Programmation fonctionnelle

David Delahaye et Philippe Janssen

HAI102I

Organisation

- 8 séances de cours
- 22 séances de TD, dont 10 séances de TD en salle informatique

Modalités de Contrôle de Connaissances

- un contrôle en amphi fin octobre - coefficient 30%
- un examen en janvier - coefficient 70%

Documents

- Moodle, le cours, les fiches de TD et TD informatisés
- Bibliographie
  - *Approche fonctionnelle de la programmation*
    Guy Cousineau, Michel Mauny - Ediscience
  - *Concepts et outils de programmation*
    Thérèse Accart Hardin, Véronique Donzeau-Gouge Viguié - InterEditions
  - *Developing Applications With OCaml*
    Emmanuel Chailloux, Pascal Manoury, Bruno Pagano,
    `http://caml.inria.fr/pub/docs/oreilly-book/html/index.html`.

Introduction

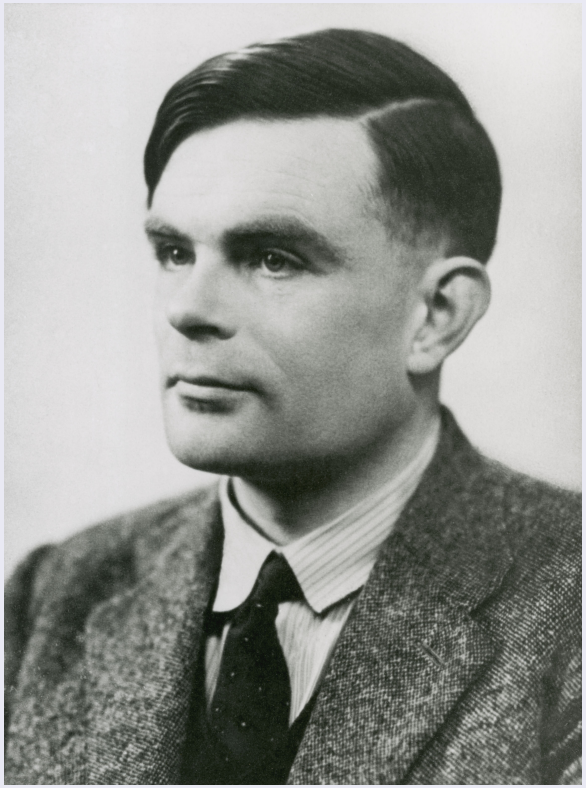
Résoudre un problème consiste à calculer des résultats à partir de données. Pour décrire un tel calcul plusieurs approches sont possibles :

- **L'approche impérative** : définir des étapes permettant de passer de l'état de départ (les données) à l'état d'arrivée (le résultat)
- **L'approche fonctionnelle** : spécifier le problème comme une fonction associant le résultat aux données, et définir cette fonction en la décomposant à partir de fonctions élémentaires.
- ...

Machines de Turing et programmation impérative

Machines de Turing :

- Fondement théorique des ordinateurs modernes et de la programmation impérative ;
- Bande ≡ mémoire adressable en lecture/écriture avec un programme stocké ;
- Automate ≡ microprocesseur.



λ-calcul et programmation fonctionnelle

λ-calcul :

- $\lambda x.M$ = fonction anonyme avec le paramètre formel x et le corps M (abstraction) ;
- $M\ N$ = appel de la fonction M avec le paramètre effectif N (application) ;
- Règle de calcul (β-réduction) : $(\lambda x.M)\ N \rightarrow_\beta M[x := N]$.

A. Church

Thèse de Church-Turing

Équivalence des machines de Turing et du λ-calcul (Turing, 1937)

- Une fonction est calculable par une machine de Turing, si et seulement si elle est calculable en utilisant le λ-calcul.

Thèse de Church-Turing

- Une fonction calculable par n'importe quelle méthode effective de calcul est aussi calculable par une machine de Turing.

En d'autres termes

- Tous les langages de programmation généralistes ont le même pouvoir de calcul et sont donc équivalents du point de vue de la calculabilité.

Mais les langages de programmation ne sont pas nés égaux

Différents pouvoirs d'expression

- Différentes représentations des données ;
- Différents modèles d'exécution ;
- Différents mécanismes d'abstraction.

D'autres caractéristiques désirables

- Sûreté de l'exécution ;
- Efficacité de l'exécution ;
- Maintenabilité du code.

Nous allons apprendre OCaml !

Spécificités

- Typage statique et inférence de type ;
- Typage fort (correct vis-à-vis de la sémantique) :
  - Un programme qui compile ne fera aucune erreur de type à l'exécution ;
- Polymorphisme et ordre supérieur ;
- Types structurés :
  - Types tuples, Types concrets, Enregistrements.
- Langage modulaire :
  - Objets, Modules.
- *Garbage collector*.

Installation

- Pour l'essayer : Try OCaml, `https://try.ocamlpro.com/` ;
- Pour l'installer : `https://ocaml.org/`.

Plan du cours

Plan du cours

Type

Types élémentaires

Les expressions de OCaml sont typées. Un type est défini par :

- un domaine : ensemble de valeurs que peuvent prendre les objets du type
- des opérations : ensemble de fonctions et opérations qu'on peut appliquer aux valeurs du type

Typage statique

En OCaml, on écrit des expressions qui correspondent à la définition ou à l'application de fonctions. Avant d'évaluer une expression OCaml vérifie si son typage est correct, c'est à dire si l'application des fonctions et opérateurs est conforme à leur signature.

Types numériques

`int` : les entiers relatifs

- Domaine : un sous-ensemble de $\mathbb{Z}$ ($[-2^{62}, 2^{62}-1]$ sur machine 64 bits)
- Opérations arithmétiques `int`$\times$`int`$\longrightarrow$`int` : + - * / mod

  Attention / et mod désignent le quotient et le reste de la division entière

  7 / 3 vaut 2 et 7 mod 3 vaut 1

`float` : les réels

- Domaine : un sous-ensemble de $\mathbb{R}$.

  Notation en virgule flottante : 1.264e2 126.4

  Attention le point est obligatoire : **2** est de type `int` et **2.** de type `float`
- Opérations `float`$\times$`float`$\longrightarrow$`float` : +. -. *. /. et ** (puissance)

  Attention le point est obligatoire : 2.3 + 3.1 n'est pas correct

  il faut écrire 2.3 +. 3.1
- Fonctions `float`$\longrightarrow$`float` : sqrt, ceil, floor , log, exp, sin ...

Types numériques

`int` : les entiers relatifs

- Domaine : un sous-ensemble de $\mathbb{Z}$ ($[-2^{62}, 2^{62}-1]$ sur machine 64 bits)
- Opérations arithmétiques `int`×`int`⟶`int` : + - * / mod

  Attention / et mod désignent le quotient et le reste de la division entière

  7 / 3 vaut 2 et 7 mod 3 vaut 1

`float` : les réels

- Domaine : un sous-ensemble de $\mathbb{R}$.

  Notation en virgule flottante : 1.264e2 126.4

  Attention le point est obligatoire : **2** est de type `int` et **2.** de type `float`
- Opérations `float`×`float`⟶`float` : **+.** **-.** ***.** **/.** et ** (puissance)

  Attention le point est obligatoire : 2.3 + 3.1 n'est pas correct

  il faut écrire 2.3 **+.** 3.1
- Fonctions `float`⟶`float` : sqrt, ceil, floor , log, exp, sin ...

Conversion de types

Les types `int` et `float` sont disjoints ; on ne peut pas les additionner, multiplier ... 3.2 * 2 n'est pas correct.
Cependant, il existe des fonctions de conversion entre ces types :

- `float_of_int` : `int` $\longrightarrow$ `float`
- `int_of_float` : `float` $\longrightarrow$ `int`

3.2 *. `float_of_int`(2) est correct.
`int_of_float` est la valeur plancher : `int_of_float`(4.7) vaut 4

Les types booléen et chaîne de caractères

`bool` : les booléens

- Domaine : $\{true, false\}$
- Opérations : `not`, `&&`, `||`

a	`not` *a*
`true`	`false`
`false`	`true`

a	*b*	*a* `&&` *b*	*a* `||` *b*
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Ne pas confondre erreur et false
not true vaut **false** alors que **not 1** n'a pas de sens et donc pas de valeur.

`string` : les chaînes de caractères

- Domaine : suites de caractères entre guillemets `"au"` `"semestre 1"`
- Opération : `^` concaténation
  `"au" ^ "semestre 1"` vaut `"ausemestre 1"`

Les types booléen et chaîne de caractères

`bool` : les booléens

- Domaine : $\{true, false\}$
- Opérations : `not`, `&&`, `||`

a	`not` *a*
`true`	`false`
`false`	`true`

a	*b*	*a* `&&` *b*	*a* `||` *b*
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Ne pas confondre erreur et false
not true vaut **false** alors que **not 1** n'a pas de sens et donc pas de valeur.

`string` : les chaînes de caractères

- Domaine : suites de caractères entre guillemets "au" "semestre 1"
- Opération : ^ concaténation

  "au" ^ "semestre 1" vaut "ausemestre 1"

Les opérateurs de comparaisons

- $=, <>, <, >, <=, >=$
- Le résultat de ces opérations est de type booléen, les deux opérandes sont de type identique quelconque. La signature de ces fonctions est :

$$\texttt{'a} \times \texttt{'a} \longrightarrow \texttt{bool}$$

  où `'a` désigne un type quelconque
- 1<>4 vaut `true`, "trois" $<$ "un" vaut `true`
- 4=4.0 et "un"=1 ne sont pas corrects

Expressions

Syntaxe des expressions de base

Une expression est :

- une constante
- (`exp1` op `exp2`) où op est un opérateur et `exp1` et `exp2` des expressions
- `fonc(exp)` où `fonc` est une fonction et `exp` une expression

De manière classique, on supprime certaines parenthèses en utilisant les propriétés et priorités des opérateurs.

Exemples

- `23/(3+4)` `ceil(12.61)` `"un" ^ "deux"` `(3.1 < sqrt(12.)) && true` sont des expressions syntaxiquement correctes
- `(3 + ) 2` n'est pas syntaxiquement correcte
- `1 && 1` est correcte du point de vue de la syntaxe, mais pas du typage

Expressions

Syntaxe des expressions de base

Une expression est :

- une constante
- (`exp1` op `exp2`) où op est un opérateur et `exp1` et `exp2` des expressions
- `fonc(exp)` où `fonc` est une fonction et `exp` une expression

De manière classique, on supprime certaines parenthèses en utilisant les propriétés et priorités des opérateurs.

Exemples

- 23/(3+4) ceil(12.61) "un" ^ "deux" (3.1 < sqrt(12.)) && true sont des expressions syntaxiquement correctes
- (3 +) 2 n'est pas syntaxiquement correcte
- 1 && 1 est correcte du point de vue de la syntaxe, mais pas du typage

Expressions

Syntaxe des expressions de base

Une expression est :

- une constante
- (`exp1` op `exp2`) où op est un opérateur et `exp1` et `exp2` des expressions
- `fonc(exp)` où `fonc` est une fonction et `exp` une expression

De manière classique, on supprime certaines parenthèses en utilisant les propriétés et priorités des opérateurs.

Exemples

- 23/(3+4) ceil(12.61) "un" ^ "deux" (3.1 < sqrt(12.)) && true sont des expressions syntaxiquement correctes
- (3 +) 2 n'est pas syntaxiquement correcte
- 1 && 1 est correcte du point de vue de la syntaxe, mais pas du typage

Expressions

Syntaxe des expressions de base

Une expression est :

- une constante
- (`exp1` op `exp2`) où op est un opérateur et `exp1` et `exp2` des expressions
- `fonc(exp)` où `fonc` est une fonction et `exp` une expression

De manière classique, on supprime certaines parenthèses en utilisant les propriétés et priorités des opérateurs.

Exemples

- 23/(3+4) ceil(12.61) "un" ^ "deux" (3.1 < sqrt(12.)) && true sont des expressions syntaxiquement correctes
- (3 +) 2 n'est pas syntaxiquement correcte
- 1 && 1 est correcte du point de vue de la syntaxe, mais pas du typage

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"
```

```
# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"
```

```
# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"
```

```
# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur
;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"
```

```
# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur
;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"

# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

L'interpréteur OCaml

Un programme OCaml est une expression. Lorsqu'une expression est saisie, l'interpréteur OCaml vérifie sa syntaxe et son type. S'ils sont corrects il évalue l'expression et affiche sa valeur et son type :

```
# expression ;;
- : type = valeur
```

où # est l'invite de l'interpréteur

;; est la marque de fin d'expression

Exemple de session

```
# 23/(3+4);;
- : int = 3

# ceil(12.61);;
- : float = 13.

# "un" ^ "deux";;
- : string = "undeux"
```

```
# (3.1 < sqrt(12.)) && true;;
- : bool = true

# (3 + ) 2;;
Error: Syntax error: ')'

# 1 && 1;;
Error: This expression has type int
```

Définition, environnement

Environnement

Un environnement est un ensemble de couples `(nom,valeur)`.
L'environnement initial est constitué des définitions de base. Par exemple dans l'environnement initial le nom `true` est lié à la valeur booléenne `"vrai"`.
On peut enrichir l'environnement en ajoutant des liaisons `(nom,valeur)`. Après quoi, ce nom pourra être utilisé dans les expressions.

Remarque

En OCaml les noms commencent par une lettre minuscule.

Valeur d'une expression

La valeur d'une expression dépend de l'environnement dans lequel elle est évaluée.
`Val(exp, Env)`, la valeur de l'expression `exp` dans l'environnement `Env` est :

- si exp est une constante, `Val(exp, Env)` est la valeur de cette constante
- si `exp` est un nom `ident`
  - si il existe une liaison `(ident, v)` dans `Env`, `Val(exp, Env)=v`
  - sinon il y a une erreur, `Val(exp, Env)` n'est pas défini
- si `exp` est de la forme `(exp1 op exp2)`
  soient `Val(exp1, Env)=v1` et `Val(exp2, Env)=v2`
  `Val(exp, Env)` est le résultat de l'application de l'opération `op` aux opérandes `v1` et `v2`
- si `exp` est de la forme `f(e)`
  soit `Val(e, Env)=v` ;
  `Val(exp, Env)` est le résultat de l'application de la fonction `f` avec l'argument `v`.

Ajout d'une définition à l'environnement

Définition globale

let *ident* = *exp*

où `ident` est un nom et `exp` une expression
L'évaluation de cette définition dans un environnement `Env` consiste à :

- calculer `v=val(exp, Env)`
- ajouter la liaison `(ident,v)` à `Env`
- afficher le nom, le type et la valeur de `ident`

Exemple

```
# let annee = 2021;;
val annee : int = 2021
# annee+1;;
- : int = 2022
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
```

Ajout d'une définition à l'environnement

Définition globale

```
let ident = exp
```

où `ident` est un nom et `exp` une expression
L'évaluation de cette définition dans un environnement `Env` consiste à :

- calculer `v=val(exp, Env)`
- ajouter la liaison `(ident,v)` à `Env`
- afficher le nom, le type et la valeur de `ident`

Exemple

```
# let annee = 2021;;
val annee : int = 2021
# annee+1;;
- : int = 2022
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
```

Ajout d'une définition à l'environnement

Définition globale

```
let ident = exp
```

où `ident` est un nom et `exp` une expression
L'évaluation de cette définition dans un environnement `Env` consiste à :

- calculer `v=val(exp, Env)`
- ajouter la liaison `(ident,v)` à `Env`
- afficher le nom, le type et la valeur de `ident`

Exemple

```
# let annee = 2021;;
val annee : int = 2021
# annee+1;;
- : int = 2022
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
```

Ajout d'une définition à l'environnement

Définition globale

```
let ident = exp
```

où `ident` est un nom et `exp` une expression
L'évaluation de cette définition dans un environnement `Env` consiste à :

- calculer `v=val(exp, Env)`
- ajouter la liaison `(ident,v)` à `Env`
- afficher le nom, le type et la valeur de `ident`

Exemple

```
# let annee = 2021;;
val annee : int = 2021
# annee+1;;
- : int = 2022
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
```

Définition locale

On peut définir un nom localement à une expression

let *ident*= *exp1* **in** *exp2*

où `ident` est un nom, `exp1` et `exp2` sont des expressions
L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env` $\cup$ `{(ident,v1)})`
- afficher `v2` et son type

Remarque : l'environnement `Env` n'est pas modifié.

