Foire aux questions sur Caml

Contacter l'auteur *Pierre.Weis@inria.fr*

Fichier créé en octobre 1995.

Table des matières:

Questions d'ordre général

Qu'est-ce que Caml ?

Caml est un langage de programmation. C'est un langage fonctionnel car les briques de base des programmes sont les fonctions. C'est un langage fortement typé, ce qui signifie que les objets manipulés appartiennent à un ensemble identifié par un nom qu'on appelle son type. En Caml, les types sont gérés et manipulés par la machine, sans intervention de l'utilisateur (types synthétisés).

Le langage est disponible sur à peu près toutes les machines Unix, sur PC (à partir de 386) et sur Macintosh. Une utilisation confortable nécessite 2 MO de mémoire libre et 2 MO de mémoire sur disque dur.

Un bref tour d'horizon des principales caractéristiques de Caml.
Plus de détails sur la distribution de Caml.

Que signifie le nom ``Caml'' ?

``Caml'' est un acronyme en langue anglaise: ``Categorical Abstract Machine Language'', c'est-à-dire langage de la machine abstraite catégorique. La CAM est une machine abstraite capable de définir et d'exécuter les fonctions, et qui est issue de considérations théoriques sur les relations entre la théorie des catégories et le lambda-calcul. Le premier compilateur du langage générait du code pour cette machine abstraite (en 1984).
La deuxième filiation de ce nom est ML (acronyme pour Meta Language): Caml est aussi issu de ce langage de programmation crée par Robin Milner en 1978, et qui servait à programmer les tactiques de preuves dans le système de preuves LCF.

Doit-on écrire Caml ou CAML ?

Facile de répondre à cette question: il suffit de lire le fichier que vous êtes en train de consulter! Il y a un nombre incroyable d'occurrences du mot Caml, et les rares occurrences du mot CAML apparaissent dans le texte de, et la réponse à, la question «Doit-on écrire Caml ou CAML ?». Que faut-il en conclure ? On écrit Caml bien sûr ! Mais voilà une explication détaillée.

``Caml'' est un acronyme en langue anglaise: ``Categorical Abstract Machine Language'', c'est-à-dire langage de la machine abstraite catégorique. Comme tout acronyme, il devrait s'écrire en lettres capitales comme INRIA ou SNCF. Mais ce nom tout en capitales est trop «mastok» et inélégant, c'est pourquoi nous écrivons Caml (plus élégant encore serait `C{\smallcaps aml}`). Ce parti pris de l'élégance s'est révélé payant avec l'introduction d'Objective Caml que personne ne s'avise d'écrire Objective CAML (à ma connaissance).

Donc, on écrit Caml pour être élégant et l'on pense CAML pour dire du bien de cet outil extraordinaire!

Est-ce un langage compilé ou interprété ?

Caml est un langage compilé. Cependant tous les systèmes Caml (on entend par ``système'' l'ensemble compilateur+bibliothèques associées) proposent une boucle d'interaction ``toplevel'', qui ressemble à s'y méprendre à un interprète. En effet dans le système interactif, l'utilisateur tape des morceaux de programmes (des ``phrases'' Caml) qui sont traitées instantanément par le système, qui les compile, les exécute et écrit les résultats à la volée.

Pour l'interactivité, est-ce analogue à perl ?

C'est-à-dire est-ce qu'on tape la phrase à tester sur la ligne de commandes ?

Non. On lance le système interactif, puis on interagit avec lui. Par exemple, si vous utilisez le système Caml Light, sous Unix, vous taperez

```
$ camllight
```

pour appeler le système interactif. Puis vous taperez la ou les phrases Caml que vous voulez tester. Par exemple:

```
$ camllight
>       Caml Light version 0.74

#1 + 2;;
- : int = 3
#
```

Comment arrêter le système interactif ?

Il est souvent possible d'interrompre un programme ou le système Caml en appuyant sur une combinaison de touches qui dépend du système d'exploitation: sous Unix provoquer une interruption (en général Contrôle-C), sur le Macintosh taper Commande-., sous Windows utiliser le menu Caml.

Comment quitter le système interactif ?

Taper:

```
quit();;
```

ou bien indiquer une fin de fichier (CTRL-D sous Unix, CTRL-Z sous DOS, ...)

Pour terminer une phrase Caml, il faut toujours taper 2 point-virgules ?

Oui, puisque c'est l'indication de fin de phrase! Et il faut même ajouter un retour chariot!

Comment compiler un programme ?

On écrit le programme dans un fichier (dont le nom se termine par le suffixe `".ml"`), puis on demande au compilateur indépendant (ou compilateur ``batch'') de compiler ce fichier. Dans ce cas, il est évident qu'on n'a plus aucune aide du système interactif pour l'impression des résultats, et qu'il faut les imprimer soi-même.

Par exemple: si le fichier toto.ml contient

```
let x = 2;;

print_int (x * x);;

exit 0;;
```

On le compile (sous Unix) par la commande

```
$ocamlc toto.ml
```

qui crée un programme compilé exécutable (pour Caml Light le compilateur indépendant s'appelle `camlc`). On lance ensuite ce programme exécutable par la commande adéquate du système d'exploitation (par défaut le programme exécutable a pour nom `a.out` sous Unix):

```
$ a.out
4$
```

Les phrases du fichier ont été exécutées dans l'ordre de leur présentation dans le fichier.

Quelle est la différence entre Caml V3.1, Caml Light, Objective Caml ?

