(1) Premières fonctions

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2022/2023

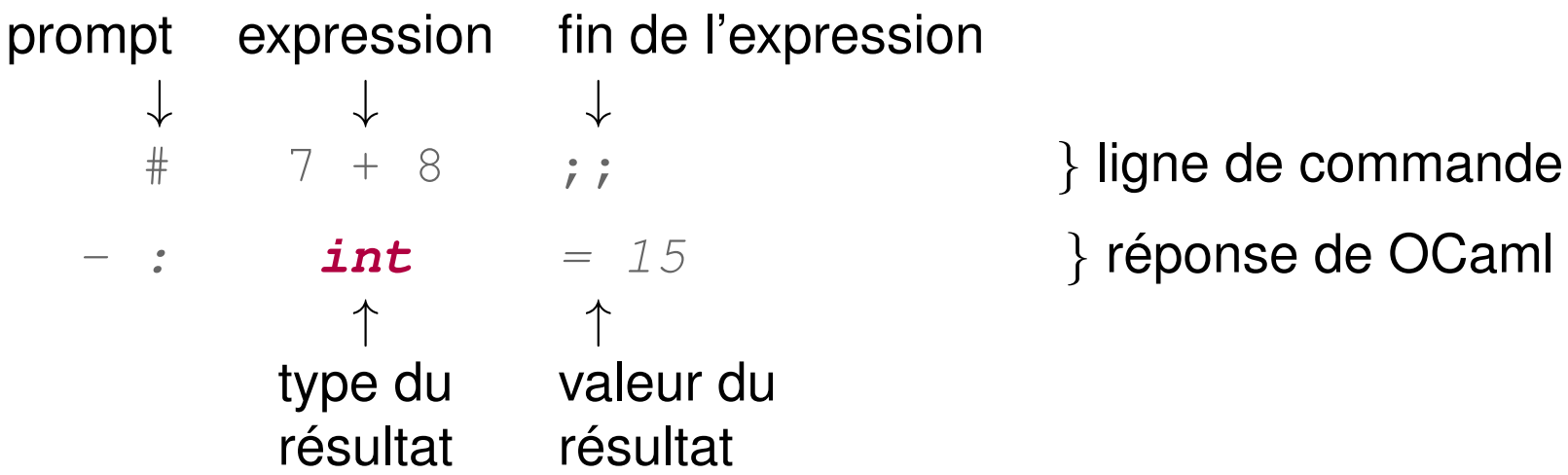
Mathieu Jaume – Adrien Koutsos

Programmation fonctionnelle

- programmer sans affectation
  - pas de manipulation explicite de la mémoire
  - pas de notion de variable au sens de celui utilisé en programmation impérative

  mais construction d'environnements permettant de donner des noms (des identificateurs) à des valeurs
  - $\neq$ notion de mémoire
- « programme fonctionnel » : suite de définitions
  - les définitions permettent de construire un environnement d'évaluation
  - « exécuter un programme fonctionnel » c'est évaluer une expression dans un environnement d'évaluation
- utilisation du noyau fonctionnel du langage OCaml
  - `https://ocaml.org`
  - OCaml offre des constructions impératives (variables, affectation, etc.), modulaires (modules, foncteurs, etc.) et objets (classes, héritage, etc.) ... qui sortent du cadre de ce cours

