Application de fonctions

Considérons la fonction

```
# let puissance_8 n =
      let n2 = n * n in
      let n4 = n2 * n2 in
      n4 * n4;;
val puissance_8 : int -> int = <fun>
```

Pour appliquer `puissance_8` à l'argument `2`, on écrit

```
# (puissance_8 2);;
- : int = 256
```

On notera l'utilisation inhabituelle des parenthèses.

Il est possible d'appeler une fonction dans une autre :

```
# let puissance_16 n =
      let n8 = (puissance_8 n) in
      n8 * n8;;
val puissance_16 : int -> int = <fun>
```

Fonctions locales

Une fonction locale est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let concat u =
      let aux u =
          u ^ "a" ^ u
      in                                  # (concat "cd");;
      (aux u) ^ (aux ("b" ^ u));;         - : string = "cdacdbcdabcd"
val concat : string -> string = <fun>
```

Il est également possible de réaliser des définitions de fonctions (locales) simultanées :

```
# let f x =
      let g y =
          y + 1 = x
      and h x =
          2 * x
      in
      (g (h x));;
val f : int -> bool = <fun>
```

Expressions conditionnelles

Une expression conditionnelle est une expression de la forme

`if COND then EXP1 else EXP2`

où `COND` est une expression dont la valeur est de type `bool` et `EXP1` et `EXP2` sont deux expressions dont les valeurs sont **toutes deux d'un même type**.

```
# if (3 >= 2) || ("aab" <= "aa") then
      "ABC"
   else
       "CDE";;
- : string = "ABC"
```

Toute expression conditionnelle **possède une valeur** :

1. lorsque `COND` s'évalue en `true`, l'expression conditionnelle à pour valeur `EXP1` ;
2. lorsque `COND` s'évalue en `false`, l'expression conditionnelle à pour valeur `EXP2`.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'imbriquer les expressions conditionnelles.

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur Caml la comprend sans ambiguïté.

Il n'a pas besoin de marqueur de fin (comme le } de certains langages).

Dessiner l'arbre syntaxique.

Demi-expressions conditionnelles

Une demi-expression conditionnelle est une expression de la forme

`if COND then EXP`

où `COND` est une expression dont la valeur est de type `bool` et `EXP` est une expression.

Le système complète implicitement et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

`if COND then EXP else ()`

En conséquence, la valeur de `EXP` doit être de type `unit`.

```
# let f x =
      if x >= 9 then
          ();;
val f : int -> unit = <fun>
```

est équivalent à

```
# let f x =
      if x >= 9 then
          ()
      else
          ();;
val f : int -> unit = <fun>
```

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP;;
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où `E1`, ..., `En` et `S` sont des types.

Plus précisément,

- pour tout `1` $\leqslant$ `i` $\leqslant$ `n`, `Ei` est le type attendu du `i`$^{\text{e}}$ argument de `F` ;
- `S` est le type de retour de `F`.

`E1 -> ... -> En -> S` est un type particulier, dit type fonction.

Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =
      (int_of_char y) + x;;
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le constructeur de types `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Cependant, lorsque l'on en met pas, la convention stipule que le parenthésage est fait **de droite à gauche**.

Ainsi, le type `int -> char -> int` est équivalent au type dénoté par l'expression totalement parenthésée `(int -> (char -> int))`.

Applications partielles de fonctions

Considérons une fonction `f` à deux paramètres de types `E1` et `E2` et à type de retour `S`.

Deux faits évidents :

1. la fonction `f` est de type `E1 -> E2 -> S` ;
2. si `e1` et `e2` sont des valeurs de types respectifs `E1` et `E2`, l'application de `f` à `e1` et `e2` par `(f e1 e2)` renvoie une valeur de type `S`.

Il est cependant possible d'**appliquer `f` à seulement `e1`** par `(f e1)`. On obtient ainsi une valeur de type `E2 -> S` qui est donc une **fonction**.

La fonction `(f e1)` est ainsi une fonction qui se comporte comme `f` lorsque son 1er paramètre est fixé à la valeur `e1`.

On dit que `(f e1)` est une application partielle de `f` à des arguments.

Applications partielles de fonctions

Il y a donc équivalence entre l'appel

```
(f e1 e2)
```

et l'appel

```
((f e1) e2).
```

Plus généralement, si `f` est une fonction de type

```
E1 -> ... -> En -> S
```

et `e1`, . . ., `ek` sont des valeurs de types respectifs `E1`, . . ., `Ek` avec $k \leqslant n$, l'application partielle

