Cours/UPMC/LI220/7

Programmation de droite à gauche *et vice-versa*

Pascal Manoury

PARACAMPLUS

Déjà parus :

UPMC/LI101 : annales corrigées

UPMC/LI102 : cours *(épuisé)*

UPMC/LI362 : annales corrigées *(épuisé)*

LiSP : Principes d'implantation de Scheme et Lisp

UPMC/LI362 : annales corrigées, nouvelle édition

UPMC/LI220 : cours, 1ère édition *(épuisé)*

Missions de Paracamplus :

Paracamplus a pour vocation de procurer, à tous les étudiants en université, des ouvrages (annales corrigées, compléments de cours) et des environnements informatiques de travail, bref, des ressources pédagogiques propres à permettre, accélérer et entretenir l'apprentissage des connaissances enseignées.

Les ouvrages des éditions Paracamplus ont un format défini pour tenir commodément dans une poche et vous accompagner pendant vos déplacements en transports en commun. Ils bénéficient surtout d'une conception nous permettant de vous les proposer à des coûts très abordables.

Rafraîchir les annales tous les ans, procurer les compléments appropriés à chaque cours, vous proposer les meilleurs outils pour apprendre, tels sont les buts de ces ouvrages.

Découvrez-nous plus avant sur notre site **www.paracamplus.com**.

SARL Paracamplus
7, rue Viollet-le-Duc, 75009 Paris – France
ISBN 978-2-916466-06-4

Table des matières

Préambule

Chère lectrice, cher lecteur, vous trouverez dans les pages de ce livre l'étude d'un certain nombre d'aspects de la programmation auxquels l'apprenti programmeur est confronté. Nous y considérons deux éléments de la programmation : l'expression des calculs et l'organisation de leurs données. Il existe, pour chacun de ces éléments plusieurs manières de faire. L'objectif de ce livre est d'aborder ces différentes manières ainsi que d'en montrer les interactions.
Le matériel de ce livre reprend celui d'un cours dispensé aux étudiants de deuxième année du cursus de la licence d'informatique de l'université Pierre et Marie Curie [1]. Ce livre s'adresse donc *a priori* à des apprentis programmeurs ayant déjà été initiés aux arcanes de la programmation, aux mystères de l'utilisation de l'un ou l'autre de ses langages. À quelques rares exceptions près, l'auteur a choisi d'illustrer son propos au moyen d'un seul langage de programmation : *Objective Caml* (que nous appellerons par la suite *OCAML*, pour faire court). Celui-ci autorise en effet, dans un cadre syntaxique uniforme, la variété des tournures « d'expression des calculs et d'organisation de leurs données » que nous souhaitons aborder.
Toutefois, il n'est pas nécessaire pour lire et comprendre ce livre de déjà connaître le dialecte utilisé. Les éléments de syntaxe nécessaires sont introduits au fur et à mesure, une annexe les résume en quelques fiches thématiques et un index des symboles et mots clés du langage vous aidera à retrouver les informations utiles.

Ces quelques mots de préambule étant posés, nous vous invitons, chère lectrice, cher lecteur, à suivre notre pérégrination « de droite à gauche (*et vice-versa*) » dans quelques contrées du monde de la programmation.

1. Le cours intitulé *Programmes et Données Génériques* pour lequel ce livre peut servir de référence.

Chapitre 1

Récurrence et itération

L'histoire [1] de l'informatique attribue des *styles* aux langages de programmation : impératif, fonctionnel, objet, logique, etc. Tous concourent au *même objectif* : décrire des calculs qui seront exécutés par une machine. Ainsi, quel que soit leur diversité, tous les langages mettent en œuvre une *même ressource* matérielle : les micros-processeurs et elurs périphériques [2]. Tous les langages partagent une communauté de moyens pour décrire les calculs. Par exemple, on retrouve partout, sous une forme ou une autre, la *structure de contrôle* de l'alternative que l'on désigne en général avec le mot clé `if`. L'autre structure de contrôle omniprésente est la *répétition* d'un calcul. Mais ici, la forme qu'elle prend peut beaucoup plus varier selon le style de programmation adopté. C'est aux moyens d'exprimer cette répétition que nous allons nous intéresser dans ce chapitre.
Nous verrons comment, dans les langages de programmation, les *définitions récursives* et l'utilisation de *boucles* suivent ce même dessein de répéter des calculs. Et on montrera incidemment comment l'adoption d'un style n'est pas inexorablement liée au choix d'un langage. Toutefois, l'architecture d'un langage peut faciliter l'expression d'un style plutôt qu'un autre.
Nous limitons notre étude aux deux grands styles : impératif et fonctionnel. Et nous l'illustrons par une variation autour du thème de la définition dans un langage de programmation de la fonction de calcul de la factorielle d'un nombre entier.

