

classe `Array`). Cette fenêtre est subdivisée en trois parties : La partie de gauche affiche la liste des variables d'instances de l'objet ou une liste des objets contenus dans une liste ou un dictionnaire³. La partie de droite correspond à un éditeur qui affiche et édite l'objet sélectionné dans la liste de gauche. La partie du bas est un éditeur de textes qui permet d'exécuter des expressions `Smalltalk` relative à l'objet inspecté.

Dans cet exemple, la partie gauche est composée de la variable système `self`, de l'ensemble de toutes les variables d'instance de la liste (`all inst vars`), et de cinq numéros correspondant aux index du tableau. La partie de droite affiche le contenu du cinquième élément du tableau, soit la valeur 1.

Erreurs

Lors d'une évaluation, des erreurs peuvent survenir. Ces erreurs peuvent être regroupées en trois types :

- Référence à un objet inconnu : Dans ce cas, Squeak affiche une liste d'options comme l'illustre la figure 8.4. Dans chaque cas, l'évaluateur traite cet objet selon notre sélection.
- Message inconnu : Cette erreur apparaît lorsque nous essayons d'envoyer un message à un objet qui ne l'implémente pas (figure 8.5). Comme l'illustre la figure, Squeak propose plusieurs solutions :
 - *Proceed* : l'évaluateur fait comme si le message existait.
 - *Abandon* : abandonne l'évaluation.
 - *Debug* : ouvre le débogueur pour permettre de comprendre le problème.
 - *Create* : permet de créer la méthode non implémentée.

De plus, Squeak montre l'état de la pile d'exécution au moment de l'erreur.

3. Un dictionnaire est une collection d'associations `clé -> valeur`.

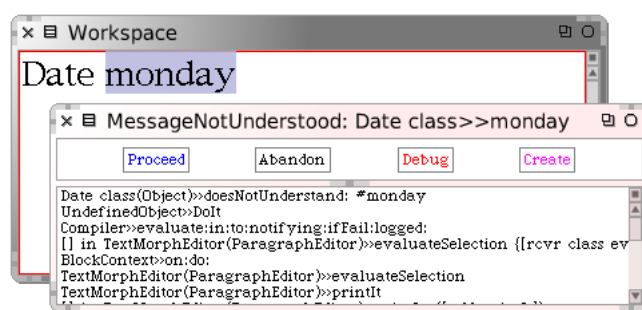


FIGURE 8.5 – Envoi d'un message inconnu.

- Une fenêtre d'erreur s'ouvre dans le cas où Squeak détecte une mauvaise utilisation de certains objets. Le titre de ce message indique la nature de l'erreur.

Navigation

Le langage Smalltalk 80 possède initialement environ 700 classes. Squeak fournit des classes supplémentaires qui sont surtout spécialisées dans le développement multimédia. En tout, la version 3.9 de Squeak possède 2040 classes. Toutes ces classes sont regroupées en catégories qui sont des ensembles de classes. Par exemple, la catégorie **Graphics-Primitives** contient les classes **Point**, **Rectangle**, etc. Cette structuration aide à retrouver la définition d'une classe donnée. Mais devant le nombre considérable de classes proposées, Squeak offre un outil graphique, le navigateur système (*system browser*), permettant d'accéder rapidement à une classe et de naviguer dans sa définition. Cet outil se situe dans l'onglet déroulant *Tools* et il correspond à une fenêtre composée de cinq sous-fenêtres :

- La première fenêtre affiche les catégories des classes du système.
- En sélectionnant une catégorie de classes, la deuxième fenêtre affiche les classes contenues dans la catégorie sélectionnée, et la cinquième fenêtre affiche le prototype d'une définition de classes. De plus, la deuxième fenêtre comprend trois boutons *instance*, *?* et *class*. Le premier bouton et le troisième bouton permettent de spécifier le type d'interfaces et de méthodes désirées. En effet, une classe étant un objet dans Squeak, elle admet donc des méthodes. Par exemple, le message **new** envoyé à la classe **Set** permet de créer un objet représentant un ensemble. Ces méthodes sont accessibles à travers le troisième bouton. Le premier bouton permet de connaître les méthodes définies pour les instance de cette classe. Le deuxième bouton permet d'afficher les commentaires sur la classe sélectionnée.
- En sélectionnant une classe, la troisième fenêtre affiche la liste des protocoles ou interfaces qu'offrent cette classe, alors que la cinquième fenêtre se sépare en deux parties pour afficher dans la partie supérieure la définition de la classe (les variables d'instances, les variables de classe, ...) et pour afficher dans la partie inférieure les commentaires de la classe. À l'instar des catégories, une interface regroupe plusieurs méthodes de la classe. Par exemple, la classe **Set** offre une interface *adding* qui, comme son nom l'indique, regroupe un ensemble de méthodes permettant d'ajouter un élément à un ensemble. L'interface *-- all --* regroupe en fait l'ensemble des méthodes possibles de la classe sélectionnée.
- En sélectionnant une interface, la quatrième fenêtre affiche la liste des méthodes (appartenant à cette interface) offertes par la classe, alors que la cinquième affiche le prototype d'une définition de méthodes. Par exemple, l'interface *adding* propose, entre autres, la méthode **add:**. Pour ajouter une méthode à cette classe, il est conseillé de s'inspirer du prototype de l'interface à laquelle la méthode appartiendra.
- En sélectionnant une méthode donnée, la cinquième et dernière fenêtre affiche la définition de cette méthode.

Aussi, Squeak permet de naviguer dans la hiérarchie des classes grâce à l'outil *hierarchy browser*. Cet outil est très semblable au navigateur système excepté que les deux premières fenêtres de catégories et de classes sont remplacées par une unique fenêtre affichant la hiérarchie des classes du système. Cet outil est accessible en double-cliquant sur le nom d'une classe du navigateur système (donc en double-cliquant sur un élément de la deuxième fenêtre du navigateur système).

L'utilisation de ces deux navigateurs jumelée avec une fenêtre de travail (*Workspace*) permet de découvrir toutes la puissance de ce langage.

8.2 Langage Smalltalk

8.2.1 Principes de base

Le langage Smalltalk s'articule autour de trois notions fondamentales : les classes, les objets et les messages. Plus précisément, tout le langage est basé sur l'expression suivante :

objet message

c'est-à-dire à l'envoi d'un message à un objet. Comme nous allons le constater tout au long de ce chapitre, tout en Smalltalk (ou presque) est défini à partir de cette simple expression.

Par ailleurs, Smalltalk est fondé sur les quatre principes suivants :

Toute entité Smalltalk est un objet

Les objets sont les éléments de base du langage Smalltalk. Ils sont analogues aux valeurs calculables des autres langages. Chaque objet est décrit par deux composantes :

- Un espace mémoire privé (état de l'objet). Tout accès aux données privées de l'objet se fait par l'intermédiaire de ses méthodes.
- Un ensemble d'opérations (comportement de l'objet). Cet ensemble constitue ce qui est communément appelé l'interface de l'objet avec le système.

Cette description unifie les notions de données et de procédures des langages impératifs.

Tout objet Smalltalk est instance d'une classe

Une classe est un modèle qui décrit la structure et le comportement commun à un ensemble d'objets créés selon ce modèle par instanciation. Elle décrit la structure de la partie « données » de ses instances, les champs (variables d'instances), ainsi que la manière dont les instances réalisent leur interface. D'après le premier principe, une classe est aussi un objet : elle est instance d'une autre classe qui lui est associée. Cette classe est dite méta-classe.

Tout objet est activé à la réception d'un message

L'unique structure de contrôle de l'objet est la transmission de messages. Un envoi de message mentionne le destinataire, un sélecteur et des arguments éventuels. En recevant un message, l'objet exécute la méthode associée au sélecteur sur les valeurs représentées par les arguments.

Toute classe est sous-classe d'une autre classe

Il est possible de décrire un sous-ensemble d'objets, appartenant à une même classe et ayant des particularités propres, par une sous-classe de la classe spécialisée. Cette classe spécialisée est alors désignée par super-classe. L'ensemble des classes forment donc une hiérarchie. La spécialisation en Smalltalk permet d'avoir une représentation en arbre de cette hiérarchie. Les noeuds pendants correspondent aux classes les plus fines, tandis que la racine est la classe la plus générale : **Object**. Cette classe fait exception à ce principe. Elle est sous-classe d'aucune autre classe. Remarquons

que dans l'environnement Squeak, la classe `Object` n'est pas la plus générale (`ProtoObject` est cette classe générale), mais ceci est un détail mineur.

8.2.2 Objets

Les objets sont les éléments de base de `Smalltalk`. Ils sont analogues aux différentes structures de données des autres langages. Par exemple, les expressions suivantes définissent des objets `Smalltalk` :

```
'Hello World'
1234
$A
#(1 2 3)
#(3 'chaine' ($A $B $C))
```

Le premier exemple est une chaîne de caractères, le deuxième est un entier, le troisième est le caractère `A`, le quatrième est un tableau contenant trois entiers alors que le cinquième est un tableau contenant à la fois un entier, une chaîne de caractères et un autre tableau de caractères.

Dans `Smalltalk`, comme c'est le cas dans tous les langages orientés objets, les messages sont utilisés pour activer les méthodes d'un objet. Les messages sont similaires aux appels de fonctions dans les langages de programmation structurée. À l'instar de ces derniers, dans lesquels un programme est décrit comme un ensemble de déclarations suivies d'un ensemble d'appels de fonctions, un programme `Smalltalk` correspond à un ensemble de déclarations de classes et d'objets suivies d'un ensemble d'envois de messages vers ces objets et ces classes. La particularité de `Smalltalk` est que les classes sont traitées comme des objets auxquels il est possible d'envoyer un message pour invoquer une de leurs méthodes.

8.2.3 Messages

Un envoi de message est composé de trois parties : l'objet récepteur, le sélecteur et les arguments. Dans l'exemple suivant, le récepteur est le tableau d'entiers, le sélecteur est `at:` et l'argument est l'entier 2 :

```
#(1 3 5 7) at: 2
```

Le résultat d'un envoi de message est toujours un objet. Dans l'exemple précédent, le résultat retourné est le deuxième élément du tableau, c'est-à-dire l'objet 3.

Messages unaires

Les messages dont l'ensemble d'arguments est vide sont dits messages unaires. Ils sont dits unaires car ils s'appliquent à l'objet récepteur et ce sans arguments. Par exemple, les expressions décrites dans la figure 8.6 utilisent des messages unaires.

Dans la figure 8.6 le résultat des différents messages est affiché à droite du message concerné. Le premier message correspond à l'envoi du sélecteur `open` à la variable globale `Transcript`. Ce message a pour effet d'ouvrir une nouvelle fenêtre *Transcript*. Le deuxième message retourne la classe de l'objet 10 résultant de l'expression $(6 + 4)$. Le troisième et le quatrième message retournent respectivement la date et l'heure courante. Le cinquième message retourne la taille de la chaîne de caractères `'Hello'`. Le sixième message permet de calculer la factorielle de 6. Le septième message retourne la classe de l'objet `#(1 2)`, soit `Array`. L'avant-dernier message retourne l'inverse d'un objet chaîne de caractères. Le dernier message teste si un objet caractère est une majuscule.

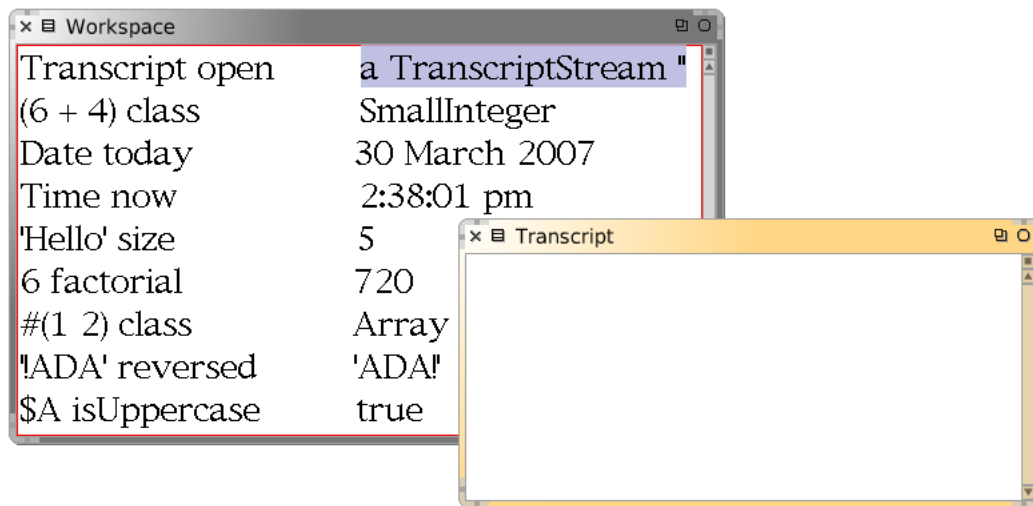


FIGURE 8.6 – Exemples de messages unaires.

Messages arithmétiques

Dans **Smalltalk**, les opérateurs arithmétiques sont aussi considérés comme des messages. Par exemple, l'expression $(3 + 4)$ est en fait un message dont l'objet entier 3 est le récepteur, l'opérateur + est le sélecteur (message binaire, car un récepteur et un argument) et l'entier est l'argument. Le résultat de ce message est l'objet entier 7.

Cet aspect des messages arithmétiques rend **Smalltalk** différent des autres langages, concernant l'évaluation des expressions arithmétiques. En effet, l'évaluation de ces expressions, dans **Smalltalk**, se fait de gauche à droite. Par exemple l'expression $3+4*2$ s'évalue en 14 et non en 11. La raison en est simple. Cette expression correspond en fait à deux messages imbriqués :

- Le premier message correspond à $3+4$. Le résultat de ce message est l'objet entier 7.
- Le deuxième message est $7*2$. Son évaluation résulte en l'objet entier 14.

Pour instaurer des propriétés sur l'évaluation, il est possible d'utiliser les parenthèses pour imposer un certain ordre. Par exemple, l'expression $3+(4*2)$ correspond au message dont le récepteur est l'objet entier 3, le sélecteur est l'opérateur + et l'argument est le résultat de l'expression $(4*2)$. Le résultat de cette évaluation est l'objet entier 11. De même, ce résultat est obtenu en changeant l'expression comme suit : $4*2+3$.

La figure 8.7 illustre l'évaluation de plusieurs expressions arithmétiques.

Dans ces exemples, les opérateurs //, \ et / représentent respectivement la division entière, le reste de la division entière et la division rationnelle.

Notons que dans **Smalltalk**, certaines classes représentent des objets « auto-décrits » comme **Integer** et **Float**, et n'ont pas, à ce titre, de variables d'instances. En effet, les objets sont, en général, caractérisés par des variables d'instances.

Les messages arithmétiques sont aussi désignés par « messages binaires ».

Messages à mots-clés

Ce sont les messages qui ont un nombre non-nul d'arguments autre que le récepteur. Supposons que l'on veuille accéder au quatrième élément d'un tableau. Dans ce cas, les messages unaires ne sont pas appropriés puisqu'ils ne permettent pas de spécifier des arguments. Les messages qui ont un ou plusieurs arguments sont appelés messages à mots-clés. La figure 8.8 illustre l'utilisation de tels messages dans **Smalltalk**. Le résultat de ces messages est affiché dans la partie droite de la fenêtre.

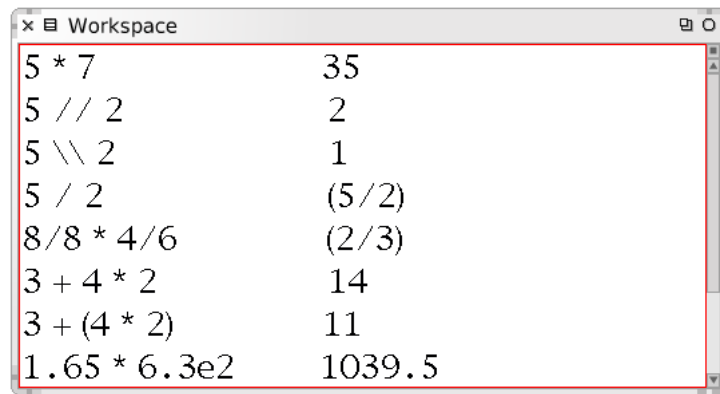


FIGURE 8.7 – Exemples de messages arithmétiques.