Ce sont différents systèmes Caml: différents compilateurs et différentes bibliothèques. De plus tous ces systèmes proposent leurs propres extensions au langage.

Ces systèmes partagent un grand nombres de traits en commun: c'est le coeur du langage Caml. Ainsi la syntaxe de base du langage est commune à tous. Les systèmes de modules sont tous différents, le plus simple est celui de Caml Light, le plus complexe (mais le plus puissant) celui de Objective Caml. Les compilateurs sont souvent de technologie différente (code natif ou byte-code), certains systèmes proposent aussi plusieurs compilateurs. Certains systèmes tournent sur plus de machines que d'autres. Actuellement le système Caml Light est le plus répandu. Il tourne sur presque toutes les machines Unix, sur les PC et sur les Macintosh.

Le système Objective Caml est plus expérimental. Il dispose de deux compilateurs étroitement couplés: un compilateur vers code natif (optimisant le code produit mais plus lent à compiler, c'est `ocamlopt`) et un compilateur byte-code (code compilé plus lent mais très grande vitesse de compilation, c'est `ocamlc`).

Pourrais-je avoir des messages en Français ?

Vous pouvez choisir la langue dans laquelle Caml Light rédige ses messages.
Pour cela, il vous faut préciser votre langue à Caml Light (le nom des langues est celui utilisé par Internet):

- en Unix: définir la variable d'environnement `LANG`, ou appeler le système avec l'option `-lang`.
- sur PC utiliser l'option `-lang` sur la ligne de commande ou dans le fichier `CAMLWIN.INI`.
- sur Macintosh éditer les ressources de l'application.

Les langues actuellement disponibles sont:

- fr: le français.
- es: l'espagnol.
- de: l'allemand.
- it: l'italien.
- src: l'anglais.

L'anglais est la langue par défaut pour tout message non traduit (ou bien pour toute langue inconnue). Si votre langue n'est pas encore disponible, et que vous êtes volontaire pour effectuer la traduction, prenez contact avec l'équipe Caml (en écrivant à `caml-light@inria.fr`).

Où trouver de la documentation ?

Livres d'introduction en Français:

- Pour le programmeur (introduction progressive au langage, nombreux exemples de taille conséquente, orientation programmation): Pierre Weis et Xavier Leroy, Le langage Caml (Intereditions, ISBN 2-7296-0493-6).
- Pour le professeur et le programmeur intéressé par la théorie des langages: Thérèse Hardin et Véronique Donzeau-Gouge, Concepts et Outils de Programmation -- le style fonctionnel, le style impératif avec Caml et Ada (Intereditions, ISBN 2-7296-0419-7).
- Enfin pour une orientation algorithmique exprimée dans un langage fonctionnel: Guy Cousineau et Michel Mauny, approche fonctionnelle de la programmation (Ediscience, ISBN 2-84074-114-8).
- Un manuel de référence du langage: Xavier Leroy et Pierre Weis, Manuel de Référence du langage Caml (Intereditions, ISBN 2-7296-0492-8).
- consultez le serveur de l'INRIA au sujet de Caml:

  `http://pauillac.inria.fr/caml/index-fra.html`

Où signaler une erreur du système Caml ?

Avant tout, vérifiez s'il vous plaît qu'il s'agit effectivement d'une erreur de Caml, non d'un aspect documenté du langage inconnu de vous. En cas d'erreur rétive, il vous faut signaler l'erreur à l'équipe des implémenteurs de Caml (`caml-bugs@inria.fr`). N'oubliez pas de préciser votre machine et la version de Caml que vous utilisez. Si possible, essayez de circonscrire l'erreur en écrivant un petit exemple reproduisant l'erreur. Merci d'avance.

Quelle est la différence entre Caml et Caml Light ?

Le système Caml V3.1 est l'ancêtre de Caml Light. Ce système possède de nombreux traits intéressant (et plutôt complexes), difficiles à implémenter. C'est pourquoi Caml V3.1 ne tourne que sur certaines machines Unix. Au contraire, le système Caml Light est beaucoup plus simple et portable, et tourne sur presque toutes les machines Unix, PC ou Macintosh. De plus, de nombreux traits utiles de Caml V3.1 sont maintenant disponibles en Caml Light, si bien que la différence entre ces deux systèmes s'amenuise. Vous pourrez trouver des détails ici.

Peut-on faire du graphisme en Caml ?

Objective Caml et Caml Light disposent d'une bibliothèque de commandes graphiques indépendantes du matériel (le même programme tourne aussi bien sous Unix, Macintosh ou PC).

Sur le Macintosh et le PC la bibliothèque graphique est intégrée à l'application. En revanche sous Unix, les primitives graphiques sont fournies par une bibliothèque externe qui se trouve dans les contributions, qu'il faut donc préalablement compiler et installer. On lance alors un système interactif comportant le graphique à l'aide de la commande

```
$ camllight camlgraph
```

```
ocamlmktop -o ocamlgraph graphics.cma
```

```
ocamlgraph
```

Dans tous les cas, on a accès aux primitives graphiques en ouvrant le module du graphique (`#open "graphics";;` pour Caml Light, (`open Graphics;;` pour Objective Caml). On fait apparaître la fenêtre de dessin en faisant appel à la fonction `open_graph` qui prend en argument une chaîne de description de la géométrie de la fenêtre (par défaut une chaîne vide assure un comportement raisonnable).
Ainsi, un programme utilisant le graphique comportera ces deux premières lignes:

```
#open "graphics";;
open_graph "";;
```

La taille de l'écran graphique dépend de l'implémentation; on la fixe en précisant la géométrie de la fenêtre dans la chaîne de caractères argument de `open_graph`.

