Programmation Fonctionnelle
Cours 05

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

October 19, 2014

Type Unit

Unit = la 0-uplet !

```
# print_string "Hello␣world\n";;
Hello world
- : unit = ()
```

- il a une seule valeur possible: ( )
  elle représente la n-uplet de 0 éléments,
- il est le type des expressions dont l'intérêt n'est pas dans leur valeur mais dans leur e□ets de bord
  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel
- on peut enchainer des expressions de type `unit` en utilisant `;`

```
# print_string "Hello␣";print_string "world\n";;
Hello world
- : unit = ()
```

Unit = la 0-uplet !

```
# print_string "Hello␣world\n";;
Hello world
- : unit = ()
```

- il a une seule valeur possible: ( )
  elle représente la n-uplet de 0 éléments,
- il est le type des expressions dont l'intérêt n'est pas dans leur valeur mais dans leur e□ets de bord
  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel
- on peut enchainer des expressions de type `unit` en utilisant `;`

```
# print_string "Hello␣";print_string "world\n";;
Hello world
- : unit = ()
```

Unit = la 0-uplet !

```
# print_string "Hello␣world\n";;
Hello world
- : unit = ()
```

- il a une seule valeur possible: ( )
  elle représente la n-uplet de 0 éléments,
- il est le type des expressions dont l'intérêt n'est pas dans leur valeur mais dans leur e□ets de bord
  - modification de l'état du système (e.g. affichage à l'écran, écriture sur un fichier, changement de la mémoire...)
  - on sort ici du cadre purement fonctionnel
- on peut enchainer des expressions de type `unit` en utilisant ;

```
# print_string "Hello␣";print_string "world\n";;
Hello world
- : unit = ()
```

Unit (fonctions)

print_char : char->unit	a□che un caractère
print_int : int->unit	a□che un entier
print_float : float->unit	a□che un nombre réel
print_string : string->unit	a□che une chaine de caractères
print_endline : string->unit	a□che une chaine suivie d'un changement de ligne
print_newline : unit->unit	a□che un changement de ligne
read_line : unit->string	lit une chaine de caractères
read_int : unit->unit	lit un entier
read_float : unit->float	lit un nombre réel

- plus sur `unit` quand on étudiera les traits impératifs de OCaml.

Unit (exemples)

```
(*notez la difference !*)
# let effet = print_endline "hello";;
hello
val effet : unit = ()

# let string = "hello";;
val string : string = "hello"

# string^" world";;
- : string = "hello world"

# effet^" world";;
  ^^^
Error: This expression has type unit but an expression
was expected of type string

# string ; 3;;
  ^^^
Warning 10: this expression should have type unit.
- : int = 3

# effet ; 3;;
- : int = 3
```

Unit (exemples)

```
# read_line ();;
Hello
- : string = "Hello"

# let hello = print_endline "Comment␣tu␣t'appelle?" ;
              let s = read_line () in
                 print_endline ("Bonjour␣"^s^"!");;
Comment tu t'appelle?
Michele
Bonjour Michele!
val hello : unit = ()

(*Qu'est-ce que fait cette fonction ?*)
# let rec perroquet x =
             print_endline "Je␣suis␣un␣perroquet" ;
             let s = read_line () in
                   print_endline s ;  perroquet ();;
val perroquet : unit -> 'a = <fun>
```

Unit (Exercice)

- Écrire une function qui choisit un entier n au hasard (utiliser la fonction `Random.int`) et demande à l'utilisateur de deviner n. Si l'utilisateur choisit un entier plus grand (resp. plus petit) le programme lui a□che `trop grand` (resp. `trop petit`) et répète la demande. Le programme s'arrête lorsque l'utilisateur devine le nombre n.

