Impression de valeurs et manipulation de fichiers en OCaml

D'après un chapitre du livre *Apprentissage de la programmation en Ocaml* de C. Dubois et V. Ménisier-Morain, Hermès, 2004.

1 Impression à l'écran

Le langage OCaml ne fournit pas de fonction `print` générale permettant d'imprimer n'importe quelle valeur. En effet, imprimer une valeur nécessite d'utiliser un algorithme qui diffère selon la nature, donc le type, de la valeur à imprimer. OCaml fournit donc un certain nombre de fonctions d'impression prédéfinies. Ainsi pour imprimer un entier, on utilisera la fonction `print_int` de type `int -> unit`, pour un flottant, la fonction `print_float` de type `float -> unit`, pour une chaîne de caractères, la fonction `print_string` de type `string -> unit` et pour un caractère la `print_char` de type `char -> unit`. C'est au programmeur de réaliser les fonctions qui impriment les valeurs plus complexes.

Toutes ces fonctions ont `unit` comme type de résultat. Ceci est caractéristique des opérations qui agissent par effet de bord, comme l'affectation, la modification d'un champ mutable. En effet l'affichage ne produit aucune valeur, son effet est de modifier l'aspect de l'écran en affichant une valeur supplémentaire, comme ci-dessous:

```
#print_string "Bonjour, c'est encore moi !";;
Bonjour, c'est encore moi !- : unit = ()
```

La réponse de OCaml est un peu confuse, détaillons-la : on trouve d'abord `Bonjour, c'est encore moi !`, la chaîne de caractères que l'on a voulu imprimer, puis immédiatement accolée, la réponse habituelle de OCaml donnant le type, ici `unit`, et la valeur, ici `()`, de la phrase tapée.

En ajoutant le caractère de retour à la ligne[1], `'\n'`, à la chaîne à afficher, on y verra un peu plus clair comme dans l'exemple suivant.

```
#print_string "Bonjour, c'est encore moi !\n";;
Bonjour, c'est encore moi !
- : unit = ()
```

On pourra aussi avoir recours à la fonction `print_newline` de type `unit -> unit`. qui prend donc en argument la valeur `()`, seule valeur du type `unit`, et dont l'unique effet consiste à imprimer un retour à la ligne.

```
#print_newline ();;

- : unit = ()
```

On peut également utiliser la fonction `printf`[2] du module `Printf` de la bibliothèque standard, qui permet d'afficher des valeurs selon un format donné, avec une grande souplesse. Le format peut être vu comme une chaîne de caractères *à trous* qui seront remplis par les valeurs des expressions qui suivent. Chaque *trou* comporte une indication de type :

- `%c` pour un caractère,
- `%s` pour une chaîne de caractères,
- `%i` pour un entier,

[1] *retour chariot* dans le vocabulaire des machines à écrire
[2] Cette fonction est proche de la fonction C de même nom

- `%f` pour un flottant,
- `%.3f` pour un flottant dont on ne veut afficher que 3 chiffres après la virgule,
- `%5d` pour un affichage en écriture décimale sur 5 caractères (on complétera par des blancs si nécessaires) etc . (Consulter le manuel utilisateur pour les détails)

```
# Printf.printf "%.3f\n" 14.000013;;
14.000
- : unit = ()
# Printf.printf "%5d\n" 14;;
   14
- : unit = ()
# Printf.printf "la valeur du code ascii de %c est %i\n"
    'a' (int_of_char 'a');;
la valeur du code ascii de a est 97
- : unit = ()
```

Dans le dernier exemple, le premier *trou* marqué par `%c` est rempli avec la valeur de `'a'`, le second *trou* marqué par `%i` est rempli avec la valeur de `int_of_char 'a'`. Cette fonction permet de mélanger textes et valeurs en un seul ordre d'impression.

Enfin, on remarquera que cette fonction prend un nombre variable d'arguments : le format qui, lui, est obligatoire, et zéro ou plusieurs autres arguments (de préférence autant qu'il y a de trous dans le format). OCaml fait un certain nombre de vérifications en particulier sur le nombre de trous et arguments, et la compatibilité entre les types des trous et des arguments correspondants. La fonction `Printf.printf` relève pour nous des primitives magiques, impossibles à écrire dans la partie du langage présenté dans cet ouvrage.

2 Séquences

La notion de séquence permet d'enchaîner l'évaluation de deux expressions. Intuitivement, elle permet donc d'exprimer l'idée *fais ceci d'abord et cela ensuite*.

Une séquence est une expression de la forme suivante : `expr_1 ; expr_2` où `expr_1` et `expr_2` sont des expressions quelconques
Le type de `expr_1 ; expr_2` est le type de `expr_2`, néanmoins `expr_1` doit être une expression bien typée (de préférence de type `unit`).
Évaluer `expr_1 ; expr_2` signifie que l'on évalue `expr_1` puis `expr_2`. La valeur de la séquence `expr_1 ; expr_2` est la valeur de `expr_2`

On ne se servira de la séquence que pour enchaîner successivement des écritures à l'écran et dans un fichier.

3 Fichiers et opérations de base

Nous terminons ce chapitre par la présentation des fichiers, structures de données fondamentales en informatique, et qui relèvent plutôt de la programmation impérative.