Exemple

Pour calculer $\frac{\sqrt{2}}{\sqrt{2}+0,5}$ on peut évaluer l'expression OCaml :

```
# let x = sqrt(2.) in x/.(x+.0.5);;
- : float = 0.73879612503625857
# x +. 1.;;
Error: Unbound value x
```

Définition locale

On peut définir un nom localement à une expression

```
let ident= exp1 in exp2
```

où `ident` est un nom, `exp1` et `exp2` sont des expressions
L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env` $\cup$ `{(ident,v1)})`
- afficher `v2` et son type

Remarque : l'environnement `Env` n'est pas modifié.

Exemple

Pour calculer $\frac{\sqrt{2}}{\sqrt{2}+0,5}$ on peut évaluer l'expression OCaml :

```
# let x = sqrt(2.) in x/.(x+.0.5);;
- : float = 0.73879612503625857
# x +. 1.;;
Error: Unbound value x
```

Définition locale

On peut définir un nom localement à une expression

```
let ident= exp1 in exp2
```

où `ident` est un nom, `exp1` et `exp2` sont des expressions
L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env` $\cup$ `{(ident,v1)})`
- afficher `v2` et son type

Remarque : l'environnement `Env` n'est pas modifié.

Exemple

Pour calculer $\frac{\sqrt{2}}{\sqrt{2}+0,5}$ on peut évaluer l'expression OCaml :

```
# let x = sqrt(2.) in x/.(x+.0.5);;
- : float = 0.73879612503625857
# x +. 1.;;
Error: Unbound value x
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Remarque

- Les définitions locales sont des expressions ; on peut les imbriquer
- Il est possible de redéfinir la valeur associée à un identificateur

```
# let x = 2 in let y = x+1 in x+y;;
- : int = 5
# let x = let y = 3 in y*2;;
val x : int = 6
# let annee = 2021;;
val annee : int = 2021
# let anneeDerniere = annee-1;;
val anneeDerniere : int = 2020
# let annee = 2022;;
val annee : int = 2022
# anneeDerniere;;
- : int = 2020
# let annee = true;;
val annee : bool = true
```

Il est possible de regrouper plusieurs liaisons dans une définition :

let *id1* = *exp1* **and** *id2* = *exp2* **in** *exp3*

L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env)`
- calculer et afficher `val(exp3,Env` $\cup$ `{(id1,v1),(id2,v2)})`

```
# let x = 1 and y = 2 in x+y;;
- : int = 3
```

Différence avec l'imbrication des définitions :

```
# let x = 1 in let y = x in x+y;;
- : int 2
# let x = 1 and y = x in x+y;;
 Error: Unbound value x
```

Il est possible de regrouper plusieurs liaisons dans une définition :

```
let id1 = exp1 and id2 = exp2 in exp3
```

L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env)`
- calculer et afficher `val(exp3,Env` $\cup$ `{(id1,v1),(id2,v2)})`

```
# let x = 1 and y = 2 in x+y;;
- : int = 3
```

Différence avec l'imbrication des définitions :

```
# let x = 1 in let y = x in x+y;;
- : int 2
# let x = 1 and y = x in x+y;;
 Error: Unbound value x
```

Il est possible de regrouper plusieurs liaisons dans une définition :

```
let id1 = exp1 and id2 = exp2 in exp3
```

L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env)`
- calculer et afficher `val(exp3,Env` $\cup$ `{(id1,v1),(id2,v2)})`

```
# let x = 1 and y = 2 in x+y;;
- : int = 3
```

Différence avec l'imbrication des définitions :

```
# let x = 1 in let y = x in x+y;;
- : int 2
# let x = 1 and y = x in x+y;;
 Error: Unbound value x
```

Il est possible de regrouper plusieurs liaisons dans une définition :

```
let id1 = exp1 and id2 = exp2 in exp3
```

L'évaluation de cette expression dans un environnement `Env` consiste à :

- calculer `v1=val(exp1,Env)`
- calculer `v2=val(exp2,Env)`
- calculer et afficher `val(exp3,Env` $\cup$ `{(id1,v1),(id2,v2)})`

```
# let x = 1 and y = 2 in x+y;;
- : int = 3
```

Différence avec l'imbrication des définitions :

```
# let x = 1 in let y = x in x+y;;
- : int 2
# let x = 1 and y = x in x+y;;
 Error: Unbound value x
```

Expression conditionnelle

La syntaxe d'une expression conditionnelle est

if *cond* **then** *exp1* **else** *exp2*

- cond est une expression de type `bool`
- exp1 et exp2 sont deux expressions **de même type**.

`val( if cond then exp1 else exp2, Env) =`

`val(exp1,Env)` si `val(cond,Env)=true`

`val(exp2,Env)` si `val(cond,Env)=false`

```
# if 3<4 then 1.2 else 8.1;;
- : float = 1.2
# let x=1;;
val x : int = 1
# let y = if x<0 then "negatif" else "positif ou nul";;
val y : string = "positif ou nul"
```

Expression conditionnelle

La syntaxe d'une expression conditionnelle est

if *cond* **then** *exp1* **else** *exp2*

- cond est une expression de type `bool`
- exp1 et exp2 sont deux expressions **de même type**.

```
val( if cond then exp1 else exp2, Env) =
    val(exp1,Env) si val(cond,Env)=true
    val(exp2,Env) si val(cond,Env)=false
```

```
# if 3<4 then 1.2 else 8.1;;
- : float = 1.2
# let x=1;;
val x : int = 1
# let y = if x<0 then "negatif" else "positif ou nul";;
val y : string = "positif ou nul"
```

Expression conditionnelle

La syntaxe d'une expression conditionnelle est

```
if cond then exp1 else exp2
```

- cond est une expression de type `bool`
- exp1 et exp2 sont deux expressions **de même type**.

```
val( if cond then exp1 else exp2, Env) =
    val(exp1,Env) si val(cond,Env)=true
    val(exp2,Env) si val(cond,Env)=false
```

```
# if 3<4 then 1.2 else 8.1;;
- : float = 1.2
# let x=1;;
val x : int = 1
# let y = if x<0 then "negatif" else "positif ou nul";;
val y : string = "positif ou nul"
```

Plan du cours

Définition de fonction

On peut ajouter à un environnement une nouvelle fonction.

Pour définir la fonction $\begin{array}{rccc} f: & A & \longrightarrow & B \\ & x & \longmapsto & r \end{array}$, on écrit

let $f: A \to B =$ **function**
$x \to r$

- `f` est un identificateur, nom de la fonction
- `x` est un identificateur, le paramètre formel de la fonction
- `A` et `B` sont des noms de type
- `r` est une expression

Lors de l'évaluation de cette définition dans un environnement, OCaml vérifie si, x étant de type `A`, le type de r est bien `B` et ajoute à l'environnement la liaison (f, "la fonction $x \mapsto r$").

Exemples

```
# let double : int → int = function x → 2*x;;
val double : int → int = <fun>
#   double(3);;
-  : int = 6
```

La signature de la fonction n'est pas obligatoire. Si elle n'est pas précisée OCaml la calcule à partir des fonctions et opérations utilisées dans l'expression.

```
# let carre = function x → x *. x;;
val carre : float → float = <fun>
```

Dans un premier temps, on indiquera la signature pour bien spécifier la fonction que l'on souhaite définir.

```
#   carre ;;
-  : float → float = <fun>
```

Le type d'une fonction est sa signature.

Exemples

```
# let double : int → int = function x → 2*x;;
val double : int → int = <fun>
#   double(3);;
- : int = 6
```

La signature de la fonction n'est pas obligatoire. Si elle n'est pas précisée OCaml la calcule à partir des fonctions et opérations utilisées dans l'expression.

```
# let carre = function x → x *. x;;
val carre : float → float = <fun>
```

Dans un premier temps, on indiquera la signature pour bien spécifier la fonction que l'on souhaite définir.

```
#   carre ;;
- : float → float = <fun>
```

Le type d'une fonction est sa signature.

Exemples

```
# let double : int → int = function x → 2*x;;
val double : int → int = <fun>
#   double(3);;
- : int = 6
```

La signature de la fonction n'est pas obligatoire. Si elle n'est pas précisée OCaml la calcule à partir des fonctions et opérations utilisées dans l'expression.

```
# let carre = function x → x *. x;;
val carre : float → float = <fun>
```

Dans un premier temps, on indiquera la signature pour bien spécifier la fonction que l'on souhaite définir.

```
#  carre ;;
- : float → float = <fun>
```

Le type d'une fonction est sa signature.

Fonction à plusieurs paramètres

Produit cartésien

Rappel A et B étant deux ensembles, $A \times B = \{(a, b) \mid a \in A, b \in B\}$.
En OCaml :

- le produit cartésien est noté *****
- **T1 * T2** est le type dont les éléments sont les couples formés d'une valeur de type `T1` et d'une valeur de type `T2`
- **(v1,v2)** est le couple formé par la valeur de `v1` et la valeur de `v2`.

Fonction à plusieurs paramètres

Pour définir la fonction $\begin{array}{rccc} f : & A \times B & \longrightarrow & C \\ & (x, y) & \longmapsto & r \end{array}$ on écrit :

```
let f: A*B -> C = function
               (x,y) -> r
```

Exemples

```
# (1,true);;
- : int * bool = (1, true)
# let moyenne : float * float -> float = function
    (x, y) -> (x +. y) /. 2. ;;
val moyenne : float * float -> float = <fun>
# moyenne(1.5, 2.);;
- : float = 1.75
# (* teste si la moyenne de 2 reels est positive *)
  let moyPositive : float * float -> bool = function
    (x, z) -> moyenne(x, z) > 0.;;
val moyPositive : float * float -> bool = <fun>
# moyPositive(-2.5, sqrt(5.));;
- : bool = false
```

Exemples

```
# (1,true);;
- : int * bool = (1, true)
# let moyenne : float * float -> float = function
      (x, y) -> (x +. y) /. 2. ;;
val moyenne : float * float -> float = <fun>
# moyenne(1.5, 2.);;
- : float = 1.75
# (* teste si la moyenne de 2 reels est positive *)
  let moyPositive : float * float -> bool = function
      (x, z) -> moyenne(x, z) > 0.;;
val moyPositive : float * float -> bool = <fun>
# moyPositive(-2.5, sqrt(5.));;
- : bool = false
```

Exemples

```
# (1,true);;
- : int * bool = (1, true)
# let moyenne : float * float -> float = function
      (x, y) -> (x +. y) /. 2. ;;
val moyenne : float * float -> float = <fun>
# moyenne(1.5, 2.);;
- : float = 1.75
# (* teste si la moyenne de 2 reels est positive *)
  let moyPositive : float * float -> bool = function
      (x, z) -> moyenne(x, z) > 0.;;
val moyPositive : float * float -> bool = <fun>
# moyPositive(-2.5, sqrt(5.));;
- : bool = false
```

Exemples

```
# (1,true);;
- : int * bool = (1, true)
# let moyenne : float * float -> float = function
    (x, y) -> (x +. y) /. 2. ;;
val moyenne : float * float -> float = <fun>
# moyenne(1.5, 2.);;
- : float = 1.75
# (* teste si la moyenne de 2 reels est positive *)
  let moyPositive : float * float -> bool = function
    (x, z) -> moyenne(x, z) > 0.;;
val moyPositive : float * float -> bool = <fun>
# moyPositive(-2.5, sqrt(5.));;
- : bool = false
```

Exemples

```
# (1,true);;
- : int * bool = (1, true)
# let moyenne : float * float -> float = function
      (x, y) -> (x +. y) /. 2. ;;
val moyenne : float * float -> float = <fun>
# moyenne(1.5, 2.);;
- : float = 1.75
# (* teste si la moyenne de 2 reels est positive *)
  let moyPositive : float * float -> bool = function
      (x, z) -> moyenne(x, z) > 0.;;
val moyPositive : float * float -> bool = <fun>
# moyPositive(-2.5, sqrt(5.));;
- : bool = false
```

La récurrence

La récurrence est un procédé classique pour définir des objets ou des fonctions.

Définition

On définit `objet` par récurrence en spécifiant :

- Les **cas de base** : des constantes donnant la valeur de `objet` sans faire appel à la récurrence.
- Les **équations de récurrence** : des définitions de `objet` qui font appel à d'autres valeurs de `objet`.

Une récurrence sera dite **correcte** si dans l'utilisation des équations de récurrence, la suite des appels récursifs est finie. C'est à dire si le processus d'appel de la définition, qui appelle la définition avec une deuxième valeur, qui appelle la définition avec une troisième valeur, qui ..., se termine toujours.

La récurrence

La récurrence est un procédé classique pour définir des objets ou des fonctions.

Définition

On définit `objet` par récurrence en spécifiant :

- Les **cas de base** : des constantes donnant la valeur de `objet` sans faire appel à la récurrence.
- Les **équations de récurrence** : des définitions de `objet` qui font appel à d'autres valeurs de `objet`.

Une récurrence sera dite **correcte** si dans l'utilisation des équations de récurrence, la suite des appels récursifs est finie. C'est à dire si le processus d'appel de la définition, qui appelle la définition avec une deuxième valeur, qui appelle la définition avec une troisième valeur, qui ..., se termine toujours.

La récurrence

La récurrence est un procédé classique pour définir des objets ou des fonctions.

Définition

On définit `objet` par récurrence en spécifiant :

- Les **cas de base** : des constantes donnant la valeur de `objet` sans faire appel à la récurrence.
- Les **équations de récurrence** : des définitions de `objet` qui font appel à d'autres valeurs de `objet`.

Une récurrence sera dite **correcte** si dans l'utilisation des équations de récurrence, la suite des appels récursifs est finie. C'est à dire si le processus d'appel de la définition, qui appelle la définition avec une deuxième valeur, qui appelle la définition avec une troisième valeur, qui ..., se termine toujours.

Exemple

Je définis `ExprF` par récurrence, en disant :
Un `ExprF` est soit :

- Cas de Base1 : un nombre
- Cas de Base2 : un identificateur
- Équation de récurrence1 : **(**`ExprF ExprF`**)**
- Équation de récurrence2 : **(fun** `id` $\rightarrow$ `ExprF`**)** où `id` est un identificateur

Par application de la définition on reconnaît les phrases qui

- sont des `ExprF` :
  `sqrt ; 8 ; (x y) ; (sqrt 8) ; (fun x` $\rightarrow$ `(sqrt x)) ;`
  `((fun x` $\rightarrow$ `x) 8) ; (fun x` $\rightarrow$ `(fun y` $\rightarrow$ `(x y)))`
- ne sont pas des `ExprF` :
  `(f 8 2) ; (x + 3) ; (fun (x y)` $\rightarrow$ `x)`

Définition récursive d'une fonction

Ce procédé peut être utilisé pour définir des fonctions.

Factorielle $(n) = 1 \times 2 \times \cdots \times (n-1) \times n$

La fonction `factorielle` peut être définie par récurrence.
n étant un entier naturel, la valeur de `factorielle(n)` est :

- Cas de Base : quand $n = 0$, `factorielle(n)` vaut `1`
- Équation de récurrence : quand $n > 0$ `factorielle(n)` vaut `n * factorielle( n-1 )`

On obtient la définition :

$$\begin{array}{rccll} \text{factorielle} : & \mathbb{N} & \longrightarrow & \mathbb{N} & \\ & n & \longmapsto & 1 & \text{si } n=0 \\ & & & n*\text{factorielle}(n-1) & \text{sinon} \end{array}$$

Définition récursive d'une fonction

Ce procédé peut être utilisé pour définir des fonctions.

Factorielle $(n) = 1 \times 2 \times \cdots \times (n-1) \times n$

La fonction `factorielle` peut être définie par récurrence.
n étant un entier naturel, la valeur de `factorielle(n)` est :

- Cas de Base : quand $n = 0$, `factorielle(n)` vaut `1`
- Équation de récurrence : quand $n > 0$ `factorielle(n)` vaut `n * factorielle( n-1 )`

On obtient la définition :

$$\begin{array}{rccll} \text{factorielle} : & \mathbb{N} & \longrightarrow & \mathbb{N} & \\ & n & \longmapsto & 1 & \text{si } n=0 \\ & & & n*\text{factorielle}(n-1) & \text{sinon} \end{array}$$

Définition récursive d'une fonction

Ce procédé peut être utilisé pour définir des fonctions.

Factorielle $(n) = 1 \times 2 \times \cdots \times (n-1) \times n$

La fonction `factorielle` peut être définie par récurrence.
n étant un entier naturel, la valeur de `factorielle(n)` est :

- Cas de Base : quand $n = 0$, `factorielle(n)` vaut `1`
- Équation de récurrence : quand $n > 0$ `factorielle(n)` vaut `n * factorielle( n-1 )`

On obtient la définition :

$$\begin{array}{rccll} \text{factorielle} : & \mathbb{N} & \longrightarrow & \mathbb{N} & \\ & n & \longmapsto & 1 & \text{si } n=0 \\ & & & n*\text{factorielle}(n-1) & \text{sinon} \end{array}$$

Définition récursive d'une fonction

Ce procédé peut être utilisé pour définir des fonctions.

Factorielle $(n) = 1 \times 2 \times \cdots \times (n-1) \times n$

La fonction `factorielle` peut être définie par récurrence.
n étant un entier naturel, la valeur de `factorielle(n)` est :

- Cas de Base : quand $n = 0$, `factorielle(n)` vaut `1`
- Équation de récurrence : quand $n > 0$ `factorielle(n)` vaut `n * factorielle( n-1 )`

On obtient la définition :

$$\begin{array}{lccll} \text{factorielle} : & \mathbb{N} & \longrightarrow & \mathbb{N} & \\ & \text{n} & \longmapsto & 1 & \text{si n=0} \\ & & & \text{n*factorielle(n-1)} & \text{sinon} \end{array}$$

Traduction en OCaml

On traduit cette définition en OCaml en indiquant avec le mot-clé **rec** que la définition est récursive :

```
# let rec factorielle: int → int = function
    n → if n=0 then 1
          else n*factorielle(n-1);;
val factorielle : int → int = <fun>
# factorielle(5);;
- : int = 120
```

La fonction `factorielle` est définie sur $\mathbb{N}$ alors que le type `int` correspond à $\mathbb{Z}$. Pour que la définition de la fonction OCaml soit totale on peut préciser qu'il y a une erreur lorsque l'argument est un entier négatif.