Premier exemple : session interactive

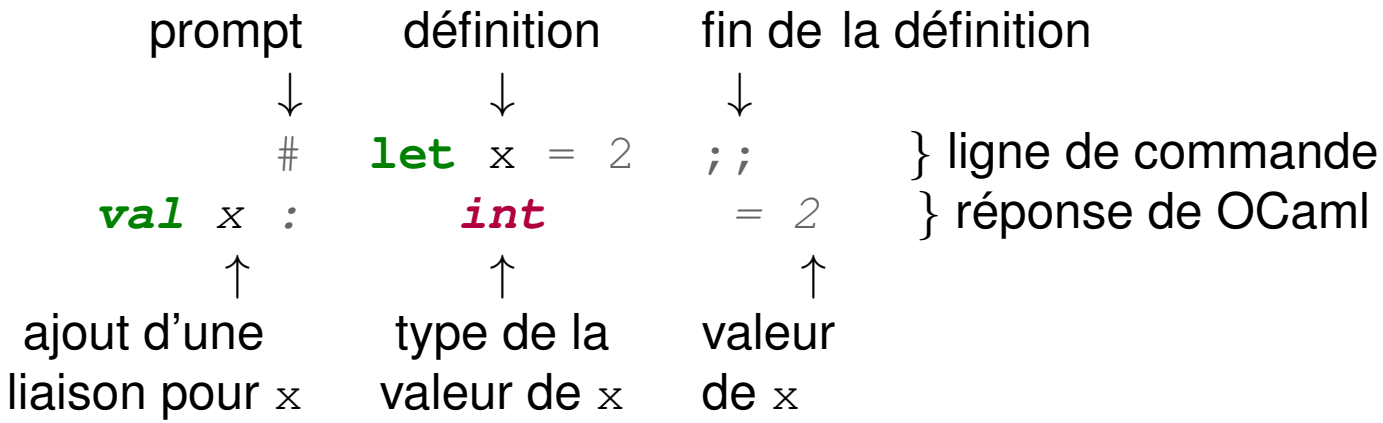



```
# 7 + 8;;
- : int = 15
# "deux" ^ "trois";;
- : string = "deuxtrois"
# "deux" + 3;;
 This expression has type string but is here used with type int
# x;;
 Unbound value x
```

les expressions sont évaluées dans l'**environnement d'évaluation** courant

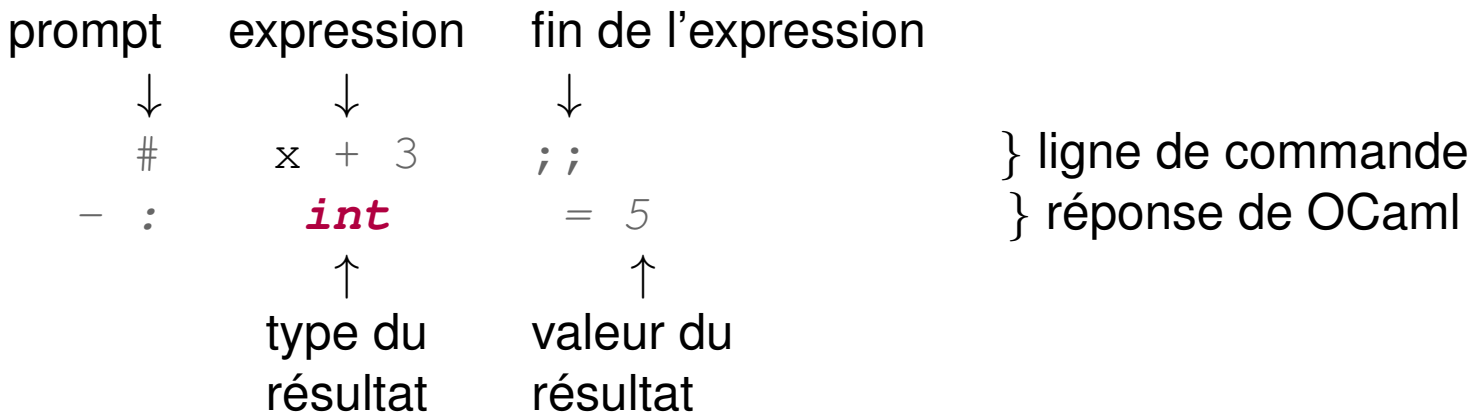
Evaluation dans un environnement d'évaluation

- construction d'un environnement d'évaluation (définitions)



ajout dans l'**environnement d'évaluation** de la **liaison** `(x,2)` qui associe la **valeur** 2 à l'**identificateur** `x`

- évaluation dans un environnement d'évaluation



l'environnement d'évaluation n'est pas modifié

Environnement d'évaluation

- toute expression est évaluée dans un environnement (d'évaluation)
- **environnement** : liste de **liaisons** (identificateur, valeur)
- l'environnement par défaut est l'environnement courant :
  - contient un ensemble de liaisons prédéfinies
    - ★ fonctions courantes, etc.
  - on peut enrichir l'environnement courant
    - ★ à partir de bibliothèques
    - ★ en ajoutant ses propres liaisons
- **ajout d'une liaison : définition**

`let` identificateur = expression

1. évaluation de l'expression (dans l'environnement courant)
   - ★ résultat de l'évaluation : valeur
2. ajout de la liaison (identificateur, valeur) dans l'environnement courant

