Informatique *Cours*
Programmation fonctionnelle avec OCaml - Traits impératifs Ipesup

Objectifs

- Découvrir les principaux traits impératifs d'OCaml : structures modifiables, boucles, exceptions et entrées-sorties.
- Savoir manipuler des références, des tableaux et gérer les effets de bord.
- Apprendre à lire, écrire et contrôler les erreurs dans des programmes interactifs.

1 Structures de données modifiables

L'un des principes essentiels de la programmation impérative est la manipulation (création, modification) de *structures de données modifiables*. OCaml en propose plusieurs types.

1.1 Enregistrements modifiables

On peut définir des produits nommés (*enregistrements*) dont certains champs sont modifiables grâce au mot-clé `mutable`.

Exemple

Le type suivant décrit un étudiant avec un identifiant et un âge modifiable :

```
type student = {id : int; mutable age : int}
```

La création d'un enregistrement modifiable se fait comme pour un enregistrement immuable :

```
let e = { id = 12134; age = 21 }
```

L'expression `e.age <- <expr>` modifie le champ `age`.
La fonction suivante incrémente l'âge d'un étudiant :

```
let birthday e = e.age <- e.age + 1
```

La bibliothèque standard définit le type prédéfini `'a ref`, qui représente un enregistrement polymorphe à champ unique modifiable :

```
type 'a ref = { mutable contents : 'a }
```

Ces enregistrements sont appelés *références* et permettent variables modifiables.

Pour les manipuler, OCaml fournit :

- la fonction `ref` pour créer une référence ;
- l'opérateur `!x` pour lire son contenu ;
- l'opérateur `x := <expr>` pour modifier sa valeur.

Exemple

```
# let x = ref 10 ;;
val x : int ref = {contents = 10}
# !x ;;
- : int = 10
```

Comparaison physique et structurelle

OCaml distingue deux types d'égalité :
- l'*égalité physique* `==`, qui compare les adresses mémoire ;
- l'*égalité structurelle* `=`, qui compare les contenus.

```
# let v1 = ref 5;;
val v1 : int ref = {contents = 5}
# let v2 = ref 5;;
val v2 : int ref = {contents = 5}
# v1 == v2;;
- : bool = false
# v1 = v2;;
- : bool = true
```

Ici, `v1` et `v2` contiennent la même valeur mais sont stockées à des adresses différentes.

Références polymorphes

Pour garantir la sûreté de typage, les références doivent avoir un type *clos* (sans variables de type libres).
La déclaration suivante provoque une erreur, car la liste vide est polymorphe :

```
let l = ref []
$ ocamlopt -o test test.ml
File "test.ml", line 1, characters 4-5:
1 | let l = ref []
    ^
Error: The type of this expression, '_weak1 list ref,
contains type variables that cannot be generalized
```

1.2 Tableaux

Une autre structure modifiable très utilisée est le *tableau*.
OCaml fournit le type prédéfini `'a array` et de nombreuses fonctions pour créer et manipuler des tableaux polymorphes.

La syntaxe `[| e1; e2; ...; en |]` crée un tableau dont tous les éléments sont du même type. Les indices vont de 0 à `n - 1` :

```
let t = [| 5; 1; 3; 4 |]
```

Le type de `t` est `int array`.
La fonction `Array.length` renvoie sa taille.

L'accès et la modification se font via `t.(i)` et `t.(i) <- <expr>`.
`Array.make n v` crée un tableau de `n` cases initialisées à `v`

```
# t.(1) <- 10;;
- : unit = ()
# t.(1);;
- : int = 10
# let t = Array.make 4 3.2 ;;
val t : float array = [|3.2; 3.2; 3.2; 3.2|]
```

Partage

Lorsqu'une donnée contient des références, il faut veiller à ne pas partager la même cellule mémoire entre plusieurs enregistrements.

```
type student = {id : int; age : int ref}
let v1 = { id = 100; age = ref 30 }
let v2 = { v1 with id = 200 } (* partage de v1.age *)
# v1.age := 40 ;;
- : unit = ()
# !(v2.age);;
- : int = 40
```

Ici, les deux enregistrements partagent la même référence : modifier `v1.age` affecte aussi `v2.age`.