L'origine du système de coordonnées est toujours en bas et à gauche. L'axe des abscisses va classiquement de la gauche vers la droite et l'axe des ordonnées du bas vers le haut. Il y a une notion de point courant et de crayon avec une taille et une couleur courantes. On déplace le crayon, sans dessiner ou en dessinant des segments de droite par les fonctions suivantes:

- `moveto x y`: déplace le crayon aux coordonnées absolues (`x`, `y`).
- `lineto x y`: trace une ligne depuis le point courant jusqu'au point de coordonnées (`x`, `y`).
- `plot x y`: trace le point argument (`x`, `y`).

Autres opérations:

- `close_graph ()`: détruit la fenêtre graphique.
- `clear_graph ()`: efface l'écran.
- `current_point`: renvoie les coordonnées du point courant.
- `size_x ()`: renvoie la taille de l'écran en abscisses.
- `size_y ()`: renvoie la taille de l'écran en ordonnées.
- `background` et `foreground` sont respectivement les couleurs du fond et du crayon.
- `set_color c`: fixe la couleur du crayon. (Les couleurs sont obtenues par la fonction `rgb` mais on dispose des couleurs prédéfinies: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.)
- `point_color x y`: renvoie la couleur du point.

Dans la fenêtre graphique, on imprime avec les fonctions:

- `draw_char c`: affiche le caractère `c` au point courant dans la fenêtre graphique.
- `draw_string s`: même chose pour la chaîne de caractères `s`.

La bibliothèque graphique permet également de surveiller un certain nombre d'événements indiquant une interaction entre la machine et l'utilisateur.

- `wait_next_event evl`: attend jusqu'à ce que l'un des événements de la liste `evl` se produise, et renvoie le statut de la souris et du clavier à ce moment-là. Les événements sont:
  - `Button_down`: le bouton de la souris a été pressé.
  - `Button_up`: le bouton de la souris a été relaché.
  - `Key_pressed`: une touche a été appuyée.
  - `Mouse_motion`: la souris a bougé.
  - `Poll`: ne pas attendre, retourner de suite.
- `button_down ()` renvoie la valeur vraie si le bouton de la souris est enfoncé, faux sinon.
- `mouse_pos ()` renvoie les coordonnées du curseur de la souris.
- `read_key ()` attend l'appui sur une touche, la renvoie.
- `key_pressed ()` renvoie la valeur vraie si une touche est enfoncée, faux sinon. Tant que la touche enfoncée n'est pas lue par `read_key ()`, la fonction `key_pressed` renvoie invariablement la valeur vraie.

La bibliothèque graphique propose également d'autres primitives, et en particulier des primitives de manipulation des images que nous ne décrirons pas ici.

Voici un exemple de petit programme illustrant l'usage de quelques fonctions graphiques: on fait évoluer une nappe de droites par translation, jusqu'à ce que l'utilisateur appuie sur une touche (dans la fenêtre graphique).

```
#open "graphics";;
```

```
open_graph "";;
```

```
type point = {mutable x : int; mutable y : int};;

type droite = {mutable origine : point; mutable extrémité : point};;

let moveto_point p = moveto p.x p.y
and lineto_point p = lineto p.x p.y;;

let dessine_droite couleur d =
 set_color couleur;
 moveto_point d.origine;
 lineto_point d.extrémité;;

let trace d = dessine_droite foreground d;;
let efface d = dessine_droite background d;;

let nombre_de_droites = 100;;

let droites =
    make_vect nombre_de_droites
     {origine = {x = 10; y = 10};
      extrémité = {x = 20; y = 100}};;

let translation_origine = {x = 3; y = 2}
and translation_extrémité = {x = -2; y = 5};;

let translate v p =
  let x1 = p.x + v.x
  and y1 = p.y + v.y in
  let nouvel_x =
   if x1 >= 0 && x1 <= size_x () then x1
   else begin v.x <- - v.x; p.x + v.x end
  and nouvel_y =
   if y1 >= 0 && y1 <= size_y () then y1
   else begin v.y <- - v.y; p.y + v.y end in
  {x = nouvel_x; y = nouvel_y};;

let nouvelle_droite d =
  {origine = translate translation_origine d.origine;
   extrémité = translate translation_extrémité d.extrémité};;

let i = ref 0;;

let step () =
  let nouvel_i = (!i + 1) mod nombre_de_droites in
  efface (droites.(nouvel_i));
  droites.(nouvel_i) <- nouvelle_droite (droites.(!i));
  trace (droites.(nouvel_i));
  i := nouvel_i;;

let main () =
 while not (key_pressed ()) do
  step ()
 done;
 read_key();;
```

Vous pouvez aussi consulter un autre exemple de programme graphique.

Comment faire des calculs en précision arbitraire en Caml ?

Caml Light dispose d'une bibliothèque de calcul en grande précision, sur les nombres rationnels.
Cette bibliothèque se trouve dans le répertoire contrib, il faut donc l'avoir compilée et installée. Ensuite on peut l'utiliser dans les programmes compilés ou lancer un système interactif comportant les grands nombres à l'aide de la commande (sous Unix):

```
$ camllight camlnum
```

Les opérations sur les grands nombres sont suffixées par le caractère `/`: par exemple l'addition se note `+/`. On crée des grands nombres par conversion à partir d'entiers ou de chaînes de caractères.