1. Cette histoire se compte en décénies : les ordinateurs font leur apparition dans les années 40 du XXème siècle, les langages de programmation prennent leur essor la décennie suivante.

2. Quelques tentatives ont cependant été faites pour construire des machines dédiées à un langage : les machines « LISP ». Mais elles n'ont pas rencontré un succés pérenne.

1.1 Définitions récursives

Une définition *mathématique* de la fonction factorielle pourra se présenter ainsi :

$$\begin{cases} 0! & = & 1 \\ (n+1)! & = & (n+1) \times n! \end{cases}$$

C'est une définition *récursive* : dans la deuxième équation, la valeur de $(n+1)!$ est donnée en fonction de la valeur de $n!$. On dit que cette définition est *par récurrence* sur son argument. On peut même ajouter qu'il s'agit d'une récurrence *structurelle* qui repose sur la structure de l'ensemble des nombres entiers naturels : *tout entier naturel est soit* 0 *; soit le successeur d'un autre nombre entier naturel*. Pour le second cas, l'écriture $n+1$ est interprétée comme « *le successeur de* n ».
Voici une autre présentation de la définition :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

C'est à nouveau une définition récursive : dans le second cas, la valeur de $n!$ est donnée en fonction de la valeur de $(n-1)!$. Elle utilise une *alternative* (si ... sinon) plutôt qu'une analyse de la structure des entiers naturels.
Dans ces deux définitions, le processus de calcul est le même :
– ou bien on veut la valeur de la factorielle en 0 et l'on pose que c'est 1 ;
– ou bien on veut la valeur de la factorielle d'un entier naturel non nul (c'est ce qu'exprime l'écriture $n+1$ dans la première définition) et on l'obtient en multipliant cet entier naturel par la valeur de la factorielle de son *prédécesseur* (c'est ce qu'exprime l'écriture $n-1$ dans la seconde défintion).

Voyons à présent comment ces définitions récursives de la factorielle s'incarnent dans quelques langages de programmation.

Définition en Scheme

```
(define (fac n)
   (if (= n 0)
       1
       (* n (fac (- n 1)))))
```

Scheme est un langage d'*expressions* (en notation *préfixée complètement parenthésée*) qui dénotent des valeurs. Ainsi, dans cette définition on donne simplement l'expression de la valeur de la factorielle dans les deux cas de l'alternative : est-ce que l'argument est nul ou non ?

Définition en Basic (VBA)

```
Function fac (ByVal n As Integer) As Integer
    If n = 0 Then
      fac = 1
```

```
    Else
      fac = n * (fac (n-1))
    End If
End Function
```

On retrouve dans cette définition ce que nous avions en Scheme : l'alternative et les deux expressions attachées chacune à un cas. Outre les variations anecdotiques de syntaxe (notation infixe des expressions arithmétiques, écriture plus explicite de l'aternative : `If ... Else ... End If`), on trouve en plus, *l'affectation* de la valeur de la fonction (`fac = ...`). Cette *instruction* est ici nécessaire car le modèle de calcul d'un programme Basic n'est pas celui d'un pur langage d'expressions comme l'est Scheme : il faut donc explicitement indiquer que les expressions arithmétiques `1` et `n * (fac (n-1))` constituent la valeur de la fonction.

Définition en C/Java

```
int fac (int n) {
  if (n == 0)
    return 1;
  else
    return (n * (fac (n-1)));
}
```

Ici, pour les mêmes raisons, mais sous une autre forme, on retrouve l'assignation explicite de la valeur de la fonction : elle est syntaxiquement exprimée par l'instruction `return`.