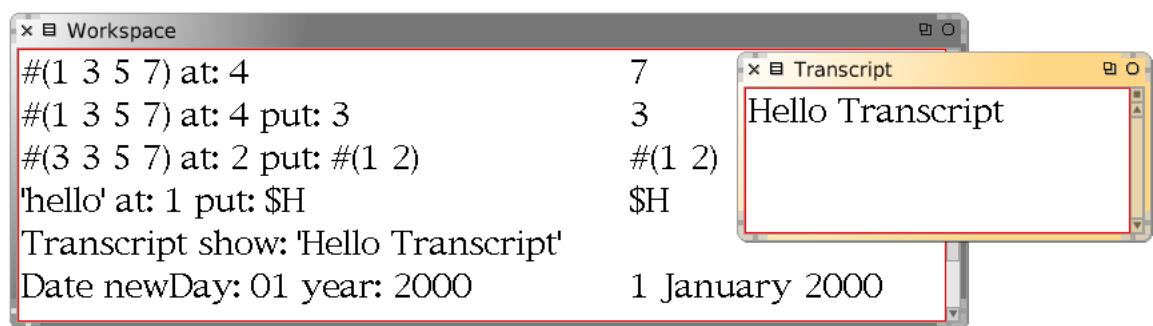


FIGURE 8.8 – Exemples de messages à mots-clés.

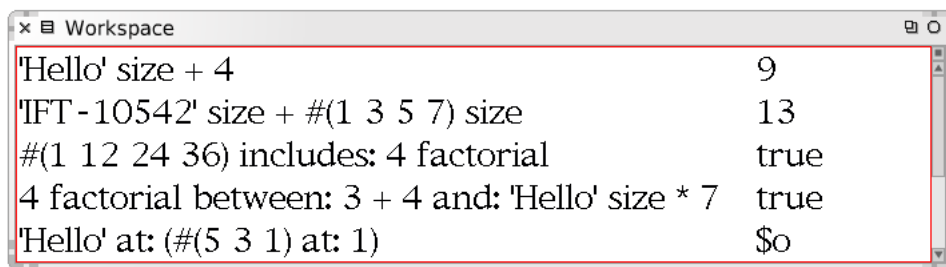


FIGURE 8.9 – Exemples d’envois de messages complexes.

Le premier message de la figure 8.8 permet de retourner le quatrième élément de l’objet tableau. Le deuxième et le troisième messages permettent de modifier respectivement le quatrième et le deuxième élément de deux objets tableaux. Le résultat de ce message est la nouvelle valeur de cet élément. Il convient de noter que dans le deuxième et le troisième messages le sélecteur correspond à `at: put:`. Le message suivant utilise le même sélecteur pour modifier le premier caractère d’une chaîne de caractères. L’avant dernier message permet d’afficher un texte dans une fenêtre *Transcript*, alors que le dernier permet de retourner une date à partir de l’année et de la position en jours de cette date dans l’année.

Expressions complexes

Nous avons mentionné que les messages, comme les fonctions retournent toujours un objet comme résultat. Donc, il est possible de remplacer un objet qui apparaît dans une expression par un message qui rend un objet de même type. La figure 8.9 illustre différentes expressions Smalltalk composées de plusieurs envois de messages imbriqués.

Le premier exemple utilise deux messages. L’un destiné à l’objet chaîne de caractères `'Hello'`, et l’autre destiné à l’objet entier résultant du premier message c’est-à-dire de, `'Hello' size`. Ce dernier retourne comme résultat l’objet entier 5. Cet objet est le récepteur du message `+ 4` qui retourne l’objet entier 9. De même, le deuxième exemple additionne la taille d’une chaîne de caractères avec celle d’un tableau. Le troisième exemple teste si l’objet résultant du message unaire `4 factorial` existe dans le tableau. Le résultat de cette expression est l’objet booléen `true`. Le quatrième exemple teste quant à lui si l’objet entier résultant du message `4 factorial` appartient à l’intervalle délimité par l’objet entier résultant du message `3 + 4` et de l’objet entier résultant du message complexe `'Hello' size * 7`. Le dernier exemple accède à un élément particulier d’une chaîne de caractères à partir d’un tableau de valeurs entières.

Comme mentionné précédemment, l’ordre d’évaluation des expressions se fait de gauche à droite. Cependant, lorsque l’expression comprend différents types de sous-expressions, l’ordre d’évaluation se fait en respectant les règles suivantes :

1. Les littéraux et les variables sont évalués avant les envois de messages. Les littéraux sont des expressions décrivant des objets constants (nombres, caractères, chaînes de caractères, tableaux de littéraux).
2. Les messages unaires sont évalués en premier.
3. Suivent les messages arithmétiques (aussi appelés « messages binaires »).
4. Puis les messages à mots-clés.

Notez que les expressions entre parenthèses modifient cet ordre et ont la plus haute priorité.

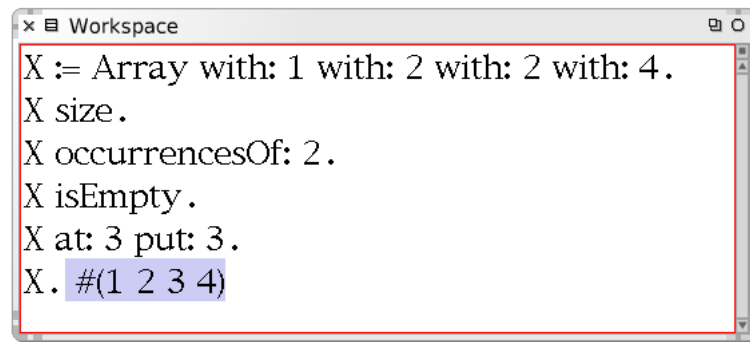


FIGURE 8.10 – Exemples de séquences de messages.

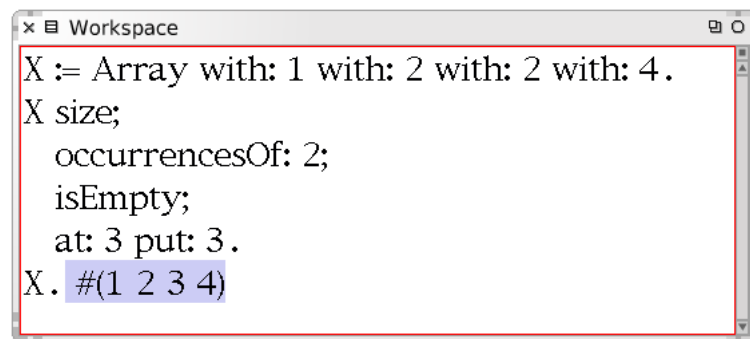


FIGURE 8.11 – Exemples d’envois de messages en cascade.

Séquences et cascades

Une évaluation Smalltalk consiste en une séquence de messages envoyés à divers objets. Chaque instruction (envoi de message) est séparée des autres par un point. La figure 8.10 donne l’exemple d’une séquence de messages envoyés à un objet tableau. La première instruction définit une variable globale *X* comme un tableau initialisé aux valeurs 1, 2, 2 et 4. Les instructions qui suivent envoient des messages à cette variable. La dernière instruction évalue le contenu de cette variable. Le résultat de cette évaluation correspond à celui de toute la séquence.

Lorsque plusieurs messages sont envoyés au même objet récepteur, il est possible d’utiliser une abréviation : l’envoi en *cascade*. Dans ce cas, il n’est pas nécessaire de répéter l’objet récepteur à chaque instruction modulo le remplacement du point par un point-virgule. L’objet retourné à la suite d’envois en cascade est le résultat de la dernière évaluation. La figure 8.11 présente le même exemple que celui de la figure 8.10 mais cette fois-ci en cascade. Mise à part la première et la dernière instruction qui respectivement définit la variable globale *X* et évalue celle-ci, les autres instructions sont présentées en cascade. La variable *X*, à laquelle sont destinés les messages, n’est utilisée qu’une seule fois.

8.2.4 Variables

Dans Smalltalk, il y a deux types de variables. Les variables temporaires et les variables globales. Les *variables temporaires*, dont leur nom doit être une chaîne de caractères alphanumériques débutant par une lettre minuscule, sont toujours déclarées entre deux barres verticales. Smalltalk élimine ces variables dès qu’elles ne sont plus utilisées.

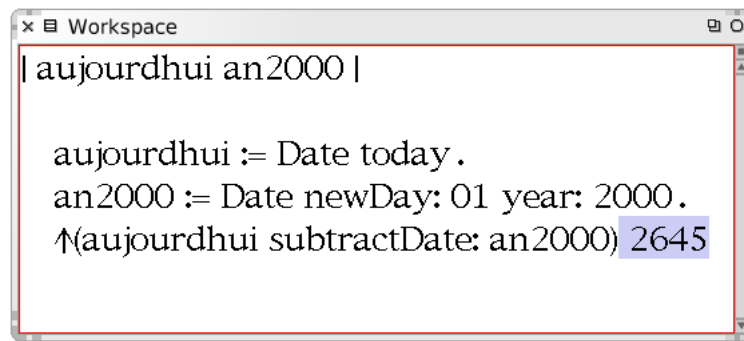


FIGURE 8.12 – Exemples d'utilisation de variables temporaires.

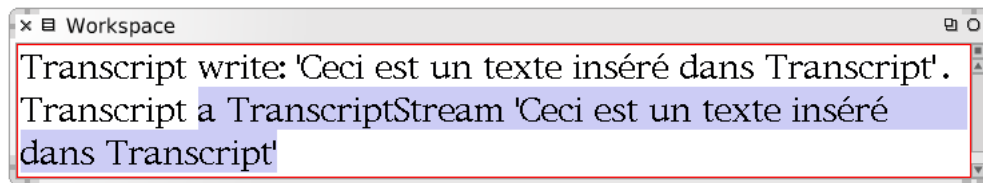


FIGURE 8.13 – Exemples d'utilisation de variables globales.

La figure 8.12 utilise deux variables temporaires pour calculer le nombre de jours qui se sont écoulés depuis l'an 2000. La première instruction déclare les deux variables temporaires `aujourd'hui` et `an2000`. Les deux instructions qui suivent définissent ces variables comme des objets instances de la classe `Date`. La variable `aujourd'hui` représente la date courante, grâce au message unaire `today`. La variable `an2000` désigne le premier jour de l'an 2000, et ce grâce au message à mot-clé `newDay: year:.` L'accent circonflexe (^) représenté par une flèche vers le haut en Squeak, utilisé dans la dernière instruction, est un message unaire indiquant la valeur à retourner comme résultat de l'exécution de la séquence. Cette instruction utilise le sélecteur `subtractDate:` qui permet de retourner le nombre de jours entre deux dates.

Contrairement aux variables temporaires, les *variables globales*, dont leur nom doit être une chaîne de caractères alphanumériques débutant par une lettre majuscule, ne seront pas automatiquement éliminées. Dans cette catégorie de variables, nous pouvons citer certains objets globaux comme `Transcript`, `Processor`, `Smalltalk`, etc. La figure 8.13 illustre l'évaluation de la variable globale `Transcript`. Dans cet exemple, la valeur retournée est le contenu stocké dans le flux.

Notons que la variable globale `Smalltalk` est un dictionnaire qui renferme tous les objets globaux définis dans le système. Par conséquent, lorsque nous choisissons de rendre une variable globale, celle-ci est rajoutée au dictionnaire `Smalltalk`.

8.2.5 Structures de contrôle

À l'instar des autres langages, Smalltalk proposent plusieurs structures de contrôle nécessaires pour la programmation des expressions conditionnelles, itératives, etc. Cependant, à la différence des autres langages, Smalltalk définit ces structures de contrôle à l'aide de messages et de sélecteurs, permettant au programmeur d'avoir une vision uniforme de la syntaxe du langage.

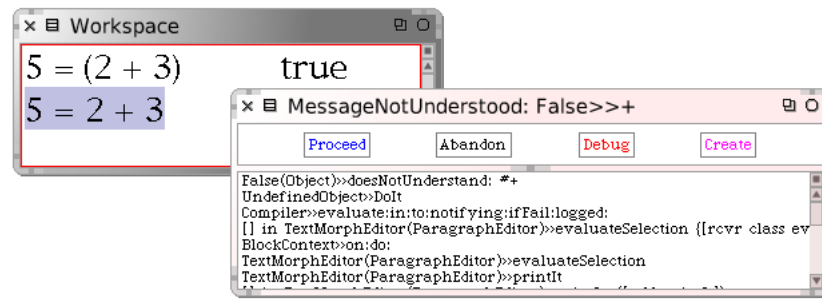


FIGURE 8.14 – Exemples de comparaisons d'objets.

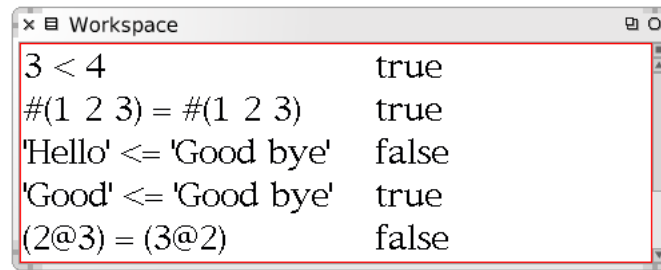


FIGURE 8.15 – Autres exemples de comparaisons d'objets.

Comparaison d'objets

La comparaison d'objets en Smalltalk s'effectue au moyen d'envois de messages. Les opérateurs relationnels ($=$, $<$, $<=$, etc.) sont implémentés comme des messages arithmétiques. L'utilisation des parenthèses est parfois utile pour imposer un ordre d'évaluation d'une expression. La figure 8.14 illustre deux exemples d'expressions comparant des objets. L'évaluation de la première expression rend l'objet Booléen **true**. En effet, l'objet récepteur 5 reçoit le sélecteur $=$ avec comme argument l'objet entier 5, résultat du message $(2 + 3)$. Par contre, l'évaluation de la deuxième expression est problématique. En effet, le résultat du message $5 = 2$ est l'objet **false**. Ce dernier ne dispose pas de méthode $+$. Par conséquent, il ne comprend pas le message qui lui est envoyé. Par ailleurs, notons que tous les objets peuvent répondre au message $=$. Ce message teste si la structure des deux objets (récepteur et argument) est identique ou non. Ce message peut être surchargé (redéfini) pour expliciter la notion d'égalité entre deux instances.

La figure 8.15 présente plusieurs exemples de comparaisons entre différents objets. La deuxième expression illustre l'égalité entre deux structures tableaux. Leurs éléments de même indice étant égaux, ces deux tableaux le sont aussi. Le dernier exemple illustre l'égalité entre deux points. En effet, le symbole $@$ permet de créer des objets instances de la classe **Point**. Par ailleurs, notons que Smalltalk définit un autre symbole d'égalité : le $==$. Ce message teste si le receveur et le paramètre correspondent au même objet. Ce message ne doit jamais être surchargé.

Test de l'état d'un objet

Plusieurs objets sont dotés de méthodes qui permettent d'avoir quelques informations sur leur état. Ces méthodes ont un identifiant qui commence, en général, par le mot **is**. La figure 8.16 illustre plusieurs méthodes de test de l'état de plusieurs objets. La première expression teste si le caractère **a** est en majuscule. La deuxième teste si le premier caractère de la chaîne 'Hello' est une voyelle. La troisième teste si la valeur entière 7 est impaire, alors que la dernière expression

teste si l'objet tableau récepteur est vide.