```
(f e1 ... ek)
```

est une fonction de type

```
Ek+1 -> ... -> En -> S
```

Applications partielles de fonctions — exemple

```
# let distr a b c =
      a * b + a * c;;
val distr : int -> int
    -> int -> int = <fun>
```

Cette fonction représente $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

```
# let distr_3 = (distr 3);;
val distr_3 : int -> int
    -> int = <fun>
```

Cette fonction représente $f_3 : \mathbb{Z}^2 \to \mathbb{Z}$ vérifiant $(b, c) \mapsto 3b + 3c$.

```
# let distr_3_5 = (distr_3 5);;
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente $f_{3,5} : \mathbb{Z} \to \mathbb{Z}$ vérifiant $c \mapsto 15 + 3c$.

La fonction `distr_3_5` peut aussi être définie directement par l'une ou l'autre des deux manières suivantes :

```
# let distr_3_5 c = (distr 3 5 c);;
val distr_3_5 : int -> int = <fun>
```

```
# let distr_3_5 = (distr 3 5);;
val distr_3_5 : int -> int = <fun>
```

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une fonction anonyme.

La construction syntaxique

`fun P1 ... Pn -> EXP`

où `P1`, ..., `Pn` sont des paramètres et `EXP` est une expression permet de définir une fonction anonyme.

```
# ((fun a b -> (a + b) * a) 4 3);;
- : int = 28
```

Ceci définit une fonction anonyme appliquée d'emblée aux arguments `4` et `3`.

```
# let produit k =
      fun x -> x * k;;
val produit : int -> int -> int
    = <fun>
```

`(produit 10)` renvoie une fonction de type `int -> int` qui multiplie par `10` son argument.

Deviner le type d'une fonction

Le système de typage de Caml agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
# let mystere x y z =
      if x && (y 1) then
          z;;
```

Cette fonction est de type

`bool -> (int -> bool) -> unit -> unit`.

```
# let etrange x y =
      "a" ^ ((y 1) ((x 'a') + 1)) ^ "b";;
```

Cette fonction est de type

`(char -> int) -> (int -> int -> string) -> string`.

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
# let mystere x y z t =
      1. +. (x ((y (not z) (t - 1)) ^ "ab") y);;
```

Cette fonction est de type

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

```
# let etrange y =
      ((fun x -> x + 1) 2) + (y (string_of_int 3));;
```

Cette fonction est de type

```
(string -> int) -> int.
```

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des fonctions d'entrée/sortie qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée/sortie utilisent le type `unit` et son unique valeur `()`.

Rappel : l'utilisation d'entrées/sorties fait que l'on **sort du paradigme de programmation fonctionnelle pure** car elles produisent un effet (affichage ou bien attente d'une action de l'utilisateur).

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, toute fonction d'écriture renvoie `()`.

Les fonctions d'écriture principales sont

```
val print_int : int -> unit
val print_float : float -> unit
val print_char : char -> unit
val print_string : string -> unit
val print_newline : unit -> unit
```

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur entrée** par l'utilisateur, est leur seul intérêt.

En pratique, comme toute fonction possède au moins un paramètre, par convention, toute fonction de lecture accepte `()` en entrée. Une fonction qui ne possède pas de paramètre est une constante (ce qui n'est pas la nature d'une fonction de lecture).

Les fonctions de lecture principales sont

```
val read_int : unit -> int
val read_float : unit -> float
val read_line : unit -> string
```

Séquences

L'utilisation des fonctions d'entrée/sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'opérateur de séquence `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ...; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

La valeur de `E1; E2; ...; En` est ainsi celle de `En`.

De ce fait, `E1`, ..., `En-1` doivent être de type `unit`.

Séquences — exemples

```
# let add x y =
      (print_string "Appel de add");
      (print_int x);
      (print_int y);
      x + y;;
val add : int -> int -> int = <fun>
```

```
# let test_div n =
      if n mod 2 = 0 then
          (print_string "pair\n");
      if n mod 3 = 0 then
          (print_string "multiple de 3\n");;
val test_div : int -> unit = <fun>
```

Séquences et blocs

Considérons la fonction

```
# let div_decr x =
      if x mod 2 = 0 then
          (print_string "pair");
          x / 2
      else
           (print_string "impair");
          x - 1;;
Error: Syntax error
```

L'opérateur de séquence est moins prioritaire que la conditionnelle. Ainsi, cette fonction est comprise en

```
# let div_decr x =
      if x mod 2 = 0 then
          (print_string "pair");
      x / 2
      else
          (print_string "impair");
      x - 1;;
```

Ceci explique l'**erreur de syntaxe**.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de bloc. Un bloc est une expression de la forme

`begin EXP end`

où `EXP` est une expression. La valeur de `begin EXP end` est celle de `EXP`.

