Programmation Fonctionnelle
Cours 06

Michele Pagani

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

pagani@irif.fr

15 octobre 2018

Récurrence terminale

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>

# mklist 3;;
- : int list = [3; 2; 1]

# mklist 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

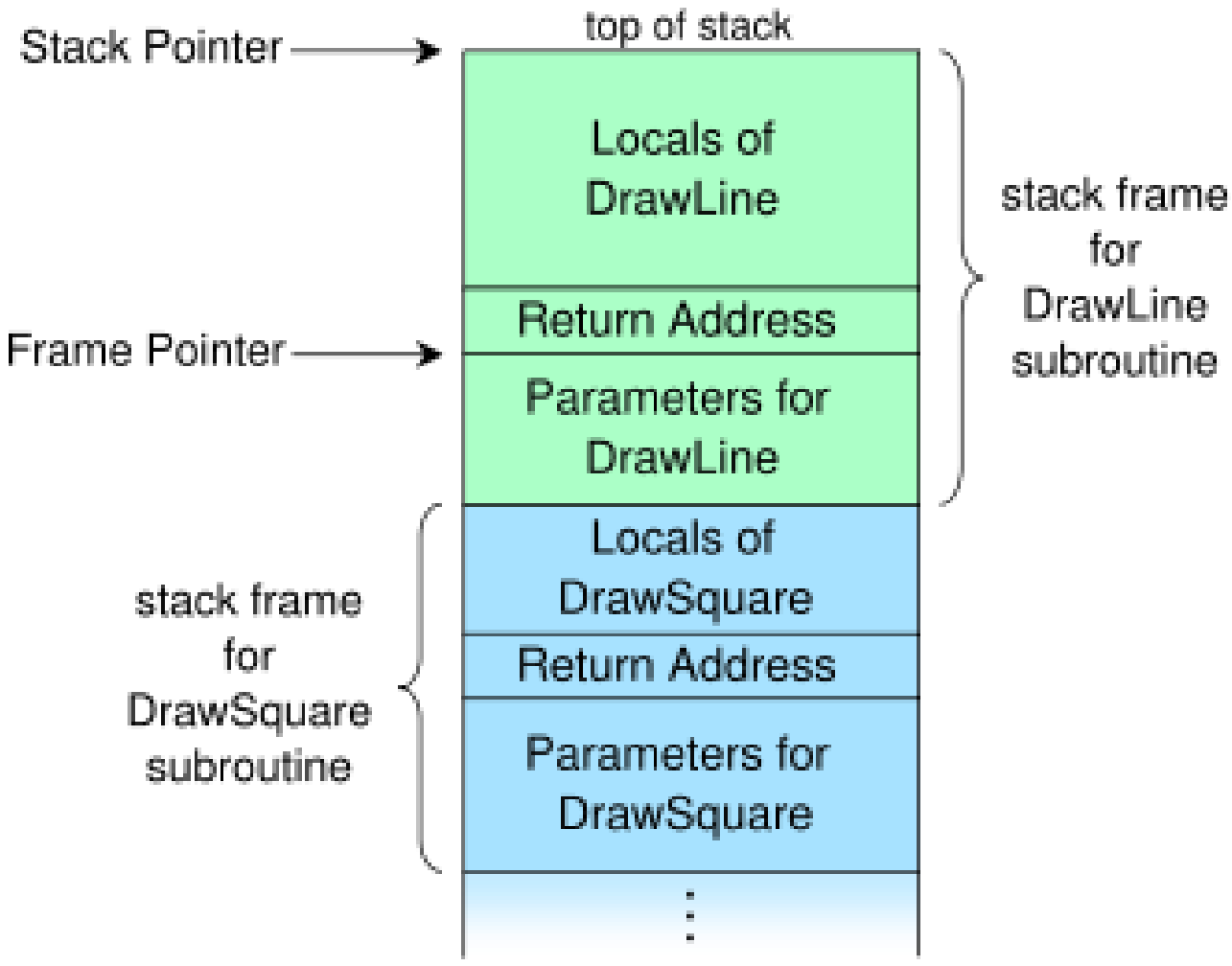
```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Exemple: `DrawSquare` appèle `DrawLine`



Merci, Wikipedia

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
    | 0 -> []
    | n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ 3::(mklist 2) $\Rightarrow$ 3::2::(mklist 1) $\Rightarrow$ 3::2::1(mklist 0)
$\Rightarrow$ 3::2::1::[] $\Rightarrow$ 3::2::[1] $\Rightarrow$ 3::[2;1] $\Rightarrow$ [3;2;1]

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriquées, on peut épuiser l'espace disponible pour la pile !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
  - appel terminal: une fonction f appèl une fonction g, mais le résultat envoyé par g est envoyé tout de suite par f,
  - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
  - l'espace sur la pile d'exécution peut être libérée avant de lancer g!
- Récurrence terminale (angl.: tail recursion): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  | 0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
    in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  | 0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
  in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with
  | 0 -> []
  | n -> n::(mklist (n-1));;
```

