Algorithmique - Programmation 1

Cours 3

Université Henri Poincaré

CESS Epinal

Automne 2008

Plan

Typage de fonctions

- Fonction prédéfinie

```
# int_of_float;;
- : float -> int = <fun>
```

- Expression fonctionnelle anonyme

```
# function x -> x + 1;;
- : int -> int = <fun>
```

- Application d'une expression fonctionnelle

```
# (function x -> x + 1) 5;;
- : int = 6
```

- Expression fonctionnelle nommée (fonction)

```
# let f = function x -> x + 1;;
val f : int -> int = <fun>
# let f x = x + 1;;
```

- Application d'une fonction en un point

```
# (f 3);;
- : int = 4
```

Typage de fonctions (suite)

- Fonctions à plusieurs paramètres :

```
# let somme2 = function x -> function y -> x + y;;

# let moy2 x y = float_of_int(x + y) /. 2.;;

# let moy3 x y z = (x +. y +. z) /. 3.;;

# moy3 2. 4.5 6.;;
- : float = 4.1666666666666696
```

Typage de fonctions (suite)

Généralisation : fonctions à n paramètres

- Type :

```
nom : t1 -> ... -> tn -> res = <fun>
```

- Définition :

```
let nom = function p1 -> ... -> function pn ->
          expr(p1, ... pn);;
```

ou

```
let nom p1 ... pn = expr(p1, ... pn);;
```

- Appel :

```
nom v1 v2 ... vn;;
```

Plan

Portée des identifiants

- Définition d'une *liaison* entre un **identifiant** et une **valeur** (expression) au moyen du mot-clé `let`
- Liaison valide toute la durée d'une *session*, tant que le même nom n'est pas réutilisé dans un autre lien
  $\longrightarrow$ Liaison **globale**
- Problème : réutilisation d'un identifiant ?
- Définition de liaisons **locales** au moyen de la syntaxe suivante :

```
let identifiant = expr1 in expr2
```

- En cas d'homonymes, un identifiant réfère à la liaison ayant la *portée la plus restreinte*
- Exemple : `let a = 3 in (a+3)/2;;`

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)

# let b = 2;;

# let c = 1;;

# let somme= a+b+c;;

# let a = 10;; (* nouvelle liaison pour a*)

# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;

# let c = 1;;

# let somme= a+b+c;;

# let a = 10;; (* nouvelle liaison pour a*)

# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;
val b :  int = 2
# let c = 1;;

# let somme= a+b+c;;

# let a = 10;; (* nouvelle liaison pour a*)

# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;
val b :  int = 2
# let c = 1;;
val c :  int = 1
# let somme= a+b+c;;

# let a = 10;; (* nouvelle liaison pour a*)

# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;
val b :  int = 2
# let c = 1;;
val c :  int = 1
# let somme= a+b+c;;
val somme :  int = 6
# let a = 10;; (* nouvelle liaison pour a*)

# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;
val b :  int = 2
# let c = 1;;
val c :  int = 1
# let somme= a+b+c;;
val somme :  int = 6
# let a = 10;; (* nouvelle liaison pour a*)
val a :  int = 10 (*liaison précédente perdue*)
# somme;;
```

Portée des identifiants (suite)

Une session avec liaisons globales

```
# let a = 3;; (* première liaison globale pour a*)
val a :  int = 3
# let b = 2;;
val b :  int = 2
# let c = 1;;
val c :  int = 1
# let somme= a+b+c;;
val somme :  int = 6
# let a = 10;; (* nouvelle liaison pour a*)
val a :  int = 10 (*liaison précédente perdue*)
# somme;;
val somme :  int = 6
(*valeur calculée avec la première liaison
pour a, b et c*)
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
val rac :  float -> float -> float -> float = <fun>
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
val rac :  float -> float -> float -> float = <fun>

# rac 2.  2.  (-.3.);;
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
val rac :  float -> float -> float -> float = <fun>

# rac 2.  2.  (-.3.);;
- :  float = 3.2915026221291814 3
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
val rac :  float -> float -> float -> float = <fun>