Définition en OCAML

```
let rec fac (n:int) : int =
  if (n = 0) then
    1
  else
    (n * (fac (n-1)))
```

À l'instar de Scheme, OCAML est un langage d'expressions. On retrouve donc dans cette définition une structure analogue à celle que nous avions en Scheme (expression de la valeur dans chaque cas de l'alternative) à quelques variantes syntaxiques près (notation infixe des opérateurs arithmétiques, écriture de l'alternative). Notez également l'indication explicite du caractère récursif de la définition : `let rec`.

En OCAML avec *filtrage*

```
let rec fac (n:int) : int =
  match n with
    0 -> 1
  | _ -> n * (fac (n-1))
```

La construction `match-with` est spécifique aux langages de la famille ML. Elle permet une écriture concise de fonctions définies par cas. Nous

utiliserons assez souvent de telles structures de contrôle par la suite avec des types de valeur plus complexes. La structure de filtrage est donnée en annexe, paragraphe A.6, page 122.

Modèle d'exécution

Quel que soit le langage choisi, la fonction récursive définie appliquée à un entier positif (ou nul) donnera lieu à un processus d'évaluation que l'on peut représenter par les étapes suivantes :

```
(fac 3)  =  3 * (fac 2)
         =  3 * (2 * (fac 1))
         =  3 * (2 * (1 * (fac 0)))
         =  3 * (2 * (1 * 1))
         =  3 * (2 * 1)
         =  3 * 2
         =  6
```

On a deux phases du calcul :

1. une *descente récursive* qui développe la suite d'applications récursives de la fonction `fac`, tant que l'argument est non nul.
2. une *remontée* des calculs lorsque la suite des développements récursifs a été achevée, c'est-à-dire, lorsque l'argument est devenu nul : c'est le *cas de base* de la récurrence.

Les types en OCAML

Le langage OCAML est un langage *fortement typé statiquement*. Ce qui signifie que toute expression du langage doit avoir un type qui est calculé avant la génération du code exécutable des programmes. Il n'y a pas moyen en OCAML de vérifier l'appartenance d'une valeur à un type pendant l'exécution, comme on peut le faire en Scheme, ni de modifier par contrainte le type d'une valeur comme on peut le faire en C ou en Java.
Dans les deux définitions de la fonction factorielle, nous avons indiqué le type de l'argument attendu (`int`) et le type du résultat (également `int`). Nous avons donné ces indications dans l'en-tête de la définition (nous soulignons) :

```
let rec fac (n:int) : int
```

La première indication est donnée pour l'argument, la seconde pour le résultat de l'application. Le type de la fonction `fac` elle-même est noté `int -> int` où la suite de symboles `->` est utilisée pour noter la *flèche fonctionnelle*. C'est la manière du langage OCAML d'exprimer que `fac` est « *une fonction des entiers dans les entiers* ».
En fait, lorsque l'on écrit des programmes en OCAML, il n'est pas obligatoire de donner ces indications de type. Le compilateur du langage est capable de les calculer lui-même. Cette phase de la compilation s'appelle *inférence de type*.

Désormais, sauf mention du contraire, le langage de programmation utilisé est OCAML.

1.2 Récursion terminale

La décomposition en *descente récursive* puis *remontée* du processus d'évaluation de l'application d'une fonction définie par récurrence n'est pas intrinsèquement liée au caractère récursif de sa définition. En effet, tout en conservant le caractère récursif d'une définition, on peut faire l'économie de la phase de *remontée* en respectant un schéma syntaxique appelé *récursion terminale*. Selon ce schéma de définition, aucun appel récursif de la fonction définie ne se trouve en position d'argument d'une autre fonction. *A contrario* :

> Les définitions récursives de `fac` données dans la partie précédente **ne sont pas** récursives terminales car l'application récursive `(fac (n-1))` est argument de la multiplication dans l'expression `n * (fac (n-1))`.