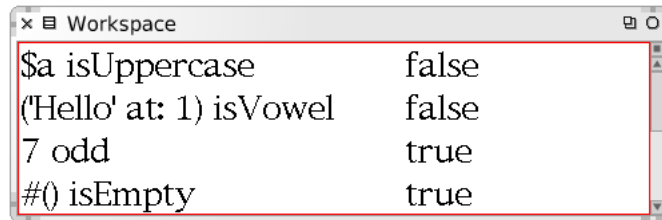


FIGURE 8.16 – Exemples de test de l'état d'objets.

Blocs

Les blocs sont des objets Smalltalk qui représentent du code compilé mais non encore évalué. Un bloc est compris entre deux crochets ([et]). Un bloc possède la même forme qu'une méthode. Il dispose d'une liste de paramètres ayant des identificateurs débutant par deux-points. Il dispose aussi de déclarations de variables temporaires. Le résultat de l'évaluation du bloc s'obtient en lui envoyant le message `unaire value` ou en lui envoyant le message à mots-clés `value:` répété plusieurs fois suivant le nombre d'arguments. Il est aussi possible de lui envoyer le message à mots-clés `valueWithArguments:` suivi d'un tableau de valeurs correspondant aux différents paramètres. La figure 8.17 propose différents exemples de blocs. La première expression retourne la classe d'un objet bloc. La deuxième affecte à une variable globale *X* un bloc qui affiche dans la fenêtre *Transcript* la chaîne de caractères 'apparition différée'. La troisième expression donne le résultat de l'évaluation de ce bloc. En effet, la chaîne de caractères est affichée dans la fenêtre appropriée. Le dernier exemple illustre l'utilisation de paramètres dans un bloc.

À partir de ces exemples, nous pouvons faire deux constatations :

- Le deuxième exemple s'apparente à une définition d'une fonction *X* qui ne prend pas d'arguments.
- La troisième expression s'apparente, quant à elle, à une application de cette fonction à un argument vide.

En fait, il est possible d'utiliser la notion de bloc pour définir des fonctions et les appliquer à des arguments. La figure 8.18 illustre ces faits. Le premier exemple définit une variable globale *Pair* à laquelle il lie un bloc dont le « rôle » est de tester si une valeur entière passée en argument est paire. La variable *Pair* s'apparente donc à une fonction. Les deux expressions qui suivent « appliquent »

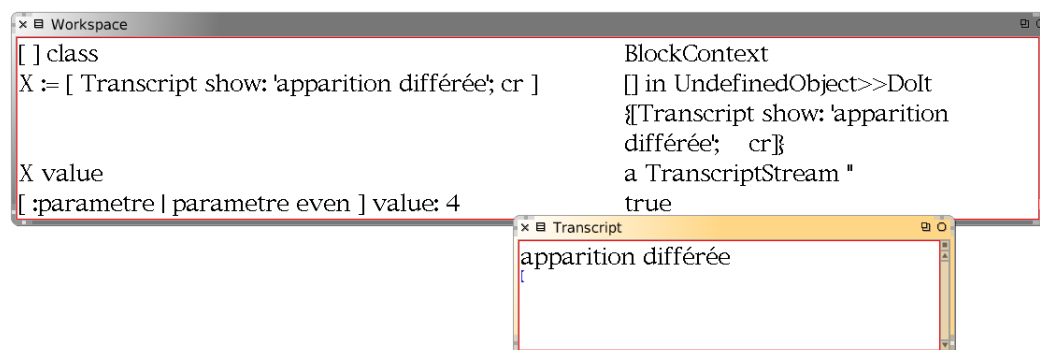


FIGURE 8.17 – Exemples de blocs.

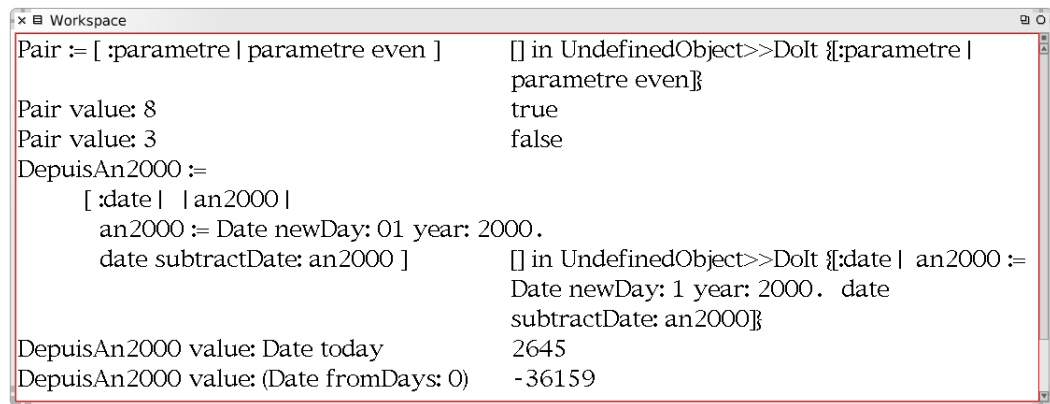


FIGURE 8.18 – Utilisation des blocs comme abstractions fonctionnelles.

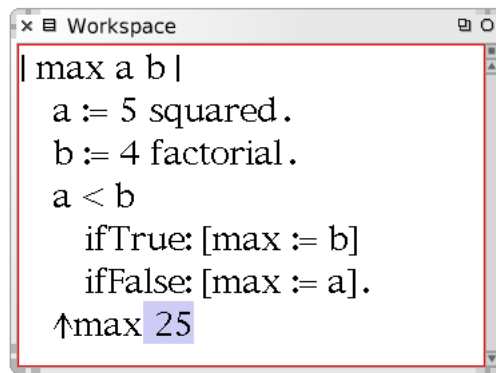


FIGURE 8.19 – Utilisation des blocs dans des conditionnelles.

cette fonction à deux arguments. La quatrième expression définit une « fonction » qui prend comme argument un bloc et qui retourne le nombre de jours qui se sont écoulés depuis l’an 2000 jusqu’à la date en argument. Cet exemple illustre aussi les déclarations de variables temporaires à l’intérieur d’un bloc.

Instructions conditionnelles

La plupart des langages de programmation utilisent l’instruction conditionnelle `if ...then ...else` comme un moyen de définir un flot de contrôle conditionnel. **Smalltalk** utilise les blocs et certains messages pour réaliser le même effet que cette instruction. La figure 8.19 illustre l’utilisation d’instructions conditionnelles en **Smalltalk**. Examinons cet exemple de plus près. Le message de comparaison `a < b` retourne une valeur Booléenne, `false` ou `true`, qui devient le récepteur du message à deux mots-clés `ifTrue: [max := b] ifFalse: [max := a]`. Ce message envoyé à un objet Booléen `true` retourne l’évaluation du bloc en argument de `ifTrue`: alors qu’envoyé à un objet `false`, elle retourne l’évaluation du bloc en argument de `ifFalse`. Puisque le message de comparaison `a < b` retourne la valeur `false`, par conséquent l’envoi du message à cette valeur résulte en la liaison de la variable temporaire `max` avec la variable temporaire `a`. Cette valeur est retournée comme résultat de l’évaluation de l’expression au complet.

Les autres messages qui exécutent les blocs de code d’une manière conditionnelle sont `ifFalse:`, `ifTrue:`, `ifTrue:` et `ifFalse:`. Pour connaître leur définition, il suffit d’accéder, via le navigateur

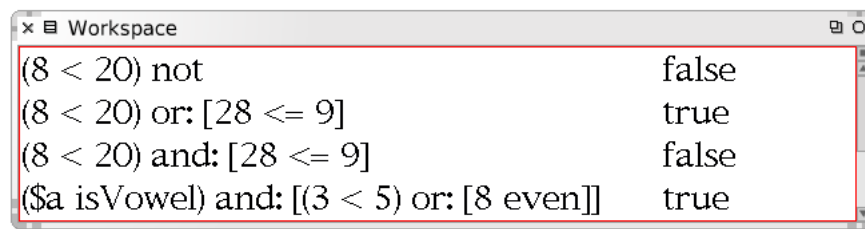


FIGURE 8.20 – Expressions Booléennes.

système (*system browser*), aux définitions des classes `Boolean`, `False` et `True`.

Expressions Booléennes

Il est souvent utile de construire des tests complexes. Pour ce faire, `Smalltalk` offre les trois messages `not`, `and:` et `or:`. La figure 8.20 illustre l'utilisation de ces messages. À l'instar des instructions Booléennes, ces messages sont envoyés à des valeurs `true` ou `false` résultats de tests de comparaison. Par exemple, la première expression s'évalue comme suit : Le message `(8 < 20)` est évalué retournant l'objet Booléen `true`. Cet objet est le récepteur d'un message `not` dont l'évaluation retourne un Booléen représentant la négation de l'objet Booléen récepteur. Par conséquent, le résultat de l'évaluation de toute l'expression est l'objet Booléen `false`. Le deuxième exemple s'évalue comme suit : Le message `(8 < 20)` est évalué retournant l'objet Booléen `true`. Cet objet est le récepteur d'un message `or:` avec comme argument le bloc `[28 <= 9]` dont l'évaluation retourne un Booléen. Puisque cette évaluation vaut `false`, alors le message `or:` est envoyé avec comme argument l'objet Booléen `false`. Par conséquent le résultat de l'évaluation de toute l'expression est l'objet Booléen `true`. Le quatrième exemple s'évalue de la même manière à la différence qu'il comporte un bloc de plus, imbriqué dans le premier. Par ailleurs, notons que les messages `and:` et `or:` n'évaluent pas toujours leur second argument. Le premier évalue un bloc de code si le récepteur est `true` tandis que le second évalue un bloc de code seulement si le récepteur est `false`.

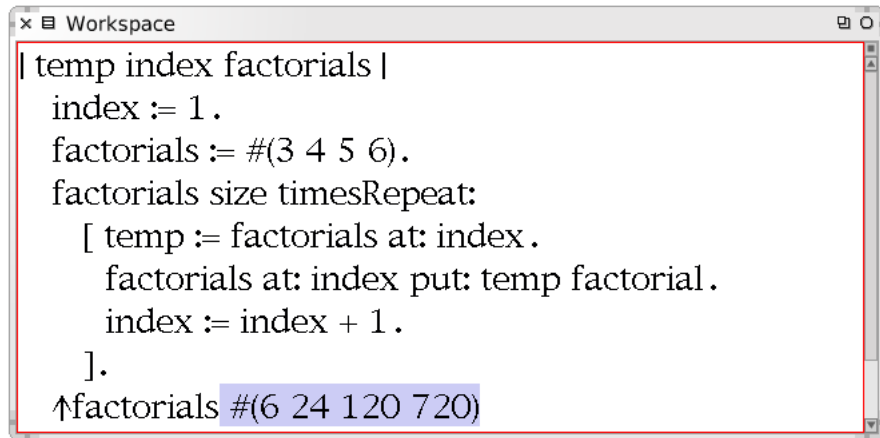
Les boucles

`Smalltalk` propose plusieurs messages pour écrire des boucles simples.

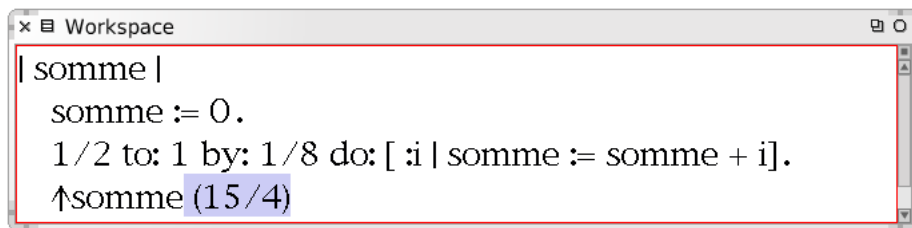
Le message `timesRepeat:` Ce message répète n fois le bloc en argument. La valeur n correspond à l'objet entier récepteur. La figure 8.21 illustre l'utilisation de ce message. Cet exemple transforme une liste d'entiers donnée en une liste d'entiers dont les éléments correspondent aux factorielles des éléments du premier tableau.

Le message `to: by: do:` `Smalltalk` offre un certain nombre de messages pour écrire facilement des instructions d'itérations. Examinons l'exemple décrit par la figure 8.22. Ce message prend trois arguments. Il prend le récepteur comme une limite inférieure et utilise le premier argument comme limite supérieure. Le deuxième argument est le pas d'itération. Le troisième est le bloc de code à évaluer à chaque étape. L'itération affecte les valeurs de $1/2$ jusqu'à 1 avec un pas de $1/8$ à l'argument du bloc c'est-à-dire i . L'argument d'un bloc est déclaré dans la première partie du bloc précédé par deux points et séparé du reste du bloc par une barre verticale (`()`).

Les messages `whileFalse:` et `whileTrue:` Le message `whileFalse:` est envoyé avec un bloc de code comme argument à un autre bloc de code récepteur. Le message évalue son bloc argument tant que le bloc récepteur est évalué à `false`. Lorsqu'il s'évalue à `true`, le message renvoie la dernière expression du bloc argument évaluée. La figure 8.23 illustre l'utilisation de ce message. Le

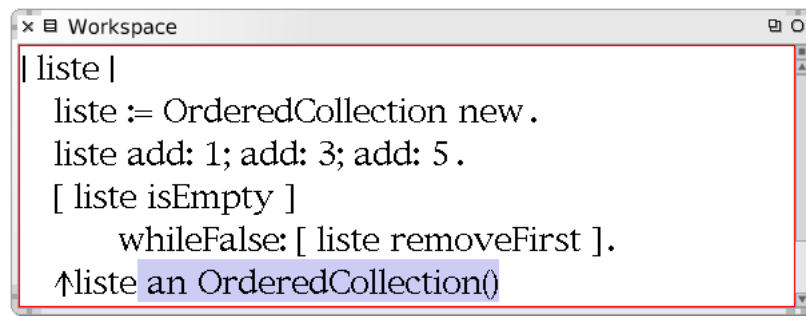
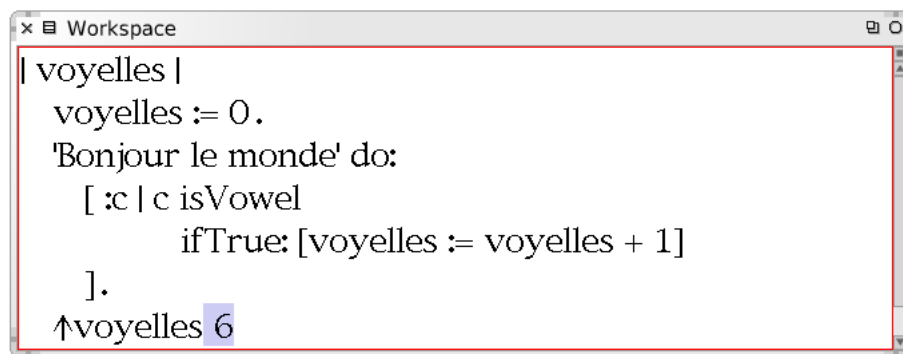


```
Workspace
| temp index factorials |
index := 1.
factorials := #(3 4 5 6).
factorials size timesRepeat:
    [ temp := factorials at: index.
      factorials at: index put: temp factorial.
      index := index + 1.
    ].
↑factorials #(6 24 120 720)
```

FIGURE 8.21 – Exemple d'utilisation du message `timesRepeat:`.

```
Workspace
| somme |
somme := 0.
1/2 to: 1 by: 1/8 do: [ :i | somme := somme + i ].
↑somme (15/4)
```

FIGURE 8.22 – Exemple d'utilisation du message `to: by: do:`.

FIGURE 8.23 – Exemple d'utilisation du message `whileFalse:`.FIGURE 8.24 – Exemple d'utilisation de l'itérateur `do:`.

message `whileTrue:` est identique au message `whileFalse:` excepté que le message évalue son bloc argument tant que le bloc récepteur est évalué à `true`.