Evaluation dans un environnement d'évaluation

```
# let x = 2 + 3 ;;
val x : int = 5
```
} définition
} ajout de la liaison (x,5)

```
# let y = x + 4 ;;
val y : int = 9
```
} définition
} ajout de la liaison (y,9)

```
# x * y ;;
- : int = 45
```
} expression

```
# x * z ;;
Error: Unbound value z
```
} expression
} erreur

```
# let x = 6;;
val x : int = 6
```
} définition
} ajout de la liaison (x,6)

```
# x ;;
- : int = 6
```
} expression

```
# y ;;
- : int = 9
```
} expression
} la valeur de y est inchangée

Types de base

- **`int`** : entiers relatifs machine
  - sur lesquels portent les opérateurs `+`, `-`, `*`, `/`, **`mod`** ...
- **`float`** : flottants
  - sur lesquels portent les opérateurs `+.`, `-.`, `*.`, `/.`, `exp`, `log`, ...
- **`bool`** : booléens `true` et `false`
  - sur lesquels portent les opérateurs `&&` (et), `||` (ou), `not`
  - qui peuvent être retournés par exemple par des opérateurs de comparaison `<`, `<=`, `>`, `>=`, ou `=` qui s'appliquent sur des entiers ou des flottants
- **`char`** : caractères
- **`string`** : chaînes de caractères
  - sur lesquelles porte l'opérateur `^` de concaténation

Annotation de type dans les définitions

- on peut indiquer le type de la valeur définie

```
# let x : int = 2 * 3 ;;
val x : int = 6

# let x : int = 2.0 *. 3.0 ;;
Error: This expression has type float but an expression
was expected of type int
```

  - OCaml est un **langage fortement typé** : pas de conversion implicite de type ... mais conversion explicite possible

    exemple : `float_of_int : int -> float`

```
# (float_of_int 6);;
- : float = 6.
```

Définition d'une fonction

exemple : fonction successeur sur les entiers relatifs : $succ : x \mapsto x + 1$

- définition de `succ`

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

  - `succ` : identificateur de la fonction définie
  - identificateur `x` : paramètre de la fonction
  - l'expression `x + 1` est le corps de la fonction

- `int -> int` est le type de la fonction `succ`
  - le paramètre de `succ` est de type `int`
  - le résultat de l'application de `succ` à un argument est de type `int`

- définition de `succ` avec les annotations de type

```
# let succ (x : int) : int = x + 1 ;;
val succ : int -> int = <fun>
```

- **signature** de la fonction `succ`

```
succ (x : int) : int
```

Définition d'une fonction

exemple : fonction successeur sur les entiers relatifs : $succ : x \mapsto x + 1$

- définition de `succ`

```
# let succ (x : int) : int = x + 1 ;;
val succ : int -> int = <fun>
```

- ajout de la liaison (`succ`,v_{succ}) dans l'environnement courant
  - v_{succ} est la valeur (la clôture) associée à la fonction
    - ★ désignée par `<fun>`
    - ★ $\approx$ description du calcul à effectuer lors de l'application de cette fonction à un argument

Définition d'une fonction

$$\texttt{let}\ f\ (x : t_x) : t_r = e_f$$

- f : nom de la fonction
- x : paramètre de la fonction
- t_x : type du paramètre de la fonction
- e_f : corps de la fonction
- t_r : type du résultat de la fonction
  - type de l'expression e_f lorsque le type de x est t_x
- $t_x \rightarrow t_r$ est le type de la fonction f

remarque : les indications de type ne sont pas obligatoires avec OCaml **mais** dans ce cours nous les demandons (expliciter les types permet au début d'éviter bien des erreurs)

Application d'une fonction : exemples

1. définition de la fonction `succ` :

```
# let succ (x : int) : int = x + 1;;
  val succ : int -> int = <fun>
```

⇝ ajout d'une liaison pour l'identificateur `succ` dans l'environnement

2. calcul du successeur de `8` : évaluation de l'expression `(succ 8)`

```
# (succ 8);;
- : int = 9
```

3. définition d'une valeur pour l'identificateur `y`

```
# let y = 4;;
  val y : int  = 4
```

⇝ ajout de la liaison `(y,4)` dans l'environnement

4. calcul du successeur de `y+2` : évaluation de l'expression `(succ(y+2))`

```
# (succ (y + 2));;
- :  int = 7
```