```
# let rec factorielle: int → int = function
    n → if n<0 then (failwith "factorielle")
          else if n=0 then 1
          else n*factorielle(n-1);;
# factorielle (-3);;
Exception: Failure "factorielle".
```

Ce problème sera abordé en fin de cours (partie exceptions).

Traduction en OCaml

On traduit cette définition en OCaml en indiquant avec le mot-clé **rec** que la définition est récursive :

```
# let rec factorielle: int → int = function
    n → if n=0 then 1
          else n*factorielle(n-1);;
val factorielle : int → int = <fun>
# factorielle(5);;
- : int = 120
```

La fonction `factorielle` est définie sur $\mathbb{N}$ alors que le type `int` correspond à $\mathbb{Z}$. Pour que la définition de la fonction OCaml soit totale on peut préciser qu'il y a une erreur lorsque l'argument est un entier négatif.

```
# let rec factorielle: int → int = function
    n → if n<0 then (failwith "factorielle")
          else if n=0 then 1
          else n*factorielle(n-1);;
 # factorielle (-3);;
Exception: Failure "factorielle".
```

Ce problème sera abordé en fin de cours (partie exceptions).

Arrêt d'un calcul récursif

Arrêt du calcul de factorielle

Pour toute valeur $n \in \mathbb{N}$ le calcul de `factorielle(n)` termine. En effet la suite des valeurs de l'argument est la suite `n, n-1, n-2, ...` décroissante, admettant pour minimum 0. Et la valeur de `factorielle(0)` est définie directement (sans récurrence) par le cas de base.

Exemple de mauvaise définition récursive

La fonction g, dont l'argument est $n \in \mathbb{Z}$, est définie récursivement par :

- Cas de Base : quand $n = 0$ `g(n)` vaut 1
- Équation de récurrence : quand $n \neq 0$ `g(n)` vaut `n * g(n-2)`

La manière de spécifier ceci en Mathématiques pour `g(n)` est

$$g : \begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ n & \longmapsto & \begin{cases} 1 & \text{si n=0} \\ n*g(n-2) & \text{sinon} \end{cases} \end{array}$$

La définition semble correcte pour un entier positif. C'est le cas pour un entier pair, mais pour un entier impair le calcul ne s'arrête pas :

`g(3) = 3*g(1) = 3*(1*g(-1)) = 3*(1*(-1*g(-3)))= ...`

Arrêt d'un calcul récursif

Arrêt du calcul de factorielle

Pour toute valeur $n \in \mathbb{N}$ le calcul de `factorielle(n)` termine. En effet la suite des valeurs de l'argument est la suite `n, n-1, n-2, ...` décroissante, admettant pour minimum 0. Et la valeur de `factorielle(0)` est définie directement (sans récurrence) par le cas de base.

Exemple de mauvaise définition récursive

La fonction g, dont l'argument est $n \in \mathbb{Z}$, est définie récursivement par :

- Cas de Base : quand $n = 0$ `g(n)` vaut `1`
- Équation de récurrence : quand $n \neq 0$ `g(n)` vaut `n * g(n-2)`

La manière de spécifier ceci en Mathématiques pour `g(n)` est

$$\begin{array}{rccll} g: & \mathbb{Z} & \longrightarrow & \mathbb{Z} & \\ & n & \longmapsto & 1 & \text{si } n=0 \\ & & & n*g(n-2) & \text{sinon} \end{array}$$

La définition semble correcte pour un entier positif. C'est le cas pour un entier pair, mais pour un entier impair le calcul ne s'arrête pas :

`g(3) = 3*g(1) = 3*(1*g(-1)) = 3*(1*(-1*g(-3)))= ...`

Arrêt d'un calcul récursif

Arrêt du calcul de factorielle

Pour toute valeur $n \in \mathbb{N}$ le calcul de `factorielle(n)` termine. En effet la suite des valeurs de l'argument est la suite `n, n-1, n-2, ...` décroissante, admettant pour minimum 0. Et la valeur de `factorielle(0)` est définie directement (sans récurrence) par le cas de base.

Exemple de mauvaise définition récursive

La fonction g, dont l'argument est $n \in \mathbb{Z}$, est définie récursivement par :

- Cas de Base : quand $n = 0$ `g(n)` vaut `1`
- Équation de récurrence : quand $n \neq 0$ `g(n)` vaut `n * g(n-2)`

La manière de spécifier ceci en Mathématiques pour `g(n)` est

$$
\begin{array}{lccll}
g: & \mathbb{Z} & \longrightarrow & \mathbb{Z} & \\
 & n & \longmapsto & 1 & \text{si } n=0 \\
 & & & n*g(n-2) & \text{sinon}
\end{array}
$$

La définition semble correcte pour un entier positif. C'est le cas pour un entier pair, mais pour un entier impair le calcul ne s'arrête pas :

`g(3) = 3*g(1) = 3*(1*g(-1)) = 3*(1*(-1*g(-3)))= ...`

Schéma des fonctions récursives sur les entiers naturels

Un schéma fréquent et correct de définition récursive d'une fonction

$$\begin{array}{rccc} f: & \mathbb{N}\times\ldots & \longrightarrow & \ldots \\ & (n,\ldots) & \longmapsto & \ldots \end{array}$$ est :

- **Cas de base** On indique la valeur de $V_B = f(n,\ldots)$ quand l'argument n vaut 0 ou, parfois, 1 ....
- **Récurrence** Quand n est différent du cas de base, on définit la valeur de $f(n,..)$ en fonction de la valeur de la fonction f pour $n-1$.

D'où le schéma fréquent

$$\begin{array}{rccll} \text{f}: & \mathbb{N}\times\ldots & \longrightarrow & \ldots & \\ & \text{(n,..)} & \longmapsto & V_B & \textbf{si n=0} \\ & & & \ldots f(n-1,\ldots)\ldots & \textbf{sinon} \end{array}$$

Ce schéma est correct car en l'appliquant la suite des arguments de f est une suite d'entiers naturels strictement décroissante et donc finie.

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{lccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, somCarrés(n) vaut 0
- **Récurrence** : quand $n > 0$ comment définir somCarrés(n) en fonction de somCarrés(n-1) ?

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?

somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
$0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int -> int = function
    n ->
        if n=0 then 0
        else n*n + somCarres (n-1);;
```

somCarrés(2)=

Exemples de définitions récursives

Somme des carrés

somCarrés : $\begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=

Exemples de définitions récursives

Somme des carrés

somCarrés : $\begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int -> int = function
    n ->
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int -> int = function
    n ->
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=2*2+somCarrés(1)=4+1*1+somCarrés(0)=4+1+0=5

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=2*2+somCarrés(1)=4+1*1+somCarrés(0)=4+1+0=5

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=2*2+somCarrés(1)=4+1*1+somCarrés(0)=4+1+0=5

Exemples de définitions récursives

Somme des carrés

$$\begin{array}{rccc} \text{somCarrés} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & 0^2 + 1^2 + 2^2 + \cdots + n^2 \end{array}$$

- **Cas de base** : quand $n = 0$, `somCarrés(n)` vaut 0
- **Récurrence** : quand $n > 0$ comment définir `somCarrés(n)` en fonction de `somCarrés(n-1)` ?
  somCarrés(n) $= 0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$
  $0^2 + 1^2 + \cdots + (n-1)^2 + n^2 =$ somCarrés(n-1)$+n^2$

```
let rec somCarres : int → int = function
    n →
      if n=0 then 0
      else n*n + somCarres (n-1);;
```

somCarrés(2)=2*2+somCarrés(1)=4+1*1+somCarrés(0)=4+1+0=5

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \quad \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$$\begin{array}{rccl} nbPairs: & \mathbb{Z}\times\mathbb{Z} & \longrightarrow & \mathbb{N} \\ & (a,b) & \longmapsto & \text{Nombre d'entiers pairs dans l'intervalle}[a,b] \end{array}$$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=

Nombre d'entiers pairs

$$\begin{array}{rccl} nbPairs: & \mathbb{Z}\times\mathbb{Z} & \longrightarrow & \mathbb{N} \\ & (a,b) & \longmapsto & \text{Nombre d'entiers pairs dans l'intervalle}[a,b] \end{array}$$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

Nombre d'entiers pairs

$nbPairs : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$

$(a, b) \longmapsto$ Nombre d'entiers pairs dans l'intervalle$[a, b]$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

Nombre d'entiers pairs

$$\begin{array}{rccl} nbPairs: & \mathbb{Z}\times\mathbb{Z} & \longrightarrow & \mathbb{N} \\ & (a,b) & \longmapsto & \text{Nombre d'entiers pairs dans l'intervalle}[a,b] \end{array}$$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

Nombre d'entiers pairs

$$\begin{array}{rccl} nbPairs: & \mathbb{Z}\times\mathbb{Z} & \longrightarrow & \mathbb{N} \\ & (a,b) & \longmapsto & \text{Nombre d'entiers pairs dans l'intervalle}[a,b] \end{array}$$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

Nombre d'entiers pairs

$$\begin{array}{rccl} nbPairs: & \mathbb{Z}\times\mathbb{Z} & \longrightarrow & \mathbb{N} \\ & (a,b) & \longmapsto & \text{Nombre d'entiers pairs dans l'intervalle}[a,b] \end{array}$$

- **Cas de base** quand $b < a$ nbPairs(a,b) vaut 0
- **Récurrence** quand $a \leqslant b$, on a 2 cas selon la parité de b :
  - quand b est pair, nbPairs(a,b) vaut 1+nbpairs(a,b-1)
  - quand b est impair, nbPairs(a,b) vaut nbpairs(a,b-1)

```
let rec nbPairs: int*int -> int = function
    (a,b) ->
        if b<a then 0
        else if b mod 2=0 then 1 + nbPairs(a, b-1)
        else nbPairs(a,b-1);;
```

nbPairs(2,4)=1+nbPairs(2,3)=1+nbPairs(2,2)=1+1+nbPairs(2,1)=1+1+0=2

PGCD de 2 entiers

Rappels

Soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$.

- Il existe de manière unique 2 entiers q et r tels que $a = bq + r$ avec $0 \leqslant r < b$.
- $q = a/b$, $r = a \ mod \ b$.
- a divise b si `b mod a = 0`
- Tout nombre a au moins pour diviseurs : 1 et lui même.
- Le plus grand diviseur commun à deux entiers est appelé leur pgcd.

Les diviseurs de 12 sont 1,2,3,4,6,12
Les diviseurs de 30 sont 1,2,3,5,6,10,15,30
pgcd(12,30) est 6

PGCD de 2 entiers

Rappels

Soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$.

- Il existe de manière unique 2 entiers q et r tels que $a = bq + r$ avec $0 \leqslant r < b$.
- $q = a/b$, $r = a \ mod \ b$.
- a divise `b` si `b mod a = 0`
- Tout nombre a au moins pour diviseurs : 1 et lui même.
- Le plus grand diviseur commun à deux entiers est appelé leur pgcd.

Les diviseurs de 12 sont 1,2,3,4,6,12
Les diviseurs de 30 sont 1,2,3,5,6,10,15,30
`pgcd(12,30)` est 6

PGCD de 2 entiers

Rappels

Soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$.

- Il existe de manière unique 2 entiers q et r tels que $a = bq + r$ avec $0 \leqslant r < b$.
- $q = a/b$, $r = a \ mod \ b$.
- a divise `b` si `b mod a = 0`
- Tout nombre a au moins pour diviseurs : 1 et lui même.
- Le plus grand diviseur commun à deux entiers est appelé leur pgcd.

Les diviseurs de 12 sont 1,2,3,4,6,12
Les diviseurs de 30 sont 1,2,3,5,6,10,15,30
`pgcd(12,30)` est 6

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Propriété

Le pgcd de a et b est aussi pgcd de b et r.
(car x divise a et b si et seulement si x divise b et r).

Récurrence

- Pour calculer `pgcd( a , b )` on calcule `pgcd( b , a mod b )`.
- Quel est le cas de base ?
  Cas où la récurrence n'est pas applicable : $b = 0$.
  Quelle est la valeur de `pgcd(a,0)` ? a

En OCaml

```
let rec pgcd: int*int -> int = function
    (a,b) ->
        if b=0 then a
        else pgcd (b, a mod b);;
```

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de pgcd(...,0).

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1

pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3

pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument *b* est remplacé par *a mod b* qui est un entier naturel strictement inférieur à *b*. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Schéma de récurrence correct ?

Dans cet exemple, l'équation de récurrence ne lie pas pgcd(...,b) à pgcd(...,b-1). Cependant l'arrêt est assuré car à chaque appel récursif, l'argument b est remplacé par $a \ mod \ b$ qui est un entier naturel strictement inférieur à b. La suite des valeurs du $2^{ème}$ argument est décroissante admettant pour minimum 0. Et on connaît la valeur de `pgcd(...,0)`.

Suite des appels

pgcd(25,9)=pgcd(9,25 mod 9)=pgcd(9,7)=pgcd(7,2)=pgcd(2,1)=pgcd(1,0)=1
pgcd(30,9)= pgcd(9,3)= pgcd(3,0)= 3
pgcd(9,30)=pgcd(30,9)= ...=3

Retour sur les fonctions

Portée des identificateurs

Le corps de la définition d'une fonction peut contenir des identificateurs. Hormis ceux correspondant aux paramètres formels de la fonction, ces identificateurs doivent être définis dans l'environnement, global ou local.

```
let f : int → int = function
        x → let y = 5 in h(x)+y+z;;
```

Dans cette définition

- `x` est un paramètre formel
- `y` est lié au nombre 5, de type `int`, dans l'environnement local de la fonction
- `h` et `z` sont deux identificateurs définis en dehors de la fonction. Pour qu'il n'y ait pas d'erreur, dans l'environnement global
  - `z` doit être lié à un nombre de type `int`
  - `h` à une fonction de type $int \rightarrow int$.

Que se passe-t-il si après avoir défini `f`, on redéfinit les identificateurs `z` et/ou `h` ?

Retour sur les fonctions

Portée des identificateurs

Le corps de la définition d'une fonction peut contenir des identificateurs. Hormis ceux correspondant aux paramètres formels de la fonction, ces identificateurs doivent être définis dans l'environnement, global ou local.

```
let f : int → int = function
        x → let y = 5 in h(x)+y+z;;
```

Dans cette définition

- `x` est un paramètre formel
- `y` est lié au nombre 5, de type `int`, dans l'environnement local de la fonction
- `h` et `z` sont deux identificateurs définis en dehors de la fonction. Pour qu'il n'y ait pas d'erreur, dans l'environnement global
  - `z` doit être lié à un nombre de type `int`
  - `h` à une fonction de type $int \to int$.