Tout d'abord, je définis un imprimeur pour le type `num`, puis j'effectue le calcul 1/3 + 2/3:

```
#open "num";;
#open "format";;
#let print_num n = print_string (string_of_num n);;
print_num : num -> unit = <fun>
#install_printer "print_num";;
- : unit = ()
#num_of_string "2/3";;
- : num = 2/3
#let n = num_of_string "1/3" +/ num_of_string "2/3";;
n : num = 1
```

Nous pouvons maintenant définir la fonction factorielle:

```
#let rec fact n =
 if n <= 0 then (num_of_int 1) else num_of_int n */ (fact (n - 1));;
fact : int -> num = <fun>
#fact 100;;
- : num =
 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000
```

Comment mesurer le temps en Caml ?

Le plus simple est d'utiliser la fonction `time : unit -> float` du module `sys`, qui donne le temps écoulé depuis le lancement du programme (ou du système interactif): vous utiliserez donc `sys__time` en Caml Light et `Sys.time` en Objective Caml.
Sous Unix, vous pouvez aussi utiliser l'interface avec le système (module unix, fonction `unix__time` pour Caml Light et `Unix.time` pour Objective Caml). Plus précis, mais plus complexe à mettre en oeuvre.

Comment installer OCaml sur Mac OS X ?

Suivez les indications de la page http://caml.inria.fr/pub/old_caml_site/caml-macosx-howto/index.html.

Syntaxe

Quelle est la syntaxe du langage ?

Commencez par consulter le guide simplifié de ``La syntaxe de Caml''. Lisez aussi quelques programmes simples, comme ceux du fichier Quelques lignes de Caml. En dernière analyse, difficile de répondre à cette question autrement qu'en renvoyant au manuel!

Un bon conseil est d'écrire ses premiers programmes par modification de programmes qui marchent déjà. Pour cela, il faut aller fouiller dans le répertoire des exemples de la distribution (répertoires `examples/*`) ou encore jetter un coup d'oeil au fichier Quelques lignes de Caml.

Pour vous donner une idée de la syntaxe et de la sémantique du langage, je vous commente ci-dessous le plus petit exemple donné dans la distribution du système Caml Light: le programme `examples/basics/fib.ml` (j'ai ajouté des numéros en début de ligne pour faciliter l'explication):

```
1: (* The Fibonacci function, once more. *)

2: let rec fib n =
3:   if n < 2 then 1 else fib(n - 1) + fib(n - 2)
4: ;;

5: if sys__interactive then () else
6: if vect_length sys__command_line <> 2 then begin
7:   print_string "Usage: fib <number>";
8:   print_newline()
9: end else begin
10:   try
11:     print_int(fib(int_of_string sys__command_line.(1)));
12:     print_newline()
13:   with Failure "int_of_string" ->
14:     print_string "Bad integer constant";
15:     print_newline()
16: end
17: ;;
```

- La ligne 1 est un commentaire.
- La ligne 2 indique la définition (`let`) récursive (`rec`) d'une fonction appelée `fib` et de paramètre `n`. Le corps de `fib` est l'expression qui suit le signe = (lignes 3 et 4). À la ligne 3 on teste l'argument `n` et on renvoit `1` ou le résultat d'un appel récursif selon le cas.
- La ligne 4 termine la première phrase du fichier par une marque de fin de phrase `;;`
- La ligne 5 commence la deuxième phrase du fichier qui se termine à la ligne 17.
- La ligne 5 teste si le fichier est exécuté dans le système interactif ou comme un programme indépendant. Si la variable `sys__interactive` est vraie, on est dans le système interactif et on laisse le soin à l'utilisateur d'appeler fib sur l'argument de son choix: on ne fait rien (``()'').
- En revanche, si le programme est utilisé comme une commande indépendante, on exécute le code de la ligne 6 à la ligne 17.
- La ligne 6 consiste à tester le nombre d'arguments fournis à la commande : s'il n'y a pas exactement un argument (en plus du nom de la commande qui est fourni implicitement), on affiche un message d'erreur (`print_string "Usage:..."`), suivi (le séquencement est indiqué par un point-virgule en Caml) d'un retour à la ligne (`print_newline()`).
- Sinon, un argument a été fourni, et on exécute les lignes 10 à 17. On imprime le résultat du calcul de `fib` sur l'argument fourni (`print_int (fib ...)`), suivi d'un retour-chariot dans les lines 11 et 12.

try ... with

Le `try` de la ligne 10, avec sa partie `with` associée (ligne 13 à 15) est la construction Caml qui sert à la gestion des erreurs: la construction ``try expression with filtrage'' permet d'exécuter ``expression'' en surveillant les erreurs qui peuvent se produire pendant cette exécution (on envisage les différents cas d'erreur dans la partie ``filtrage'' du ``try ... with'').