Mode de passage

Les arguments de fonction sont passés *par valeur* : l'expression est d'abord évaluée, puis sa valeur copiée. Cependant, comme la plupart des valeurs sont des pointeurs vers des structures allouées, la « copie » concerne le pointeur lui-même. Ainsi, une fonction peut modifier une donnée passée en argument si celle-ci contient une référence :

```
# let t = ref 0;;
val t : int ref = {contents = 0}
# let f x = x := 100 ;;
val f : int ref -> unit = <fun>
# f t ;;
- : unit = ()
# !t ;;
- : int = 100
```

`Array.init n f` crée un tableau de `n` éléments initialisés par `f i` :

```
# let t = Array.init 5 (fun i -> 2 * i) ;;
val t : int array = [|0; 2; 4; 6; 8|]
```

OCaml fournit aussi des itérateurs comme `Array.exists` ou `Array.for_all`, ainsi que `Array.iter` et `Array.map` :

```
# Array.exists (fun x -> x < 4) t ;;
- : bool = true
# Array.for_all (fun x -> x mod 2 = 0) t ;;
- : bool = true
```

1.3 Matrices

Une matrice est un tableau de tableaux, de type `'a array array`.

```
let m = [| [| 1; 2 |]; [| 3; 4 |] |]
```

L'accès à la case (i, j) se fait avec `m.(i).(j)`.

On peut créer une matrice avec `Array.make_matrix n m v`, qui initialise toutes les cases `v`.

Attention

Attention à ne pas partager les lignes :

```
let m1 = Array.make_matrix 3 4 v (* lignes indpendantes *)
let m2 = Array.make 3 (Array.make 4 v) (* lignes partages *)
```

Dans `m2`, toute modification d'une case d'une ligne est répercutée sur les autres.

2 Boucles

La programmation impérative repose naturellement sur l'usage de boucles.
OCaml propose deux types principaux : les boucles `for` et les boucles `while`.

2.1 La boucle `for`

Une boucle `for` fait varier un *indice* dans un intervalle d'entiers et exécute, à chaque itération, un bloc d'instructions délimité par les mots-clés `do` ... `done`.

Exemple

La boucle suivante fait varier `i` de 1 à 5. À chaque tour, elle ajoute `i` à une référence `acc` et affiche la nouvelle valeur.

```
let acc = ref 0
let () =
  for i = 1 to 5 do
    acc := !acc + i;
    Printf.printf "%d " !acc;
  done
```

Compilation et exécution donnent :

```
$ ocamlopt -o test test.ml
$ ./test
1 3 6 10 15
```

Remarques importantes :

- L'indice `i` n'est jamais modifiable et n'existe qu'à l'intérieur de la boucle.
- On peut inverser le sens du parcours avec `downto` au lieu de `to`.
- Le corps de la boucle doit être de type `unit`.

— Les bornes de la boucle sont évaluées une seule fois avant l'exécution.

Exemple

Dans :

```
let () =
  for i = (print_string "*"; 0) to (print_string "."; 5) do
    Printf.printf "%d" i
  done
```

les bornes sont imprimées avant la boucle, et le programme affiche : `*.012345`.

2.2 La boucle `while`

Une boucle `while` s'écrit avec le mot-clé `while` suivi d'une condition booléenne et d'un corps de type `unit` entre `do` et `done`. Elle répète l'exécution du corps tant que la condition est vraie.

Exemple

```
let acc = ref 0
let () =
  while !acc < 100 do
    acc := !acc * 2 + 1;
    Printf.printf "%d " !acc;
  done
```

Déroulement :

— La condition `!acc < 100` est évaluée avant chaque itération.
— Si elle vaut `true`, le corps s'exécute, puis la condition est testée à nouveau.
— Si elle vaut `false`, la boucle s'arrête.

Contrairement à la boucle `for`, la boucle `while` peut ne jamais se terminer si la condition reste toujours vraie.

3 Exceptions

L'évaluation d'une expression peut parfois provoquer une erreur. Par exemple, l'opération `10 + 1 / 0` déclenche :

```
# 10 + 1 / 0 ;;
Exception: Division_by_zero.
```

Ces erreurs sont représentées en OCaml par des *exceptions*, un mécanisme permettant de signaler et gérer les erreurs survenant pendant l'exécution d'un programme.

Lorsqu'une erreur est détectée, l'exception correspondante est *levée*, ce qui interrompt immédiatement le calcul en cours.

```
let () =
  begin
    print_string "before\n";
    print_int (1/0);
    print_string "after\n"
  end
```

Ce programme affiche :

```
before
Fatal error: exception Division_by_zero
```

Les exceptions permettent donc de traiter les fonctions partiellement définies ou sujettes à des erreurs d'exécution : accès hors limites, conversions invalides, divisions par zéro, etc.

```
# let x = [| 1 |];;
val x : int array = [|1|]
# x.(2);;
Exception: Invalid_argument "index out of bounds".
# int_of_string "hello";;
Exception: Failure "int_of_string".
```

Toutes les exceptions appartiennent au type prédéfini `exn` :

```
# Division_by_zero;;
- : exn = Division_by_zero
```

Définir des exceptions

On peut définir de nouvelles exceptions avec la commande `exception` :

```
exception Fin
exception E of int
```