Pour obtenir une définition récursive terminale de la factorielle, on introduit une *fonction récursive auxiliaire* qui possède un argument supplémentaire. Le rôle de cet argument supplémentaire est de contenir le résultat, partiel, puis final de la fonction. On appelle ce type d'argument un *accumulateur*. Voici la définition de la fonction auxiliaire :

```
let rec fac_loop (n:int) (r:int) : int =
  if (n = 0) then
    r
  else
    (fac_loop (n-1) (n * r))
```

C'est l'argument `r` qui joue le rôle d'accumulateur.
Cette définition récursive est *terminale*. Mais ce n'est pas la fonction factorielle attendue. En effet, de manière générale, l'application `(fac_loop n r)` ne calcule pas la factorielle de `n` (notée `n`!) mais la valeur de `n`! × `r`. Pour que cette valeur soit égale à `n`!, il suffit de prendre $\texttt{r} = 1$ (élément neutre de la multiplication). On en déduit la définition proprement dite de la fonction `fac` en *initialisant* l'argument `r` avec la bonne valeur :

```
let fac n =
  (fac_loop n 1)
```

Exécution : la modélisation équationnelle de l'évaluation de `fac` montre que l'on a effectivement fait l'économie des étapes de *remontée* de la récursion :

```
(fac 3)  =  (fac_loop 3 1)
         =  (fac_loop (3-1) (3*1))   =  (fac_loop 2 3)
         =  (fac_loop (2-1) (2*3))   =  (fac_loop 1 6)
         =  (fac_loop (1-1) (1*6))   =  (fac_loop 0 6)
         =  6
```

Définition locale La fonction auxiliaire `fac_loop` n'a pas d'intérêt en elle-même : elle n'a pour dessein que de permettre la définition de la factorielle selon le principe de récursion terminale. On peut donc « localiser » la définition de `fac_loop` à la définition de la fonction `fac`. On utilise pour cela une construction syntaxique justement appelée une *définition locale*. On l'écrit en OCAML en utilisant les mots clé `let` et `in` (cf. paragraphe A.2, page 118 de l'annexe) :

```
let fac (n:int) : int =
  let rec fac_loop (n:int) (r:int) : int =
    if (n = 0) then
      r
    else
      (fac_loop (n-1) (n * r))
  in
    (fac_loop n 1)
```

Notez : l'argument `n` de la fonction `fac` *n'est pas le même* que l'argument `n` de l'auxiliaire `fac_loop` ; mais le calcul de `fac_loop` commence avec la même valeur dans l'appel `(fac_loop n 1)`.

1.3 Procédures récursives

Les exemples divers de définitions récursives de la fonction de calcul de la factorielle ont montré que le style récursif n'est pas lié à un langage. Nous avons cependant adopté dans tous ces exemples un *style fonctionnel* de programmation. Nous allons voir à présent que le style récursif n'est pas non plus nécessairement lié au style fonctionnel et que l'on peut utiliser des définitions récursives tout en exprimant des calculs dans un *style impératif*.
Pour illustrer ce point, nous repartons de la définition récursive terminale de notre factorielle et nous nous intéresserons au traitement de l'accumulateur : plutôt que de faire évoluer sa valeur en tant qu'argument de la fonction auxiliaire (ce qui est un style fonctionnel), nous allons la faire évoluer par *modification* explicite d'une portion de *mémoire* (valeur d'une variable) accessible depuis la fonction auxiliaire (ce qui est un style impératif). Voici comment on réalise cela en C/Java :

```
int r;

void fac_loop(int n) {
  if (n == 0) { } // on ne fait rien
  else {
    r = n * r;
    fac_loop (n-1);
```

```
  }
}

int fac(int n) {
  r = 1;
  fac_loop(n);
  return r;
}
```

Notez que la « fonction » auxiliaire `fac_loop` ne donne pas de résultat : son *type de retour* est `void`. Son calcul procède par *effet de bord* en modifiant la valeur de la variable `r` qui, elle, est la valeur donnée par la fonction `fac`.
Avant d'obtenir un résultat analogue en OCAML, il nous faut faire un petit détour par les notions de *variable*, de *valeur d'une variable*, de *modification de la valeur d'une variable* et exposer leur incarnation dans le langage OCAML.