Que se passe-t-il si après avoir défini `f`, on redéfinit les identificateurs `z` et/ou `h` ?

Retour sur les fonctions

Portée des identificateurs

Le corps de la définition d'une fonction peut contenir des identificateurs. Hormis ceux correspondant aux paramètres formels de la fonction, ces identificateurs doivent être définis dans l'environnement, global ou local.

```
let f: int → int = function
                   x → let y = 5 in h(x)+y+z;;
```

Dans cette définition

- `x` est un paramètre formel
- `y` est lié au nombre 5, de type `int`, dans l'environnement local de la fonction
- `h` et `z` sont deux identificateurs définis en dehors de la fonction. Pour qu'il n'y ait pas d'erreur, dans l'environnement global
  - `z` doit être lié à un nombre de type `int`
  - `h` à une fonction de type $int \to int$.

Que se passe-t-il si après avoir défini `f`, on redéfinit les identificateurs `z` et/ou `h` ?

Liaison statique

La liaison est effectuée à la définition de la fonction et non lors de son application.

```
# let z = 2;;
# let h: int → int = function x → x+1;;
# let f: int → int = function
        x → let y=5 in h(x)+y+z;;
# f(1);;
- : int = 9
# let h: int → int = function x → x+2;;
# let z=9;;
# f(1);;
- : int = 9
```

On dit que la portée des identificateurs est statique car l'environnement pour se passer comme est lors de la définition d'une fonction et non lors de l'application de la fonction et l'environnement au moment de l'exécution.

Liaison statique

La liaison est effectuée à la définition de la fonction et non lors de son application.

```
# let z = 2;;
# let h: int → int = function x → x+1;;
# let f: int → int = function
            x → let y=5 in h(x)+y+z;;
# f(1);;
- : int = 9
# let h: int → int = function x → x+2;;
# let z=9;;
# f(1);;
- : int = 9
```

On dit que la portée des identificateurs est statique et non dynamique.
Tout se passe comme si lors de la définition d'une fonction on mémorisait le nom, le corps de la fonction et l'environnement au moment de sa définition.

Liaison statique

La liaison est effectuée à la définition de la fonction et non lors de son application.

```
# let z = 2;;
# let h: int -> int = function x -> x+1;;
# let f: int -> int = function
        x -> let y=5 in h(x)+y+z;;
# f(1);;
- : int = 9
# let h: int -> int = function x -> x+2;;
# let z=9;;
# f(1);;
- : int = 9
```

On dit que la portée des identificateurs est **statique** et non dynamique.
Tout se passe comme si lors de la définition d'une fonction, on mémorisait le nom, le corps de la fonction et l'environnement au moment de sa définition.

Expression fonctionnelle

En OCaml, tout ou presque est expression. La syntaxe d'une définition est :

```
let nom = expression
```

que l'on définisse une constante ou une fonction et donc

```
function x → expression
```

est une expression, utilisable comme toute autre expression.
Son type est sa signature et sa valeur la fonction (son corps).

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

Ce qui importe dans une fonction c'est son corps (la correspondance qu'elle définit) et non son nom. On peut appliquer une fonction sans la nommer en écrivant :

```
# (function x → x+1) (4);;
- : int = 5
```

Simplication d'écriture des expressions

L'expression de l'application d'une fonction peut s'écrire **f x** plutôt que **f(x)**.

```
# (function x -> x+1) 4;;
- : int = 5
```

Attention aux parenthèses et à l'ordre d'application des fonctions !
Le parenthésage implicite correspond à l'associativité à gauche :

- **f a b** est interprété comme **(f a) b** (et donc f(a) (b))
- **f a op b**, où **op** est le nom d'une opération,
  est interprété comme **(f a) op b)** et donc comme **f(a) op b**.

```
# let f: int -> int = function x -> x+1;;
val f : int -> int = <fun>
# f 2*3;;
- : int = 9
# f(2)*3;;
- : int = 9
# f(2*3);;
- : int = 7
```

Simplication d'écriture des expressions

L'expression de l'application d'une fonction peut s'écrire **f x** plutôt que **f(x)**.

```
# (function x -> x+1) 4;;
- : int = 5
```

Attention aux parenthèses et à l'ordre d'application des fonctions !
Le parenthésage implicite correspond à l'associativité à gauche :

- **f a b** est interprété comme **(f a) b** (et donc f(a) (b))
- **f a op b**, où **op** est le nom d'une opération,
  est interprété comme **(f a) op b)** et donc comme **f(a) op b**.

```
# let f: int -> int = function x -> x+1;;
val f : int -> int = <fun>
# f 2*3;;
- : int = 9
# f(2)*3;;
- : int = 9
# f(2*3);;
- : int = 7
```

Simplication d'écriture des expressions

L'expression de l'application d'une fonction peut s'écrire **f x** plutôt que **f(x)**.

```
# (function x → x+1) 4;;
- : int = 5
```

Attention aux parenthèses et à l'ordre d'application des fonctions !
Le parenthésage implicite correspond à l'associativité à gauche :

- **f a b** est interprété comme **(f a) b** (et donc f(a) (b))
- **f a op b**, où **op** est le nom d'une opération,
  est interprété comme **(f a) op b)** et donc comme **f(a) op b**.

```
# let f: int → int = function x → x+1;;
val f : int → int = <fun>
# f 2*3;;
- : int = 9
# f(2)*3;;
- : int = 9
# f(2*3);;
- : int = 7
```

Simplication d'écriture des expressions

L'expression de l'application d'une fonction peut s'écrire **f x** plutôt que **f(x)**.

```
# (function x -> x+1) 4;;
- : int = 5
```

Attention aux parenthèses et à l'ordre d'application des fonctions !
Le parenthésage implicite correspond à l'associativité à gauche :

- **f a b** est interprété comme **(f a) b** (et donc f(a) (b))
- **f a op b**, où **op** est le nom d'une opération,
  est interprété comme **(f a) op b)** et donc comme **f(a) op b**.

```
# let f: int -> int = function x -> x+1;;
val f : int -> int = <fun>
# f 2*3;;
- : int = 9
# f(2)*3;;
- : int = 9
# f(2*3);;
- : int = 7
```

Arité des fonctions

Nous avons l'habitude d'utiliser la convention précédente pour les fonctions opérant sur un produit cartésien.
Par exemple, nous écrivons **pgcd(9,6)** alors qu'il faudrait écrire **pgcd((9,6))**, puisque le couple s'écrit **(9,6)**.
En fait, la fonction `pgcd` est une fonction qui n'a qu'un paramètre, un couple, élément d'un produit cartésien.
En OCaml, il est d'usage de définir des fonctions d'arité 1 sans utiliser systématiquement le produit cartésien. Pour cela, on définit une fonction dont le résultat est une fonction.
Une fonction dont la signature est $A \times B \to C$ pourra être définie comme une fonction de signature $A \to B \to C$, c'est à dire une fonction, qui étant donné un paramètre de type A renvoie une fonction de signature $B \to C$.

$$A \to (B \to C)$$

Exemple

La fonction `maxi` calculant la plus grande valeur parmi 2 entiers peut être définie avec les deux signatures :

maxi_a : $int \times int \to int$ ou maxi_b : $int \to int \to int$

Arité des fonctions

Nous avons l'habitude d'utiliser la convention précédente pour les fonctions opérant sur un produit cartésien.
Par exemple, nous écrivons **pgcd(9,6)** alors qu'il faudrait écrire **pgcd((9,6))**, puisque le couple s'écrit **(9,6)**.
En fait, la fonction `pgcd` est une fonction qui n'a qu'un paramètre, un couple, élément d'un produit cartésien.
En OCaml, il est d'usage de définir des fonctions d'arité 1 sans utiliser systématiquement le produit cartésien. Pour cela, on définit une fonction dont le résultat est une fonction.
Une fonction dont la signature est $A \times B \to C$ pourra être définie comme une fonction de signature $A \to B \to C$, c'est à dire une fonction, qui étant donné un paramètre de type A renvoie une fonction de signature $B \to C$.

$$A \to (B \to C)$$

Exemple

La fonction `maxi` calculant la plus grande valeur parmi 2 entiers peut être définie avec les deux signatures :

maxi_a : $int \times int \to int$ ou maxi_b : $int \to int \to int$

Arité des fonctions

Nous avons l'habitude d'utiliser la convention précédente pour les fonctions opérant sur un produit cartésien.
Par exemple, nous écrivons **pgcd(9,6)** alors qu'il faudrait écrire **pgcd((9,6))**, puisque le couple s'écrit **(9,6)**.
En fait, la fonction `pgcd` est une fonction qui n'a qu'un paramètre, un couple, élément d'un produit cartésien.
En OCaml, il est d'usage de définir des fonctions d'arité 1 sans utiliser systématiquement le produit cartésien. Pour cela, on définit une fonction dont le résultat est une fonction.
Une fonction dont la signature est $A \times B \to C$ pourra être définie comme une fonction de signature $A \to B \to C$, c'est à dire une fonction, qui étant donné un paramètre de type A renvoie une fonction de signature $B \to C$.

$$A \to (B \to C)$$

Exemple

La fonction `maxi` calculant la plus grande valeur parmi 2 entiers peut être définie avec les deux signatures :

maxi_a : $int \times int \to int$ ou maxi_b : $int \to int \to int$

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

Suite de l'exemple

Voici les deux définitions OCaml correspondantes

```
# let maxi_a : int*int -> int =
    function (x,y) -> if x<y then y else x;;
val maxi_a : int * int -> int = <fun>
# let maxi_b: int -> int -> int =
    function x -> function y -> if x<y then y else x;;
val maxi_b : int -> int -> int = <fun>
# maxi_a(5,2);;
- : int = 5
# maxi_b 5 2;;
- : int = 5
```

maxi_b 5 est la fonction qui fait correspondre à x le maximum entre 5 et x.

```
# maxi_b 5;;
- : int -> int = <fun>
```

Alors que **maxi_a 5** est une expression erronée.

```
# maxi_a 5;;
This expression has type int but int * int was expected
```

- Pour alléger l'écriture la séquence : *function x → function y →* peut être remplacée par : *fun x y →*

```
# let maxi_b: int → int → int =
    fun x y → if x<y then y else x;;
```

- La fonction `max` est prédéfinie en OCaml. Elle correspond à la version `maxi_b` à ceci près qu'elle opère sur n'importe quel type, comme les opérateurs de comparaison.

```
# max;;
- : 'a → 'a → 'a = <fun>
# max 7.5;;
- : float → float = <fun>
# max "un" "deux";;
- : string = "un"
```

On aurait pu la définir en écrivant :

```
# let max = fun x y →
      if x<y then y else x;;
val max : 'a → 'a → 'a = <fun>
```

- Pour alléger l'écriture la séquence : *function x → function y →* peut être remplacée par : *fun x y →*

```
# let maxi_b: int → int → int =
    fun x y → if x<y then y else x;;
```

- La fonction `max` est prédéfinie en OCaml. Elle correspond à la version `maxi_b` à ceci près qu'elle opère sur n'importe quel type, comme les opérateurs de comparaison.

```
# max;;
- : 'a → 'a → 'a = <fun>
# max 7.5;;
- : float → float = <fun>
# max "un" "deux";;
- : string = "un"
```

On aurait pu la définir en écrivant :

```
# let max = fun x y →
      if x<y then y else x;;
val max : 'a → 'a → 'a = <fun>
```

- Pour alléger l'écriture la séquence : *function x → function y →* peut être remplacée par : *fun x y →*

```
# let maxi_b: int → int → int =
    fun x y → if x<y then y else x;;
```

- La fonction `max` est prédéfinie en OCaml. Elle correspond à la version `maxi_b` à ceci près qu'elle opère sur n'importe quel type, comme les opérateurs de comparaison.

```
# max;;
- : 'a → 'a → 'a = <fun>
# max 7.5;;
- : float → float = <fun>
# max "un" "deux";;
- : string = "un"
```

On aurait pu la définir en écrivant :

```
# let max = fun x y →
      if x<y then y else x;;
  val max : 'a → 'a → 'a = <fun>
```

Plan du cours

Le type liste

Notion de liste

- Une `liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
- Une liste permet de regrouper des valeurs et de les traiter comme un tout : par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif.

Définition récursive des listes

Une liste est

- soit la liste vide
- soit une liste non vide qui est un couple composé :

Le type liste

Notion de liste

- Une `liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
- Une liste permet de regrouper des valeurs et de les traiter comme un tout : par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif.

Définition récursive des listes

Une liste est

- soit la liste vide
- soit une liste non vide qui est un couple composé :

Le type liste

Notion de liste

- Une `liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
- Une liste permet de regrouper des valeurs et de les traiter comme un tout : par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif.

Définition récursive des listes

Une liste est

- soit la liste vide
- soit une liste non vide qui est un couple composé :
  - du premier élément de la liste (appelé *Tête de la liste*)
  - et de la liste constituée des autres éléments (le 2ème, le 3ème, ...) (appelée *Queue de la liste*)

Le type liste

Notion de liste

- Une `liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
- Une liste permet de regrouper des valeurs et de les traiter comme un tout : par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif.

Définition récursive des listes

Une liste est

- soit la liste vide
- soit une liste non vide qui est un couple composé :
  - du premier élément de la liste (appelé *Tête de la liste*)
  - et de la liste constituée des autres éléments (le 2ème, le 3ème, ...) (appelée *Queue de la liste*)

Le type liste

Notion de liste

- Une `liste` est une structuration des données correspondant à une séquence de valeurs toutes de même type et de longueur arbitraire.
- Une liste permet de regrouper des valeurs et de les traiter comme un tout : par exemple une série de relevés de températures pourra être représentée par une liste de nombre réels.
- L'ordre des éléments d'une liste est significatif.

Définition récursive des listes

Une liste est

- soit la liste vide
- soit une liste non vide qui est un couple composé :
  - du premier élément de la liste (appelé *Tête de la liste*)
  - et de la liste constituée des autres éléments (le 2ème, le 3ème, ...) (appelée *Queue de la liste*)

Les listes en OCaml

- **'a list** est le type des listes dont les éléments sont de type **'a**.
  Par exemple **int list** est le type des listes d'entiers
- **Domaine** : une liste est représentée par la séquence encadrée par des crochets de ses éléments séparés par des points-virgules.
  Par exemple [1 ; 6 ; 2] est la liste composée des trois entiers 1, 6 et 2.
  La liste vide, composée d'aucun élément, est notée [].
- **Opérations** :

  List.hd : $'a\ list \setminus \{[\,]\} \longrightarrow 'a$
  $li \longmapsto$ Le 1^{er} élément de la liste li

  List.tl : $'a\ list \setminus \{[\,]\} \longrightarrow 'a\ list$
  $li \longmapsto$ La liste li privée de son 1^{er} élément

  :: : $'a \times 'a\ list \longrightarrow 'a\ list$
  $(e, li) \longmapsto$ La liste dont la tête est e et la queue li

  Les fonctions `List.hd` et `List.tl` ne sont pas définies pour la liste vide.

  La fonction :: est utilisée sous forme infixe.
  :: et [ ] sont les constructeurs du type liste.

Les listes en OCaml

- **'a list** est le type des listes dont les éléments sont de type **'a**.
  Par exemple **int list** est le type des listes d'entiers
- **Domaine** : une liste est représentée par la séquence encadrée par des crochets de ses éléments séparés par des points-virgules.
  Par exemple [1 ; 6 ; 2] est la liste composée des trois entiers 1, 6 et 2.
  La liste vide, composée d'aucun élément, est notée [].
- **Opérations** :

  List.hd : $'a\ list \setminus \{[\,]\} \longrightarrow 'a$
  $li \longmapsto$ Le 1^{er} élément de la liste li

  List.tl : $'a\ list \setminus \{[\,]\} \longrightarrow 'a\ list$
  $li \longmapsto$ La liste li privée de son 1^{er} élément

  :: : $'a \times 'a\ list \longrightarrow 'a\ list$
  $(e, li) \longmapsto$ La liste dont la tête est e et la queue li

  Les fonctions `List.hd` et `List.tl` ne sont pas définies pour la liste vide.

  La fonction :: est utilisée sous forme infixe.
  :: et [ ] sont les constructeurs du type liste.