Une exception peut être levée explicitement à l'aide de l'expression `raise <constr>` :

```
# let f x = if x < 0 then raise (E x) else 10 / x ;;
val f : int -> int = <fun>
# f (-4);;
Exception: E (-4).
```

Une expression `raise <constr>` est toujours bien typée, quel que soit le contexte. Dans l'exemple ci-dessus, `raise (E x)` est donc considérée comme une expression de type `int`.
La bibliothèque standard fournit également la fonction `failwith`, équivalente à `raise (Failure m)` :

```
failwith "message" (* lve Failure "message" *)
```

Pour intercepter une exception et éviter qu'elle ne stoppe le programme, on utilise la construction `try ... with` :

```
try <expr>
with
| <motif1> -> <expr1>
...
| <motifk> -> <exprk>
```

Exemple

```
let test x y =
  try
    let q = x / y in
    Printf.printf "quotient = %d\n" q
  with
    Division_by_zero -> Printf.printf "error\n"
```

```
# test 3 0; print_string "suite\n";;
error
suite
- : unit = ()
```

Ici, l'exception `Division_by_zero` est capturée, ce qui permet au programme de continuer son exécution normalement après l'erreur.

4 Entrées-Sorties

OCaml propose de nombreux mécanismes et fonctions pour réaliser des opérations d'entrée-sortie. Cette section présente les plus courants.

Fonctions d'affichage

Pour afficher des valeurs de base, on utilise `print_int`, `print_float`, `print_string` et `print_char`. Le module `Printf` offre des fonctions plus générales comme `Printf.printf`, permettant un affichage formaté.
Un appel `Printf.printf fmt e1 ... en` prend en premier argument une chaîne de formatage `fmt`, puis les valeurs à afficher. La chaîne de formatage indique comment insérer ces valeurs dans le texte à l'aide de spécificateurs.
Les plus utilisés sont :

- `%d` pour un entier,
- `%s` pour une chaîne,
- `%f` pour un flottant,
- `%b` pour un booléen.

Exemple :

```
let x = 42
let r = x > 10
let () = Printf.printf "Un entier %d et un boolen %b" x r
```

Ce programme affiche : `Un entier 42 et un booléen true.`

Fonctions de saisie

OCaml fournit aussi plusieurs fonctions pour lire des données saisies au clavier.

- `read_line : unit -> string` lit une phrase et renvoie la chaîne saisie (sans retour chariot).
- `read_int : unit -> int` lit une chaîne et tente de la convertir en entier. Elle lève `Failure "int_of_string"` en cas d'échec.
- `read_float : unit -> float` lit et convertit une chaîne en nombre flottant. Elle lève `Failure "float_of_string"` si la conversion échoue.

Les canaux d'entrée-sortie

Comme sous Unix, OCaml représente les dispositifs d'entrée-sortie sous forme de *canaux*. Il existe deux types :

- `in_channel` pour la lecture ;
- `out_channel` pour l'écriture.

Les valeurs prédéfinies `stdin`, `stdout` et `stderr` représentent respectivement le clavier, la sortie standard (terminal) et la sortie d'erreur.

Les fichiers sont également manipulés via des canaux :

- `open_in : string -> in_channel` ouvre un fichier en lecture (exception `Sys_error` si le fichier n'existe pas) ;
- `open_out : string -> out_channel` ouvre un fichier en écriture.

Principales opérations :

- `input_line : in_channel -> string` lit une ligne depuis un canal (sans le retour chariot) et lève `End_of_file` en fin de fichier ;
- `output_string : out_channel -> string -> unit` écrit une chaîne dans un canal de sortie.

Il est essentiel de fermer les canaux après usage :

- `close_in` ferme un canal d'entrée ;
- `close_out` ferme un canal de sortie.

Cela libère les ressources système et garantit l'enregistrement correct des données.

Exemple

```
(* Lecture et criture avec des canaux *)
let () =
  let input = open_in "source.txt" in
  let output = open_out "copie.txt" in
  try
    while true do
      let ligne = input_line input in
      output_string output (ligne ^ "\n")
    done
  with End_of_file ->
    close_in input;
    close_out output
```

Ce programme lit le fichier `source.txt` ligne par ligne et en écrit une copie dans `copie.txt`.

Arguments d'un programme

Les programmes OCaml peuvent recevoir des arguments depuis la ligne de commande. Ils sont accessibles via le tableau `Sys.argv` du module `Sys`.

- `Sys.argv.(0)` contient le nom du programme exécuté ;
- `Sys.argv.(1)`, `Sys.argv.(2)`, etc. contiennent les arguments passés.

Exemple :

```
$ ./test 100
```

On obtient alors :

```
Sys.argv.(0) = "./test"
Sys.argv.(1) = "100"
```

Pour connaître le nombre d'arguments fournis, on utilise `Array.length Sys.argv`.