5. définition d'une valeur pour l'identificateur `z`

```
# let z = (succ (y + 5));;
  val z : int  = 10
```

⇝ ajout de la liaison `(z,10)` dans l'environnement

Application d'une fonction : exemples

1. définition de la fonction `succ` :

```
# let succ (x : int) : int = x + 1;;
  val succ : int -> int = <fun>
```

⇝ ajout d'une liaison pour l'identificateur `succ` dans l'environnement

2. définition d'une valeur pour l'identificateur `x`

```
# let x = 18;;
  val x : int  = 18
```

⇝ ajout de la liaison `(x,18)` dans l'environnement

3. calcul du successeur de `8` : évaluation de l'expression `(succ 8)`

```
# (succ 8);;
- : int = 9
```

   - lors de l'évaluation de `(succ 8)`, le corps `x+1` de la fonction `succ` est évalué
     - ★ lors de l'évaluation du corps `x+1` de la fonction `succ`, la valeur associée à l'identitificateur `x` est celle de l'argument (8) ... et pas celle de l'identificateur `x` dans l'environnement courant (18)
     - ★ l'identificateur `x` dans le corps `x+1` de la fonction est un **identificateur lié**

Application d'une fonction : appel par valeur

modèle d'évaluation

- définition de la fonction f — `let` $f\ (x : t_x) : t_r = e_f$
- application de la fonction f à un argument e_a — $(f\ e_a)$
  - e_a est une expression de type t_x
  1. évaluation de l'expression e_a dans l'environnement courant
     $\rightsquigarrow$ résultat de l'évaluation : valeur v_a
  2. évaluation de l'expression e_f dans **un** environnement augmenté de la liaison (x, v_a)
     $\rightsquigarrow$ résultat de l'évaluation = valeur de l'expression $(f\ e_a)$

     ⋆ **un** environnement : lequel ? cf. exercice 1.3 TME1

Application d'une fonction : appel par valeur

appel par valeur : évaluation de l'application $(f\ e_a)$

1. évaluation de l'expression e_a en argument en une valeur v_a
2. (puis) évaluation du corps de la fonction f avec la valeur v_a liée au paramètre de f

avantage : si le paramètre apparaît plusieurs fois dans le corps de la fonction f il n'est évalué qu'une seule fois

```
let f (x : int) : int = x + x
```

inconvénient : si le paramètre n'apparaît pas dans le corps de la fonction f il est quand même évalué

```
let g (x : int): int = 3
```

- l'évaluation de `(g (fact 100000))` en la valeur 3 prend beaucoup de temps !
- l'évaluation de `(g (foo x))` ne termine pas si l'évaluation de `(foo x)` ne termine pas

... au programmeur d'éviter ces situations

Application d'une fonction : syntaxe

- l'application de la fonction f à l'argument e_a s'écrit $\boxed{(f\ e_a)}$
  - **ne s'écrit pas** $f(e_a)$!
- l'application de la fonction f à l'argument e_a peut s'écrire $\boxed{f\ e_a}$
  - sans parenthèse l'expression $e_1\ e_2\ e_3$ est interprétée par l'application $((e_1\ e_2)\ e_3)$
    1. la fonction e_1 est appliquée à l'argument $e_2 \rightsquigarrow$ le résultat de l'évaluation est une fonction (cf. cours 2 et 3) ...
    2. ... qui est appliquée à l'argument e_3 pour obtenir le résultat final
  - sans parenthèse l'expression $e_1\ e_2\ \cdots\ e_{n-1}\ e_n$ est interprétée par l'application $(((e_1\ e_2)\ \cdots\ e_{n-1})\ e_n)$
- conseil : expliciter le parenthésage des expressions

Application d'une fonction : syntaxe

- $e_1\ e_2\ e_3$ est interprétée par l'application $((e_1\ e_2)\ e_3)$
  1. la fonction e_1 est appliquée à l'argument $e_2 \rightsquigarrow$ le résultat de l'évaluation est une fonction ...
  2. ... qui est appliquée à l'argument e_3 pour obtenir le résultat final
- *exemple* : fonction opposé sur les entiers relatifs : $op : x \mapsto -x$

```
let op (x : int) : int = -x
```

  - `op op 3` n'est pas une expression bien typée (erreur)
    - ★ `op op 3` est interprétée par l'application ((`op op`) `3`)
      ... mais `op` est une fonction dont l'argument est de type `int` et son argument `op` est de type `int -> int`
  - calcul de l'opposé de l'opposé de 3 : `(op (op 3))`

```
# (op (op 3));;
- : int = 3
```