L'itérateur `do:` C'est l'itérateur le plus simple. Dans l'exemple de la figure 8.24, il oblige la chaîne de caractères d'itérer sur elle-même et de passer chacun de ses caractères comme argument au bloc de code. De la même manière, cet itérateur peut parcourir un tableau, ou plus généralement une collection.

L'itérateur `select:` Cet itérateur est encore plus puissant que le précédent. Il itère sur son récepteur et retourne tous les éléments pour lesquels l'argument bloc de code s'évalue en `true`. Dans l'exemple de la figure 8.25, le résultat est la chaîne de voyelles du récepteur `'Bonjour le monde'`. Dans cet exemple, le message `size` permet de calculer le nombre d'éléments retenus, c'est-à-dire le nombre de voyelles que contient la chaîne.

L'itérateur `reject:` Le message `reject:` fonctionne de la même manière que le précédent, à la différence qu'il répond avec les éléments pour lesquels l'argument bloc de code s'évalue en `false`. La figure 8.26 illustre l'utilisation de ce message.

L'itérateur `collect:` Le message `collect:` évalue le bloc de code pour chaque élément de son récepteur et répond par une collection de tous les résultats retournés par le bloc.

Les différences entre les trois messages (`select:`, `reject:` et `collect:`) sont illustrées dans la figure 8.27.

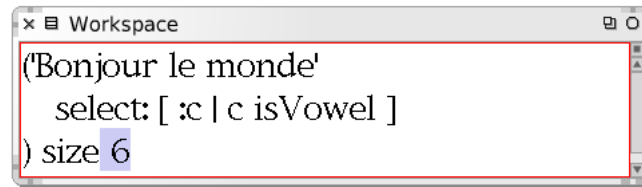
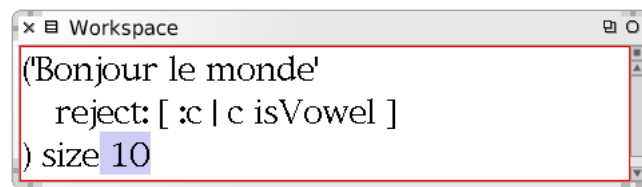
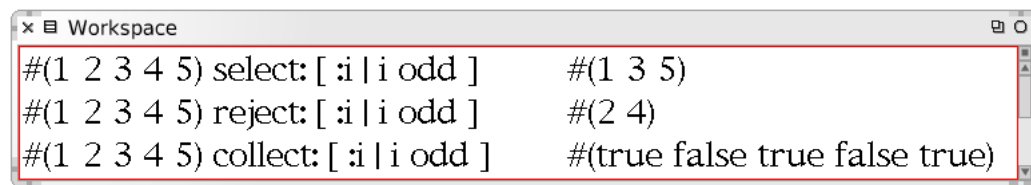
FIGURE 8.25 – Exemple d'utilisation de l'itérateur `select:`.FIGURE 8.26 – Exemple d'utilisation de l'itérateur `reject:`.

FIGURE 8.27 – Comparaison entre les différents itérateurs.

8.2.6 Classes

Dans cette section, nous présentons les différentes étapes nécessaires pour la définition d'une nouvelle classe. À cette fin, il est nécessaire de choisir une classe à partir de laquelle notre classe va être définie. En effet, d'après le quatrième principe énoncé au début de ce chapitre, toute classe *Smalltalk* est sous-classe d'une autre classe. La seule exception à cette règle est la classe *Object*⁴. Si aucune classe existante n'admet un comportement proche de celui de notre classe, alors c'est à partir de la classe *Object* qu'il faut la définir.

Les différentes étapes de définition d'une nouvelle classe peuvent être résumées comme suit :

- Choix d'une catégorie de classes. Les catégories sont des ensembles de classes. La sélection d'une catégorie liste les classes de la catégorie et indique dans la fenêtre de texte un modèle de message de création d'une classe de cette catégorie. C'est ce modèle que nous éditons pour créer la définition d'une classe.
- Déclaration du nom de la classe ainsi que celui de sa super-classe.
- Déclarations des variables d'instances et des variables de classes.
- Choix d'un « *poolDictionary* », c'est-à-dire d'un ensemble de dictionnaires partagés par plusieurs classes et accessibles simplement par les instances.
- Définition des différents protocoles regroupant chacun une partie des méthodes ou des messages qu'offrent la classe. Deux types de méthodes sont disponibles :
 - Les méthodes d'instances. Elles sont destinées aux instances d'une classe.
 - Les méthodes de classes. Ce sont des messages qui sont envoyés aux classes elles-mêmes. Elles servent à la création d'instances, à l'initialisation des variables de classes, etc.
- Implantation de ces méthodes.

Toute cette procédure se fait à l'aide du navigateur système (notons que le navigateur de la hiérarchie des classes peut être utile pour déterminer la super-classe de la classe que nous désirons créer.). Le navigateur système sert à la fois à la consultation des classes et à la création de nouvelles classes. Cette création se fait par insertion de la classe nouvellement définie dans une catégorie de classes existante. Par conséquent, le système *Smalltalk* est composé principalement d'une base initiale de classes qui se développe au fur et à mesure que de nouvelles classes sont ajoutées, ce qui augmente graduellement les possibilités du langage.

L'exemple choisi est assez simple. Il s'agit de définir une nouvelle classe *MonitoredArray* qui a comme particularité de garder en mémoire le nombre de fois que le tableau est utilisé et le nombre d'accès pour chaque cellule du tableau. Une analogie pourrait être faite avec l'accès à un site Web. En effet, dans ce dernier, il existe des outils permettant de compter le nombre de visites du site, ainsi que le nombre de visites pour chaque page.

La première étape est de définir les variables d'instances et les variables de classes nécessaires pour cette classe. La figure 8.28 présente la définition de cette classe. La première ligne précise que cette classe est définie à partir de la classe *Array*. Les deux lignes qui suivent définissent respectivement les variables d'instances et les variables de classes. Pour cette classe, une variable d'instance est nécessaire : La variable *atCounts* permet de garder une trace du nombre d'accès aux différents éléments du tableau. De même, une variable de classe, *MonitoredArrayCount*, est nécessaire pour calculer le nombre d'instances créées à partir de la classe *MonitoredArray*. Les valeurs attribuées pour la catégorie et le « *poolDictionaries* » sont prises par défaut.

Une fois, la classe déclarée, il faut définir les méthodes qu'elle implante. À cette fin, il faut tout d'abord définir des protocoles ou interfaces afin de regrouper logiquement les différentes méthodes. La figure 8.29 définit la méthode de classe *new:* permettant de créer une instance de la classe à définir. Cette méthode appartient à l'interface *instance creation*. Le rôle de cette méthode est de créer une instance de tableau, de l'initialiser et de retourner ce tableau comme résultat. Pour créer ce tableau, cette méthode fait appel à la méthode *new:* définie dans une classe mère (une super-classe). Dans ce cas, il s'agit de la méthode *new:* de la classe *Array*. La variable de liaison *super* désigne la classe mère de la classe courante. Dans ce cas, elle désigne la classe *Array*.

4. Rappelons qu'avec Squeak, c'est plutôt la classe *ProtoObject*.

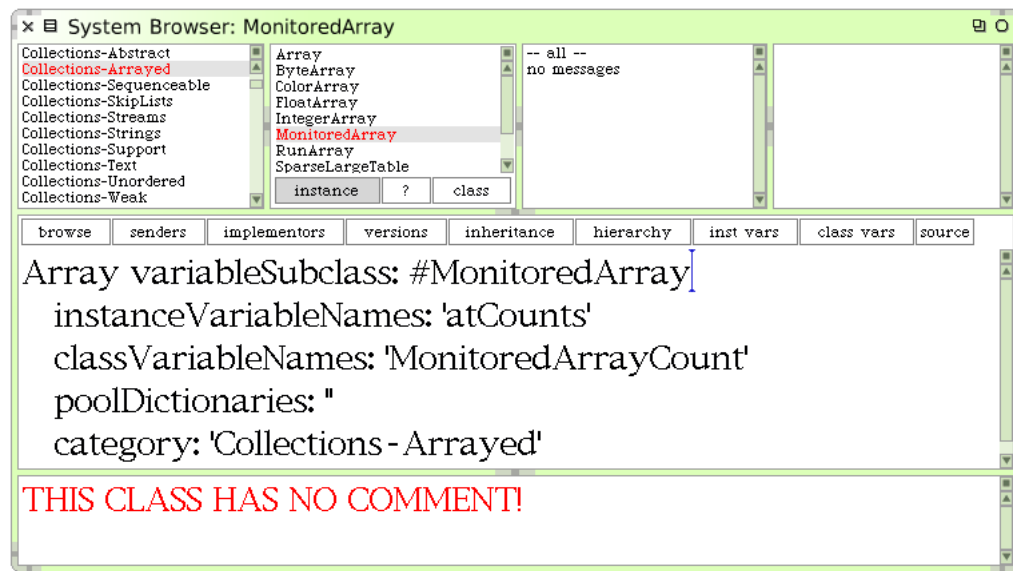


FIGURE 8.28 – Déclaration de la nouvelle classe

Une fois le tableau créé, il faut l'initialiser grâce à l'utilisation de la méthode `initialize` définie ultérieurement.

La méthode `initialize` est une méthode d'instance définie dans l'interface `initialization`. La figure 8.30 présente la définition de cette méthode. Elle consiste à initialiser l'instance définie par la méthode de classe `new`. Cette initialisation concerne particulièrement la variable d'instance `atCounts`. En effet, cette méthode attribue à cette variable un tableau de même dimension que celui de l'instance en cours. Ce tableau est initialisé à des valeurs égales à 0. La variable `self` désigne l'objet courant, c'est-à-dire celui à qui est destiné le message.

La deuxième méthode d'instance, `at:`, est présentée dans la figure 8.31. Cette méthode permet, à partir d'une valeur passée en paramètre, d'accéder à un élément particulier du tableau. Cette méthode étant définie dans la classe parente, elle est utilisée pour retourner la valeur désirée. Néanmoins, cette méthode doit aussi incrémenter la valeur d'un élément du tableau `atCounts`, permettant de garder une information précise du nombre d'accès à cet élément.

La figure 8.32 permet de définir la dernière méthode d'instance, `accessCounts`, permettant d'accéder à la valeur de la variable d'instance `atCounts`.

La première version de la classe `MonitoredArray` est ainsi définie. Cette version ne garde en mémoire que le nombre d'accès aux éléments du tableau de chaque instance de la classe. Il reste à compléter la définition de la classe pour garder en mémoire le nombre d'instances créées.

La figure 8.33 complète la définition de la méthode de classe `new`: précédemment définie. En plus du traitement présenté dans la figure 8.29, cette méthode met à jour la variable de classe `MonitoredArrayCount`. S'il s'agit de la première instanciation, cette variable est initialisée à la valeur 1, sinon elle est incrémentée à chaque instanciation future.

Aussi, il faut disposer d'une méthode permettant d'accéder à la valeur de la variable `MonitoredArrayCount`. Cette méthode de classe est définie dans la figure 8.34. Son traitement consiste à retourner la valeur de la variable `MonitoredArrayCount`.

L'exemple présenté dans la figure 8.35 permet de présenter une utilisation de la version finale de la classe `MonitoredArray`. Le premier exemple qui y figure définit un objet instance de la classe `MonitoredArray` et disposant de vingt éléments. Puis deux itérations permettent de faire plusieurs accès aux différentes positions du tableau. Ces accès mettent à jour la variable d'instance `atCounts`. La valeur de cette variable est retournée dans la dernière expression de l'exemple. Ce

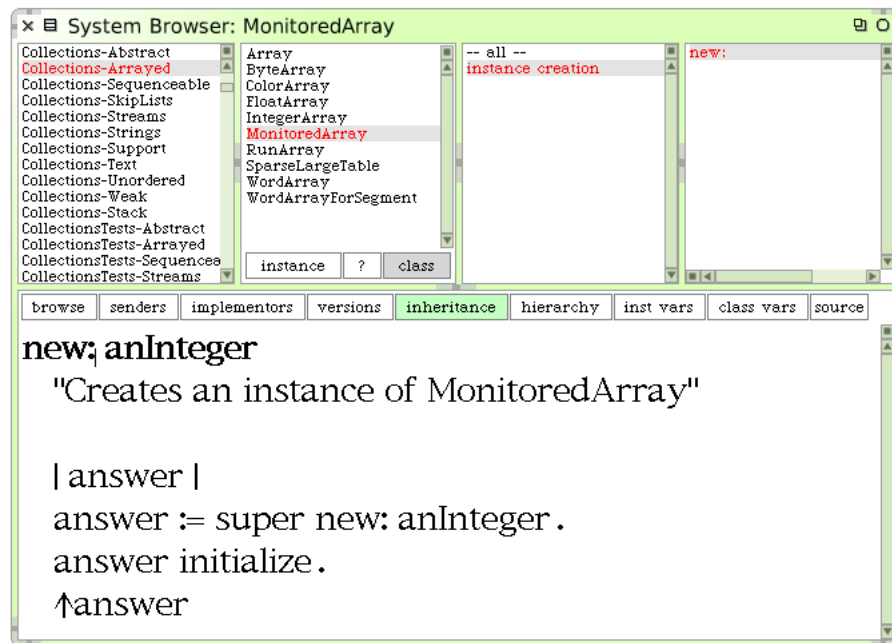


FIGURE 8.29 – Définition d'une méthode de classe.