Les listes en OCaml

- **'a list** est le type des listes dont les éléments sont de type **'a**.
  Par exemple **int list** est le type des listes d'entiers
- **Domaine** : une liste est représentée par la séquence encadrée par des crochets de ses éléments séparés par des points-virgules.
  Par exemple [1 ; 6 ; 2] est la liste composée des trois entiers 1, 6 et 2.
  La liste vide, composée d'aucun élément, est notée [].
- **Opérations** :

  List.hd : 'a list $\setminus \{[\,]\}$ $\longrightarrow$ 'a
  li $\longmapsto$ Le 1^{er} élément de la liste li

  List.tl : 'a list $\setminus \{[\,]\}$ $\longrightarrow$ 'a list
  li $\longmapsto$ La liste li privée de son 1^{er} élément

  :: : 'a $\times$ 'a list $\longrightarrow$ 'a list
  (e, li) $\longmapsto$ La liste dont la tête est e et la queue li

  Les fonctions `List.hd` et `List.tl` ne sont pas définies pour la liste vide.

  La fonction :: est utilisée sous forme infixe.
  :: et [ ] sont les constructeurs du type liste.

Les listes en OCaml

- **'a list** est le type des listes dont les éléments sont de type **'a**.
  Par exemple **int list** est le type des listes d'entiers
- **Domaine** : une liste est représentée par la séquence encadrée par des crochets de ses éléments séparés par des points-virgules.
  Par exemple [1 ; 6 ; 2] est la liste composée des trois entiers 1, 6 et 2.
  La liste vide, composée d'aucun élément, est notée `[]`.
- **Opérations** :

  List.hd : $'a\ list \setminus \{[\,]\} \longrightarrow 'a$

  $li \longmapsto$ Le 1^{er} élément de la liste li

  List.tl : $'a\ list \setminus \{[\,]\} \longrightarrow 'a\ list$

  $li \longmapsto$ La liste li privée de son 1^{er} élément

  :: : $'a \times 'a\ list \longrightarrow 'a\ list$

  $(e, li) \longmapsto$ La liste dont la tête est e et la queue li

  Les fonctions `List.hd` et `List.tl` ne sont pas définies pour la liste vide.

  La fonction :: est utilisée sous forme infixe.
  :: et [ ] sont les constructeurs du type liste.

Les listes en OCaml

- **'a list** est le type des listes dont les éléments sont de type **'a**.
  Par exemple **int list** est le type des listes d'entiers
- **Domaine** : une liste est représentée par la séquence encadrée par des crochets de ses éléments séparés par des points-virgules.
  Par exemple [1 ; 6 ; 2] est la liste composée des trois entiers 1, 6 et 2.
  La liste vide, composée d'aucun élément, est notée `[]`.
- **Opérations** :

  List.hd : $'a\ list \setminus \{[\,]\} \longrightarrow 'a$

  $li \longmapsto$ Le 1^{er} élément de la liste li

  List.tl : $'a\ list \setminus \{[\,]\} \longrightarrow 'a\ list$

  $li \longmapsto$ La liste li privée de son 1^{er} élément

  :: : $'a \times 'a\ list \longrightarrow 'a\ list$

  $(e, li) \longmapsto$ La liste dont la tête est e et la queue li

  Les fonctions `List.hd` et `List.tl` ne sont pas définies pour la liste vide.

  La fonction :: est utilisée sous forme infixe.
  :: et [ ] sont les constructeurs du type liste.

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Expressions avec des listes

expression	Type et valeur
[4 ;2 ;1]	int list = [4 ; 2 ; 1]
List.tl [4 ;2 ;1]	int list = [2 ; 1]
List.hd [4 ;2 ;1]	int = 4
List.tl (List.tl [4 ;2 ;1])	int list = [1]
List.hd (List.hd [4 ;2 ;1])	ERREUR
8 : : 3	ERREUR
8 : : [3]	int list = [8 ; 3]
8 : : [3.5]	ERREUR
List.tl [7]	int list = []
List.hd (List.tl [7])	ERREUR
List.tl (List.tl [7])	ERREUR
List.tl [7]=[]	bool = true
List.hd [5 ;8] : : List.tl [5 ;8]	int list = [5 ;8]
6 : : List.tl [5 ;8]	int list = [6 ;8]

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste

  deuxieme : 'a list $\longrightarrow$ 'a

  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme : 'a list -> 'a = fun
  l -> List.hd (List.tl l)
```

- Supprimer le deuxième élément d'une liste

  oterDeuxieme : 'a list $\longrightarrow$ 'a list

  li $\longmapsto$ Liste *li* sans son deuxième élément

```
let oterDeuxieme : 'a list -> 'a list = fun
  l -> List.hd l :: List.tl (List.tl l)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste
  deuxieme : 'a list $\longrightarrow$ 'a
  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme: 'a list -> 'a = fun
    li -> List.hd (List.tl li)
```

- Supprimer le deuxième élément d'une liste
  oterDeuxieme : 'a list $\longrightarrow$ 'a list
  li $\longmapsto$ Liste *li* sans son deuxième élément

```
let oterDeuxieme: 'a list -> 'a list = fun
    li -> List.hd li :: List.tl (List.tl li)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste
  deuxieme : 'a list $\longrightarrow$ 'a
  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme: 'a list -> 'a = fun
    li -> List.hd (List.tl li)
```

- Supprimer le deuxième élément d'une liste
  oterDeuxieme : 'a list $\longrightarrow$ 'a list
  li $\longmapsto$ Liste *li* sans son deuxième élément

```
let oterDeuxieme: 'a list -> 'a list = fun
    li -> List.hd li :: List.tl (List.tl li)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste
  deuxieme : 'a list $\longrightarrow$ 'a
  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme: 'a list -> 'a = fun
        li -> List.hd (List.tl li)
```

- Supprimer le deuxième élément d'une liste
  oterDeuxieme : 'a list $\longrightarrow$ 'a list
  li $\longmapsto$ Liste li sans son deuxième élément

```
let oterDeuxieme: 'a list -> 'a list = fun
        li -> List.hd li :: List.tl(List.tl li)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste
  deuxieme : 'a list $\longrightarrow$ 'a
  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme: 'a list -> 'a = fun
        li -> List.hd (List.tl li)
```

- Supprimer le deuxième élément d'une liste
  oterDeuxieme : 'a list $\longrightarrow$ 'a list
  li $\longmapsto$ Liste li sans son deuxième élément

```
let oterDeuxieme: 'a list -> 'a list = fun
        li -> List.hd li :: List.tl(List.tl li)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Exemples de fonctions sur les listes

- Accéder au deuxième élément d'une liste
  deuxieme : 'a list $\longrightarrow$ 'a
  li $\longmapsto$ Deuxième élément de *li*

```
let deuxieme: 'a list -> 'a = fun
        li -> List.hd (List.tl li)
```

- Supprimer le deuxième élément d'une liste
  oterDeuxieme : 'a list $\longrightarrow$ 'a list
  li $\longmapsto$ Liste li sans son deuxième élément

```
let oterDeuxieme: 'a list -> 'a list = fun
        li -> List.hd li :: List.tl(List.tl li)
```

```
# deuxieme [3;2;1];;
- : int = 2
# oterDeuxieme [3;2;1];;
- : int list = [3;1]
```

Schéma des algorithmes récursifs sur les listes

La plupart des algorithmes sur les listes utilisent un schéma récursif, basé sur la définition récursive des listes. Pour calculer la valeur d'une fonction

$$\begin{array}{rccc} g : & \text{Liste} \times \ldots & \longrightarrow & \ldots \\ & (li, \ldots) & \longmapsto & \ldots \end{array}$$

le schéma de définition récursive est :

- **Le cas de base** donne la valeur `VB` de `g` quand la liste `li` est vide.
- **La récurrence** définit la valeur de `g(li,..)` en fonction de `g(List.tl(li),..)`

D'où le schéma fréquent

$$\begin{array}{rccll} g : & \text{Liste} \times \ldots & \longrightarrow & \ldots & \\ & (\text{li},..) & \longmapsto & V_B & \textbf{si li=[ ]} \\ & & & \ldots g(List.tl, \ldots) \ldots & \textbf{sinon} \end{array}$$

Ce schéma est correct car la taille de l'argument liste décroît strictement à chaque appel récursif et la valeur est connue pour une taille de liste nulle.

Schéma des algorithmes récursifs sur les listes

La plupart des algorithmes sur les listes utilisent un schéma récursif, basé sur la définition récursive des listes. Pour calculer la valeur d'une fonction

$$\begin{array}{rccc} g : & \text{Liste} \times \ldots & \longrightarrow & \ldots \\ & (li, \ldots) & \longmapsto & \ldots \end{array}$$

le schéma de définition récursive est :

- **Le cas de base** donne la valeur `VB` de `g` quand la liste `li` est vide.
- **La récurrence** définit la valeur de `g(li,..)` en fonction de `g(List.tl(li),..)`

D'où le schéma fréquent

$$\begin{array}{rccll} g : & \text{Liste} \times \ldots & \longrightarrow & \ldots & \\ & (\text{li},..) & \longmapsto & V_B & \textbf{si li=[ ]} \\ & & & \ldots g(List.tl, \ldots) \ldots & \textbf{sinon} \end{array}$$

Ce schéma est correct car la taille de l'argument liste décroît strictement à chaque appel récursif et la valeur est connue pour une taille de liste nulle.

Schéma des algorithmes récursifs sur les listes

La plupart des algorithmes sur les listes utilisent un schéma récursif, basé sur la définition récursive des listes. Pour calculer la valeur d'une fonction

$$\begin{array}{rccc} g: & \text{Liste} \times \ldots & \longrightarrow & \ldots \\ & (li, \ldots) & \longmapsto & \ldots \end{array}$$

le schéma de définition récursive est :

- **Le cas de base** donne la valeur `VB` de `g` quand la liste `li` est vide.
- **La récurrence** définit la valeur de `g(li,..)` en fonction de `g(List.tl(li),..)`

D'où le schéma fréquent

$$\begin{array}{rccll} g: & \text{Liste} \times \ldots & \longrightarrow & \ldots & \\ & (\text{li},..) & \longmapsto & V_B & \textbf{si li=[ ]} \\ & & & \ldots g(List.tl, \ldots) \ldots & \textbf{sinon} \end{array}$$

Ce schéma est correct car la taille de l'argument liste décroît strictement à chaque appel récursif et la valeur est connue pour une taille de liste nulle.

Schéma des algorithmes récursifs sur les listes

La plupart des algorithmes sur les listes utilisent un schéma récursif, basé sur la définition récursive des listes. Pour calculer la valeur d'une fonction

$$\begin{array}{rccc} g: & \text{Liste} \times \ldots & \longrightarrow & \ldots \\ & (li, \ldots) & \longmapsto & \ldots \end{array}$$

le schéma de définition récursive est :

- **Le cas de base** donne la valeur `VB` de `g` quand la liste `li` est vide.
- **La récurrence** définit la valeur de `g(li,..)` en fonction de `g(List.tl(li),..)`

D'où le schéma fréquent

$$\begin{array}{rccll} \text{g}: & \text{Liste} \times \ldots & \longrightarrow & \ldots & \\ & \text{(li,..)} & \longmapsto & V_B & \textbf{si li=[ ]} \\ & & & \ldots g(List.tl, \ldots) \ldots & \textbf{sinon} \end{array}$$

Ce schéma est correct car la taille de l'argument liste décroît strictement à chaque appel récursif et la valeur est connue pour une taille de liste nulle.

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

$$\begin{array}{rccl} \textit{longueur} : & \text{'a list} & \longrightarrow & \mathbb{N} \\ & \textit{li} & \longmapsto & \text{Taille de la liste li} \end{array}$$

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list -> int = function
    li -> if li=[] then 0
          else 1+longueur (List.tl li)
```

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

longueur : 'a list $\longrightarrow$ $\mathbb{N}$
li $\longmapsto$ Taille de la liste li

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list → int = function
    li → if li=[] then 0
         else 1+longueur (List.tl li)
```

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

$$\begin{array}{rccl} \textit{longueur} : & \text{'a list} & \longrightarrow & \mathbb{N} \\ & \textit{li} & \longmapsto & \text{Taille de la liste li} \end{array}$$

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list -> int = function
    li -> if li=[] then 0
          else 1+longueur (List.tl li)
```

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

longueur : 'a list $\longrightarrow$ $\mathbb{N}$
li $\longmapsto$ Taille de la liste li

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list → int = function
    li → if li=[] then 0
         else 1+longueur (List.tl li)
```

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

$$\begin{array}{rccl} \textit{longueur} : & \text{'a list} & \longrightarrow & \mathbb{N} \\ & \textit{li} & \longmapsto & \text{Taille de la liste li} \end{array}$$

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list → int = function
    li → if li=[] then 0
         else 1+longueur (List.tl li)
```

Algorithmes récursifs sur les listes

Calcul du nombre d'éléments d'une liste

$$\begin{array}{rccl} longueur : & \text{'a list} & \longrightarrow & \mathbb{N} \\ & li & \longmapsto & \text{Taille de la liste li} \end{array}$$

- Quand `li` est la liste vide, quelle est la longueur de `li` ? 0
- Quand `li` n'est pas vide, connaissant `longueur(List.tl(li))`, quelle est la longueur de li ? 1+longueur(List.tl(li))

```
let rec longueur : 'a list → int = function
    li → if li=[] then 0
         else 1+longueur (List.tl li)
```

```
# longueur [4;3;8];;
- : int = 3
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list -> int list = function
    li -> if li=[] then []
          else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list -> int list = function
    li -> if li=[] then []
          else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list -> int list = function
    li -> if li=[] then []
          else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?

  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list → int list = function
    li → if li=[] then []
         else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list -> int list = function
    li -> if li=[] then []
          else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list → int list = function
    li → if li=[] then []
         else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list -> int list = function
    li -> if li=[] then []
          else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list → int list = function
    li → if li=[] then []
         else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Ajouter 1 à chaque élément d'une liste d'entiers

$ajout1$: int list $\longrightarrow$ int list

$li \longmapsto$ liste obtenue à partir de li en ajoutant 1 à chacun de ses éléments

---

- Ajouter 1 à chaque élément de la liste vide est la liste vide
- Connaissant ajout1(List.tl(li)), quelle est la liste ajout1(li) ?
  C'est la liste
  dont la tête est la tête de li + 1
  dont la queue est ajout1(List.tl(li))
  c'est à dire la liste List.hd(li)+1 : : ajout1(List.tl(li))

```
let rec ajout1: int list → int list = function
    li → if li=[] then []
         else (1+List.hd li) :: ajout1 (List.tl li)
```

```
# ajout1 [3;1;8];;
- : int list = [4;2;9]
```

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

- La concaténation des listes [1 ; 2] et [5 ; 3 ; 6] est la liste [1 ; 2 ; 5 ; 3 ; 6]
- La concaténation des listes [2] et [2 ; 5 ; 5 ; 2] est la liste [2 ; 2 ; 5 ; 5 ; 2]
- La concaténation des listes [1 ; 2] et [ ] est la liste [1 ; 2]
- La concaténation des listes [ ] et [3 ; 2] est la liste [3 ; 2]

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

- La concaténation des listes [1 ; 2] et [5 ; 3 ; 6] est la liste [1 ; 2 ; 5 ; 3 ; 6]
- La concaténation des listes [2] et [2 ; 5 ; 5 ; 2] est la liste [2 ; 2 ; 5 ; 5 ; 2]
- La concaténation des listes [1 ; 2] et [ ] est la liste [1 ; 2]
- La concaténation des listes [ ] et [3 ; 2] est la liste [3 ; 2]

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

- La concaténation des listes [1 ; 2] et [5 ; 3 ; 6] est la liste [1 ; 2 ; 5 ; 3 ; 6]
- La concaténation des listes [2] et [2 ; 5 ; 5 ; 2] est la liste [2 ; 2 ; 5 ; 5 ; 2]
- La concaténation des listes [1 ; 2] et [ ] est la liste [1 ; 2]
- La concaténation des listes [ ] et [3 ; 2] est la liste [3 ; 2]

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

- La concaténation des listes [1 ; 2] et [5 ; 3 ; 6] est la liste [1 ; 2 ; 5 ; 3 ; 6]
- La concaténation des listes [2] et [2 ; 5 ; 5 ; 2] est la liste [2 ; 2 ; 5 ; 5 ; 2]
- La concaténation des listes [1 ; 2] et [ ] est la liste [1 ; 2]
- La concaténation des listes [ ] et [3 ; 2] est la liste [3 ; 2]

Algorithme de concaténation de deux listes

Concaténation de Listes

La concaténation des listes `li1` et `li2` est la liste formée des éléments de `li1` (dans le même ordre), puis des éléments de `li2` (dans le même ordre).