  - `op 3 + 8` s'évalue à 5 `=((op 3)+8) = -3+8`
  - `op (3 + 8)` s'évalue à -11 `=(op (3+8)) = (op 11)=-11`

Expressions conditionnelles

if e_b **then** e_1 **else** e_2

1. évaluation de l'expression e_b en une valeur booléenne v_b
   - sinon l'expression **if** e_b **then** e_1 **else** e_2 est mal typée (erreur)
2.
   - si v_b est la valeur `true` alors évaluation de l'expression e_1 en une valeur v_1 de type t_1 ...
   - si v_b est la valeur `false` alors évaluation de l'expression e_2 en une valeur v_2 de type t_2 ...

   ... c'est le résultat de l'évaluation de l'expression **if** e_b **then** e_1 **else** e_2

- pour pouvoir attribuer un type unique à **if** e_b **then** e_1 **else** e_2 , les expressions e_1 et e_2 doivent avoir le même type : $t_1 = t_2$
- *remarque* : en utilisant uniquement le noyau fonctionnel de OCaml, la partie **else** est obligatoire
- *exemple* : valeur absolue d'un entier relatif

```
let abs (x : int) : int = if x > 0 then x else - x
# (abs (-2));;
- : int = 2
```

Définitions récursives

- programmation impérative : boucles **`while`**, **`for`**, etc.
  - itérer/répéter une séquence de calcul en faisant varier la valeur en mémoire de certaines variables (avec des affectations)
- programmation fonctionnelle : pas d'affectation, pas de manipulation explicite de la mémoire $\rightsquigarrow$ pas de boucles **`while`**, **`for`**, etc.
- itérer/répéter une séquence de calcul sans affectation ?
  - exécutions successives de la séquence de calcul dans des environnements différents
  - définition d'une fonction
    - ★ dont le corps est la séquence de calcul à itérer
    - ★ dont les paramètres sont les identificateurs des valeurs que l'on souhaite faire varier
  - modification de l'environnement : appels successifs à une même fonction avec des arguments différents

$\rightsquigarrow$ **définition de fonctions récursives**

Définitions récursives : exemples

- factorielle d'un entier naturel :

$$0! = 1 \qquad n! = \underbrace{1 \times 2 \times \cdots \times (n-1)}_{(n-1)!} \times n = (n-1)! \times n \text{ (pour } n > 0)$$

  - *exemple* : $5! = \underbrace{1 \times 2 \times 3 \times 4}_{4!} \times 5 = 120$

```
let rec fact (n : int) : int =
  if n = 0 then 1 else (fact (n - 1)) * n
```

```
# (fact 3);;
- : int = 6
```

```
(fact 3)
= if 3=0 then 1 else (fact (3-1))*3          = (fact 2) * 3
= (if 2=0 then 1 else (fact (2-1))*2)*3      = ((fact 1) * 2) * 3
= ((if 1=0 then 1 else (fact (1-1))*1)*2)*3  = (((fact 0) * 1) * 2) * 3
= ((if 0=0 then 1 else (fact (0-1))*1)*2)*3  = ((1 * 1) * 2) * 3 = 6
```

Définition de fonctions récursives : exemples

```
let rec fact (n:int):int = if n=0 then 1 else (fact (n-1)) * n
```