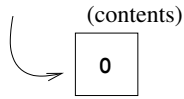
Liaison et référence

En OCAML, les définitions `let` et `let-in` créent une *liaison entre un nom et une valeur*. Cette liaison est en quelque sorte définitive, elle est *non modifiable*. Ce n'est pas le cas des langages C, BASIC ou Java ni même du langage Scheme. Dans ces langages, la valeur associée à un identificateur[3] peut être modifiée par une instruction dite d'*affectation*. C'est, dans l'exemple ci-dessus le symbole = de la ligne `r = n * r;`. En Scheme, cette opération s'appelle `set!`.
En OCAML, la possibilité de pouvoir modifier la valeur associée à un identificateur doit être explicitement indiquée. Dans le cas qui nous intéresse, on utilise la *primitive*[4] `ref` (pour *ref*erence). Par exemple `ref 0` construit une valeur modifiable dont le type n'est pas `int`, mais `int ref`. La définition `let n = ref 0` associe à `n` non pas directement la valeur `0`, mais une *référence vers* la valeur `0`. La situation de `n` est alors la suivante :
– la valeur de `n` est une *référence*, c'est-à-dire, une adresse mémoire, un *pointeur* qui est immuable ;
– la valeur référencée (le « contenu » de l'adresse) est accessible par *indirection* : on utilise la notation pointée `n.contents` ;
– la valeur référencée est modifiable par une opération d'affectation qui prend la forme : `n.contents <- n.contents + 1` (on dit abruptement *« ajouter 1 à `n` »* mais cela implique des mécanismes plus complexes en mémoire : accès au contenu d'une adresse puis modification de ce contenu).

3. En informatique, le terme « identificateur » est synonyme de celui de « nom ».
4. Pour un langage de programmation, une fonction est appelée *primitive* lorsqu'elle ne peut être définie dans le langage lui-même. En OCAML, les fonctions primitives sont définies en C (en général).

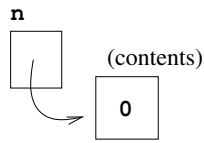
On trouve en C un mécanisme voisin avec le type « pointeur vers » et l'opérateur de type `*`.

Représentation graphique de la valeur `ref 0`



La flèche vers la cellule contenant `0` représente la référence, ou le pointeur vers cette valeur : c'est une adresse en mémoire.

Représentation graphique de la définition `let n = ref 0`



La flèche, qui est la valeur de `ref 0` (adresse), est mémorisée dans une cellule mémoire identifiée par `n` du point de vue du programmeur.
Nous reviendrons au chapitre 5, paragraphe 5.1 sur le type `'a ref`; voir également le paragraphe A.19, page 136 de l'annexe de ce livre.
Remarque : le langage OCAML ne dispose pas d'opérateur de *référencement* comme le `&` de C. En OCAML, le programmeur ne manipule jamais directement les adresses mémoire, ce sont les mécanismes du langage (compilation, bibliothèque d'exécution) qui s'en chargent. C'est la même chose en Scheme ou en Java.

Variable L'informatique a emprunté aux mathématiques le terme de « variable ». Une variable est un objet assez complexe qui relie le domaine des noms (ou identificateurs) avec celui des valeurs.
En mathématique, une variable *désigne*[5] une valeur qui doit exister, mais que l'on peut ne pas connaître exactement. Dans certaines circonstance, on peut *remplacer* dans une formule une variable par une valeur déterminée. Et dans ce cas, la variable disparaît de la formule.
En informatique, les choses sont un peu différentes. Dans les programmes informatiques, les variables ne disparaissent pas, mais on peut faire « comme si » on les remplaçait. Par exemple, dans l'expression

```
let x = 3 in
  x + x
```

tout se passe comme si on calculait la valeur de `x + x` où l'on remplace `x` par `3`, c'est-à-dire, comme si l'on calculait `3 + 3`. Il est, en informatique, un autre moyen de « remplacer » une variable par une valeur : c'est l'affectation. Mais notez que lorsque l'on décrit une affectation, par

5. Pour faire « riche », on dit *dénote*.

exemple, `x = 3` (pour reprendre la notation la plus courante des langages de programmation) on ne dit pas que « `3` remplace `x` », mais que « `3` **devient la valeur de** `x` ». Ainsi, en informatique, une variable est composée de deux entités toujours distinctes :
– son nom, que les compilateurs remplacent par des adresses en mémoire ;
– sa valeur, qui est celle que l'on trouve à cette adresse.
C'est cette distinction qui est explicitée par la syntaxe curieuse de l'affectation en OCAML : `x.contents <- 3`.
Pour plus de détails sur ce point, voir en annexe le paragraphe A.4, page 120.