Concaténation

- La concaténation des listes [1 ; 2] et [5 ; 3 ; 6] est la liste [1 ; 2 ; 5 ; 3 ; 6]
- La concaténation des listes [2] et [2 ; 5 ; 5 ; 2] est la liste [2 ; 2 ; 5 ; 5 ; 2]
- La concaténation des listes [1 ; 2] et [ ] est la liste [1 ; 2]
- La concaténation des listes [ ] et [3 ; 2] est la liste [3 ; 2]

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est :
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est et dont la queue est

```
# let rec concat = fun l1 l2 →
    if l1=[] then l2
    else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est :
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est et dont la queue est

```
# let rec concat = fun l1 l2 →
    if l1=[] then l2
    else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est et dont la queue est

```
# let rec concat = fun l1 l2 →
    if l1=[] then l2
    else List.hd l1 :: concat (List.tl l1) l2;;
val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est et dont la queue est

```
# let rec concat = fun l1 l2 →
    if l1=[] then l2
    else List.hd l1 :: concat (List.tl l1) l2;;
val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est la tête de l1 et dont la queue est concat(List.tl(l1),l2). C'est à dire la liste List.hd(l1) : : concat (List.tl(l1),l2)

```
# let rec concat = fun l1 l2 →
      if l1=[ ] then l2
      else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est la tête de l1 et dont la queue est concat(List.tl(l1),l2). C'est à dire la liste List.hd(l1) : : concat (List.tl(l1),l2)

```
# let rec concat = fun l1 l2 →
      if l1=[ ] then l2
      else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est la tête de l1 et dont la queue est concat(List.tl(l1),l2). C'est à dire la liste List.hd(l1) : : concat (List.tl(l1),l2)

```
# let rec concat = fun l1 l2 →
      if l1=[ ] then l2
      else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est la tête de l1 et dont la queue est concat(List.tl(l1),l2). C'est à dire la liste List.hd(l1) : : concat (List.tl(l1),l2)

```
# let rec concat = fun l1 l2 →
      if l1=[ ] then l2
      else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Algorithme de concaténation

```
let rec concat = fun l1 l2 →
```

- Choisir le paramètre par rapport auquel sera définie la récursivité : l1
- Le résultat de la concaténation de la liste vide avec l2 est : l2
- Si l1 n'est pas vide, le résultat de la concaténation de l1 et l2 est la liste dont la tête est la tête de l1 et dont la queue est concat(List.tl(l1),l2). C'est à dire la liste List.hd(l1) : : concat (List.tl(l1),l2)

```
# let rec concat = fun l1 l2 →
        if l1=[ ] then l2
        else List.hd l1 :: concat (List.tl l1) l2;;
- : val concat : 'a list → 'a list → 'a list = <fun>

# concat [4;3;8] [9;2];;
- : int list = [4; 3; 8; 9; 2]
```

Le filtrage

L'expression de *filtrage* ou *matching* est une généralisation de l'expression conditionnelle. Sa valeur est sélectionnée selon un ensemble de filtres (ou pattern).

- **Syntaxe** :

  match *exp* **with**
  | *f1* → *exp1*
  | *f2* → *exp2*
  ..
  | *fn* → *expn*

  `exp` et `expi` sont des expressions. Les `expi` doivent être de même type `fi` sont des *filtres (pattern)*

- **Valeur** :
  `val(expi)` où `fi` est le premier filtre *"compatible"* avec `exp`.

Un filtre est une expression construite à partir de constantes, noms, et constructeurs de type (produit cartésien et liste) et le symbole _

Exemple de filtres constants

```
# match true with
    | false → true
    | true → false ;;
- : bool = false
```

Le filtrage permet de définir des fonctions par cas :

```
# let neg = function
    b → match b with
        | false → true
        | true → false ;;
val neg : bool → bool = <fun>
```

Cette écriture peut être simplifiée en supprimant "`b -> match b with `"

```
# let neg = function
    | false → true
    | true → false ;;
```

Exemple de filtres constants

```
# match true with
  | false → true
  | true → false ;;
- : bool = false
```

Le filtrage permet de définir des fonctions par cas :

```
# let neg = function
    b → match b with
      | false → true
      | true → false ;;
val neg : bool → bool = <fun>
```

Cette écriture peut être simplifiée en supprimant "b -> match b with "

```
# let neg = function
  | false → true
  | true → false ;;
```

Exemple de filtres constants

```
# match true with
  | false → true
  | true → false ;;
- : bool = false
```

Le filtrage permet de définir des fonctions par cas :

```
# let neg = function
    b → match b with
      | false → true
      | true → false ;;
val neg : bool → bool = <fun>
```

Cette écriture peut être simplifiée en supprimant "`b -> match b with `"

```
# let neg = function
  | false → true
  | true → false ;;
```

Exemple de filtre avec nom

On peut utiliser le filtrage pour définir par cas la fonction factorielle :

```
# let rec fact = function
  | 0 → 1
  | n → n * fact(n-1);;
```

Les filtres sont testés par ordre d'apparition : si le 1^{er} filtre n'est pas compatible (si $n \neq 0$) on teste le second filtre qui lui est compatible.

L'ordre des clauses est important.

Ici en inversant l'ordre des clauses la définition devient fausse :

```
# let rec factb = function
  | n → n * factb(n-1)
  | 0 → 1;;
    ^ Warning : this match case is unused.
# factb 1;;
 Stack overflow during evaluation (looping recursion?).
```

Exemple de filtre avec nom

On peut utiliser le filtrage pour définir par cas la fonction factorielle :

```
# let rec fact = function
  | 0 -> 1
  | n -> n * fact(n-1);;
```

Les filtres sont testés par ordre d'apparition : si le 1^{er} filtre n'est pas compatible (si $n \neq 0$) on teste le second filtre qui lui est compatible.
L'ordre des clauses est important.
Ici en inversant l'ordre des clauses la définition devient fausse :

```
# let rec factb = function
  | n -> n * factb(n-1)
  | 0 -> 1;;
    ^ Warning : this match case is unused.
# factb 1;;
Stack overflow during evaluation (looping recursion?).
```

Exemple de filtre avec constructeur

```
# let ou = function
  | (true, true) -> true
  | (true, false) -> true
  | (false, true) -> true
  | (false, false) -> false;;
```

On peut regrouper deux filtres en un filtre utilisant constructeur et nom :

```
# let ou = function
  | (true, x) -> true
  | (false, x) -> x;;
```

Exemple de filtre avec constructeur

```
# let ou = function
  | (true, true) -> true
  | (true, false) -> true
  | (false, true) -> true
  | (false, false) -> false;;
```

On peut regrouper deux filtres en un filtre utilisant constructeur et nom :

```
# let ou = function
  | (true, x) -> true
  | (false, x) -> x;;
```

Suite de l'exemple

Lorsque dans une clause un nom n'est pas utilisé dans la partie droite, on peut le remplacer par le symbole _ signifiant que la valeur liée n'a pas d'importance

```
# let ou = function
    | (true,_) -> true
    | (false,x) -> x;;
```

En changeant l'ordre des clauses on obtient une définition plus simple :

```
# let ou = function
    | (false,false) -> false
    | (_,_) -> true;;
```

qu'on peut encore simplifier

```
# let ou = function
    | (false,false) -> false
    | _ -> true;;
```

Suite de l'exemple

Lorsque dans une clause un nom n'est pas utilisé dans la partie droite, on peut le remplacer par le symbole _ signifiant que la valeur liée n'a pas d'importance

```
# let ou = function
  | (true,_) -> true
  | (false,x) -> x;;
```

En changeant l'ordre des clauses on obtient une définition plus simple :

```
# let ou = function
  | (false,false) -> false
  | (_,_) -> true;;
```

qu'on peut encore simplifier

```
# let ou = function
  | (false,false) -> false
  | _ -> true;;
```

Suite de l'exemple

Lorsque dans une clause un nom n'est pas utilisé dans la partie droite, on peut le remplacer par le symbole _ signifiant que la valeur liée n'a pas d'importance

```
# let ou = function
  | (true,_) -> true
  | (false,x) -> x;;
```

En changeant l'ordre des clauses on obtient une définition plus simple :

```
# let ou = function
  | (false,false) -> false
  | (_,_) -> true;;
```

qu'on peut encore simplifier

```
# let ou = function
  | (false,false) -> false
  | _ -> true;;
```

Filtrage partiel

OCaml vérifie si l'ensemble des filtres couvre la totalité du domaine du type :

```
# let traduction = function
    | "vrai" → true
    | "faux" → false ;;
Warning : this pattern-matching is not exhaustive.
Here is an example of a case that is not matched :""
```

En terminant par une clause dont le filtre est _, on couvre tous les cas :

```
# let traduction = function
    | "vrai" → true
    | "faux" → false
    | _ → (failwith "traduction : cas non defini") ;;
```

Filtrage partiel

OCaml vérifie si l'ensemble des filtres couvre la totalité du domaine du type :

```
# let traduction = function
  | "vrai" → true
  | "faux" → false ;;
Warning : this pattern-matching is not exhaustive.
Here is an example of a case that is not matched :""
```

En terminant par une clause dont le filtre est _, on couvre tous les cas :

```
# let traduction = function
  | "vrai" → true
  | "faux" → false
  | _ → (failwith "traduction : cas non defini") ;;
```

Filtre avec garde

Un filtre ne peut pas contenir plusieurs occurrences d'un même identificateur :

```
# let egal = function
  | (x, x) -> true
  | _ -> false ;;
Error: Variable x is bound several times in this matching
```

Dans ce cas on peut ajouter une condition à un filtre. La syntaxe de la clause est :

$$f \ \textbf{when} \ cond \rightarrow exp$$

où f est un filtre, cond une expression booléenne et exp une expression

```
# let egal = function
  | (x, y) when x=y -> true
  | _ -> false ;;
```

Filtre avec garde

Un filtre ne peut pas contenir plusieurs occurrences d'un même identificateur :

```
# let egal = function
  | (x, x) → true
  | _ → false ;;
Error: Variable x is bound several times in this matching
```

Dans ce cas on peut ajouter une condition à un filtre. La syntaxe de la clause est :

f **when** *cond* → *exp*

où `f` est un filtre, `cond` une expression booléenne et `exp` une expression

```
# let egal = function
  | (x, y) when x=y → true
  | _ → false ;;
```

Exemple de filtre sur les listes

Les listes sont définies avec les constructeurs [ ] et : : qu'on peut utiliser dans le filtrage :

```
# let rec longueur = function
    |   []    → 0
    | _::q  → 1+longueur q;;
- : val longueur : 'a list → int = <fun>

#  let rec concat = fun
     li1 li2 →
       match li1 with
         |   []    → li2
         | t::q → t :: concat q li2;;
- : val concat : 'a list → 'a list → 'a list = <fun>
```

Exemple de filtre sur les listes

Les listes sont définies avec les constructeurs [ ] et : : qu'on peut utiliser dans le filtrage :

```
# let rec longueur = function
    |   []    → 0
    | _::q  → 1+longueur q;;
- : val longueur : 'a list → int = <fun>

#   let rec concat  = fun
     li1  li2 →
       match li1 with
       |   []    → li2
       | t::q  → t :: concat q li2;;
- : val concat : 'a list → 'a list → 'a list = <fun>
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme `[ ]`, **appartient e li** est Faux
- Quand `li` est de la forme `t : : q` avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme `t : : q` (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
    | [] -> false
    | t::_ when t=e -> true
    | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
      | [] -> false
      | t::_ when t=e -> true
      | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
      | [] -> false
      | t::_ when t=e -> true
      | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
    | [] -> false
    | t::_ when t=e -> true
    | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
    | [] -> false
    | t::_ when t=e -> true
    | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li →
  match li with
    | [] → false
    | t::_ when t=e → true
    | _::q → appartient e q;;

- : val appartient : 'a → 'a list → bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li ->
    match li with
      | [] -> false
      | t::_ when t=e -> true
      | _::q -> appartient e q;;

- : val appartient : 'a -> 'a list -> bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li →
    match li with
    | [] → false
    | t::_ when t=e → true
    | _::q → appartient e q;;

- : val appartient : 'a → 'a list → bool = <fun>
```

```
# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

appartenance à une liste

on veut définir la fonction `appartient` de sorte que :

appartient e li = Vrai si e appartient à la liste li,
Faux sinon

- Quand `li` est de la forme [ ], **appartient e li** est Faux
- Quand `li` est de la forme t : : q avec $t = e$, **appartient e li** est Vrai
- Dans les autres cas, quand `li` est de la forme t : : q (avec $t \neq e$) **appartient e li** si et seulement si **appartient e q**

```
# let rec appartient = fun e li →
  match li with
  | [] → false
  | t::_ when t=e → true
  | _::q → appartient e q;;

- : val appartient : 'a → 'a list → bool = <fun>

# appartient 2 [3;8;21;1];;
- : bool = false
# appartient 2 [3;8;2;1];;
- : bool = true
```

Fonction concaténation

La fonction concaténation de 2 listes est prédéfinie en tant qu'opérateur @ :

```
# [6;4;3] @ [3;1];;
- : int list = [6; 4; 3; 3; 1]
```

Simplification d'écriture

Il est d'usage de noter la déclaration de fonction

```
# let f = function x → ...    ==>    # let f x = ...
# let f = fun x y → ...       ==>    # let f x y = ...
```

```
# let rec appartient e li =
  match li with
  | [] → false
  | t::_ when e=t → true
  | _::q → appartient e q;;
```

Récursion terminale - accumulateurs

Pour certaines fonctions récursives il peut être intéressant de définir une fonction intermédiaire avec des paramètres supplémentaires pour faire apparaître une récursivité terminale.

Factorielle

```
# let rec fact1 n =
  if n=0 then 1
  else n*fact1(n-1)
```

Trace de l'évaluation de fact1(3) :

```
fact1(3)
3*fact1(2)
3*(2*fact1(1))
3*(2*(1*fact1(0)))
3*(2*(1*1))
3*(2*1)
3*2
6
```

```
# let rec factl n a =
  if n=0 then a
  else factl (n-1) (n*a)
# let fact2 n = factl n 1
```

Trace de l'évaluation de fact2(3) :

```
fact2(3)
factl 3 1
factl 2 3
factl 1 6
factl 0 6
6
```

Récursion terminale - accumulateurs

Pour certaines fonctions récursives il peut être intéressant de définir une fonction intermédiaire avec des paramètres supplémentaires pour faire apparaître une récursivité terminale.

Factorielle

```
# let rec fact1 n =
  if n=0 then 1
  else n*fact1(n-1)
```

Trace de l'évaluation de fact1(3) :

```
fact1(3)
3*fact1(2)
3*(2*fact1(1))
3*(2*(1*fact1(0)))
3*(2*(1*1))
3*(2*1)
3*2
6
```

```
# let rec factl n a =
  if n=0 then a
  else factl (n-1) (n*a)
# let fact2 n = factl n 1
```

Trace de l'évaluation de fact2(3) :

```
fact2(3)
factl 3 1
factl 2 3
factl 1 6
factl 0 6
6
```

Récursion terminale - accumulateurs

Pour certaines fonctions récursives il peut être intéressant de définir une fonction intermédiaire avec des paramètres supplémentaires pour faire apparaître une récursivité terminale.

Factorielle

```
# let rec fact1 n =
    if n=0 then 1
    else n*fact1(n-1)
```

Trace de l'évaluation de fact1(3) :

```
fact1(3)
3*fact1(2)
3*(2*fact1(1))
3*(2*(1*fact1(0)))
3*(2*(1*1))
3*(2*1)
3*2
6
```

Récursion terminale - accumulateurs

Pour certaines fonctions récursives il peut être intéressant de définir une fonction intermédiaire avec des paramètres supplémentaires pour faire apparaître une récursivité terminale.

Factorielle

```
# let rec fact1 n =
    if n=0 then 1
    else n*fact1(n-1)
```

Trace de l'évaluation de fact1(3) :

```
fact1(3)
3*fact1(2)
3*(2*fact1(1))
3*(2*(1*fact1(0)))
3*(2*(1*1))
3*(2*1)
3*2
6
```

```
# let rec factl n a =
    if n=0 then a
    else factl (n-1) (n*a)
# let fact2 n = factl n 1
```

Trace de l'évaluation de fact2(3) :