(fact 3) = (fact 2) ∗ 3 = ((fact 1) ∗ 2) ∗ 3 = (((fact 0) ∗ 1) ∗ 2) ∗ 3

= ((1 ∗1) ∗ 2) ∗ 3 = (1 ∗2) ∗ 3 = 2 ∗3 = 6

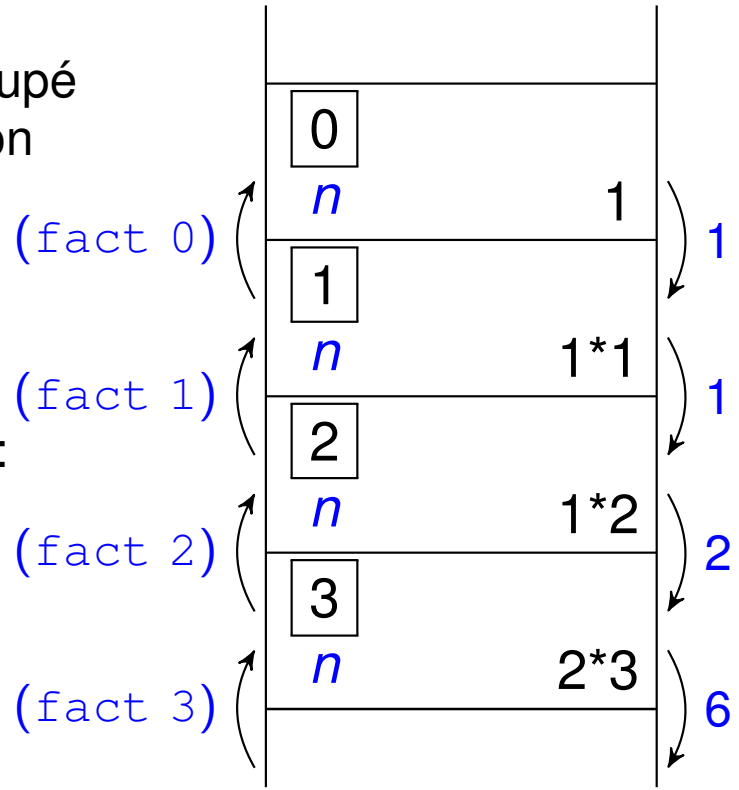
(fact 0) (fact 1) (fact 2)

(fact 1) (fact 2) (fact 3)

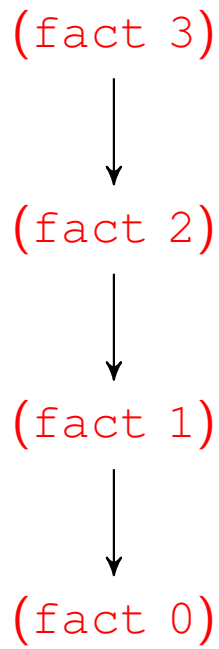
l'espace mémoire occupé dans la pile d'exécution est proportionnel au nombre d'appels récursifs

si *n* est « trop » grand :

`Stack overflow`



Pile d'exécution



Arbre des appels

Définitions récursives : exemples

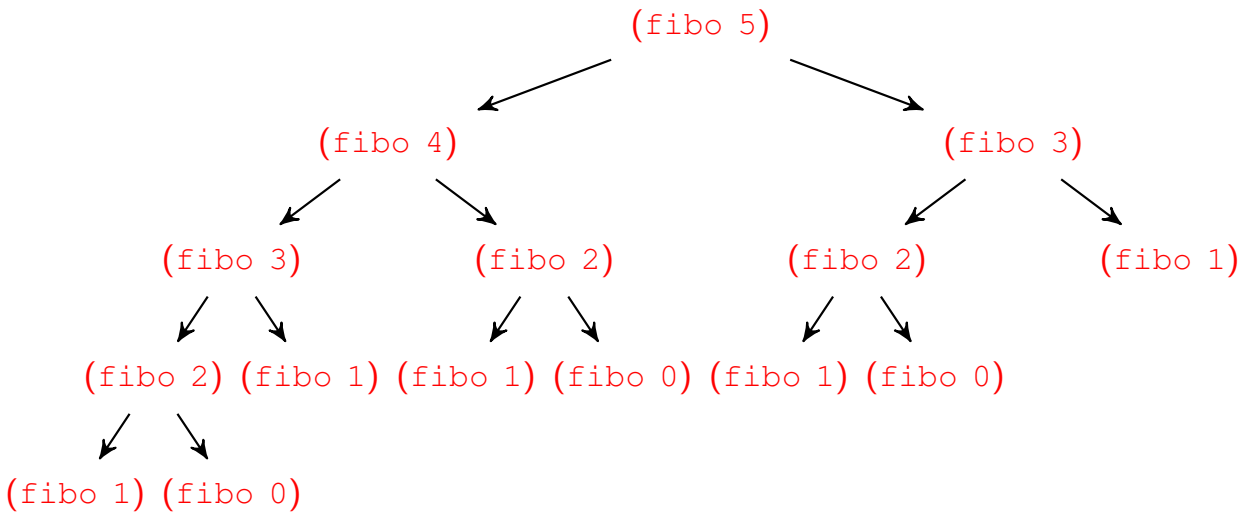
- suite de Fibonacci $F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$ (pour $n \geq 2$)

```
let rec fibo (n : int) : int =
  if n = 0 then 0
           else if n = 1 then 1
                         else (fibo (n-1)) + (fibo (n-2))

# (fibo 6);;
- : int = 8
```