Dans l'exemple précédent, si l'utilisateur du programme ne l'a pas appelé avec un argument qui peut être converti en entier (il a lancé par exemple `$fib coucou`), alors la fonction int_of_string de la ligne 11 échoue avec l'exception (`Failure "int_of_string"`). Cette erreur est alors rattrapée dans la partie ``with'' du ``try'' qui affiche un message d'erreur approprié (lignes 14 et 15).

begin ... end

Les mots-clefs `begin` et `end` sont équivalents à des parenthèses; on les utilisent surtout pour entourer les séquences d'instructions (expressions séparées par des ; comme celles des lignes 7 et 8) ou encore les constructions ``match ... with'' et les ``try ... with'', comme aux lignes 9 et 16.

raise exception

La primitive `raise` sert à déclencher des exceptions (signaler des erreurs) en cas d'impossibilité de poursuivre les calculs. Ces exceptions devront être rattrapées par une construction ``try ... with'' englobante pour traiter l'erreur (soit en interrompant le programme après avoir imprimé un message, soit en poursuivant le traitement d'une autre manière). Si personne ne rattrape l'erreur l'échec se propage jusqu'à l'arrêt complet de l'évaluation. Par exemple:

```
#print_string "Bonjour";
#raise (Failure "division par zéro");
#print_string " à tous ";;
BonjourException non rattrapée: Failure "division par zéro"
```

Comment définir une fonction ?

En Caml, les définitions de fonctions sont très proches de la syntaxe des mathématiques: on introduit la définition par le mot-clef `let`, suivi du nom de la fonction et de ses arguments; enfin la formule de calcul de l'image de l'argument est introduite par un signe =.

```
#let successeur (n) = n + 1;;
successeur : int -> int = <fun>
```

Variantes et autres cas de définitions de fonctions:

- les parenthèses autour de l'argument sont facultatives.
- les arguments peuvent être multiples.
- les fonctions peuvent être récursives: utiliser `let rec` au lieu de `let`.
- les fonctions peuvent être anonymes.
- les fonctions peuvent manipuler des n-uplets.

Comment définir une procédure ?

Rappelons qu'on appelle traditionnellement *procédure*, toute commande qui produit un *effet* (par exemple une impression au terminal ou la modification d'une case de la mémoire) mais n'a pas de résultat mathématiquement significatif.

En Caml il n'y a pas à proprement parler de différence entre procédure et fonction: une procédure est un cas particulier de fonction qui a pour résultat la valeur triviale ``rien" c'est-à-dire `()`. Par exemple la primitive `print_string` qui imprime une chaîne de caractères au terminal ne rend que `()` pour tout résultat, indiquant ainsi qu'elle a effectué le travail demandé.

Les procédures qui ne nécessitent pas d'argument significatif ont encore une fois `()` pour argument. Par exemple, la procédure `print_newline` qui passe une ligne au terminal n'a pas d'argument significatif: elle est de type `unit -> unit`.
Les procédures avec argument sont définies exactement comme des fonctions. Par exemple:

```
#let message s = print_string s; print_newline();;
message : string -> unit = <fun>
#message "Bonjour";;
Bonjour
- : unit = ()
```

Remarquez qu'il est impossible de définir en Caml une procédure sans argument: sa définition donnerait lieu à son exécution, et on ne pourrait plus l'appeler ultérieurement. Dans l'exemple suivant `double_newline` vaut `()`, et son évaluation ne produit pas de retour chariots.

```
#let double_newline = print_newline(); print_newline();;


double_newline : unit = ()
#double_newline;;
- : unit = ()
```

La définition et l'utilisation correcte de la procédure serait:

```
#let double_newline () = print_newline(); print_newline();;
double_newline : unit -> unit = <fun>
#double_newline;;
- : unit -> unit = <fun>
#double_newline();;


- : unit = ()
```

Comment définir une fonction à plusieurs arguments ?

Il suffit d'écrire la liste des arguments successifs lors de la déclaration de la fonction. Par exemple:

```
#let somme x y = x + y;;
somme : int -> int -> int = <fun>
```

et de fournir les arguments dans le bon ordre à l'application:

```
#somme 1 2;;
- : int = 3
```

Ces fonctions sont appelées fonctions curryfiées, par opposition aux fonctions ayant des n-uplets pour argument.

Comment manipuler des paires ou des n-uplets ?

En Caml, les n-uplets ont la même syntaxe qu'en mathématiques: une liste d'expressions entre parenthèses et séparées par des virgules.

```
#(1, 2);;
- : int * int = 1, 2
```

Comment accéder aux éléments des paires ou des n-uplets ?

On accède généralement aux éléments des n-uplets par filtrage, au moyen de la définition d'un couple d'identificateurs.

```
#let (x, y) = let z = 1 in ((z + 1), (z + 2));;
x : int = 2
y : int = 3
```

Comment définir une fonction manipulant des n-tuplets ?

Les fonctions peuvent prendre en argument ou rendre en résultat des n-uplets de valeurs, sans aucune restriction.

```
#let add (x, y) = x + y;;
add : int * int -> int = <fun>
#add (1, 2);;
- : int = 3
#let div_mod x y = (x quo y, x mod y);;
div_mod : int -> int -> int * int = <fun>
#div_mod 15 7;;
- : int * int = 2, 1
```

(Note: pour coder des fonctions à plusieurs arguments, l'usage veut qu'on définisse des fonctions curryfiées plutôt que des fonctions avec n-uplets d'arguments: on écrira `let f x y = ...`, plutôt que `let f (x, y) = ...`.)

Comment définir une fonction anonyme ?

Une fonction peut n'avoir pas de nom: on parle alors d'une valeur fonctionnelle ou d'une fonction anonyme. Une valeur fonctionnelle est introduite par le mot clef `function`, suivi de l'argument de la fonction, d'une flèche `->` et du corps de la fonction. Par exemple

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

Comment appliquer une fonction ?