```
fact2(3)
factl 3 1
factl 2 3
factl 1 6
factl 0 6
6
```

Récursion terminale - accumulateurs

Pour certaines fonctions récursives il peut être intéressant de définir une fonction intermédiaire avec des paramètres supplémentaires pour faire apparaître une récursivité terminale.

Factorielle

```
# let rec fact1 n =
  if n=0 then 1
  else n*fact1(n-1)
```

Trace de l'évaluation de fact1(3) :

```
fact1(3)
3*fact1(2)
3*(2*fact1(1))
3*(2*(1*fact1(0)))
3*(2*(1*1))
3*(2*1)
3*2
6
```

```
# let rec factl n a =
  if n=0 then a
  else factl (n-1) (n*a)
# let fact2 n = factl n 1
```

Trace de l'évaluation de fact2(3) :

```
fact2(3)
factl 3 1
factl 2 3
factl 1 6
factl 0 6
6
```

Fibonacci

```
# let rec f1 = function
  | 0 -> 0
  | 1 -> 1
  | n -> f1 (n-1) + f1 (n-2)
```

Trace de l'évaluation de f1(4) :

$$
\begin{aligned}
& f1(4) \\
& f1(3)+f1(2) \\
& (f1(2)+f1(1))+f1(2) \\
& ((f1(1)+f1(0))+f1(1))+f1(2) \\
& ((1+f1(0))+f1(1))+f1(2) \\
& ((1+0)+f1(1))+f1(2) \\
& (1+f1(1))+f1(2) \\
& (1+1)+f1(2) \\
& 2+f1(2) \\
& 2+(f1(1)+f1(0)) \\
& 2+(1+0) \\
& 3
\end{aligned}
$$

```
# let rec fibl n v1 v2 =
    if n=0 then v1
    else fibl (n-1) v2 (v2+v1)
# let f2 n = fibl n 0 1
```

Trace de l'évaluation de f2(4) :

```
f2 4
fibl 4 0 1
fibl 3 1 1
fibl 2 1 2
fibl 1 2 3
fibl 0 3 5
3
```

Fibonacci

```
# let rec f1 = function
| 0 → 0
| 1 → 1
| n → f1 (n−1) + f1 (n−2)
```

Trace de l'évaluation de f1(4) :

$$
\begin{aligned}
& f1\,(4)\\
& f1\,(3)+f1\,(2)\\
& (f1\,(2)+f1\,(1))+f1\,(2)\\
& ((f1\,(1)+f1\,(0))+f1\,(1))+f1\,(2)\\
& ((1+f1\,(0))+f1\,(1))+f1\,(2)\\
& ((1+0)+f1\,(1))+f1\,(2)\\
& (1+f1\,(1))+f1\,(2)\\
& (1+1)+f1\,(2)\\
& 2+f1\,(2)\\
& 2+(f1\,(1)+f1\,(0))\\
& 2+(1+0)\\
& 3
\end{aligned}
$$

```
# let rec fibl n v1 v2 =
  if n=0 then v1
  else fibl (n−1) v2 (v2+v1)
# let f2 n = fibl n 0 1
```

Trace de l'évaluation de f2(4) :

```
f2 4
fibl 4 0 1
fibl 3 1 1
fibl 2 1 2
fibl 1 2 3
fibl 0 3 5
3
```

Fibonacci

```
# let rec f1 = function
| 0 → 0
| 1 → 1
| n → f1 (n−1) + f1 (n−2)
```

Trace de l'évaluation de f1(4) :

$f1(4)$
$f1(3)+f1(2)$
$(f1(2)+f1(1))+f1(2)$
$((f1(1)+f1(0))+f1(1))+f1(2)$
$((1+f1(0))+f1(1))+f1(2)$
$((1+0)+f1(1))+f1(2)$
$(1+f1(1))+f1(2)$
$(1+1)+f1(2)$
$2+f1(2)$
$2+(f1(1)+f1(0))$
$2+(1+0)$
3

```
# let rec fibl n v1 v2 =
  if n=0 then v1
  else fibl (n−1) v2 (v2+v1)
# let f2 n = fibl n 0 1
```

Trace de l'évaluation de f2(4) :

```
f2 4
fibl 4 0 1
fibl 3 1 1
fibl 2 1 2
fibl 1 2 3
fibl 0 3 5
3
```

Fibonacci

```
# let rec f1 = function
| 0 → 0
| 1 → 1
| n → f1 (n−1) + f1 (n−2)
```

Trace de l'évaluation de f1(4) :

$f1(4)$
$f1(3)+f1(2)$
$(f1(2)+f1(1))+f1(2)$
$((f1(1)+f1(0))+f1(1))+f1(2)$
$((1+f1(0))+f1(1))+f1(2)$
$((1+0)+f1(1))+f1(2)$
$(1+f1(1))+f1(2)$
$(1+1)+f1(2)$
$2+f1(2)$
$2+(f1(1)+f1(0))$
$2+(1+0)$
3

```
# let rec fibl n v1 v2 =
  if n=0 then v1
  else fibl (n−1) v2 (v2+v1)
# let f2 n = fibl n 0 1
```

Trace de l'évaluation de f2(4) :

```
f2 4
fibl 4 0 1
fibl 3 1 1
fibl 2 1 2
fibl 1 2 3
fibl 0 3 5
3
```

Fibonacci

```
# let rec f1 = function
| 0 → 0
| 1 → 1
| n → f1 (n−1) + f1 (n−2)
```

Trace de l'évaluation de f1(4) :

$f1(4)$
$f1(3)+f1(2)$
$(f1(2)+f1(1))+f1(2)$
$((f1(1)+f1(0))+f1(1))+f1(2)$
$((1+f1(0))+f1(1))+f1(2)$
$((1+0)+f1(1))+f1(2)$
$(1+f1(1))+f1(2)$
$(1+1)+f1(2)$
$2+f1(2)$
$2+(f1(1)+f1(0))$
$2+(1+0)$
3

```
# let rec fibI n v1 v2 =
  if n=0 then v1
  else fibI (n−1) v2 (v2+v1)
# let f2 n = fibI n 0 1
```

Trace de l'évaluation de f2(4) :

f2 4
fibI 4 0 1
fibI 3 1 1
fibI 2 1 2
fibI 1 2 3
fibI 0 3 5
3

Inversion de liste

```
# let rec inv1=
    | [] → []
    | t::q → inv1 q @ [t]

inv1 [1;2;3]
(inv1 [2;3]) @ [1]
((inv1 [3]) @ [2]) @ [1]
(((inv1 [])@ [3]) @ [2]) @ [1]
(([]@ [3]) @ [2]) @ [1]
([3] @ [2]) @ [1]
[3;2] @ [1]
[3;2;1]
```

```
# let rec invl l lr =
    match l with
    | [] → lr
    | t::q → invl q (t::lr)
# let inv2 l = invl l []

inv2 [1;2;3]
invl [2;3] [1]
invl [3] [2;1]
invl [] [3;2;1]
[3;2;1]
```

Inversion de liste

```
# let rec inv1=
    | [] → []
    | t::q → inv1 q @ [t]
```

```
inv1 [1;2;3]
(inv1 [2;3]) @ [1]
((inv1 [3]) @ [2]) @ [1]
(((inv1 [])@ [3]) @ [2]) @ [1]
(([]@ [3]) @ [2]) @ [1]
([3] @ [2]) @ [1]
[3;2] @ [1]
[3;2;1]
```

```
# let rec invl l lr =
  match l with
    | [] → lr
    | t::q → invl q (t::lr)
# let inv2 l = invl l []
```

```
inv2 [1;2;3]
invl [2;3] [1]
invl [3] [2;1]
invl [] [3;2;1]
[3;2;1]
```

Inversion de liste

```
# let rec inv1=
    | [] → []
    | t::q → inv1 q @ [t]

inv1 [1;2;3]
(inv1 [2;3]) @ [1]
((inv1 [3]) @ [2]) @ [1]
(((inv1 [])@ [3]) @ [2]) @ [1]
(([]@ [3]) @ [2]) @ [1]
([3] @ [2]) @ [1]
[3;2] @ [1]
[3;2;1]
```

Inversion de liste

```
# let rec inv1=
    | [] → []
    | t::q → inv1 q @ [t]

inv1 [1;2;3]
(inv1 [2;3]) @ [1]
((inv1 [3]) @ [2]) @ [1]
(((inv1 [])@ [3]) @ [2]) @ [1]
(([]@ [3]) @ [2]) @ [1]
([3] @ [2]) @ [1]
[3;2] @ [1]
[3;2;1]
```

```
# let rec invl l lr =
    match l with
    | [] → lr
    | t::q → invl q (t::lr)
# let inv2 l = invl l []

inv2 [1;2;3]
invl [2;3] [1]
invl [3] [2;1]
invl [] [3;2;1]
[3;2;1]
```

Inversion de liste

```
# let rec inv1=
    | [] → []
    | t::q → inv1 q @ [t]

inv1 [1;2;3]
(inv1 [2;3]) @ [1]
((inv1 [3]) @ [2]) @ [1]
(((inv1 [])@ [3]) @ [2]) @ [1]
(([]@ [3]) @ [2]) @ [1]
([3] @ [2]) @ [1]
[3;2] @ [1]
[3;2;1]
```

```
# let rec invl l lr =
    match l with
    | [] → lr
    | t::q → invl q (t::lr)
# let inv2 l = invl l []

inv2 [1;2;3]
invl [2;3] [1]
invl [3] [2;1]
invl [] [3;2;1]
[3;2;1]
```

Version finale de l'inversion d'une liste

```
# let inv l =
    let rec aux l li=
      match l with
        | [] -> li
        | t::q -> aux q (t::li)
    in aux l []
```

Plan du cours

Polymorphisme

Principe

- Fournir une interface unique à des entités pouvant avoir différents types.
- Objectif : permettre de factoriser du code.
- Par exemple : avoir une seule fonction de tri, et non une fonction de tri pour les entiers, pour les chaînes de caractères, etc.

Différents polymorphismes

1. Polymorphisme ad hoc (surcharge) : n'existe pas en OCaml.
2. Polymorphisme paramétrique : celui du noyau fonctionnel d'OCaml.
3. Polymorphisme d'inclusion (sous-typage) : existe aussi en OCaml.

Nous verrons (2) dans ce cours.

Historiquement

Le λ-calcul polymorphe de Reynolds

- John C. Reynolds, *Towards a Theory of Type Structure*, 1974 : *We suggest that a solution to [the polymorphic sort function] problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the syntactic checking of type correctness.*

Le système F de Girard

- Quelques années avant Reynolds, vers 1970-1971, le logicien (français) Jean-Yves Girard avait inventé le même λ-calcul typé polymorphe sous le nom de système F.
- Les motivations de Girard n'étaient pas la programmation générique, mais l'étude de l'arithmétique du second ordre via une interprétation fonctionnelle.

Le polymorphisme (paramétrique) en pratique

Polymorphisme de deuxième classe

- Définitions polymorphes mais valeurs monomorphes.
- Exemples de langages :
  - Les génériques en Ada, Java, C#.
  - Le typage de Hindley-Milner dans les langages de la famille ML (SML, OCaml, Haskell, etc.), avec inférence des types, des lambdas sur les types et des instanciations.

Polymorphisme de première classe

- Les paramètres de fonctions peuvent avoir des quantifications sur les types.
- Exemples de langages : extensions récentes d'OCaml et de Haskell.

Un premier exemple

La fonction identité

```
# let id x = x;;
val id : 'a → 'a = <fun>
# id 1;;
- : int = 1
# id true;;
- : bool = true
```

- On remarque que même en présence de polymorphisme, OCaml parvient à inférer le type des définitions (ici de fonction).
- L'inférence concerne aussi bien les variables de type de la définition polymorphe que les types effectifs (instanciation) lors de l'application de la définition polymorphe.
- Ceci est rendu possible grâce à une restriction syntaxique du polymorphisme, qui consiste à mettre les quantifications sur les types en tête du type (polymorphisme prénexe).

Un autre exemple

Une fonction sur les listes

```
# let cons_last x l = l @ [x];;
val cons_last : 'a → 'a list → 'a list = <fun>
# cons_last 5 [1; 2; 3; 4];;
- : int list = [1; 2; 3; 4; 5]
# cons_last 'e' ['a'; 'b'; 'c'; 'd'];;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e']
```

- On avait déjà vu de nombreuses fonctions polymorphes sur les listes.
- C'est le cas car les listes sont une structure de données polymorphe.
- Nous allons en voir d'autres avec les types sommes.

Types sommes

Définition

- Appelés aussi types concrets ou types algébriques.
- Type formé d'une liste de cas possibles pour une valeur de ce type.
- Chaque cas comporte un nom de cas, appelé constructeur, et une (éventuelle) valeur associée (l'argument du constructeur).
- En mathématiques, ce type est appelé union disjointe (ou somme disjointe ou encore somme cartésienne).

Union disjointe

- L'union disjointe de deux ensembles A et B consiste à réunir non pas directement A et B, mais deux ensembles disjoints, copies de A et B de la forme $\{\alpha\} \times A$ et $\{\beta\} \times B$, où α et β sont deux symboles quelconques distincts servant à identifier les ensembles A et B (par exemple 0 et 1) et $\times$ désigne le produit cartésien.

Un cas dégénéré : les types énumérés

Type couleur

- Un cas dégénéré consiste à définir un type dont les constructeurs n'ont pas d'argument (constructeurs constants).
- On parle alors de type énuméré, c'est-à-dire un type somme dont on énumère effectivement les constantes.

```
# type couleur = Bleu | Blanc | Rouge;;
type couleur = Bleu | Blanc | Rouge
# Bleu;;
- : couleur = Bleu
```

- Le symbole « | » se lit « ou ».
- Les noms `Bleu`, `Blanc` et `Rouge` sont les constructeurs du type `couleur`.
- **Attention :** les noms de constructeurs commencent obligatoirement par une majuscule. C'est une contrainte syntaxique (pas une convention).

Un cas dégénéré : les types énumérés

Type couleur

- Lorsqu'une fonction prend en paramètre une expression de type couleur, le filtrage (*pattern matching*) permet de savoir dans quel cas on est.
- On peut ainsi écrire la fonction `rgb`, qui étant donné une expression de type couleur en argument, rend le code RGB (*Red Green Blue*) de la couleur :

```
# let rgb = function
    | Bleu → (0, 0, 255)
    | Blanc → (255, 255, 255)
    | Rouge → (255, 0, 0);;
val rgb : couleur → int * int * int = <fun>
# rgb Blanc;;
- : int * int * int = (255, 255, 255)
```

- Le premier « | » est facultatif, mais c'est préférable de le mettre pour éviter de provoquer des erreurs de syntaxe en cas de copier-coller.

Types sommes : le cas général

Type nombre

- Dans le cas général, les constructeurs ont des arguments : à la définition du type somme on indique leur type après le mot-clé `of`.
- Par exemple, on peut définir le type `number`, qui réunit entiers et flottants au sein d'un même ensemble :

```
# type number =
    | Int of int
    | Float of float;;
type number = Int of int | Float of float
# let i = Int 1;;
val i : number = Int 1
# let f = Float 1.5;;
val f : number = Float 1.5
```

Types sommes : le cas général

Type nombre

- De même, on peut écrire des fonctions sur ce type.
- Par exemple, on peut écrire la fonction `add_number`, qui fait l'addition de deux expressions de type `number` et rend une expression de type `number` :

```
# let add_number n1 n2 =
    match (n1, n2) with
    | (Int i1, Int i2) -> Int (i1 + i2)
    | (Int i, Float f) -> Float ((float_of_int i) +. f)
    | (Float f, Int i) -> Float (f +. (float_of_int i))
    | (Float f1, Float f2) -> Float (f1 +. f2);;
val add_number : number -> number -> number = <fun>
# add_number i f;;
- : number = Float 2.5
```

Types sommes imbriqués

Type carte

- Les types sommes sont des types comme les autres et peuvent donc être imbriqués dans des types sommes ou d'autres types.
- Par exemple, nous pouvons définir les cartes d'un jeu de cartes de la façon suivante (plusieurs modélisations sont possibles, nous en donnons une) :

```
# type enseigne = Trefle | Carreau | Coeur | Pique;;
type enseigne = Trefle | Carreau | Coeur | Pique
# type valeur = As | Deux | Trois | Quatre | Cinq | Six | Sept
| Huit | Neuf | Dix | Valet | Dame | Roi;;
type valeur =
    As | Deux | Trois | Quatre | Cinq | Six | Sept | Huit
  | Neuf | Dix | Valet | Dame | Roi
# type carte = Carte of enseigne * valeur;;
type carte = Carte of enseigne * valeur
```

Types sommes imbriqués

Type carte