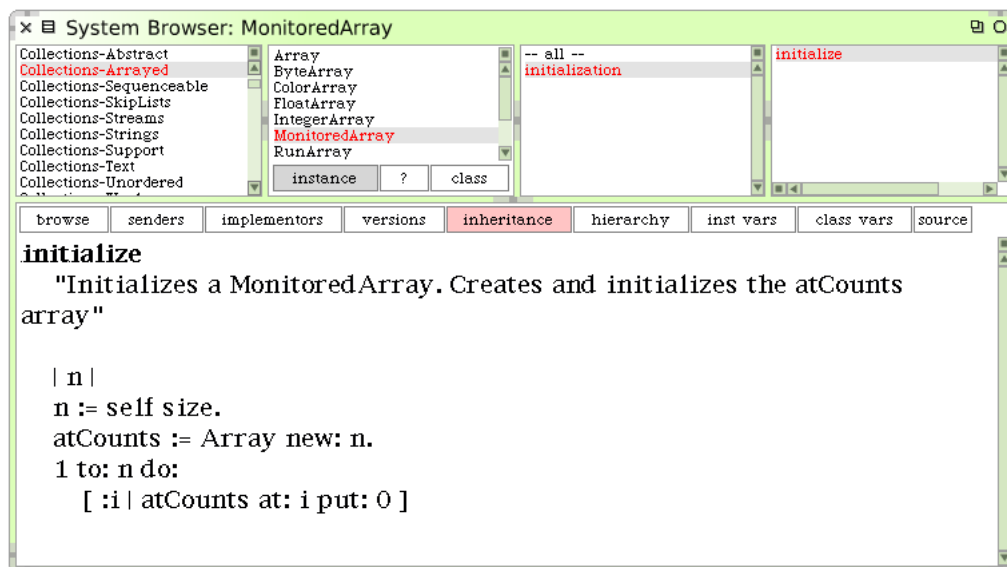
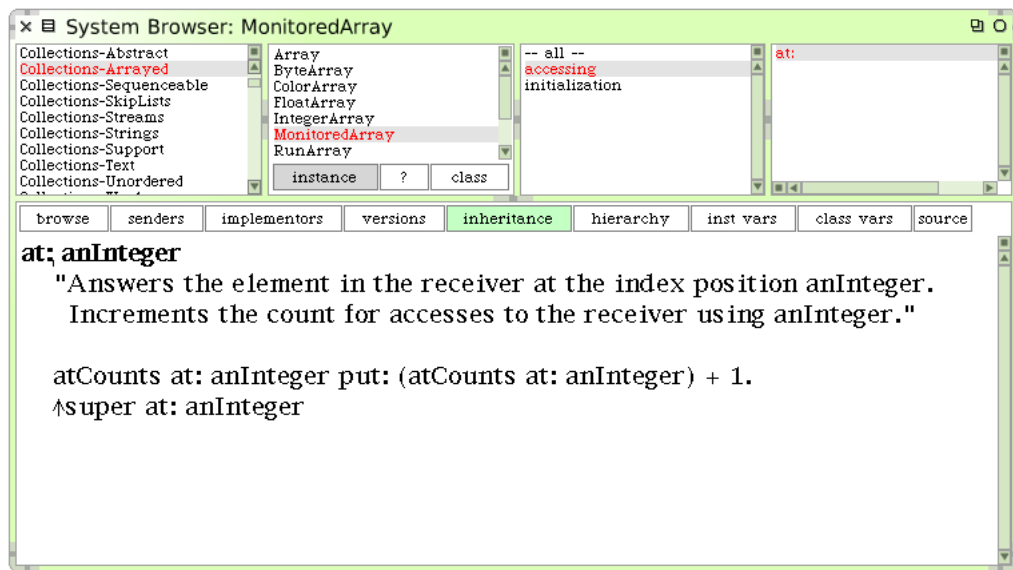
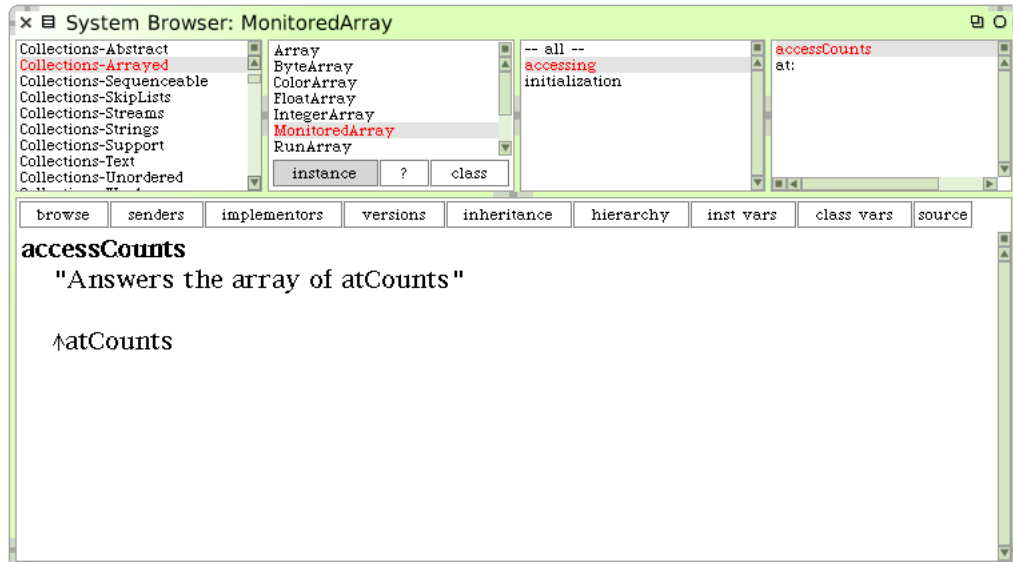


FIGURE 8.30 – Définition de la méthode initialize.

FIGURE 8.31 – Définition de la méthode `at:`.FIGURE 8.32 – Définition de la méthode `accessCounts`.

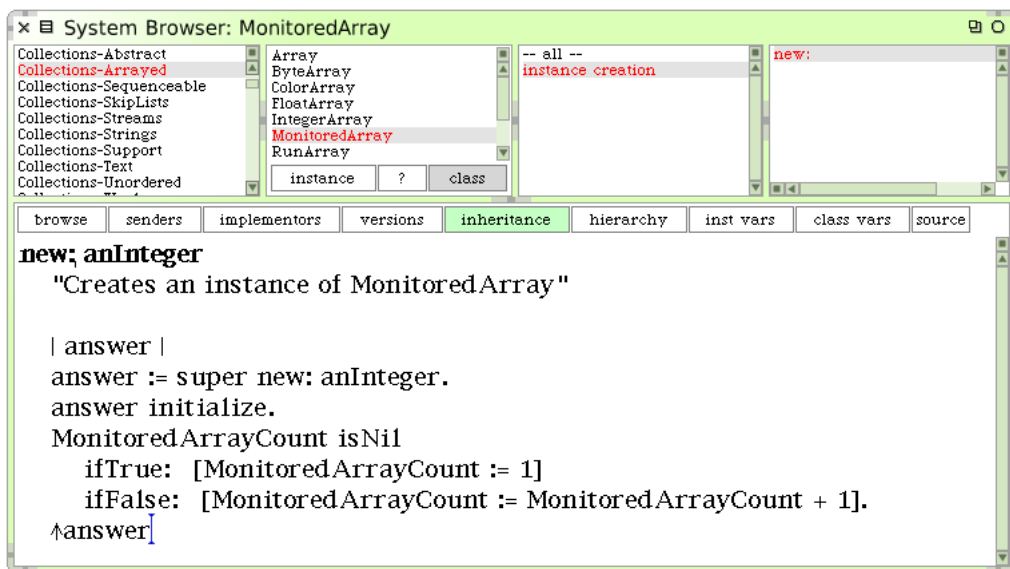


FIGURE 8.33 – Modification de la méthode de classe new:.

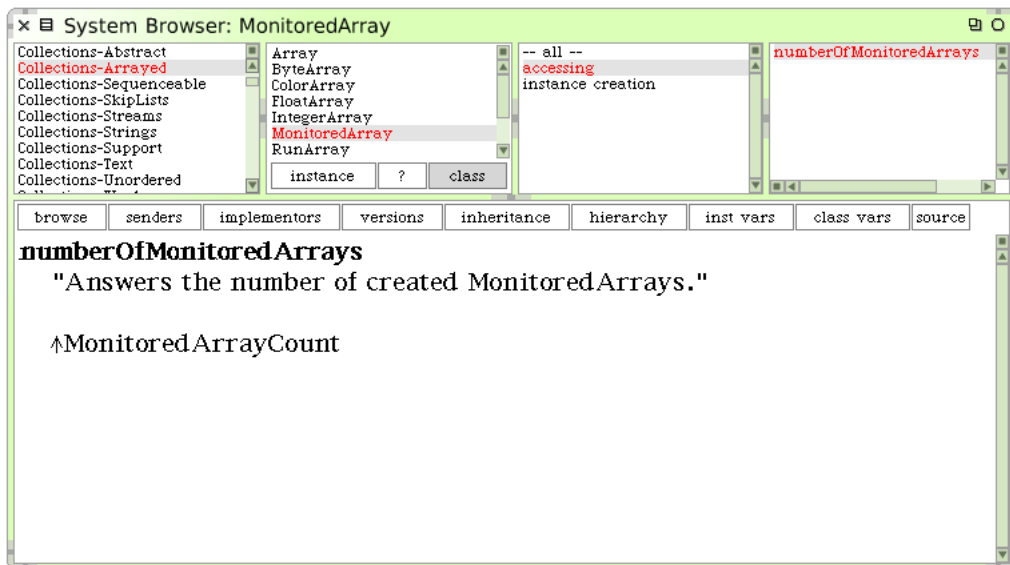


FIGURE 8.34 – Définition de la méthode de classe numberOfMonitoredArrays.

```

x Workspace
" Exemples d'utilisation de la classe MonitoredArray "
"Exemple 1"

| array |
array := MonitoredArray new: 20.
1 to: 10 do:
    [ :m | 1 to: 10 do:
        [ :n | array at: m+n ]
    ].
↑array accessCounts          "Resultat:" #(0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1)

"Exemple 2"
[
| array1 array2 |
array1 := MonitoredArray new: 20.
1 to: 10 do:
    [ :m | 1 to: 10 do:
        [ :n | array1 at: m+n ]
    ].

array2 := MonitoredArray new: 20.
1 to: 10 do:
    [ :m | 1 to: 10 do:
        [ :n | array2 at: m+n ]
    ].

↑MonitoredArray numberOfMonitoredArrays          "Resultat:" 2

```

FIGURE 8.35 – Exemple d'utilisation de la classe `MonitoredArray`.

tableau spécifie que le premier élément n'a jamais été consulté. Par contre, le troisième élément a été consulté deux fois, etc. Le deuxième exemple est simplement pour illustrer le comptage des instances de la classe `MonitoredArray` créées. Nous créons deux instances temporaires de cette classe et effectuons à ces deux tableaux le même traitement que l'exemple précédent. Ensuite, nous vérifions le nombre d'instances de cette classe créées. Remarquons que le résultat dans cet exemple est 2 car nous avons effectué les deux exemples séparément (pas durant la même session de travail). Si nous avions effectué ces deux exemples consécutivement, nous aurions obtenu le résultat 3.

8.3 Collection

Smalltalk est surtout caractérisé par la quantité volumineuse de classes prédéfinies. L'apprentissage de ce langage passe surtout par la connaissance et la maîtrise de l'interface de ces classes. Dans cette section, nous présentons certaines classes utiles et usuelles afin de vous donner la base nécessaire pour consulter vous-même d'autres classes qui pourraient vous intéresser.

La classe *Collection* permet tout simplement de décrire n'importe quelle collection d'objets que ce soit un tableau d'objets, un ensemble d'objets, un dictionnaire associant des objets à d'autres objets, etc. Il s'agit d'une classe abstraite. Par conséquent, on ne peut l'instancier. Elle propose une interface commune à ces différentes sous-classes permettant :

- d'itérer sur une collection d'objets,

- de rechercher un objet dans une collection,
- d'ajouter ou de supprimer des objets dans une collection,
- d'accéder ou de modifier des éléments d'une collection,
- et bien plus encore.

Les différentes sous-classes de la classe *Collection* sont caractérisées la plupart du temps par la manière d'organiser les objets (ordonnés ou non, etc.), d'y accéder (par un index, par une clé, etc.), d'en ajouter (taille de la collection fixe ou variable), d'en supprimer, etc.

Cette classe admet trois sous-classes principales :

1. *Bag* : Classe caractérisée par le fait que les objets qu'elle comprend sont stockés d'une manière non ordonnée, qu'on ne peut accéder à ces objets par un index, qu'elle accepte des occurrences multiples d'un même objet, et qu'elle peut avoir une taille variable.
2. *SequenceableCollection* : Classe abstraite caractérisée par le fait que les objets qu'elle comprend peuvent être accédés par un index. Elle admet plusieurs sous-classes extrêmement utiles que nous allons voir plus loin.
3. *Set* : Classe admettant les mêmes caractéristiques que la classe *Bag* à la différence qu'elle n'admet qu'une occurrence de chaque objet.

Par ailleurs, pour chaque classe (non abstraite) de la classe *Collection*, il existe deux manières de définir des instances :

1. **new**
Ce message permet de définir une instance de la classe en question.

```
Set new.
SortedCollection new.
OrderedCollection new
```
2. **with:**, **with:with:**, **with:with:with:** et **with:with:with:with:**
Ces messages permettent de créer des instances initialisées respectivement avec un, deux, trois ou quatre objets.

```
Array with: 1.
Set with: 1 with: 2.
SortedCollection with: 4 with: 3 with: 1.
OrderedCollection with: 2 with: 3 with: 1 with: 4
```

Remarquons aussi que toutes sous-classes de *Collection* ayant une taille fixe peut être créée par le message :

- **new: n**
Ce message permet de définir une instance de taille *n*.

```
Array new: 10
```

Certaines méthodes sont héritées de la classe *Object*, comme par exemple :

- **asString**
Permet d'avoir une version chaîne de caractères d'une collection donnée.
- **at: anInteger**
Retourne l'objet qui se trouve à la position *anInteger*.
- **at: anInteger put: anObject**
Met à jour l'élément à la position *anInteger* par l'objet *anObject*.

Voici quelques messages pouvant, lorsque c'est possible⁵, être adressés à des instances de sous-classes de la classe *Collection* :

- **add: anObject**
Ajoute une occurrence de l'objet *anObject*.

5. Par exemple, le message **add:** ne peut être adressé à un tableau puisque la classe *Array* représente des collections de taille fixe. Par contre, ce message peut être adressé à un ensemble pour en ajouter un élément.

— `addAll: aCollection`

Ajoute tous les éléments se trouvant dans *aCollection*.

```
| s |
s := Set with: 1 with: 2.
s addAll: #( 1 3 2 3 2 4 ).
^s
```

— `asArray`, `asBag`, `asOrderedCollection`, `asSet` et `asSortedCollection`

Tous ces messages permettent de transformer une forme de collection en une autre. Par exemple, pour passer d'un tableau d'objets à un ensemble d'objets, rien de plus simple :

```
#( 1 5 3 5 3 3 ) asSet
```

En utilisant adéquatement ces messages, nous pouvons obtenir certains traitements intéressants. Par exemple, pour ne conserver qu'une occurrence de chaque élément d'un tableau *t* (tout en considérant l'ordre non important), nous pouvons faire ce qui suit :

```
| t |
t := #( 1 5 3 5 3 3 ).
t := t asSet asArray.
^t
```

— `asSortedCollection: aBloc`

Permet de passer d'une collection d'objets à une collection triée suivant une « fonction » de comparaison spécifiée par le bloc *aBloc*. Par exemple, pour avoir une version triée en ordre décroissant d'un tableau :

```
#( 1 3 5 2 8 6 ) asSortedCollection: [ :a :b | a > b ]
```

Une manière de trier un tableau *t* pourrait se faire comme suit :

```
| t |
t := #( 1 3 5 2 8 6 ).
t := ( t asSortedCollection: [ :a :b | a < b ] ) asArray.
^t
```

— `includes: anObject`

Teste si *anObject* est membre de la collection.

— `inject: initialValue into: binaryBloc`

Est équivalent à « **fold** *f v l* » avec *f* représentée par *binaryBloc*, *v* représentée par *initialValue* et *l* représentée par la collection qui reçoit le message.

Par exemple, pour faire la somme des éléments d'un tableau :

```
#( 1 3 5 2 8 6 ) inject: 0 into: [ :sousTotal :nouveau | sousTotal +
nouveau ]
```

Pour comprendre l'exemple qui suit, consultez la définition de la méthode `occurrencesOf` :

```
#( 1 3 2 5 2 8 6 2 ) inject: 0 into: [ :sousTotal :nouveau | sousTotal
+ ( (nouveau = 2) ifTrue: [1] ifFalse: [0] ) ]
```

— `isCollection`

Retourne `true` pour signaler qu'il s'agit bien d'une collection.

— `isEmpty`

Teste si la collection est vide.

— `occurrencesOf: anObject`

Retourne le nombre d'occurrences de *anObject*.

— `remove: anObject`

Supprime une occurrence de l'objet *anObject*. Il existe une version de ce message avec le mot-clé `ifAbsent: aBloc`. Dans ce cas, le bloc *aBlock* est évalué si *anObject* n'existe pas dans la collection.