On applique les fonctions comme en mathématiques en écrivant le nom de la fonction suivi de son argument entre parenthèses: f (x). En pratique les parenthèses ne sont obligatoires que lorsque l'expression argument ``x'' est complexe. On les omet donc lorsqu'il s'agit d'une constante ou d'une variable: on écrit `fib 2` au lieu de `fib (2)` et `fact x` au lieu de `fact (x)`.
Difficulté: les parenthèses restent obligatoires pour les arguments complexes.

Arguments syntaxiquement complexes

Lorsque l'expression argument d'une fonction est plus complexe qu'un simple identificateur, il faut mettre cet argument entre parenthèses. Cela est valable en particulier si l'argument d'une fonction est un nombre négatif: il faut le parenthéser.
L'application de `f` à `-1` s'écrit donc `f (-1)`

et non `f -1` qui est interprété exactement comme `x - 1` (c'est donc une soustraction).

Comment appliquer une fonction dans une opération ?

Il est prudent, mais pas nécessaire, de parenthéser un appel de fonction qui apparaît dans l'un des opérandes d'une opération binaire. On peut donc écrire `fact x + 1` pour signifier `(fact x) + 1`.

Mon programme boucle sans raison ?

Si l'argument d'une fonction est une opération il faut impérativement parenthéser cette opération:
pour appliquer `f` à `x + 1` on doit écrire `f (x + 1)`

et non `f x + 1` qui est une addition qui signifie `(f x) + 1`. Quand on oublie les parenthèses, le vérificateur de types trouve souvent une erreur dans le programme. Ce n'est malheureusement pas toujours le cas:

```
let rec fact x =
 if x = 0 then 1 else x * fact x - 1;;
```

`x * fact x - 1` est interprété comme `x * (fact x) - 1` et non comme `x * fact (x - 1)`, et le programme boucle.

Quelle est la différence entre fun et function ?

Les fonctions s'introduisent normalement par le mot-clé `function`. Chaque paramètre de la fonction est introduit par une nouvelle construction `function`. Par exemple

```
function x -> function y -> ...
```

définit une fonction à deux paramètres `x` et `y`. Les fonctions qui procèdent par filtrage s'introduisent également par le mot-clé `function`.

Le mot-clé `fun` introduit des fonctions curryfiées (à plusieurs arguments successifs). Par exemple

```
fun x y -> ...
```

introduit une fonction à deux paramètres `x` et `y` équivalente à

```
function x -> function y -> ...
```

Difficulté: lors d'un filtrage introduit par `fun`, il faut parenthéser les filtres dont les constructeurs ont des arguments (constructeurs ``fonctionnels''). En effet, si `C` est un constructeur à un argument, alors

```
fun C x -> ...
```

est interprété comme une fonction à deux arguments ``C'' et ``x''. Il faut noter l'application par des parenthèses pour lever l'ambiguïté:

```
#type compteur = Compteur of int;;
Type compteur defined.
#let f = fun Compteur c -> c + 1;;
Entrée interactive:
>let f = fun Compteur c -> c + 1;;
>           ^^^^^^^^
Le constructeur Compteur exige un argument.
#let f = fun (Compteur c) -> c + 1;;
f : compteur -> int = <fun>
```

Notez qu'avec l'utilisation du mot-clé `function`, ce problème disparaît.

```
#let f = function Compteur c -> c + 1;;
f : compteur -> int = <fun>
```

Sémantique

Comment définir une fonction récursive ?

Il faut explicitement signaler la récursion: on utilise ``let rec'' au lieu de ``let''. Par exemple, ou, ou encore.

Mon programme ne prend pas le bon cas de filtrage ?

Il n'y a sans doute pas d'erreur dans le compilateur! Cela provient la plupart du temps d'une erreur de parenthésage dans un filtrage emboîté dans un autre filtrage.

Comment faire des filtrages emboîtés ?

Il faut absolument parenthéser un filtrage qui se trouve à l'intérieur d'un autre filtrage. En effet le filtrage le plus interne ``capture'' toutes les clauses de filtrage qui le suivent. Par exemple:

```
let f = function
 | 0 -> match ... with | a -> ... | b -> ...
 | 1 -> ...
 | 2 -> ...;;
```

est compris comme

```
let f = function
 | 0 ->
     match ... with
     | a -> ...
     | b -> ...
     | 1 -> ...
     | 2 -> ...;;
```

Cette erreur peut se produire dans tous les cas de constructions qui impliquent du filtrage: ``function'', ``match .. with'' et ``try ... with''. En général on parenthèse le filtrage fautif à l'aide des mots-clefs `begin` et `end`. On écrira donc:

```
let f = function
 | 0 ->
     begin match ... with
     | a -> ...
     | b -> ...
     end
 | 1 -> ...
 | 2 -> ...;;
```

Ma fonction n'est jamais appliquée ?

Cela provient la plupart du temps d'un argument oublié: du fait du caractère fonctionnel de Caml, il n'y a pas d'erreur lorsqu'on demande l'évaluation d'une fonction sans tous ses arguments: dans ce cas une valeur fonctionnelle est retournée, mais la fonction n'est évidemment pas appelée. Exemple caricatural: `print_newline` sans argument ne provoque pas d'erreur mais ne fait rien.

```
#print_newline;;
- : unit -> unit

#print_newline ();;

- : unit = ()
```

Mon tableau est modifié sans raison ?