Retour à la factorielle

Nous allons donc remplacer la fonction récursive auxiliaire `fac_loop` par une *procédure* récursive : c'est-à-dire une portion de code nommée et invocable comme une fonction, mais qui, plutôt que de donner une valeur comme résultat de son application, procède par *effet de bord* en modifiant la valeur d'une variable. Cette variable dont la valeur est modifiable remplace l'accumulateur de la version fonctionnelle.
Voici une première version du code correspondant qui adopte une structure proche de celle que nous avons donnée en C/Java :

```
let (r:int ref) = ref 0

let rec loop (n:int) : unit =
  if (n = 0) then ()      (* on ne "fait" rien *)
  else (
    r.contents <- n * r.contents;
    loop (n-1)
  )

let fac (n:int) : int = (
  r.contents <- 1;
  loop n;
  r.contents
)
```

Notez comment ici également, on n'utilise pas le mot clé `return` pour spécifier la valeur de la fonction `fac`. Celle-ci est simplement donnée par la valeur de la dernière expression de la *séquence* qui définit la fonction, à savoir : `r.contents`.
Sur la séquence en OCAML, voir le paragraphe A.5, page 121 de l'annexe de ce livre.

La définition de la fonction `loop` mérite commentaires Le premier est que nous venons de qualifier `loop` de « fonction » et non de « procédure » comme nous avions annoncé vouloir faire. Est-ce une faute de notre part ? Non, car si `loop` dans cette version procédurale ne délivre

pas une valeur entière, elle en a tout de même une. Un peu comme en C, nous avions attribué le type `void` à la procédure `loop`, nous lui attribuons en OCAML le type `unit`. C'est un *vrai* type en OCAML contrairement au `void` de C/Java. Le type `unit` de OCAML contient une valeur et une seule notée `()`. C'est elle qui apparaît dans la première branche de l'alternative `if (n = 0)`. D'ailleurs, cette valeur est également la valeur de retour de l'opération d'affectation. Ce sera en général la valeur attribuée à toute construction OCAML relevant de l'aspect impératif de la programmation.
Le type `unit` et son utilisation sont décrits au paragraphe A.10, page 126 de l'annexe de ce livre.

Localisation La variable `r` et la fonction récursive `loop` n'ayant de pertinence que pour le calcul de la factorielle, il est préférable d'en donner des définitions *locales* à celle de `fac` :

```
let fac (n : int) : int =
  let (r: int ref) = ref 1 in
  let rec loop (n:int) : unit =
    if (n = 0) then
      ()
    else (
      r.contents <- n * r.contents;
      loop (n-1)
    )
  in
    loop n;
    r.contents
```

Notez qu'ici, l'initialisation `r.contents <- 1` a disparu, elle est remplacée par la définition locale `let (r: int ref) = ref 1` en début de définition de `fac`.

Exécution : le tableau ci-dessous présente l'évolution du contenu de `r` au long des appels récursifs de `loop` :

(loop n)	r.contents
	1
(loop 3)	3
(loop 2)	6
(loop 1)	6
(loop 0)	6
()	6

Inverser le test Plutôt qu'indiquer de ne rien faire lorsque `n=0`, autant dire ce qu'il faut faire lorsque ça n'est pas le cas. C'est-à-dire, puisque l'on travaille avec des entiers positifs, lorsque `(n > 0)`. L'alternative qui contrôle les appels récursifs, teste donc une condition de *continuation* à la place d'une condition d'arrêt :

```
let fac (n:int) : int =
```

```
let (r: int ref) = ref 1 in
let rec loop (n:int) : unit =
  if (n > 0) then (
    r.contents <- n * r.contents;
    (loop (n-1))
  )
in
 (loop n);
 r.contents
```

Remarque : syntaxiquement, on peut se passer *ici* de l'alternant `else ()` : c'est le seul cas licite de `if` *unilatère* en OCAML car le type du résultat est `unit`. Une utilisation unilatère de l'alternative de la forme `if` e_1 `then` e_2 doit toujours *être équivalente* à `if` e_1 `then` e_2 `else ()`.