- **removeAll:** *aCollection*
Supprime tous les éléments de la collection se trouvant dans *aCollection*.
- **size**
Retourne la taille de la collection réceptrice du message.
- (**select:**, **reject:**, **collect:**, **do:**) *aBloc*
Voir notes de cours :)
- **detect:** *aBloc*
Applique le bloc *aBloc* à chaque élément de la collection et retourne le premier objet pour lequel cette application retourne **true**. Il existe une version de ce message avec le mot-clé **ifNone:** *exceptionBlock*. Dans ce cas, le bloc *exceptionBlock* est évalué s'il n'existe pas d'élément qui satisfait *aBloc*.

```
#( $s $m $a $l $l ) detect: [ :e | e isVowel ].  
'PasdeSéparateurs' detect: [ :e | e isSeparator ] ifNone: [ 'Aucun  
séparateurs!' ]
```

Remarques Certaines définitions de méthodes de la classe *Collection* utilisent des messages particuliers :

- **add:**
Dans ce cas, le message **subclassResponsability** défini dans la classe *Object* est utilisé et il peut donc être adressé à n'importe quel objet, pour signaler que la méthode **add:** doit être définie dans une sous-classe. Comme nous pouvons le constater, cette méthode est définie dans les sous-classes *Bag*, *OrderedCollection*, *SortedCollection*, *Set* et *Dictionary*. En effet, pour prendre un exemple, le traitement pour ajouter un élément dans un ensemble (teste si l'élément existe déjà pour éviter des occurrences multiples) est différent de celui pour l'ajouter dans une collection triée (ajouter l'élément à une position précise pour maintenir la collection triée). Pour s'en convaincre, il suffit de consulter les définitions de cette méthode dans ces deux classes. Cette méthode est aussi définie dans la classe *ArrayedCollection* pour signaler, dans ce cas, que le message ne peut être envoyé à une collection de taille fixe. Ceci est réalisé en utilisant le message **shouldNotImplement** défini aussi dans la classe *Object*. Par ailleurs, les méthodes **do:** et **remove:ifAbsent:** doivent aussi être définies dans une des sous-classes.
- **with:**
Dans ce cas, un autre message défini dans la classe *Object* est utilisé : **yourself**. Lorsqu'il est envoyé à un objet quelconque, ce message retourne l'objet lui-même. Pour s'en rendre compte, il suffit de consulter sa définition dans la classe *Object*. Ce message est particulièrement utile lors d'envoi de messages en cascade. Prenons l'exemple suivant :

Set with: 1

Voici les différentes étapes qui seront réalisées par l'expression en cascade **^(self new) add: 1; yourself** correspondant à la définition de la méthode **with:** avec l'argument 1 :

1. Une instance de la classe *Set* est créée, soit **s**.
2. **s** reçoit le message **add: 1**. Cet envoi résulte que l'objet **s**, un *Set*, contient désormais la valeur 1.
3. **s** reçoit le message **yourself**, ce qui aura pour conséquence de retourner **s**, soit un *Set* avec la valeur 1. Ceci correspond bien au résultat voulu.

Il aurait été possible de définir cette méthode sans l'usage du message **yourself**.

```
with: anObject  
"Answer an instance of me containing anObject."  
| s |
```

```
s := self new add: anObject.
^s
```

— `collect: aBloc`

Dans ce cas, un autre message défini dans la classe *Object* est utilisé : `species`. Lorsqu'il est envoyé à un objet quelconque, ce message retourne la classe de cet objet, ou une classe similaire. Pour s'en rendre compte, il suffit de consulter sa définition dans la classe *Object*. En fait, le message `collect:`, lorsqu'il est envoyé à un objet instance d'une classe donnée, retourne une nouvelle instance de cette même classe contenant l'application du bloc *aBloc* aux différents éléments de la collection réceptrice. Par exemple, si nous envoyons ce message à un tableau, il nous retourne un tableau. À un ensemble, il nous retourne un ensemble, etc. Exemple :

```
'hello' collect: [ :e | e asUppercase ]
#( 1 2 3 4 5 ) collect: [ :e | e + 1 ]
( Set with: 1 with: 2 with: 3 ) collect: [ :e | e asString ]
```

Par conséquent, la première chose à faire dans la méthode `collect:` est de créer un objet instance de la même classe que celle du récepteur du message.

```
newCollection := self species new.
```

Par ailleurs, dans certain cas, la méthode `species` est redéfinie. Par exemple, dans la classe *Interval*, cette méthode est redéfinie et spécifie que la classe à retourner est *Array*. Ceci s'explique par le fait que si, par exemple, nous appliquons un traitement particulier aux éléments d'un intervalle, le résultat de cette application peut ne pas correspondre à un autre intervalle. Par conséquent, il est approprié, dans ce cas, de retourner comme résultat un tableau d'éléments. Pour s'en convaincre, il suffit de consulter, par exemple, la définition de la méthode `collect:`, dans la classe *SequenceableCollection*, dans laquelle un objet de classe similaire à celui récepteur du message `collect:` est défini.

```
self species new: self size
```

Par exemple, en envoyant ce message à un intervalle, nous obtenons un tableau en résultat.

```
( 1 to: 10 ) collect: [ :e | e * e ]
```

8.3.1 Bag

Cette classe n'est pas abstraite et peut donc être instanciée. Elle permet de stocker des objets d'une manière non ordonnée, en acceptant des occurrences multiples d'un même objet, et peut avoir une taille variable. Vu que les éléments sont ajoutés d'une manière non ordonnée, nous ne pouvons pas accéder à un élément précis avec le message `at:` ou ajouter un élément à une position précise avec le message `at:put:`. Aussi, cette classe comprend une variable d'instance nommée *contents* dont le rôle est de stocker les objets du *Bag*. Puisque ce dernier accepte des occurrences multiples d'un même objet, il serait judicieux de représenter les éléments d'un *Bag* comme un dictionnaire associant pour chaque objet son nombre d'occurrences. Pour s'en convaincre, il suffit de consulter la définition de la méthode de classe `contentsClass` dans laquelle nous pouvons constater que la variable d'instance *contents* est effectivement représentée comme un dictionnaire. À partir de là, nous pouvons consulter les différentes méthodes de la classe *Bag* et constater qu'elles comprennent des messages, comme `removeKey:`, provenant de la classe *Dictionary*.

En plus des méthodes héritées ou redéfinies, la classe *Bag* propose les méthodes suivantes :

— `add: anObject withOccurrences: anInteger`

Ajoute *anObject* « *anInteger* » fois dans le *Bag*.

— `valuesAndCounts`

Retourne le contenu de la variable d'instance *contents*. Par conséquent, cela retourne un dictionnaire. À partir de là, nous pouvons utiliser toutes les méthodes de la classe *Dictionary*.

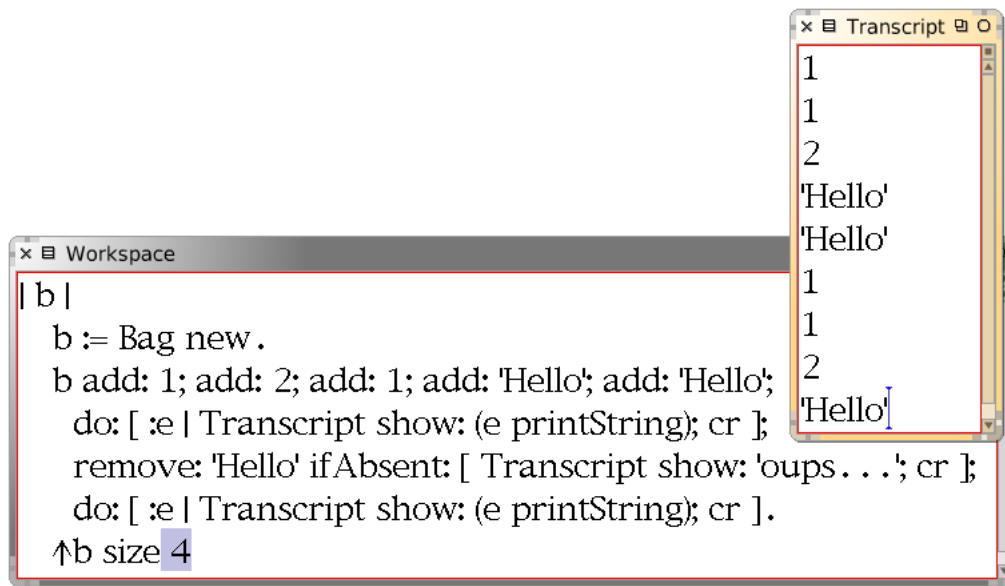


FIGURE 8.36 – Exemples d'utilisation de la classe Bag.

— size

Retourne le nombre total d'objets dans le Bag en tenant compte des occurrences multiples.

Il suffit de consulter la définition de cette méthode pour s'en rendre compte.

Cette classe est illustrée par la figure 8.36.

Méthodes héritées mais non valides :

at:, at:put:.

8.3.2 SequenceableCollection

Il s'agit d'une classe abstraite dans le seul but est de factoriser le comportement de ses sous-classes. Étant abstraite, nous ne pouvons pas l'instancier. Elle permet de préciser que les collections qu'elle décrit sont indexées, c'est-à-dire que nous pouvons accéder à ses éléments par un index. Parmi les sous-classes que nous traiterons figurent *ArrayedCollection*, *Interval* et *OrderedCollection*.

En plus des méthodes définies par ses super-classes, la classe *SequenceableCollection* propose les méthodes suivantes :

— , aCollection

Permet de concaténer l'objet récepteur avec *aCollection*. Ce dernier doit être une collection indexée (instance d'une sous-classe de *SequenceableCollection*).

```

#( 1 3 5 2 8 6 ), #( 3 1 6 )
#( 1 4 6 ), 'hello'
'Hello', ' World'
'Hello ', #( $w $o $r $l $d )
#( 1 2 3 4 5 ), ( 6 to: 10 )
( 1 to: 5 ), ( 6 to: 10 )

```

— after: anObject

Retourne l'objet qui suit l'objet *anObject*.

```
( OrderedCollection with: 4 with: 5 with: 6 ) after: 5; yourself
```

- Il existe une version de ce message avec le mot-clé `ifAbsent: aBloc` permettant d'évaluer ce bloc dans le cas où `anObject` n'existe pas.
- `atAll: aCollection put: anObject`
Met l'objet `anObject` à toutes les positions précisées par la collection `aCollection`.

```

#( 1 4 2 6 3 7 4 ) atAll: #( 2 4 6 ) put: 0

```
 - `atAllPut: anObject`
Met l'objet `anObject` à toutes les positions de la collection réceptrice du message.

```

#( 1 4 2 6 3 7 4 ) atAllPut: 0

```
 - `before: anObject`
Retourne l'objet qui précède l'objet `anObject`.
Il existe une version de ce message avec le mot-clé `ifAbsent: aBloc` permettant d'évaluer ce bloc dans le cas où `anObject` n'existe pas.
 - `copyFrom: start to: stop`
Retourne une nouvelle collection contenant les éléments du receveur qui se trouve entre les index `start` et `stop`.
 - `findFirst: aBloc`
Applique le bloc `aBloc` à chaque élément de la collection réceptrice du message et retourne l'index du premier élément pour lequel cette application vaut `true`.

```

#( 1 2 3 $a 4 $b 5 ) findFirst: [ :e | e isCharacter ]

```
 - `findLast: aBloc`
Applique le bloc `aBloc` à chaque élément de la collection réceptrice du message et retourne l'index du dernier élément pour lequel cette application vaut `true`.
 - `first`
Retourne le premier élément de la collection.
 - `last`
Retourne le dernier élément de la collection.
 - `indexOf: anObject`
Retourne l'index de l'objet `anObject`. Retourne 0 si l'objet n'existe pas.
 - `replaceFrom: start to: stop with: aCollection`
Remplace les éléments du receveur qui se trouve entre les index `start` et `stop` par ceux de la collection `aCollection`. Le nombre d'éléments remplacés doit être égal à la taille de la collection en argument.

```

#( 1 3 4 6 5 ) replaceFrom: 2 to: 4 with: #( 2 3 4 )

```
 - `reversed`
Retourne une version de la collection dont les éléments sont inversés.
 - `with: aCollection do: aBloc`
Retourne le résultat de l'application du bloc `aBloc` aux différentes paires de valeurs, dont le premier élément est pris de la collection réceptrice du message et le deuxième élément est pris de `aCollection`. Ces deux collections doivent avoir la même taille.

```

| d |
d := Dictionary new.
#( 1 2 3 4 5 ) with: #( $a $b $c $d $e ) do: [ :a :b | d at: a put: b ].
^d

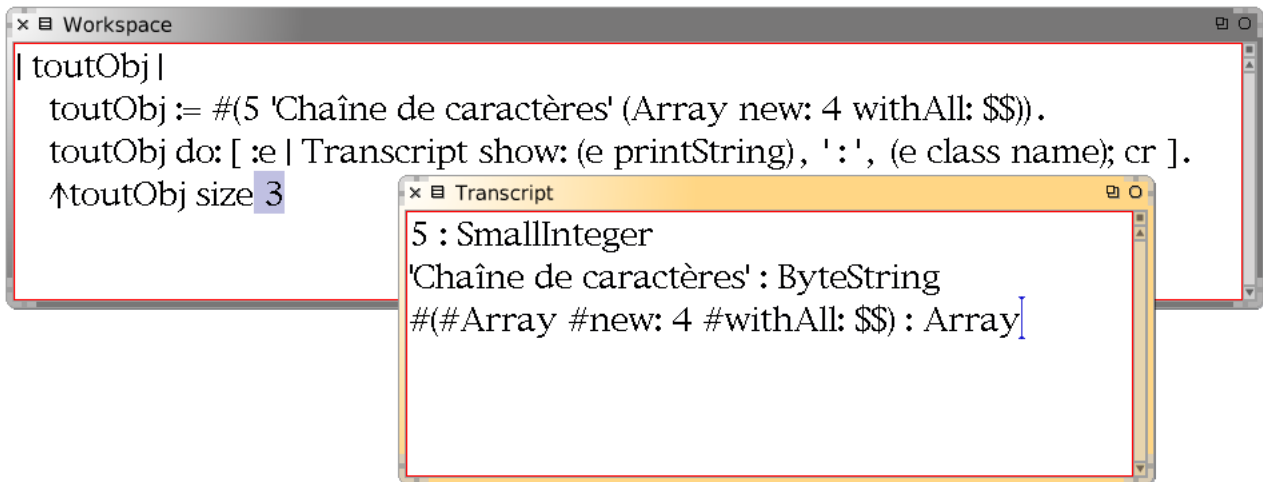
```

ArrayedCollection

Il s'agit d'une classe abstraite dans le seul but est de factoriser le comportement de ses sous-classes. Étant abstraite, nous ne pouvons pas l'instancier. Elle permet de préciser que les collections qu'elle décrit sont de taille fixe. Parmi les sous-classes que nous traiterons figurent `Array` et `String`.

Méthodes héritées mais non valides :

```
add:, remove:, remove:ifAbsent:, removeAll:.
```

FIGURE 8.37 – Exemples d'utilisation de la classe `Array`.

Array La classe `Array`, étant sous-classe des classes `ArrayedCollection`, `SequenceableCollection` et `Collection`, permet de représenter des collections d'objets qui peuvent être accédés par un index. Ces collections ont une taille fixe. La taille est précisée par les méthodes de classes `new:`, `with:`, `with:with:`, `with:with:with:` et `with:with:with:with:`.

En plus des méthodes héritées, la classe `Array` propose la méthode suivante :

— `isArray`

Retourne `true` pour indiquer que l'objet récepteur du message est bien un tableau.

La figure 8.37 illustre l'utilisation de cette classe.

String La classe `String`, étant sous-classe des classes `ArrayedCollection`, `SequenceableCollection` et `Collection`, permet de représenter des collections de caractères qui peuvent être accédés par un index. Ces collections ont une taille fixe. Un objet instance de `String` est spécifié par une suite de caractères entourée du symbole `'`.

`'Bonjour le monde'`

En plus des méthodes qui sont héritées par ses super-classes, la classe `String` propose les méthodes suivantes :

— `asByteArray`

Retourne un tableau contenant les codes ASCII des différents caractères du `String` récepteur.

`'Bonjour le monde' asByteArray`

— `asDate`, `asInteger`, `asLowercase`, `asNumber` et `asUppercase`

Retourne une version particulière du `String` récepteur.

`'12/18/2000' asDate.`

`'12' asInteger.`

`'Bonjour le monde' asLowercase.`

`'3.5' asNumber.`

`'Bonjour le monde' asUppercase`

— `isString`

Retourne `true` pour indiquer que l'objet récepteur du message est bien une chaîne de caractères.