# rac 2.  2.  (-.3.);;
- :  float = 3.2915026221291814 3

# delta;;
```

Portée des identifiants (suite)

Une session avec liaisons locales

```
# let rac a b c = let delta = (b *.  b -.  4.*.a*.c)
                  in (-.(b) +.  sqrt delta)/.2.*.a;;
val rac :  float -> float -> float -> float = <fun>

# rac 2.  2.  (-.3.);;
- :  float = 3.2915026221291814 3

# delta;;
Unbound value delta
(* la liaison établie entre delta et la formule est
locale à l'expression du calcul de racine,
elle n'est plus valide ici *)
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) and v=z-3 in
                float_of_int u /.float_of_int v;;
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) and v=z-3 in
                float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) and v=z-3 in
                float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) in
                 let v=u-3 in
                    float_of_int u /.float_of_int v;;
```

Portée des identifiants (suite)

Imbrication de liaisons locales

```
# let f z = let u=(z+1)*(z+1) in
                let v=z-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) and v=z-3 in
                float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>

# let f z = let u=(z+1)*(z+1) in
                 let v=u-3 in
                    float_of_int u /.float_of_int v;;
val f :  int -> float = <fun>
```

Portée des identifiants (suite)

Définition (Portée)

La portée d'une liaison est la portion du code dans laquelle cette liaison est valide (i.e. où un identifiant est lié à une expression).

- En Caml, on considère 3 portées :
  1. portée globale (`let`)
  2. portée locale (`let ... in`)
  3. argument de fonction (`function ... ->`)
- En cas d'ambiguïté (plusieurs occurrences du même identifiant), priorité à celui ayant la **portée la plus restreinte**

Portée : exemple

```
# let a = 2;;

# let b = a*3 in b*b;;

# let a = 3;;

# let a = a*a in let a = a+1 in a/2;;

# a+1;;

# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;

# let a = 3;;

# let a = a*a in let a = a+1 in a/2;;

# a+1;;

# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;
- :  int = 36
# let a = 3;;

# let a = a*a in let a = a+1 in a/2;;

# a+1;;

# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;
- :  int = 36
# let a = 3;;
val a :  int = 3
# let a = a*a in let a = a+1 in a/2;;

# a+1;;

# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;
- :  int = 36
# let a = 3;;
val a :  int = 3
# let a = a*a in let a = a+1 in a/2;;
- :  int = 5
# a+1;;

# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;
- :  int = 36
# let a = 3;;
val a :  int = 3
# let a = a*a in let a = a+1 in a/2;;
- :  int = 5
# a+1;;
- :  int = 4
# let f = function a ->3*a;;

# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a : int = 2
# let b = a*3 in b*b;;
- : int = 36
# let a = 3;;
val a : int = 3
# let a = a*a in let a = a+1 in a/2;;
- : int = 5
# a+1;;
- : int = 4
# let f = function a ->3*a;;
val f : int -> int = <fun>
# f 1;;

# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a : int = 2
# let b = a*3 in b*b;;
- : int = 36
# let a = 3;;
val a : int = 3
# let a = a*a in let a = a+1 in a/2;;
- : int = 5
# a+1;;
- : int = 4
# let f = function a ->3*a;;
val f : int -> int = <fun>
# f 1;;
- : int = 3
# let a=3 and b=a+3;;
```

Portée : exemple

```
# let a = 2;;
val a :  int = 2
# let b = a*3 in b*b;;
- :  int = 36
# let a = 3;;
val a :  int = 3
# let a = a*a in let a = a+1 in a/2;;
- :  int = 5
# a+1;;
- :  int = 4
# let f = function a ->3*a;;
val f :  int -> int = <fun>
# f 1;;
- :  int = 3
# let a=3 and b=a+3;;
val a :  int = 3          val b :  int = 6
```

Plan

Le type caractère

- Valeurs représentées :
  - les caractères de l'alphabet (majuscules et minuscules)
  - les chiffres
  - les signes de ponctuation
  - les caractères spéciaux (retour à la ligne, etc)
- En tout : 256 possibilités (chaque caractère est codé comme un entier sur 8 positions binaires)

  Exemple : $'a' = 97_{10} = 1100001_2$
- Codage standard des caractères : code ASCII (0 à 127)
  Entre 128 and 255 Caml suit le standard ISO 8859-1

Le type caractère (suite)

- En Caml, type noté `char`
- Quelques fonctions prédéfinies :