Procédure récursive sans argument Ce que nous avons fait pour l'accumulateur, pourquoi ne pas le faire pour l'entier dont on veut calculer la factorielle ? Plutôt que de passer en argument à la fonction récursive un `n` diminué de `1`, on peut aussi « externaliser » cet argument, comme nous l'avons fait de l'accumulateur. Nous obtenons ainsi une fonction récursive auxiliaire sans argument qui modifie les valeurs de deux variables : l'accumulateur `r` et la variable `i` qui compte le nombre de fois où il faut *répéter* le calcul :

```
let fac (n:int) : int =
  let (r: int ref) = ref 1 in
  let (i: int ref) = ref n in
  let rec loop () : unit =
    if (i.contents > 0) then (
      r.contents <- i.contents * r.contents;
      i.contents <- i.contents - 1;
      loop ()
    )
  in
   loop ();
   r.contents
```

Notez l'ordre des affectations dans `loop` : on modifie d'abord `r` puis `i`. Procéder en sens inverse eût faussé le résultat : nous aurions alors obtenu $(n-1)!$, voire 0 avec `(fac 1)`.
Stricto sensu, la fonction auxiliaire `loop` n'est pas réellement *sans argument* puisque son type est `unit -> unit`. Elle attend en fait toujours le même argument, l'unique valeur du type `unit` notée `()` pour activer son calcul.

1.4 Définitions itératives

Du point de vue du processus de calcul, les définitions récursives des fonctions provoquent la *répétition* de la séquence de calcul exprimée

dans la définition. Les langages de programmation offrent d'autres possibilités pour provoquer la répétition d'une séquence de calcul. Ce sont les *structures de contrôles* que l'on appelle des *boucles*.
L'application de la fonction récursive auxiliaire `loop` répète la séquence

```
      r.contents <- i.contents * r.contents;
      i.contents <- i.contents - 1
```

tant que `i.contents` reste strictement positif. Le langage OCAML, et la plupart des autres langages de programmation, fournissent une construction pour pouvoir répéter une séquence tant qu'une condition est vérifiée : c'est la boucle `while`. Nous pouvons l'utiliser pour donner une définition *itérative* de la fonction `loop` :

```
let fac (n:int) =
  let (r: int ref) = ref 1 in
  let (i: int ref) = ref n in
  let loop () : unit =
    while (i.contents > 0) do
      r.contents <- i.contents * r.contents;
      i.contents <- i.contents - 1
    done
  in
    loop();
    r.contents
```

On peut même se passer ici de la définition de la fonction auxiliaire et remplacer son application par l'écriture de la boucle elle-même :

```
let fac (n:int) =
  let (r: int ref) = ref 1 in
  let (i: int ref) = ref n in
    while (i.contents > 0) do
      r.contents <- i.contents * r.contents;
      i.contents <- i.contents - 1
    done;
    r.contents
```

Itération bornée

Dans le calcul de la factorielle, si l'argument `n` est strictement positif, on effectue n multiplications. À chaque tour, on multiplie le résultat avec une valeur décroissant de 1 chaque fois (la variable `i` dont la valeur initiale est celle de `n`). On connaît donc à l'avance le nombre de fois où il faudra multipler le résultat. Les itérations dont on connaît à l'avance le nombre de répétitions sont appelées *itérations bornées*. Des langages de programmation fournissent des constructions pour de telles boucles : les boucles `for` qui permettent de poser la valeur initiale et la valeur finale d'un *indice de boucle*. Ces indices sont des valeurs entières incrémentées ou décrémentées d'une unité à chaque itération selon que l'intervalle qui leur est assigné est croissant ou décroissant.

Pour définir la factorielle, on peut utiliser un indice `i` dont l'intervalle de valeurs est croissant de 1 à `n`. On obtient alors la définition suivante :

```
let fac (n:int) : int =
  let r = ref 1 in
    for i = 1 to n do
      r.contents <- i * r.contents
    done;
    r.contents
```

Si `n` vaut 0 alors la boucle n'est pas exécutée et l'on a bien que `(fac 0)` vaut 1.