En général cela provient d'un partage physique passé inaperçu. En Caml, il n'y a jamais de recopie implicite de tableaux. Si donc l'on donne deux noms à un tableau, toute modification citant l'un des noms se répercute sur l'autre:

```
(* Définition de v *)
#let v = make_vect 3 0;;
v : int vect = [|0; 0; 0|]
(* Le tableau w est physiquement le même que v *)
#let w = v;;
w : int vect = [|0; 0; 0|]
#w.(0) <- 4;;
- : unit = ()
(* v est modifié par la modification de w *)
#v;;
- : int vect = [|4; 0; 0|]
```

Le partage physique insoupçonné apparaît également au niveau des éléments stockés dans les cases d'un tableau: quand ces cases elles-mêmes contiennent des vecteurs, le partage de ces vecteurs implique qu'une modification de l'un se répercute sur les autres (plus de détails).

Comment faire des tableaux à deux dimensions ?

La seule manière de procéder est de créer un tableau de tableaux (les tableaux de Caml sont unidimentionnels: ce ne sont que des vecteurs au sens mathématique).

Si l'on écrit naïvement la création d'un tableau à plusieurs dimensions, le résultat n'est pas celui attendu, car on crée sans le vouloir un partage physique des lignes du tableau:

```
#let matrice_2_3 = make_vect 2 (make_vect 3 0);;
matrice_2_3 : int vect vect = [|[|0; 0; 0|]; [|0; 0; 0|]|]
#matrice_2_3.(0).(0) <- 1;;
- : unit = ()
#matrice_2_3;;
- : int vect vect = [|[|1; 0; 0|]; [|1; 0; 0|]|]
```

En effet l'allocation d'un tableau consiste à calculer la valeur d'initialisation et à la mettre dans chaque case du tableau (c'est pourquoi la ligne de la matrice qui est allouée par l'expression `(make_vect 3 0)` est unique et physiquement partagée par toutes les cases du tableau `matrice_2_3`).

Solution: utiliser la primitive `make_matrix`, qui fabrique directement une matrice dont les cases sont remplies par la valeur d'initialisation fournie. L'autre solution consiste à écrire le programme qui allouera explicitement une nouvelle ligne pour chaque ligne du tableau. Par exemple:

```
let matrice_n_m n m init =
 let result = make_vect n (make_vect m init) in
 for i = 1 to n - 1 do
  result.(i) <- make_vect m init
 done;
 result;;
matrice_n_m : int -> int -> 'a -> 'a vect vect = <fun>
```

De même, la fonction `copy_vect` ne donne pas le résultat attendu pour des matrices: il faut écrire une fonction de copie qui copie explicitement le contenu de toutes les lignes de la matrice:

```
let copy_matrix m =
 let l = vect_length m in
 if l = 0 then m else
 let result = make_vect l m.(0) in
 for i = 1 to l - 1 do
  let coli = copy_vect m.(i) in
  result.(i) <- coli
 done;
 result;;
```

Qu'est-ce qu'un type abstrait ?

Un type abstrait est un type dont les constructeurs (s'il s'agit d'un type somme) ou les labels (s'il s'agit d'un type produit) ne sont pas exportés par le module qui le définit. Cela permet de découpler l'utilisation et la définition du type: on peut ainsi changer tout ou partie de l'implémentation de ce type ainsi que des opérations qui lui sont associées, sans que les utlisateurs du type abstrait ne s'en aperçoivent. Cela permet ainsi d'optimiser une bibliothèque sans en déranger les utilisateurs.
Exemple, on implémente d'abord les piles avec des listes:

```
type 'a stack == 'a list;;
let new_stack init = ref [];;
let push x s = s := x :: !s;;
```

Puis avec une structure plus complexe, utilisant des vecteurs et réallouant dynamiquement la pile en cas de débordement:

```
type 'a stack = {mutable Contenu : 'a vect; mutable Pointeur : int};;
let new_stack init =
 {Contenu = make_vect 100 init; Pointeur = 0};;
let push x s =
 s.Contenu.(s.Pointeur) <- x;
 s.Pointeur <- s.Pointeur + 1;
 if s.Pointeur = vect_length (s.Contenu) then
   begin
    let ancien = s.Contenu in
    s.Contenu <- make_vect (2 * vect_length ancien)
                           ancien.(0);
    blit_vect ancien 0 s.Contenu 0 (vect_length ancien)
   end;;
```

Comment imprimer ?

Les fonctions d'impressions pour chaque type de base se nomme print_``nom du type''. Par exemple print_int, print_float, print_string.

Une autre méthode consiste à appeler la primitive `printf` du module printf.
La fonction `printf` prend en premier argument une chaîne de caractères (le ``format''), dans laquelle on indique le type des arguments à imprimer. Conventionnellement chaque argument à imprimer est représenté dans le format par un symbole % suivi de son type symbolique. Ainsi ``%s'' désignera un argument chaîne et ``%d'' un argument entier:

```
#printf__printf "Le nombre %s est %d" "un" 1;;
Le nombre un est 1- : unit = ()
```

Il existe aussi un module d'impression enjolivée à l'aide de boîtes d'impression et de coupures de lignes, le module format (une première approche de format est décrite ici).

Pourquoi certaines sorties disparaissent-elles ?

Pour accélérer l'impression, les caractères imprimés par les fonctions d'impression ne sont pas immédiatement sortis sur le terminal (ou le fichier), mais laissés en attente dans un *tempon de sortie*. Ce tempon est vidé automatiquement chaque fois qu'il est plein, ou sur un ordre explicite du programme (fonction `flush : out_channel -> unit`), ou encore à la fin de l'exécution si un ordre de terminaison explicite du programme est exécuté (`exit 0`).