Définitions récursives : exemples

```
let rec fibo (n : int) : int =
  if n = 0 then 0
           else if n = 1 then 1
                         else (fibo (n-1)) + (fibo (n-2))
```



Arbre des appels

(fibo 5) est appelé/calculé 1 fois	(fibo 2) est appelé/calculé 3 fois
(fibo 4) est appelé/calculé 1 fois	(fibo 1) est appelé/calculé 5 fois
(fibo 3) est appelé/calculé 2 fois	(fibo 0) est appelé/calculé 3 fois

Définitions récursives : exemples

- fonctions mutuellement récursives

```
let rec pair (n : int) : bool =
  if n = 0 then true
           else (impair (n - 1))
and impair (m : int) : bool =
  if m = 0 then false
           else (pair (m - 1))

val pair : int -> bool = <fun>
  impair : int -> bool = <fun>
```

Définitions récursives

- application de fonctions récursives
  - plusieurs évaluations du corps de la fonction dans des environnements d'évaluation différents
    - ★ chaque environnement contient la valeur de l'argument utilisé lors de l'appel
  - terminaison : le cas de base (cas sans appel récursif) doit toujours être atteint au bout d'un nombre fini d'appels récursifs
- complexité en espace mémoire / en temps de calcul ?
  - pile d'exécution : stockage de valeurs utilisées pour produire le résultat à partir des résultats des appels récursifs

    ```
    Stack overflow during evaluation (looping recursion?).
    ```
  - duplication des calculs
    - ★ fonction `fibo`

⇝ **récursivité terminale** : cf. cours 6

Définitions locales

exemple : évaluation de l'expression `(fact 10) + 4 * (fact 10)`

- la sous-expression `(fact 10)` est évaluée deux fois
- pour évaluer une seule fois l'expression `(fact 10)` il faut pouvoir stocker la valeur de cette expression dans l'environnement
  1. évaluation de `(fact 10)` en une valeur v
  2. associer un identificateur id à la valeur v
  3. ajout dans l'environnement d'une liaison (id,v)
  4. évaluation de l'expression id `+4*`id

  la liaison (id,v) n'est utile que durant l'évaluation de l'expression id `+4*`id
- la liaison (id,v) peut être définie uniquement dans l'environnement d'évaluation de l'expression id `+4*`id
  - définition locale (liaison temporaire)

```
let id = (fact 10) in
id + 4 * id
```

Définitions locales

ajouter temporairement une liaison (identificateur,valeur) durant l'évaluation d'une expression

```
let identificateur = expression_id in
expression
```

- ne modifie pas l'environnement courant
  - $\neq$ différent d'une définition qui ajoute une liaison dans l'environnement courant

1. évaluation de l'expression expression_id dans l'environnement courant
   - résultat de l'évaluation : valeur
2. évaluation de l'expression expression dans l'environnement courant augmenté de la liaison (identificateur,valeur)
   - qui masque **temporairement** les éventuelles liaisons de identificateur déjà présentes dans l'environnement courant
   - résultat de l'évaluation = valeur de l'expression
     `let identificateur = expression_id in expression`

Définitions locales : exemples

```
# let x = 5;;
val x : int = 5
```

définition : ajout de la liaison (x,5)

```
# let z = x + 3 in
  2 * z;;
- : int = 16
# z;;
Error: Unbound value z
```

expression (la liaison (z,8) n'est pas ajoutée dans l'environnement courant)

```
# let x = x + 3 in
  2 * x;;
- : int = 16
# x;;
- : int = 5
```

la liaison (x,8) masque **temporairement** la liaison (x,5)

l'environnement courant contient la liaison (x,5)

```
# let y =
    let z = x + 3 in
    2 * z;;
val y : int = 16
# y;;
- : int = 16
```

définition : ajout de la liaison (y,16) dans l'environnement courant (16 est le résultat de l'évaluation dans l'environnement courant de

```
    let z = x + 3 in
    2 * z
```

)

Définitions locales

EXERCICE. calculer la valeur de l'expression :

```
let x = 1 in
let x = x + 1 in
let x = x + 2 in
x + 3
```