1.5 Généralisation

La factorielle est un cas particulier de la forme de calcul

$$f(n, f(n-1, \ldots, f(1, a)\ldots))$$

où f est la multiplication et a son élément neutre, 1.
Dans un langage tel que OCAML, les fonctions peuvent être manipulées comme d'autres valeurs (c'est ce que l'on dit lorsque l'on dit que OCAML « est un langage fonctionnel »). Une fonction peut être argument, voire résultat d'une autre fonction. On peut donc *programmer*, en toute généralité la forme de calcul utilisée par la factorielle en définissant l'*itérateur* suivant :

```
let rec iter (f: int -> int -> int) (a:int) (n:int) : int =
  if (n = 0) then
    a
  else
    (f n (iter f a (n - 1)))
```

Le type complet de `iter` s'écrit : `(int -> int -> int) -> int -> int -> int`. Pour « lire » ce type, il faut prêter attention aux parenthèses. Le premier argument de `iter` est de type `(int -> int -> int)` ; c'est donc une fonction qui à deux entiers (les deux premiers `int` du type) associe un entier (le dernier `int`). Le deuxième et le troisième argument de `iter` sont des entiers. Et le résultat de l'application de `iter` est un entier.
On peut également définir cette fonction d'itération sans définition récursive, dans un style impératif, avec une boucle d'itération bornée :

```
let iter (f: int -> int -> int) (a:int) (n:int) : int =
  let r = ref a in
    for i=1 to n do
      r.contents <- (f i r.contents)
    done;
    r.contents
```

Quelle que soit la définition donnée de la fonction générique d'itération `iter`, on peut l'appliquer pour définir la factorielle :

```
let fac (n:int) : int =
  iter ( * ) 1 n
```

Commentaire syntaxique : pour pouvoir utiliser le symbole *infixe* de la multiplication comme une fonction ordinaire il faut le placer entre deux parenthèses.

Typage et polymorphisme

Les types que nous avons indiqués dans notre définition de `iter` ne parlent que des entiers car nous avions en tête son application pour définir la factorielle. Mais si nous laissons au compilateur le soin de calculer lui-même le type de `iter`, en écrivant, sans indication de type :

```
let rec iter f a n =
  if (n = 0) then
    a
  else
    (f n (iter f a (n-1)))
```

Nous obtenons que `iter` est de type `(int -> 'a -> 'a) -> 'a -> int -> 'a`. Il est *plus général* que celui que nous avions indiqué, en ce sens que là où nous avions partout des `int`, il sont parfois remplacés par la variable de type `'a`.
La structure globale du type inféré est semblable à celle que nous avions indiqué (comparez les places des flèches et des parenthèses), mais, là où nous avions partout des `int`, l'inférence de type du compilateur nous donne tantôt également `int`, tantôt, un type noté `'a` qui correspond en fait à un type non déterminé : une *variable de type*. Un type qui contient des variables de type est appelé *polymorphe* et une fonction dont le type est polymorphe est également qualifiée de polymorphe. On peut appliquer une telle fonction à des valeurs d'un type quelconque, partout où elle attend un argument de type indéterminé. L'application est correcte du point de vue des types lorsqu'il est possible d'*instancier* une variable de type avec un même type.
Par exemple, on peut utiliser `iter` pour définir le calcul de

$$\frac{1}{n} + \ldots + \frac{1}{2} + 1$$

dont le résultat sera une valeur de type `float`. Nous utilisons pour définir la fonction itérée, les opérations sur les flottants ainsi que la primitive de conversion des entiers en flottants (`float_of_int`) :

```
let sum_frac n =
  let f n x =
    (1. /. (float_of_int n)) +. x
  in
    iter f 0. n
```

Le type de `sum_frac` est `int -> float`. L'instance du type de `iter` utilisée pour inférer ce type est :

```
(int -> float -> float) -> float -> int -> float.
```

C'est le type `(int -> 'a -> 'a) -> 'a -> int -> 'a` où `float` remplace la variable de type `'a`.