  - on exécute un calcul (n:: ...) avant de renvoyer le résultat de l'appel récursif (mklist (n−1))

- Voici une variante terminale:

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
  in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

  - aucun calcul après l'appel récursif (mkaux (n−1) (n::l))
  - on utilise une fonction auxiliaire (mkaux) avec un argument accumulateur (l)
  - attention à renverser l'ordre des elements de la liste

Récurrence terminale (exemples)

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
    in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

$$\begin{aligned}
&\texttt{mklist } 3 \Rightarrow \texttt{List.rev(mkaux 3 [])} \Rightarrow \texttt{List.rev(mkaux 2 (3::[]))} \\
&\quad \Rightarrow \texttt{List.rev(mkaux 2 (3::[]))} \Rightarrow \texttt{List.rev(mkaux 2 [3])} \\
&\Rightarrow \texttt{List.rev(mkaux 1 (2::[3]))} \Rightarrow \texttt{List.rev(mkaux 1 (2::[3]))} \\
&\quad \Rightarrow \texttt{List.rev(mkaux 1 [2;3])List.rev(mkaux 0 (1::[2;3]))} \\
&\Rightarrow \texttt{List.rev(mkaux 0 (1::[2;3]))} \Rightarrow \texttt{List.rev(mkaux 0 [1;2;3])} \\
&\quad \Rightarrow \texttt{List.rev [1;2;3]} \Rightarrow \cdots \Rightarrow [3;2;1]
\end{aligned}$$

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- consequence. . .

Récurrence terminale (exemples)

```
# let mklist n =
    let rec mkaux n l = match n with
      | 0 -> l
      | n -> mkaux (n-1) (n::l)
    in List.rev (mkaux n []);;
val mklist : int -> int list = <fun>
```

mklist 3 $\Rightarrow$ List.rev(mkaux 3 []) $\Rightarrow$ List.rev(mkaux 2 (3::[]))
$\Rightarrow$ List.rev(mkaux 2 (3::[])) $\Rightarrow$ List.rev(mkaux 2 [3])
$\Rightarrow$ List.rev(mkaux 1 (2::[3])) $\Rightarrow$ List.rev(mkaux 1 (2::[3]))
$\Rightarrow$ List.rev(mkaux 1 [2;3])List.rev(mkaux 0 (1::[2;3]))
$\Rightarrow$ List.rev(mkaux 0 (1::[2;3])) $\Rightarrow$ List.rev(mkaux 0 [1;2;3])
$\Rightarrow$ List.rev [1;2;3] $\Rightarrow \cdots \Rightarrow$ [3; 2; 1]

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- consequence...

