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 effets 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 effets 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 effets 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	affiche un caractère
print_int : int->unit	affiche un entier
print_float : float->unit	affiche un nombre réel
print_string : string->unit	affiche une chaine de caractères
print_endline : string->unit	affiche une chaine suivie d'un changement de ligne
print_newline : unit->unit	affiche 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 affiche `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 effet, `exn` est un type extensible:

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

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 effet, `exn` est un type extensible:

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

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 effet, `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 (**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)

```
1  let divide x =
2   print_endline "donner le numerateur";
3   let n = read_int () in
4   print_endline "donner le denominateur";
5   let m = read_int () in
6   n/m
7
8  let rec main () =
9   try
10   begin
11   let r = divide () in
12   print_string "la division est ";
13   print_int r
14   end
15  with
16  | Division_by_zero ->
17    print_endline "Grrr! pas choisir 0!"; main ()
18
19 main ()
```

Exceptions (complet1.ml)

```
1  (* sans exceptions, avec parcours multiples *)
2  type arbre = F | N of arbre * arbre;;
3
4  let rec hauteur a =
5    match a with
6    | F -> 0
7    | N(g,d) -> 1 + max (hauteur g) (hauteur d);;
8
9  let rec complet a =
10   match a with
11   | F -> true
12   | N(g,d)->(complet g) && (complet d)    (* parcours g,d *)
13             && (hauteur g = hauteur d);;  (* parcours g,d *)
14
15
16 complet (N(N(F,F),N(F,F)));;
17 complet (N(N(F,F),N(F,N(F,F))));;
```

Exceptions (complet2.ml)

```
1  (* avec exceptions et un seul parcours de l'arbre : *)
2  exception Incomplet
3
4  let complet a =
5    let rec haut_aux a = match a with
6      | F -> 0
7      | N(g,d) ->
8        let hg = haut_aux g
9        and hd = haut_aux d in
10       if hg = hd then (1 + hg) else raise Incomplet
11   in try
12     let _ = haut_aux a in true
13   with Incomplet -> false;;
14
15 complet (N(N(F,F),N(F,F)));;
16 complet (N(N(F,F),N(F,N(F,F))));;
```

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

- 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

- 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

- 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

- 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

- 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)

```
1  let rec fib n =
2   assert (n>=0) ;
3   match n with
4    | 0 | 1 -> n
5    | n -> fib (n-1) + fib(n-2)
6
7  let rec main () =
8   let _ = print_endline "Give␣me␣a␣non-negative␣number" in
9   let n = read_int () in
10   try let m = fib n in
11     print_string "The␣fibonacci␣is␣";
12     print_int m; print_newline ()
13   with
14    | Assert_failure _ ->
15     print_endline "Grrr␣!␣non-negative␣!"; main ()
16 ;;
17
18 main ()
```