Ce comportement, habituel dans les langages de programmation, peut occasionner la perte des dernières impressions du programme qui sont laissées pendantes dans le tempon de sortie et n'apparaissent pas au terminal ou dans le fichier de sortie.

Ce phénomène n'apparaît pas lorsqu'on utilise le système interactif parce que le système interactif vide systématiquement le tampon de sortie à la fin de chaque évaluation, avant de redonner la main à l'utilisateur. Il apparaît cependant dès que l'on écrit des programmes indépendants (exécutables générés par le compilateur Caml).

Pour régler ce problème, il suffit donc de terminer son programme par l'évaluation de la fonction `exit`:

```
exit 0;;
```

ou de donner explicitement l'ordre de vider le tempon de sortie du canal sur lequel on écrit. Par exemple pour le terminal, on écrit:

```
flush stdout;;
```

Pourquoi certaines sorties ne sont-elles pas dans le bon ordre ?

Si vous utilisez le module `format`, vous ne devez pas mélanger les ordres d'impression de `format` avec les ordres d'impression habituels, car les ordres d'impression de `format` sont retardées pour faire la mise en page et les coupures de lignes, alors que les impressions habituelles ne sont pas soumises à ce retard. Ceci explique la différence de comportement entre:

```
# print_string "before";
  print_string "MIDDLE";
  print_string "after";;
beforeMIDDLEafter- : unit = ()
```

et

```
# print_string "before";
  Format.print_string "MIDDLE";
  print_string "after";;
beforeafterMIDDLE- : unit = ()
```

Pour éviter ce problème, il ne faut pas mélanger les ordres d'impression de `format` et les ordres de plus bas niveau; c'est pourquoi on a l'habitude d'ouvrir le module `format` lorsqu'on utilise ses fonctions.

N.B: Les impressions de `format` sont vidées automatiquement à chaque sortie du système interactif, ou bien à l'aide d'un l'appel à la fonction `print_newline` qui émet un retour-charriot avant de vidée la queue d'impression de `format`.

Comment faire des entrées-sorties ?

Il faut utiliser les canaux d'entrées-sorties qui mettent Caml en relation avec le monde extérieur. On doit créer un canal avant de l'utiliser, pour cela on doit l'``ouvrir'', c'est-à-dire relier le canal à une ressource matérielle de la machine. Après usage on doit fermer les canaux.

- Les canaux d'entrées sont du type `in_channel`, on les ouvre en appliquant `open_in` à un nom de fichier. On ferme un canal d'entrée avec la primitive `close_in`. On lit une chaîne avec `input` ou `input_line`.
- Les canaux de sortie sont du type `out_channel`, ouverts avec `open_out`, fermés avec `close_out`, et on y écrit avec `output_string`, `output_char` ou `output`.

Trois canaux sont toujours ouverts par défaut:

- `std_in`, le canal d'entrée standard, généralement relié au clavier de l'ordinateur. On lit une ligne au clavier à l'aide de `input_line std_in;;`.
- `std_out`, la sortie standard, généralement reliée à l'écran. On écrit une ligne s au terminal à l'aide de `output_string std_out s;;`.
  Autres opérations sur std_out: `print_string`, `print_newline`, `print_endline`.
- `std_err`, la sortie d'erreur standard, également reliée à l'écran.
  Autres opérations sur `std_err`: `prerr_string`, `prerr_endline`.

Autres opérations:

- On peut écrire sur disque puis relire n'importe quelle valeur structurée à l'aide des primitives `output_value` et `input_value` et ces opérations préservent le partage présent dans la valeur manipulée. Attention à utiliser des fichiers en mode binaire.
- Voir le module io.

Pour accélérer l'exécution toutes les entrées-sorties de Caml sont faites en mémoire dans un *tampon* spécifique à chaque canal. Dans le cas des programmes indépendants (produits par le compilateur indépendant et son éditeur de liens), il ne faut pas oublier de vider les tempons de sortie des canaux utilisés par le programme en utilisant la fonction `flush`. À défaut, les derniers caractères écrits sur les canaux correspondants seraient perdus.
L'ordre explicite de fin de programme `exit` vide également automatiquement les tampons de sortie. Il vous suffit donc d'ajouter l'expression `exit 0;;` à la fin de votre programme pour assurer que toutes les opérations d'entrées-sorties sont complètement terminées.

Comment obtenir des nombres au hasard ?

Il faut utiliser des fonctions prédéfinies de la bibliothèque standard `random`. Pour obtenir un entier on utilise la fonction `int` et pour obtenir un flottant la fonction `float` du module `random`.

Attention au caractère procédural de ces fonctions: à chaque invocation ces ``fonctions'' renvoient un résultat différent. Ainsi la fonction `make_couple` définie par:

```
#let make_couple () = [| random__int 1000; random__int 1000 |];;
make_couple : unit -> int vect = <fun>
```

hérite de ce caractère, et chaque appel à `make_couple` renverra un vecteur différent.
Inversement, la définition

```
#let make_couple = [| random__int 1000; random__int 1000 |];;
make_couple : int vect = [|281; 407|]
```

crée un vecteur *constant* de deux nombres pris au hasard, mais choisis une fois pour toutes.

Contacter l'auteur Pierre.Weis@inria.fr