Les Exceptions,

Les Exceptions,
i.e. le type exn

Exceptions

```
# 3/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# String.get "Hello" 17;;
Exception: Invalid_argument "index␣out␣of␣bounds".
```

- une fonction d'un certain type peut ne pas être définie sur toutes les valeurs de ce type
- le mécanisme d'exceptions de OCaml permet de traiter l'application d'une fonction en dehors de son domain de définition
  - toujours préférable à l'envoi des valeurs légales mais bidon
  - arrêt du programme si l'exception n'est pas rattrapée

Exceptions

```
# 3/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# String.get "Hello" 17;;
Exception: Invalid_argument "index␣out␣of␣bounds".
```

- une fonction d'un certain type peut ne pas être définie sur toutes les valeurs de ce type
- le mécanisme d'exceptions de OCaml permet de traiter l'application d'une fonction en dehors de son domain de définition
  - toujours préférable à l'envoi des valeurs légales mais bidon
  - arrêt du programme si l'exception n'est pas rattrapée

Exceptions

```
# 3/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# String.get "Hello" 17;;
Exception: Invalid_argument "index␣out␣of␣bounds".
```

- une fonction d'un certain type peut ne pas être définie sur toutes les valeurs de ce type
- le mécanisme d'exceptions de OCaml permet de traiter l'application d'une fonction en dehors de son domain de définition
  - toujours préférable à l'envoi des valeurs légales mais bidon
  - arrêt du programme si l'exception n'est pas rattrapée

Exceptions

```
# 3/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# String.get "Hello" 17;;
Exception: Invalid_argument "index␣out␣of␣bounds".
```

- une fonction d'un certain type peut ne pas être définie sur toutes les valeurs de ce type
- le mécanisme d'exceptions de OCaml permet de traiter l'application d'une fonction en dehors de son domain de définition
  - toujours préférable à l'envoi des valeurs légales mais bidon
  - arrêt du programme si l'exception n'est pas rattrapée

Exceptions

```
# 3/0;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".

# String.get "Hello" 17;;
Exception: Invalid_argument "index␣out␣of␣bounds".
```

- une fonction d'un certain type peut ne pas être définie sur toutes les valeurs de ce type
- le mécanisme d'exceptions de OCaml permet de traiter l'application d'une fonction en dehors de son domain de définition
  - toujours préférable à l'envoi des valeurs légales mais bidon
  - arrêt du programme si l'exception n'est pas rattrapée

Exceptions (exn)

- les exceptions sont des valeurs d'un type exn:

```
type exn=
  ...
  | Division_by_zero
  | Failure of string
  | Invalid_argument of string
  | ...
```

- exn est un type somme:
  - Division_by_zero, Failure, ... sont les constructeurs,
  - parfois ils ont un argument pour transporter de l'information sur l'origine de l'exception
- en e□et, exn est un type extensible:

```
# exception Echec ;;
exception Echec
# exception Int_exception of int;;
exception Int_exception of int
```

Exceptions (raise)

- on peut lever une exception en utilisant `raise`:

```
# raise;;
- : exn -> 'a = <fun>
```

- le type 'a permets que la levée d'une exception soit compatible avec n'importe quelle autre expression:

```
# let rec fact = function
  0 -> 1
  | n -> if n>0 then n*(fact (n-1))
         else raise (Int_exception n);;
val fact : int -> int = <fun>

# fact (-1);;
Exception: Int_exception (-1).
```

- la levée d'une exception arrête l'évaluation d'une expression:

```
# (fun x -> 1) (fact (-1));;
Exception: Int_exception (-1).
```

Exceptions (raise)

- on peut lever une exception en utilisant `raise`:

```
# raise;;
- : exn -> 'a = <fun>
```

- le type `'a` permets que la levée d'une exception soit compatible avec n'importe quelle autre expression:

```
# let rec fact = function
  0 -> 1
  | n -> if n>0 then n*(fact (n-1))
         else raise (Int_exception n);;
val fact : int -> int = <fun>

# fact (-1);;
Exception: Int_exception (-1).
```

- la levée d'une exception arrête l'évaluation d'une expression:

```
# (fun x -> 1) (fact (-1));;
Exception: Int_exception (-1).
```

Exceptions (raise)

- on peut lever une exception en utilisant `raise`:

```
# raise;;
- : exn -> 'a = <fun>
```

- le type `'a` permets que la levée d'une exception soit compatible avec n'importe quelle autre expression:

```
# let rec fact = function
  0 -> 1
  | n -> if n>0 then n*(fact (n-1))
         else raise (Int_exception n);;
val fact : int -> int = <fun>

# fact (-1);;
Exception: Int_exception (-1).
```

- la levée d'une exception arrête l'évaluation d'une expression:

```
# (fun x -> 1) (fact (-1));;
Exception: Int_exception (-1).
```

Exceptions (**try** ... **with** ...)

```
try expr with
  | excep1 -> expr1
  ...
  | excepn -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue expr,
  - si pas d'exception, alors la valuer est la valeur de expr
  - sinon, on filtre la valeur de l'exception avec excep1, puis excep2, puis excep3, ...
    - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
    - sinon on propage l'exception jusqu'au prochaine `try...with...`
- les motifs doivent être du type exn
- les expressions dans le filtrage doivent être du même type

Exceptions (**try** ... **with** ...)