Récurrence terminale (exemples)

```
# mklist 100000;;
- : int list =
[100000; 99999; 99998; 99997; 99996; 99995; 99994; 99993; 99992; 99991;
 99990; 99989; 99988; 99987; 99986; 99985; 99984; 99983; 99982; 99981; 99980;
 99979; 99978; 99977; 99976; 99975; 99974; 99973; 99972; 99971; 99970; 99969;
 99968; 99967; 99966; 99965; 99964; 99963; 99962; 99961; 99960; 99959; 99958;
 99957; 99956; 99955; 99954; 99953; 99952; 99951; 99950; 99949; 99948; 99947;
 99946; 99945; 99944; 99943; 99942; 99941; 99940; 99939; 99938; 99937; 99936;
 99935; 99934; 99933; 99932; 99931; 99930; 99929; 99928; 99927; 99926; 99925;
 99924; 99923; 99922; 99921; 99920; 99919; 99918; 99917; 99916; 99915; 99914;
 99913; 99912; 99911; 99910; 99909; 99908; 99907; 99906; 99905; 99904; 99903;
 99902; 99901; 99900; 99899; 99898; 99897; 99896; 99895; 99894; 99893; 99892;
 99891; 99890; 99889; 99888; 99887; 99886; 99885; 99884; 99883; 99882; 99881;
 99880; 99879; 99878; 99877; 99876; 99875; 99874; 99873; 99872; 99871; 99870;
 99869; 99868; 99867; 99866; 99865; 99864; 99863; 99862; 99861; 99860; 99859;
 99858; 99857; 99856; 99855; 99854; 99853; 99852; 99851; 99850; 99849; 99848;
 99847; 99846; 99845; 99844; 99843; 99842; 99841; 99840; 99839; 99838; 99837;
 99836; 99835; 99834; 99833; 99832; 99831; 99830; 99829; 99828; 99827; 99826;
 99825; 99824; 99823; 99822; 99821; 99820; 99819; 99818; 99817; 99816; 99815;
 99814; 99813; 99812; 99811; 99810; 99809; 99808; 99807; 99806; 99805; 99804;
 99803; 99802; 99801; 99800; 99799; 99798; 99797; 99796; 99795; 99794; 99793;
 99792; 99791; 99790; 99789; 99788; 99787; 99786; 99785; 99784; 99783; 99782;
 99781; 99780; 99779; 99778; 99777; 99776; 99775; 99774; 99773; 99772; 99771;
 99770; 99769; 99768; 99767; 99766; 99765; 99764; 99763; 99762; 99761; 99760;
 99759; 99758; 99757; 99756; 99755; 99754; 99753; 99752; 99751; 99750; 99749;
 99748; 99747; 99746; 99745; 99744; 99743; 99742; 99741; 99740; 99739; 99738;
 99737; 99736; 99735; 99734; 99733; 99732; 99731; 99730; 99729; 99728; 99727;
 99726; 99725; 99724; 99723; 99722; 99721; 99720; 99719; 99718; 99717; 99716;
 99715; 99714; 99713; 99712; 99711; 99710; 99709; 99708; 99707; 99706; 99705;
 99704; 99703; 99702; ...]
```

Exercices

Transformez les récurrences suivantes en récurrences terminales:

```
let rec fact n = match n with
  |0 -> 1
  |n -> n*(fact (n-1));;

val fact : int -> int = <fun>


let divisors x =
   let rec aux x c =
      if c = x then [ x ]
      else if (x mod c = 0) then (c::(aux x (c+1)))
      else aux x (c+1)
   in aux x 1;;

val divisors : int -> int list = <fun>


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

val fib : int -> int = <fun>
```

Type Unit

Unit = la 0-uplet !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: ()
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

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

  - 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 !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: `()`
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

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

  - 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 !

```
# ();;
- : unit = ()
```

- il est le type avec une seule valeur possible: `()`
- l'intérêt d'une expression de type `unit` n'est pas dans sa valeur mais dans ses effets de bord

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

  - 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->int	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.