- **size**
Retourne la taille du **String** récepteur.
- **subStrings**
Retourne un tableau contenant comme éléments les différents mots du **String** récepteur du message.
 'Cette chaîne de caractères contient 7 mots' subStrings
- **subStrings: aStringOrACollectionOfCharacters**
Retourne un tableau contenant comme éléments les différents mots, séparés par les caractères séparateurs *aStringOrACollectionOfCharacters*, du **String** récepteur du message
 'Cette-chaîne-de-caractères-contient-7-mots' subStrings: '-'
 'Cette-chaîne-de-caractères-contient-7-mots' subStrings: #(\$-)

Interval

La classe **Interval** permet de représenter des intervalles d'entiers. Elle offre deux méthodes de classes permettant de définir des intervalles :

- **from: beginningInteger to: endInteger**
Retourne un intervalle dont les bornes sont *beginningInteger* et *endInteger*.
 Interval from: 1 to: 10
- **from: beginningInteger to: endInteger by: incrementInteger**
Retourne un intervalle dont les bornes sont *beginningInteger* et *endInteger* et dont les éléments sont séparés par un « saut » égal à *incrementInteger*.
 Interval from: 1 to: 10 by: 1/2

En plus des méthodes héritées par ses super-classes, cette classe propose les méthodes suivantes :

- **+ amount**
Retourne un nouvel intervalle dont les bornes sont incrémentées de *amount*.
 | i |
 i := Interval from: 1 to: 10.
 i + 5
 - **- amount**
Retourne un nouvel intervalle dont les bornes sont décrémentées de *amount*.
 - **at: anInteger**
Retourne l'élément de l'intervalle qui se trouve à la position *anInteger*.
 | i |
 i := Interval from: 1 to: 10.
 i at: 3
 - **size**
Retourne la taille de l'intervalle.
- Méthodes héritées mais non valides :

at:put:.

Signalons que la classe **Number** propose deux méthodes d'instances permettant de créer des intervalles.

- **to: aNumber**
Retourne un nouvel intervalle dont les bornes sont délimitées par le nombre récepteur du message et *aNumber*.
 1 to: 10
- **to: sNumber by: iNumber**
Retourne un intervalle dont les bornes sont délimitées par le nombre récepteur du message et *sNumber* et dont les éléments sont séparés par un « saut » égal à *iNumber*.
 1 to: 10 by: 1/2

OrderedCollection

Étant sous-classe de la classe `SequenceableCollection`, les éléments de la classe `OrderedCollection` peuvent être accédés par un index. Aussi, elles ont une taille variable et ses éléments sont ordonnés par leur séquence d'entrée. C'est la classe qui ressemble le plus à une liste générique.

En plus des méthodes héritées par ses super-classes, cette classe propose les méthodes suivantes :

- `add: newObject after: oldObject`
Ajoute l'objet *newObject* à la suite de l'objet *oldObject*.

```
( OrderedCollection with: 1 with: 2 with: 4 ) add: 3 after: 2; yourself
```
- `add: anObject afterIndex: anInteger`
Ajoute l'objet *anObject* à la suite de l'objet qui se trouve à l'index *anInteger* + 1.

```
( OrderedCollection with: 1 with: 2 with: 3 ) add: 0 afterIndex: 0;  
yourself
```
- `add: newObject before: oldObject`
Ajoute l'objet *newObject* avant l'objet *oldObject*.
- `add: anObject beforeIndex: anInteger`
Ajoute l'objet *anObject* avant l'objet qui se trouve à l'index *anInteger* + 1.
- `addAllFirst: aSequenceableCollection`
Ajoute la collection *aSequenceableCollection* au début de la collection réceptrice du message.

```
( OrderedCollection with: 4 with: 5 with: 6 ) addAllFirst: #( 1 2 3 );  
yourself
```
- `addAllLast: aSequenceableCollection`
Ajoute la collection *aSequenceableCollection* à la fin de la collection réceptrice du message.
- `addFirst: anObject`
Ajoute l'objet *anObject* au début de la collection réceptrice du message.
- `addLast: anObject`
Ajoute l'objet *anObject* à la fin de la collection réceptrice du message.
- `removeAt: anInteger`
Supprime l'élément à l'index *anInteger* de la collection réceptrice du message.

```
( OrderedCollection with: 1 with: 2 with: 3 ) removeAt: 3; yourself
```
- `removeFirst`
Supprime le premier élément de la collection réceptrice du message.

```
( OrderedCollection with: 1 with: 2 with: 3 ) removeFirst; yourself
```
- `removeLast`
Supprime le dernier élément de la collection réceptrice du message.

SortedCollection Cette classe est caractérisée par le fait que ses éléments sont triés. Par conséquent, toutes les méthodes, héritées de ses super-classes et permettant d'ajouter un objet à un index précis, ne sont plus valides.

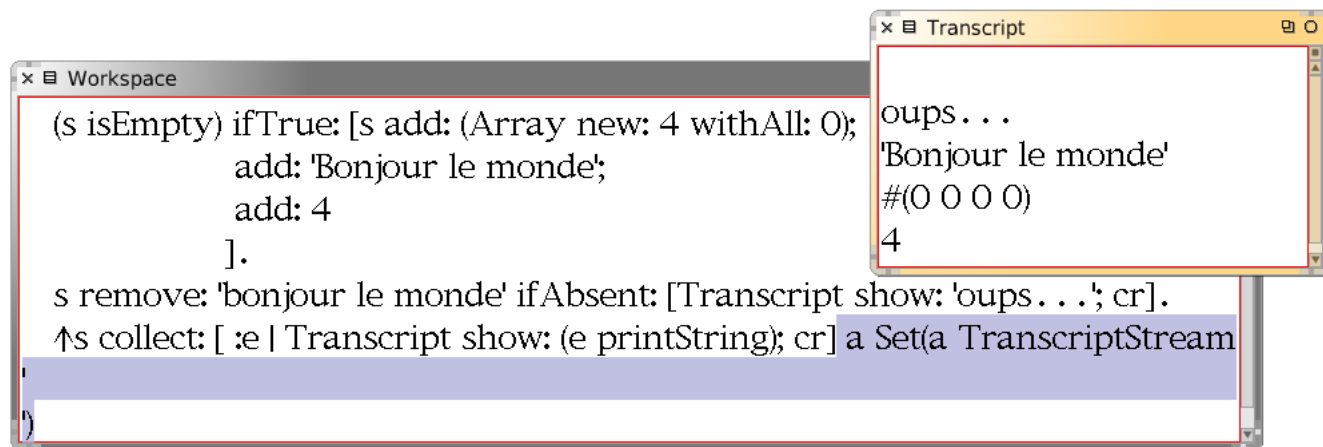
En plus des méthodes héritées par ses super-classes, cette classe propose la méthode suivante :

- `sortBlock: aBloc`
Permet de trier à nouveau la collection en fonction du bloc *aBloc*.

```
( SortedCollection with: 2 with: 1 with: 3 ) sortBlock: [ :a :b | a > b  
]
```

Méthodes héritées mais non valides :

`add:after:`, `add:before:`, `addAllFirst:`, `addAllLast:`, `addFirst:`, `addLast:`, `at:put:`.

FIGURE 8.38 – Exemples d'utilisation de la classe `Set`.

8.3.3 Set

Cette classe partage les mêmes caractéristiques que la classe `Bag` à la différence qu'elle n'admet qu'une occurrence d'un objet donné.

En plus des méthodes héritées par sa super-classe, cette classe propose la méthode suivante :

— `size`

Retourne le nombre d'éléments de l'ensemble récepteur du message.

À l'instar des tableaux, un ensemble n'est pas typé et peut contenir toutes sortes d'objets. La figure 8.38 présente des expressions manipulant des ensembles.

Méthodes héritées mais non valides :

`at:`, `at: put:`.

Dictionary

Un dictionnaire est défini comme un ensemble d'associations. Il hérite donc naturellement de la classe `Set`. Une association, instance de la classe `Association`, associe une clé à une valeur : il s'agit donc d'une paire (clé, valeur) ou (clé \mapsto valeur). La classe `Association` propose deux méthodes particulières, `key` et `value`, permettant respectivement d'accéder à la clé et à la valeur d'une association. Aussi, cette classe offre une méthode de classe, `key:value:`, permettant de créer une association.

Étant particulière, la classe `Dictionary` redéfinit plusieurs méthodes héritées qui ont, dans certains cas, une signification particulière relative aux dictionnaires.

— `add: anAssociation`

Ajoute l'association *anAssociation* au dictionnaire récepteur du message.

```
( Dictionary new ) add: ( Association key: 1 value: $a )
```

— `addAll: aCollection`

Ajoute une collection d'associations au dictionnaire récepteur du message.

```
| d1 d2 |
d1 := Dictionary new.
d2 := Dictionary new.
d1 at: 1 put: $a; at: 2 put: $b; at: 3 put: $c.
d2 at: 4 put: $d; at: 2 put: $e.
d1 addAll: d2
```

- ```

| d a |
d := Dictionary new.
a := Array with: (Association key: 4 value: $d) with: (Association
key: 2 value: $e).
d at: 1 put: $a; at: 2 put: $b; at: 3 put: $c.
d addAll: a

```
- **associationsDo: aBloc**  
Applique le bloc *aBloc* à chaque association du dictionnaire récepteur du message.

```

| d keys |
d := Dictionary new.
keys := Set new.
d at: 1 put: $a; at: 2 put: $b; at: 3 put: $c.
d associationsDo: [:a | keys add: (a key)]

```
  - **associationsSelect: aBloc**  
Applique le bloc *aBloc* à chaque association du dictionnaire récepteur du message et retourne un nouveau dictionnaire contenant les associations pour lesquelles cette application vaut **true**.

```

| d1 d2 |
d1 := Dictionary new.
d1 at: 1 put: $a; at: 2 put: $b; at: 3 put: $c.
d2 := d1 associationsSelect: [:a | (a key) > 1]

```
  - **at: aKey**  
Retourne la valeur associée à *aKey* dans le dictionnaire récepteur du message. Il existe une version **at:ifAbsent:** permettant d'évaluer un bloc dans le cas où *aKey* n'existe pas.
  - **at: aKey put: anObjet**  
Met à jour la valeur associée à *aKey* si cette dernière est déjà définie dans le dictionnaire, ou, dans le cas contraire, ajoute la nouvelle association (*aKey*, *anObject*).
  - **do: aBloc**  
Applique le bloc *aBloc* à chaque valeur contenue dans le dictionnaire récepteur du message.

```

| d |
d := Dictionary new.
d at: 1 put: $a; at: 2 put: $b; at: 3 put: $c.
d do: [:v | Transcript show: v class name; cr]

```
  - **includes: anObject**  
Retourne **true** s'il existe une valeur dans le dictionnaire égale à *anObject*. Retourne **false** sinon.
  - **includesKey: aKey**  
Retourne **true** s'il existe une clé dans le dictionnaire égale à *aKey*. Retourne **false** sinon.
  - **keyAtValue: anObject**  
Retourne la clé associée à une valeur égale à *anObject*. Il existe une version **keyAtValue:ifAbsent:** permettant d'évaluer un bloc dans le cas où *anObject* n'existe pas.
  - **keys**  
Retourne un ensemble contenant les clés du dictionnaire.
  - **keysDo: aBloc**  
Applique le bloc *aBloc* à toutes les clés du dictionnaire.
  - **keysAndValuesRemove: keyValueBlock**  
Supprime toutes les associations du dictionnaire pour lesquelles l'évaluation du bloc retourne **true**.
  - **occurrencesOf: anObject**  
Retourne le nombre d'associations dans le dictionnaire qui associent une clé à la valeur

- anObject.*
  - **removeKey: aKey**  
Supprime l'association dont la clé est *aKey* du dictionnaire. Il existe une version **removeKey:ifAbsent:** permettant d'évaluer un bloc dans le cas où *aKey* n'existe pas.
  - **select: aBloc**  
Applique le bloc *aBloc* à chaque valeur du dictionnaire récepteur du message et retourne un nouveau dictionnaire contenant les associations pour lesquelles cette application vaut **true**.
  - **values**  
Retourne une collection contenant toutes les valeurs du dictionnaire.
- Méthodes héritées mais non valides :
- remove:, remove:ifAbsent:, removeAll:.**

## 8.4 Autres classes utiles

### 8.4.1 Magnitude

À l'instar de la classe **Collection**, la classe **Magnitude** est une classe abstraite qui permet de représenter les objets pouvant être comparés, comptés et/ou mesurés. Parmi ses sous-classes, nous distinguons les classes **Association**, **Character**, **Date**, **Float**, **Fraction**, **Integer** et **Time**. Représentant des objets assez basiques et communs, leurs méthodes sont triviales : il est assez facile de deviner, à partir de leur nom, ce qu'effectue chacune d'entre elles.

### 8.4.2 UIManager

Cette classe permet de gérer et de centraliser une grande partie des opérations simples d'affichages graphiques à l'utilisateur. Nous ne pouvons pas l'instancier, mais nous pouvons utiliser son instance déjà créée en utilisant le message **UIManager default**. Cette instance connaît déjà les préférences de l'utilisateur. À partir de cette instance, nous pouvons afficher une fenêtre demandant à l'utilisateur d'entrer une réponse au clavier à l'aide de la méthode **request:** ou de la méthode **request: initialAnswer:.** Voici un exemple :

```
UIManager default request: 'Année en cours ?' initialAnswer: '2007'
```

Cette instance permet aussi d'afficher un message de confirmation à l'utilisateur (du type oui/non) à l'aide de la méthode **confirm:**. Voici un exemple :

```
UIManager default confirm: 'Êtes-vous sûr de vouloir quitter ?'
```

Beaucoup d'autres méthodes d'instances utiles sont définies dans cette classe. Il est assez simple de deviner ce qu'elles effectuent.

## Exercices

**8.1** *Quelle est la différence entre les deux expressions suivantes :*

- `3 factorial factorial`
- `3 factorial; factorial`

**8.2** *Évaluer les expressions suivantes :*

- `4 * 5; + 3 factorial`
- `4 * 5 factorial; + 3`
- `4 * 5; + 3; factorial`

**8.3** *Quelle est la différence entre les expressions `true`, `'true'` et `#true` ?*

**8.4** *Ajouter une méthode `explode` à la classe `String`. Elle transforme une chaîne de caractères en une collection de chaînes de caractères. Par exemple :*

`'hello' explode`  $\implies$  `#('h' 'e' 'l' 'l' 'o')`

**8.5** *Ajouter une méthode `implode` à la classe `OrderedCollection`. Elle transforme une collection de chaînes de caractères en une chaîne de caractères. Elle a l'effet inverse de la fonction `explode`.*

**8.6** *Écrire une méthode `somme` qui retourne la somme des éléments d'une collection.*

## Corrigés

**A.1 :** La première expression vaut 720. Elle correspond à la factorielle de 6 qui est la factorielle de 3. Par contre la deuxième expression s'évalue en 6. En effet, elle correspond à l'évaluation en cascade de la factorielle de 3.

**A.2 :** Les évaluations des différentes expressions sont :

- 10
- 7
- 24

La première correspond à la somme de 4 et de la factorielle de 3, c'est-à-dire 6. La deuxième évaluation correspond à la somme de 4 et de 3. La dernière évaluation correspond à la factorielle de 4, c'est-à-dire 24.

**A.3 :** L'expression `true` correspond à un objet de la classe `True`. Cet objet représente la valeur Booléenne vraie. L'expression `'true'` correspond à une chaîne de caractères représentant le mot « true ». L'expression `#true` correspond au symbole ayant comme identificateur le mot « true ». Pour en être sûr, il est possible d'envoyer à chaque expression le message `class` qui retourne la classe de l'objet receveur du message.