Un fichier OCaml contient des informations rangées séquentiellement, c'est-à-dire les unes derrière les autres. En ce sens, un fichier peut être vue de manière abstraite comme une liste. À la différence d'une liste, un fichier est une donnée *persistante*, stockée sur disque et qui donc existe en dehors d'une session OCaml. En revanche, les listes tout comme les tableaux sont des données qui n'existent qu'en mémoire et le temps d'exécution des programmes. D'autre part, la manipulation d'un fichier est complètement différente de celle d'une liste. A la différence d'une liste, un fichier peut être assimilé à une donnée mutable (dont la valeur peut changer). Détaillons maintenant les opérations sur les fichiers, elles font partie de la famille des opérations d'entrées-sorties.

3.1 Ouverture d'un fichier

Pour manipuler un fichier, il faut tout d'abord l'*ouvrir*. On ouvre un fichier *en lecture* si on désire consulter les informations qu'il contient, sans le modifier; on utilise pour cela la fonction prédéfinie `open_in`. Inversement, on ouvre un fichier *en écriture* si on désire modifier son contenu ou le créer, la fonction prédéfinie correspondante est alors `open_out`.

Traditionnellement, un fichier doit être vide ou est considéré comme tel lorsqu'il est ouvert en écriture. Dans ce dernier cas, le contenu initial du fichier est *écrasé*.

L'opération d'ouverture en lecture ou écriture consiste à associer l'information contenue dans un fichier sur disque, connu sous son *nom physique*, dans l'exemple ci-dessous `"mon_fichier.txt"` à un nom, `fic` dans l'exemple ci-dessous, exactement de la même façon que l'on nomme une valeur avec la construction `let` ou `let ...in`. Cet identificateur `fic` est appelé *nom logique* du fichier. C'est sous ce nom que le fichier sera manipulé dans le programme.

```
# let fic = open_in "mon_fichier.txt";;
val fic : in_channel = <abstr>
# let fic2 = open_out "résultats";;
val fic2 : out_channel = <abstr>
```

Examinons les réponses de OCaml aux opérations d'ouverture de fichiers précédentes. Les types `out_channel` et `in_channel` sont les types des noms logiques des fichiers ouverts respectivement en lecture et en écriture. Les valeurs de ces noms logiques sont des valeurs abstraites, notées `<abstr>`.

Si le fichier que l'on cherche à ouvrir en lecture n'existe pas, tel le fichier `f1` ci-dessous, l'opération d'ouverture en lecture échoue. Les demandes d'ouverture d'un fichier en lecture ou en écriture doivent être compatibles avec les droits d'accès donnés par le système d'exploitation. Par exemple, si seul l'accès en lecture est donné au fichier `f2`, l'ouverture en écriture lèvera une exception :

```
# open_in "f1";;
Exception: Sys_error "f1: No such file or directory".
# open_out "f2";;
- : out_channel = <abstr>
```

3.2 Lecture, écriture dans un fichier

Les fichiers auxquels nous nous intéressons dans ce chapitre sont des fichiers de texte, organisés en lignes. Lire ou consulter un fichier consistera à lire une ligne à la fois, séquentiellement, du début à la fin. Écrire dans un fichier consistera à modifier le contenu du fichier en ajoutant une ligne. Vu de OCaml, une ligne est une chaîne de caractères, donc de type `string`. Il existe d'autres opérations possibles sur un fichier mais on ne s'intéressera ici qu'aux opérations de lecture et d'écriture.

On lit une chaîne de caractères avec la fonction prédéfinie `input_line` de type `in_channel -> string` et on l'écrira avec la fonction prédéfinie `output_string` de type `out_channel -> string -> unit`. On remarquera le type `unit` du résultat de l'opération d'écriture, ceci souligne que l'opération d'écriture agit par effet de bord sur le fichier en le modifiant. La fonction de lecture, même si son résultat n'est pas de type `unit`, agit elle aussi en quelque sorte par effet de bord puisque chaque lecture provoquera la modification du repère de lecture dans le fichier.

De manière générale, on lit ou écrit des chaînes de caractères successives en appelant les fonctions `input_line` et `output_string` autant de fois qu'il y a de chaînes de caractères.

3.3 Fermeture d'un fichier

Quand on a effectué la ou les lecture/écriture(s) que l'on voulait faire sur un fichier, alors on le ferme avec la fonction `close_in` de type `in_channel -> unit` s'il avait été ouvert en lecture, ou `close_out` de type `out_channel -> unit` s'il avait été ouvert en écriture.

3.4 Une fonction de visualisation du contenu d'un fichier

On veut écrire une fonction `cat` qui permet de visualiser le contenu d'un fichier dont on donne le nom en argument, comme le fait la commande Unix du même nom. Lorsque l'on se trouve à la fin du fichier la fonction `input_line` échoue en levant l'exception `End_of_file`. On lit donc le fichier ligne

à ligne tant que c'est possible en imprimant aussitôt la chaîne lue et lorsque l'exception est levée le fichier est vide, il n'y a plus rien à imprimer et on ferme le fichier.

Ecrivons cette fonction récursivement. Le cas de base concerne le cas où l'on est à la fin du fichier et on ferme alors le fichier en lecture. Dans le cas général, on lit la ligne à afficher et on l'affiche puis on continue récursivement. L'appel récursif est le même que l'appel initial ce qui peut paraître surprenant mais on ne boucle pas car le repère de lecture dans le fichier a avancé, l'effet de bord nous assure que l'on se rapproche de la fin du fichier donc du cas de base de notre fonction.

```
#let cat nom_fichier =
  let f = open_in nom_fichier in
  let rec cat_rec () =
    try
      print_string (input_line f); print_newline (); cat_rec ();
    with End_of_file -> close_in f
  in cat_rec ();;
val cat : string -> unit = <fun>

#cat "résultats";;
azerty
123
toto
- : unit = ()
```