```
try expr with
  | excep1 -> expr1
  ...
  | excepn -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue expr,
  - si pas d'exception, alors la valuer est la valeur de expr
  - sinon, on filtre la valeur de l'exception avec excep1, puis excep2, puis excep3, ...
    - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
    - sinon on propage l'exception jusqu'au prochaine `try...with...`
- les motifs doivent être du type exn
- les expressions dans le filtrage doivent être du même type

Exceptions (**try** ... **with** ...)

```
try expr with
  | excep1 -> expr1
  ...
  | excepn -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue expr,
  - si pas d'exception, alors la valuer est la valeur de expr
  - sinon, on filtre la valeur de l'exception avec excep1, puis excep2, puis excep3, ...
    - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
    - sinon on propage l'exception jusqu'au prochaine `try...with...`
- les motifs doivent être du type exn
- les expressions dans le filtrage doivent être du même type

Exceptions (**try** ... **with** ...)

```
try expr with
  | excep1 -> expr1
  ...
  | excepn -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue expr,
  - si pas d'exception, alors la valuer est la valeur de expr
  - sinon, on filtre la valeur de l'exception avec excep1, puis excep2, puis excep3, ...
    - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
    - sinon on propage l'exception jusqu'au prochaine `try...with...`
- les motifs doivent être du type exn
- les expressions dans le filtrage doivent être du même type

Exceptions (**try** ... **with** ...)

```
try expr with
  | excep1 -> expr1
  ...
  | excepn -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue `expr`,
  - si pas d'exception, alors la valuer est la valeur de `expr`
  - sinon, on filtre la valeur de l'exception avec `excep1`, puis `excep2`, puis `excep3`, ...
    - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
    - sinon on propage l'exception jusqu'au prochaine `try...with...`
- les motifs doivent être du type exn
- les expressions dans le filtrage doivent être du même type

Exceptions (divide.ml)

```
let divide x =
 print_endline "donner le numerateur";
 let n = read_int () in
 print_endline "donner le denominateur";
 let m = read_int () in
 n/m

let rec main () =
 try
  begin
  let r = divide () in
  print_string "la division est ";
  print_int r
  end
 with
 | Division_by_zero ->
   print_endline "Grrr! pas choisir 0!"; main ()

main ()
```

Exceptions (complet1.ml)

```
1 (* sans exceptions,avec parcours multiples *)
2 type
```

Exceptions (complet2.ml)

```
(* avec exceptions et un seul parcours de l'arbre : *)
exception Incomplet

let complet a =
  let rec haut_aux a = match a with
    | F
```

Assertions

Mot clef assert, suivi d'une expression booléenne

Lève une exception Assert_failure quand l'expression donne false, avec nom du chier, numéro de ligne et colonne

Utile pour déclarer des invariants

Utile pour tester des invariants pendant l'exécution

On peut désactiver les assertions pendant la compilation d'un programme:

```
ocamlc noassert
```

Premier langage avec assertions: Eiffel

Assertions

- Mot clef assert, suivi d'une expression booléenne
- Lève une exception Assert_failure quand l'expression donne false, avec nom du fichier, numéro de ligne et colonne
- Utile pour déclarer des invariants
- Utile pour tester des invariants pendant l'exécution
- On peut désactiver les assertions pendant la compilation d'un programme:

  ocamlc −noassert
- Premier langage avec assertions : `Eiffel`

Assertions (Exemple)

```
let rec fib n =
 assert (n>=0) ;
 match n with
 | 0 | 1 -> n
 | n -> fib (n-1) + fib(n-2)

let rec main () =
 let _ = print_endline "Give me a non-negative number" in
 let n = read_int () in
  try let m = fib n in
    print_string "The fibonacci is ";
    print_int m; print_newline ()
  with
  | Assert_failure _ ->
    print_endline "Grrr ! non-negative !"; main ()
;;

main ()
```