La fonction précédente devient ainsi correcte en écrivant

```
# let div_decr x =
      if x mod 2 = 0 then begin
          (print_string "pair");
          x / 2
      end
      else begin
          (print_string "impair");
          x - 1
      end;;
val div_decr : int -> int = <fun>
```

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur Caml, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour obtenir un exécutable.

Pour cela, on écrit le programme dans un fichier d'extension `.ml`. On le compile par la commande

```
ocamlc -o Prog Prog.ml
```

Ceci invoque le **compilateur bytecode**.

Le **compilateur natif** produit des exécutables souvent plus performants. On y fait appel par la commande

```
ocamlopt -o Prog Prog.ml
```

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

1. on construit un fichier objet (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

   ```
   ocamlc -c F.ml
   ```

   ou bien

   ```
   ocamlopt -c F.ml
   ```

2. on appelle l'éditeur de liens par la commande

   ```
   ocamlc -o Prog F1.cmo ... Fn.cmo
   ```

   ou bien

   ```
   ocamlopt -o Prog F1.cmx ... Fn.cmx
   ```

   où les `F1.cm*`, ..., `Fn.cm*` sont les fichiers objet du projet.

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

   `open A;;`

2. Utiliser `f` aux endroits désirés dans `B.ml`.

   Dans `B.ml`, le nom de `f` devient

   `A.f`

Exemple :

```
(* A.ml *)

let double x =
2 * x;;
```

```
(* B.ml *)

open A;;

let quadruple x =
2 * (A.double x);;
```

Dans ce projet, `B.ml` inclut `A.ml`. Ainsi, la fonction `double` de `A.ml` est visible dans `B.ml` par l'identificateur `A.double`.

Espaces de noms

Un espace de nom est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple :

```
(* A.ml *)
let calcul x =
...
```

```
(* B.ml *)
let calcul x y =
...
```

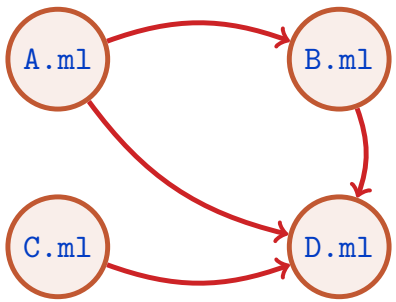
```
(* C.ml *)
open A;;
open B;;
...
(A.calcul 1)
...
(B.calcul 'a' 2)
...
```

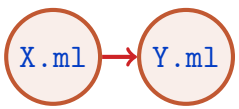
Dans ce projet, deux fonctions nommées `calcul` sont définies. Leur nom absolu n'est en revanche pas le même (`A.calcul` et `B.calcul`). Il n'y a ainsi aucune ambiguïté dans `C.ml` qui inclut les deux fichiers précédents.

Ordre de compilation

À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

Considérons le graphe d'inclusions suivant :



Toute flèche 

signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

Il faut donc compiler le projet dans l'ordre dicté par un **tri topologique** du graphe dual du graphe d'inclusions du projet.

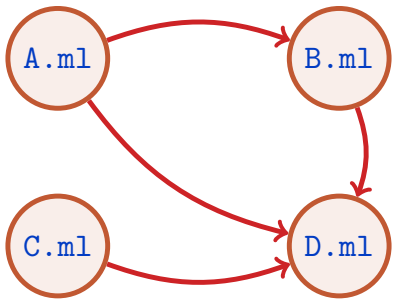
Dans cet exemple, on a les trois ordres suivants possibles :

- `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Reprenons le graphe d'inclusions



La commande `ocamldep *.ml` affiche

```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: B.cmo
B.cmx: B.cmx
C.cmo: D.cmo
C.cmx: D.cmx
D.cmo:
D.cmx:
```

Il est ainsi possible d'écrire des `Makefile` génériques en appelant, à l'intérieur, `ocamldep`.