```
#  int_of_char ;;
- :  char -> int = <fun>

#  char_of_int ;;
- :  int -> char = <fun>

#  Char.lowercase ;;
- :  char -> char = <fun>
```

- Comparaison (ordre lexicographique) au moyen de < et > (test d'égalité via =)

Le type caractère (suite)

Quelques exemples de fonctions utilisant des caractères

```
# let est_un_a car = (car = 'a' or car = 'A');;

# let voyelle = function
                  'a' -> true
                | 'e' -> true
                | 'i' -> true
                | 'o' -> true
                | 'u' -> true
                | _   -> false;;
```

Plan

Le type chaîne de caractères

- Tableau de caractères
- En Caml, type noté `string`
- *Constructeur* : les guillemets
  Exemple :
```
# "toto";;
- :  string = "toto"
```
- Opérateur de **concaténation** : `^`
  Exemple :
```
# "bon"^" appetit";;
- : string = "bon appetit"
```

Le type chaîne de caractères (suite)

- Quelques fonctions prédéfinies :

```
#  string_of_int ;;
- :  int -> string = <fun>

#  float_of_string ;;
- :  string -> float = <fun>

#  String.get ;;
- :  String -> int -> char = <fun>
```

- Exemple :

```
# String.get 123 2;;
This expression has type int but is here used with type
string

# String.get (string_of_int 123) 2;;
- : char = '3'
```

Plan

Fonctions récursives

Définition (Fonction récursive)

Une fonction f est dite récursive si sa définition contient un appel à la fonction f elle-même.

- Représentation naturelle (*e.g.* suites)
- Lisibilité, simplicité (code concis)
- Exemple : *factorielle*

$$n\,! = \begin{cases} 1 & \text{si } n = 1 \\ n \times (n-1)\,! & \text{si } n > 1 \end{cases}$$

Fonctions récursives (suite)

- Définition d'une fonction récursive à base de cas :
  - cas d'arrêt (valeur particulière)
  - cas général (appel récursif)

- Dans le cas général, la fonction s'appelle elle-même en modifiant son(ses) paramètre(s) de manière à **converger vers le cas d'arrêt**

- En Caml, une fonction est définie comme étant récursive au moyen du mot clé `rec` :

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

Fonctions récursives (suite)

- La distinction entre cas général et cas d'arrêt peut se faire au moyen d'un **filtrage** (pattern matching) :

```
# let rec fact = function 1 -> 1
                        | n -> n * fact (n-1);;
val fact : int -> int = <fun>
```

- "Isolement" de l'appel récursif :

```
# let rec fact = function 1 -> 1
                        | n -> let fr = fact (n-1)
                               in n * fr ;;
val fact : int -> int = <fun>
```

Fonctions récursives (suite)

- Attention : l'appel récursif doit converger vers le cat d'arrêt sinon risque de débordement de mémoire

  (problème détecté à l'exécution uniquement)

- De plus : lors d'une définition par filtrage, tous les cas doivent retourner le **même type de données**

- Déroulement de la fonction `fact` pour `n = 5` ?

- Fonction permettant de représenter la suite ci-dessous ?

$$\begin{cases} U_0 &= 2 \\ U_n &= 3 \times U_{n-1} \end{cases}$$

Typage de fonctions

```
# function a -> function b -> function c ->
  (a (c * 1)) - (b (c + 0));;

# let f = function u -> function v -> u + v;;

# let g = f 1 ;;

# function f -> function g -> function k ->
  f (2 * k) *. g (3.5 *. float_of_int (k) ) ;;

# let rec f = function x -> function n ->
              if x = 0 then 1
                       else x * f x (n-1);;
```