**A.4 :**

```
explode
 "Transforme une chaîne en liste de chaîne"

 ^(self asOrderedCollection collect: [:c | c asString])
```

**A.5 :**

```
implode
 "Transforme une liste de chaîne en une chaîne"

 | s |
 s := ''.
 self do: [:c | s := s , c].
 ^s
```

**A.6 :**

```
somme
 "Additionne les éléments d'une collection"

 | sum |
 sum := 0.
 self do: [:n | sum := sum + n].
 ^sum
```

Autre solution possible :

```
somme
 "Additionne les éléments d'une collection"

 ^self inject: 0 into: [:a :b | a + b]
```



# Table des matières

|          |                                                                            |          |
|----------|----------------------------------------------------------------------------|----------|
| <b>I</b> | <b>Programmation fonctionnelle</b>                                         | <b>1</b> |
| <b>1</b> | <b>Fondements de la programmation fonctionnelle</b>                        | <b>3</b> |
| 1.1      | Concepts . . . . .                                                         | 4        |
| 1.2      | Historique et caractéristiques . . . . .                                   | 5        |
| <b>2</b> | <b>ML</b>                                                                  | <b>7</b> |
| 2.1      | Premiers pas avec OCaml . . . . .                                          | 8        |
| 2.2      | Données : Types de base . . . . .                                          | 10       |
| 2.2.1    | Type <i>unit</i> . . . . .                                                 | 11       |
| 2.2.2    | Booléens . . . . .                                                         | 11       |
| 2.2.3    | Entiers . . . . .                                                          | 14       |
| 2.2.4    | Réels . . . . .                                                            | 16       |
| 2.2.5    | Caractères . . . . .                                                       | 18       |
| 2.2.6    | Chaînes de caractères . . . . .                                            | 18       |
| 2.3      | Données : Types composés . . . . .                                         | 20       |
| 2.3.1    | Tuples . . . . .                                                           | 21       |
| 2.3.2    | Enregistrements . . . . .                                                  | 23       |
| 2.3.3    | Listes . . . . .                                                           | 26       |
| 2.3.4    | Types somme, produit, algébrique et inductif . . . . .                     | 29       |
| 2.3.5    | Abréviations de types . . . . .                                            | 34       |
| 2.3.6    | Mot-clé «type» . . . . .                                                   | 35       |
| 2.4      | Données : Définition d'identificateurs, liaisons et déclarations . . . . . | 35       |
| 2.5      | Comportements : Filtrage par motifs . . . . .                              | 37       |
| 2.6      | Comportements : Fonctions . . . . .                                        | 45       |
| 2.6.1    | Fonctions récursives . . . . .                                             | 51       |
| 2.6.2    | Fonctions polymorphes . . . . .                                            | 51       |
| 2.6.3    | Filtrage par motifs . . . . .                                              | 51       |
| 2.6.4    | Fonctions anonymes . . . . .                                               | 54       |
| 2.6.5    | Curryfication . . . . .                                                    | 55       |
| 2.6.6    | Typage explicite . . . . .                                                 | 55       |
| 2.6.7    | Fonctions d'ordre supérieur . . . . .                                      | 56       |
| 2.6.8    | Variants polymorphes et types ouverts . . . . .                            | 61       |
| 2.6.9    | Paramètres étiquetés et optionnels . . . . .                               | 62       |
| 2.7      | Caractéristique impératives . . . . .                                      | 63       |
| 2.7.1    | Références . . . . .                                                       | 63       |
| 2.7.2    | Structures de contrôle . . . . .                                           | 65       |
| 2.7.3    | Enregistrements à champs modifiables . . . . .                             | 66       |
| 2.7.4    | Tableaux . . . . .                                                         | 67       |
| 2.7.5    | Exceptions . . . . .                                                       | 69       |
| 2.8      | Données + Comportements = Modules . . . . .                                | 72       |

|           |                                                             |            |
|-----------|-------------------------------------------------------------|------------|
| 2.8.1     | Module . . . . .                                            | 72         |
| 2.8.2     | Signatures . . . . .                                        | 75         |
| 2.8.3     | Foncteurs . . . . .                                         | 77         |
| 2.9       | Interaction avec l'interpréteur . . . . .                   | 82         |
| 2.10      | Conclusion . . . . .                                        | 85         |
|           | Exercices . . . . .                                         | 86         |
|           | Corrigés . . . . .                                          | 88         |
| <b>3</b>  | <b>ML : Notions avancées</b>                                | <b>93</b>  |
| 3.1       | Aspects du paradigme fonctionnel . . . . .                  | 93         |
| 3.1.1     | Représentation des grammaires . . . . .                     | 95         |
| 3.1.2     | Fermetures ( <i>closures</i> ) . . . . .                    | 96         |
| 3.1.3     | Itérateurs . . . . .                                        | 97         |
| 3.1.4     | Mémoïsation . . . . .                                       | 100        |
| 3.1.5     | Récursivité terminale . . . . .                             | 105        |
| 3.1.6     | Continuations . . . . .                                     | 106        |
| 3.1.7     | Évaluations par valeur/par nom/paresseuse . . . . .         | 110        |
| 3.1.8     | Listes paresseuses ou flots ( <i>stream</i> ) . . . . .     | 113        |
| 3.2       | Programmation déclarative : <i>LINQ/PLINQ</i> . . . . .     | 117        |
| 3.2.1     | C# : un langage fonctionnel ? . . . . .                     | 121        |
| 3.2.2     | C# : un langage définitivement fonctionnel ! . . . . .      | 122        |
| <b>II</b> | <b>Programmation orientée objet</b>                         | <b>123</b> |
| <b>4</b>  | <b>Fondements de la programmation orientée objet</b>        | <b>125</b> |
| 4.1       | Objet . . . . .                                             | 125        |
| 4.2       | Encapsulation . . . . .                                     | 125        |
| 4.3       | Classe . . . . .                                            | 126        |
| 4.4       | Instanciation . . . . .                                     | 126        |
| 4.5       | Héritage . . . . .                                          | 126        |
| 4.6       | Transmission de message . . . . .                           | 126        |
| 4.7       | Exemple . . . . .                                           | 127        |
| 4.8       | Langages orientés objet . . . . .                           | 127        |
| 4.9       | Évaluation . . . . .                                        | 128        |
| 4.10      | Conclusion . . . . .                                        | 129        |
| <b>5</b>  | <b>Objective Caml</b>                                       | <b>131</b> |
| 5.1       | Objets . . . . .                                            | 131        |
| 5.1.1     | Encapsulation . . . . .                                     | 132        |
| 5.1.2     | Valeurs de première classe . . . . .                        | 132        |
| 5.1.3     | Propriétés fonctionnelles . . . . .                         | 133        |
| 5.1.4     | Références mutuelles . . . . .                              | 134        |
| 5.1.5     | Typage explicite . . . . .                                  | 134        |
| 5.1.6     | Transtypage . . . . .                                       | 135        |
| 5.1.7     | Types ouverts (..) . . . . .                                | 136        |
| 5.1.8     | Interaction avec les aspects fonctionnels . . . . .         | 137        |
| 5.1.9     | Objets immédiats : Conclusion . . . . .                     | 140        |
| 5.2       | Classes . . . . .                                           | 140        |
| 5.2.1     | Classes = Constructeurs d'objets . . . . .                  | 140        |
| 5.2.2     | Types de classes, Types d'objets et Types ouverts . . . . . | 141        |
| 5.2.3     | Classe paramétrée . . . . .                                 | 141        |
| 5.2.4     | Déclaration de variables locales . . . . .                  | 142        |



|       |                                                                 |     |
|-------|-----------------------------------------------------------------|-----|
| 5.2.5 | Référence à l'objet courant et méthodes privées . . . . .       | 143 |
| 5.2.6 | Initialisation d'objets . . . . .                               | 144 |
| 5.2.7 | Héritage . . . . .                                              | 145 |
| 5.2.8 | Objets immédiats et héritage . . . . .                          | 149 |
| 5.3   | Aspects orientés objets du langage : Notions avancées . . . . . | 150 |
| 5.3.1 | Classes et méthodes virtuelles . . . . .                        | 150 |
| 5.3.2 | Méthodes binaires . . . . .                                     | 150 |
| 5.3.3 | Méthodes retournant <code>self</code> . . . . .                 | 152 |
| 5.3.4 | Retour aux aspects fonctionnels . . . . .                       | 153 |
| 5.3.5 | Polymorphisme et sous-typage . . . . .                          | 154 |
| 5.3.6 | Classe polymorphe (générique) . . . . .                         | 155 |
| 5.3.7 | Type d'une classe . . . . .                                     | 155 |
| 5.4   | Aspects orientés objets et modules . . . . .                    | 156 |
| 5.5   | Conclusion . . . . .                                            | 157 |
|       | Exercices . . . . .                                             | 158 |
|       | Corrigés . . . . .                                              | 159 |

### III Programmation parallèle et concurrente 161

#### 6 Introduction 163

|     |                                                                                |     |
|-----|--------------------------------------------------------------------------------|-----|
| 6.1 | Description du parallélisme . . . . .                                          | 163 |
| 6.2 | Distinction entre programmation concurrente, parallèle et distribuée . . . . . | 163 |
| 6.3 | Langages de programmation concurrente et parallèle . . . . .                   | 164 |

#### 7 OCaml parallèle et concurrent 165

|       |                                                                                |     |
|-------|--------------------------------------------------------------------------------|-----|
| 7.1   | Introduction . . . . .                                                         | 165 |
| 7.2   | Concepts et caractéristiques . . . . .                                         | 166 |
| 7.3   | Langage . . . . .                                                              | 166 |
| 7.3.1 | Abréviations et fonctions utiles . . . . .                                     | 166 |
| 7.3.2 | Apport de CML . . . . .                                                        | 167 |
| 7.3.3 | Fil d'exécution . . . . .                                                      | 168 |
| 7.3.4 | Version 5.0 et plus d'OCaml . . . . .                                          | 173 |
| 7.3.5 | Verrous (module <i>Mutex</i> ) . . . . .                                       | 185 |
| 7.3.6 | Canaux de communication (module <i>Event</i> ) . . . . .                       | 188 |
| 7.3.7 | Événements . . . . .                                                           | 189 |
| 7.3.8 | Autres fonctions manipulant les événements . . . . .                           | 201 |
| 7.3.9 | Autres fonctions définies dans le module <i>Event</i> . . . . .                | 213 |
| 7.4   | Programmation modulaire et concurrente . . . . .                               | 214 |
| 7.4.1 | Abstraction des détails d'implantation . . . . .                               | 214 |
| 7.4.2 | Définition de nouveaux modules utiles . . . . .                                | 218 |
| 7.5   | Combinaison de différents paradigmes . . . . .                                 | 219 |
| 7.5.1 | <i>PFX</i> : un cadre pour la programmation parallèle en <i>.Net</i> . . . . . | 222 |
| 7.5.2 | Exploitation du module <i>Tpl</i> . . . . .                                    | 235 |
|       | Exercices . . . . .                                                            | 244 |
|       | Corrigés . . . . .                                                             | 245 |

### IV Annexes 247

#### 8 Smalltalk 249

|       |                                          |     |
|-------|------------------------------------------|-----|
| 8.1   | Environnement de développement . . . . . | 250 |
| 8.1.1 | Installation . . . . .                   | 250 |

|       |                                  |     |
|-------|----------------------------------|-----|
| 8.1.2 | Environnement . . . . .          | 250 |
| 8.1.3 | Premiers pas . . . . .           | 252 |
| 8.2   | Langage Smalltalk . . . . .      | 256 |
| 8.2.1 | Principes de base . . . . .      | 256 |
| 8.2.2 | Objets . . . . .                 | 257 |
| 8.2.3 | Messages . . . . .               | 257 |
| 8.2.4 | Variables . . . . .              | 261 |
| 8.2.5 | Structures de contrôle . . . . . | 262 |
| 8.2.6 | Classes . . . . .                | 270 |
| 8.3   | Collection . . . . .             | 275 |
| 8.3.1 | Bag . . . . .                    | 279 |
| 8.3.2 | SequenceableCollection . . . . . | 280 |
| 8.3.3 | Set . . . . .                    | 285 |
| 8.4   | Autres classes utiles . . . . .  | 287 |
| 8.4.1 | Magnitude . . . . .              | 287 |
| 8.4.2 | UIManager . . . . .              | 287 |
|       | Exercices . . . . .              | 288 |
|       | Corrigés . . . . .               | 289 |

# Liste des tableaux

|     |                          |    |
|-----|--------------------------|----|
| 2.1 | Table de vérité. . . . . | 13 |
|-----|--------------------------|----|



# Table des figures

|      |                                                                                   |     |
|------|-----------------------------------------------------------------------------------|-----|
| 8.1  | Fenêtres au démarrage de Squeak . . . . .                                         | 251 |
| 8.2  | Évaluation à l'aide de l'action <code>print it</code> . . . . .                   | 253 |
| 8.3  | Inspection d'un objet. . . . .                                                    | 254 |
| 8.4  | Utilisation d'une référence inconnue. . . . .                                     | 254 |
| 8.5  | Envoi d'un message inconnu. . . . .                                               | 255 |
| 8.6  | Exemples de messages unaires. . . . .                                             | 258 |
| 8.7  | Exemples de messages arithmétiques. . . . .                                       | 259 |
| 8.8  | Exemples de messages à mots-clés. . . . .                                         | 259 |
| 8.9  | Exemples d'envois de messages complexes. . . . .                                  | 260 |
| 8.10 | Exemples de séquences de messages. . . . .                                        | 261 |
| 8.11 | Exemples d'envois de messages en cascade. . . . .                                 | 261 |
| 8.12 | Exemples d'utilisation de variables temporaires. . . . .                          | 262 |
| 8.13 | Exemples d'utilisation de variables globales. . . . .                             | 262 |
| 8.14 | Exemples de comparaisons d'objets. . . . .                                        | 263 |
| 8.15 | Autres exemples de comparaisons d'objets. . . . .                                 | 263 |
| 8.16 | Exemples de test de l'état d'objets. . . . .                                      | 264 |
| 8.17 | Exemples de blocs. . . . .                                                        | 264 |
| 8.18 | Utilisation des blocs comme abstractions fonctionnelles. . . . .                  | 265 |
| 8.19 | Utilisation des blocs dans des conditionnelles. . . . .                           | 265 |
| 8.20 | Expressions Booléennes. . . . .                                                   | 266 |
| 8.21 | Exemple d'utilisation du message <code>timesRepeat:</code> . . . . .              | 267 |
| 8.22 | Exemple d'utilisation du message <code>to: by: do:</code> . . . . .               | 267 |
| 8.23 | Exemple d'utilisation du message <code>whileFalse:</code> . . . . .               | 268 |
| 8.24 | Exemple d'utilisation de l'itérateur <code>do:</code> . . . . .                   | 268 |
| 8.25 | Exemple d'utilisation de l'itérateur <code>select:</code> . . . . .               | 269 |
| 8.26 | Exemple d'utilisation de l'itérateur <code>reject:</code> . . . . .               | 269 |
| 8.27 | Comparaison entre les différents itérateurs. . . . .                              | 269 |
| 8.28 | Déclaration de la nouvelle classe . . . . .                                       | 271 |
| 8.29 | Définition d'une méthode de classe. . . . .                                       | 272 |
| 8.30 | Définition de la méthode <code>initialize</code> . . . . .                        | 272 |
| 8.31 | Définition de la méthode <code>at:</code> . . . . .                               | 273 |
| 8.32 | Définition de la méthode <code>accessCounts</code> . . . . .                      | 273 |
| 8.33 | Modification de la méthode de classe <code>new:</code> . . . . .                  | 274 |
| 8.34 | Définition de la méthode de classe <code>numberOfMonitoredArrays</code> . . . . . | 274 |
| 8.35 | Exemple d'utilisation de la classe <code>MonitoredArray</code> . . . . .          | 275 |
| 8.36 | Exemples d'utilisation de la classe <code>Bag</code> . . . . .                    | 280 |
| 8.37 | Exemples d'utilisation de la classe <code>Array</code> . . . . .                  | 282 |
| 8.38 | Exemples d'utilisation de la classe <code>Set</code> . . . . .                    | 285 |