```
# let main = [Carte (Pique, As); Carte (Trefle, Valet);
  Carte (Carreau, Sept); Carte (Coeur, Roi);
  Carte (Pique, Dame)];;
val main : carte list =
  [Carte (Pique, As); Carte (Trefle, Valet);
   Carte (Carreau, Sept); Carte (Coeur, Roi);
   Carte (Pique, Dame)]
```

- On peut remarquer que le type `carte` n'a qu'un seul constructeur. Dans ce cas, ce type se contente d'agréger des types de données et ressemble davantage à un type tuple ou enregistrement (que nous verrons plus tard) qu'à un type somme.

Types sommes imbriqués

Type carte

- Comme précédemment, on peut écrire des fonctions manipulant ce type.
- Par exemple, on peut écrire une fonction qui compte le nombre de `Pique` dans une main (représentée comme une liste d'expressions de type `carte`) :

```
# let rec compte_pique = function
  | [] → 0
  | Carte (Pique, _) :: tl → 1 + (compte_pique tl)
  | _ :: tl → compte_pique tl;;
val compte_pique : carte list → int = <fun>
# compte_pique main;;
- : int = 2
```

- On remarquera la profondeur du motif `Carte (Pique, _) :: tl`, où on filtre d'abord sur les listes, ensuite sur les expressions de type `carte`, puis sur les expressions de type `enseigne`.
- Les motifs imbriqués sont un outil très puissant du filtrage d'OCaml !

Types sommes récursifs

Type des entiers de Peano

- Les types sommes peuvent également être récursifs.
- Par exemple, on peut définir les entiers de Peano, qui sont construits à partir de deux constructeurs 0 et s (successeur).
- Ainsi, dans l'arithmétique de Peano, 0 est défini comme 0, 1 comme $s(0)$, 2 comme $s(s(0))$, etc.
- En OCaml, ce type peut être défini comme suit :

```
# type nat = O | S of nat;;
type nat = O | S of nat
# let two = S (S O);;
val two : nat = S (S O)
```

Types sommes récursifs

Type des entiers de Peano

- Dans cette représentation des entiers, la somme de deux entiers peut être définie de la façon suivante :

```
# let rec add_nat n = function
    | O -> n
    | (S p) -> S (add_nat n p);;
val add_nat : nat -> nat -> nat = <fun>
# add_nat two two;;
- : nat = S (S (S (S O)))
```

Types sommes polymorphes

Type des arbres binaires

- Les types sommes peuvent également être polymorphes.
- Par exemple, on peut définir les arbres binaires comme suit (le type des éléments de l'arbre est quelconque) :

```
# type 'a tree =
    | Empty
    | Node of 'a * 'a tree * 'a tree;;
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
let t = Node (1, Node (2, Empty, Empty),
                 Node (3, Node (4, Empty, Empty),
                          Node (5, Empty, Empty)));;
# val t : int tree =
  Node (1, Node (2, Empty, Empty),
   Node (3, Node (4, Empty, Empty), Node (5, Empty, Empty)))
```

- Ici, l'utilisateur doit explicitement indiquer la variable de type (`'a`).
- On remarquera que ce type somme est également récursif.

Types sommes polymorphes

Type des arbres binaires

- On peut ensuite écrire une fonction qui construit la liste des éléments d'un arbre (avec éventuellement des doublons) :

```
# let rec list_tree = function
  | Empty → []
  | Node (e, l, r) → e :: ((list_tree l) @ (list_tree r));;
val list_tree : 'a tree → 'a list = <fun>
# list_tree t;;
- : int list = [1; 2; 3; 4; 5]
```

Types enregistrements

Définition

- Appelés aussi types *records* (en anglais).
- Un type enregistrement est formé d'une liste de champs comportant un nom de champ (le label) et une valeur associée (la valeur du champ).
- Peut être vu comme un type tuple ou un type somme avec un seul constructeur, mais ici, l'avantage est que l'accès aux champs se fait directement avec un nom.

Types enregistrements

Type personne

- Par exemple, on peut définir le type des personnes avec comme informations, leur nom, leur prénom et leur âge :

```
# type personne = {nom : string; prenom : string; age : int};;
type personne = { nom : string; prenom : string; age : int; }
# let p = {nom = "Church"; prenom = "Alonzo"; age = 92};;
val p : personne =
  {nom = "Church"; prenom = "Alonzo"; age = 92}
# p.nom;;
- : string = "Church"
# p.prenom;;
- : string = "Alonzo"
# p.age;;
- : int = 92
```

Types enregistrements

Type personne

- On peut écrire des fonctions qui manipulent des enregistrements.
- Par exemple, on peut écrire la fonction qui étant donné une expression de type `personne` en argument, rend la chaîne de caractères composée du prénom et du nom de la personne :

```
# let nom p = p.prenom ^ " " ^ p. nom;;
val nom : personne → string = <fun>
# nom p;;
- : string = "Alonzo Church"
```

Types enregistrements

Type personne

- On peut faire du filtrage sur presque tous les types de données en OCaml, et les types enregistrements en font partie.
- On peut réécrire la fonction précédente en utilisant le filtrage sur le paramètre `p` :

```
# let nom = function
    | {nom = n; prenom = p; _} -> p ^ " " ^ n;;
val nom : personne -> string = <fun>
# nom p;;
- : string = "Alonzo Church"
```

- On rappelle que le mot-clé `function` indique un argument implicite (non nommé) sur lequel on fait du filtrage.

Types enregistrements

Type personne

- Comme nous ne sommes pas intéressés par le dernier champ (on a mis « _ »), on peut ne mettre dans le motif que les champs dont on veut récupérer la valeur comme suit :

```
# let nom = function
    | {nom = n; prenom = p} -> p ^ " " ^ n;;
val nom : personne -> string = <fun>
# nom p;;
- : string = "Alonzo Church"
```

Types enregistrements

Type personne

- Si l'on veut créer un nouvel enregistrement à partir d'un autre, on peut utiliser le mot-clé `with` qui permet d'indiquer seulement les champs que nous souhaitons changer dans le nouvel enregistrement.
- Par exemple, si nous souhaitons incrémenter l'âge d'une personne, on écrira la fonction suivante :

```
# let incr_age p = {p with age = p.age + 1};;
val incr_age : personne → personne = <fun>
# incr_age p;;
- : personne = {nom = "Church"; prenom = "Alonzo"; age = 93}
```

Types abréviations

Définition

- Un type abréviation définit un alias pour une expression de type.

Un alias pour les entiers

- On peut définir des alias vers les types primitifs d'OCaml.
- Par exemple, on peut définir un alias vers le type `int`.

```
# type entier = int;;
type entier = int
# let x = (1 : entier);;
val x : entier = 1
```

- L'expression `(1 : entier)` permet d'indiquer que 1 est de type `entier`.
- C'est rare de mettre des annotations de type en OCaml (qui infère les types sans aucune aide), mais en l'absence de cette annotation ici, OCaml va inférer que 1 est de type `int` (et non `entier`).

Types abréviations

Un alias pour les entiers

- Les valeurs d'un type abréviation peuvent être considérées comme du type abrégé (OCaml ne fera aucune différence) :

```
# x + 1;;
- : int = 2
```

Types abréviations

Un alias pour le produit cartésien d'entiers

- Les expressions de type utilisées dans les types abréviations peuvent être plus complexes que de simples types primitifs.
- Par exemple, on peut définir un type abréviation pour le produit cartésien d'entiers (opérateur « * ») :

```
# type prod = int * int;;
type prod = int * int
# let c = ((1, 2) : prod);;
val c : prod = (1, 2)
```

Utilisation des types abréviations

- Les types abréviations servent dans les modules (pour définir un type exigé par l'interface du module) ou à des fins de documentation.

Plan du cours

Fonctions partiellement définies et exceptions

Fonctions partielles

- Le domaine de définition d'une fonction correspond à l'ensemble des valeurs sur lesquelles la fonction effectue son calcul.
- La fonction est dite partielle si son domaine de définition est plus petit que l'ensemble des valeurs du type d'entrée.
- Nombreuses fonctions mathématiques partielles : division, logarithme, etc.
- Ce problème se pose aussi pour les fonctions manipulant des structures de données plus complexes : premier élément d'une liste vide, factorielle d'un entier négatif, etc.

Fonctions partiellement définies et exceptions

Gestion des fonctions partielles

- Un certain nombre de situations exceptionnelles peuvent se produire durant l'exécution d'un programme, par exemple une tentative de division par zéro.
- Tenter de diviser un nombre par zéro provoquera au mieux l'arrêt du programme, au pire un état incohérent de la machine.
- La sûreté d'un langage de programmation passe par la garantie de ne pas se retrouver dans une telle situation pour ces cas particuliers.
- Les exceptions sont une manière d'y répondre.

Exemples de levées d'exceptions

Division par 0

- La division de 1 par 0 provoquera la levée d'une exception spécifique :

```
# 1/0;;
Exception: Division_by_zero.
```

- Le message indique d'une part que l'exception `Division_by_zero` a été levée, et que d'autre part elle n'a pas été récupérée.
- Cette exception fait partie des exceptions prédéfinies du langage.

Exemples de levées d'exceptions

Filtrage non exhaustif

- Souvent, le type d'une fonction ne correspond pas à son domaine de définition quand un filtrage de motif n'est pas exhaustif, c'est à dire qu'il ne filtre pas tous les cas de l'expression donnée.
- Pour prévenir une telle erreur, OCaml affiche un avertissement :

```
# let tete l = match l with t::q -> t①;;
val tete : 'a list -> 'a = <fun>
①: Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
```

- Si néanmoins le programmeur maintient sa définition incomplète, OCaml lèvera une exception en cas d'appel erroné à la fonction partielle :

```
# tete [];;
Exception: Match_failure ("//toplevel//", 1, 13).
```

Exemples de levées d'exceptions

Exception prédéfinie `Failure`

- Elle prend un argument de type `string`.
- On peut déclencher cette exception en utilisant la fonction `failwith`.
- On pourra l'utiliser pour compléter notre définition de la fonction `tete` :

```
# let tete = function
    | [] → failwith "Liste␣vide"
    | h::t → h;;
val tete : 'a list → 'a = <fun>

# tete [];;
Exception: Failure "Liste␣vide".
```

Définition d'une exception

Type et déclaration

- Les exceptions appartiennent à un type prédéfini `exn`.
- C'est un type somme extensible : on peut étendre l'ensemble des valeurs du type en déclarant de nouveaux constructeurs.
- La syntaxe de la déclaration d'une exception est la suivante :

  exception *Nom* ; ;

  ou

  exception *Nom* **of** *t* ; ;

Définition d'une exception

Exemples de déclarations d'exceptions

```
# exception A_MOI;;
exception A_MOI

# A_MOI;;
- : exn = A_MOI

# exception Depth of int;;
exception Depth of int

# Depth 4;;
- : exn = Depth 4
```

Définition d'une exception

Attention

- Les noms d'exceptions sont des constructeurs, ils commencent donc obligatoirement par une majuscule :

```
# exception minuscule;;
Syntax error
```

- Les exceptions sont monomorphes, elles n'ont pas de paramètre de type dans la déclaration du type de leur argument :

```
# exception Value of 'a;;
Unbound type parameter 'a
```

Une exception polymorphe autoriserait la définition de fonctions avec un type de retour quelconque.

Levée d'une exception

Principe

- Une exception est levée par l'intermédiaire de la fonction `raise`.
- La fonction `raise` est une fonction primitive du langage.
- Elle prend une exception comme argument et possède un type de retour entièrement polymorphe.
- Lorsqu'une exception est levée, elle interrompt le calcul en cours.
- Si aucun bloc de traitement de l'exception n'est croisé sur le chemin d'évaluation du programme, le programme se termine en indiquant qu'une exception a été levée et non rattrapée.
- Dans le cas du toplevel, l'exception est rattrapée par le toplevel qui ne se termine pas et indique qu'une exception a été levée et non rattrapée.

Levée d'une exception

Exemples de levées d'exceptions

```
# raise;;
- : exn → 'a = <fun>

# raise A_MOI;;
Exception: A_MOI.

# 1 + (raise A_MOI);;
Exception: A_MOI.

# raise (Depth 4);;
Exception: Depth 4.
```

Note

- L'expression `failwith "problème"` est une abréviation pour l'expression `raise (Failure "problème")`.

Rattrapage d'une exception

Principe

- L'intérêt de lever des exceptions réside dans la capacité de les récupérer et d'orienter la suite du calcul selon la valeur de l'exception levée.
- La construction syntaxique suivante, qui calcule la valeur d'une expression, permet la récupération d'une exception levée lors de ce calcul :

```
try expr with
| p1 → expr1
| ...
| pn → exprn
```

Rattrapage d'une exception

Principe

- Si le calcul de `expr` ne lève pas d'exception, alors le résultat est celui du calcul de `expr`.
- Sinon, la valeur de l'exception déclenchée est filtrée. La valeur de l'expression correspondante à la première branche du filtrage correcte est retournée.
- Le type des `expri` doit être le même que le type de `expr`.
- Si aucune branche du filtrage ne correspond à la valeur de l'exception alors celle-ci se propage jusqu'au précédent `try-with` sur le chemin d'évaluation.
- Si aucun bloc de traitement de l'exception n'est croisé sur le chemin d'évaluation du programme, le programme se termine en indiquant qu'une exception a été levée et non rattrapée.
- Dans le cas du toplevel, l'exception est rattrapée par le toplevel qui ne se termine pas et indique qu'une exception a été levée et non rattrapée.

Rattrapage d'une exception

Exemples de rattrapages d'exceptions

```
# try 1 with
    | A_MOI -> 0;;
- : int = 1

# try 1 + (raise A_MOI) with
    | A_MOI -> 0;;
- : int = 0
```

Rattrapage d'une exception

Exemples de rattrapages d'exceptions

```
# let id_10 n =
    if n > 10 then raise (Depth n)
    else n;;
val id_10 : int → int = <fun>

# let appel_id_10 n =
    try print_int (id_10 n); print_newline() with
    | Depth n → print_endline
      "Le␣paramètre␣doit␣être␣inférieur␣ou␣égal␣à␣10␣!";;
val appel_id_10 : int → unit = <fun>
```

Rattrapage d'une exception

Exemples de rattrapages d'exceptions

```
# appel_id_10 5;;
5
- : unit = ()

# appel_id_10 11;;
Le paramètre doit être inférieur ou égal à 10 !
- : unit = ()
```

Levée et évaluation d'une exception

Attention

- Il ne faut pas confondre calculer une exception (c'est-à-dire une valeur de type `exn`) et lever une exception qui effectue une rupture de calcul.
- Une exception étant une valeur comme les autres, elle peut être rendue comme résultat d'une fonction.

```
# let rendre x = Failure x;;
val rendre : string → exn = <fun>

# rendre "test";;
- : exn = Failure "test"

# let lever x = raise (Failure x);;
val lever : string → 'a = <fun>

# lever "test";;
Exception: Failure "test".
```

Programmation avec les exceptions

Programmer avec des ruptures de calcul

- Outre leur utilisation pour le traitement de valeurs non désirées, les exceptions permettent aussi un style de programmation et peuvent être source d'optimisations.

Produit des éléments d'une liste d'entiers

```
# exception Found_zero;;
exception Found_zero

# let rec mult_rec = function
    | [] → 1
    | 0 :: _ → raise Found_zero
    | n :: x → n * (mult_rec x);;
val mult_rec : int list → int = <fun>
```

Produit des éléments d'une liste d'entiers

```
# let mult_list l =
    try mult_rec l with
    | Found_zero → 0;;
val mult_list : int list → int = <fun>

# mult_list [1; 2; 3; 0; 5; 6];;
- : int = 0
```

- On utilise une exception pour interrompre le parcours de la liste et retourner la valeur 0 dès qu'on la rencontre.
- Ainsi tous les calculs restant en attente, à savoir les multiplications par `n` qui suivent chacun des appels récursifs, sont abandonnés. Après avoir rencontré le `raise`, le calcul reprend à partir du filtrage du `try-with`.

Plan du cours

Fonctions d'ordre supérieur

Définition

- Les fonctions d'ordre supérieur ou fonctionnelles sont des fonctions qui ont au moins une des propriétés suivantes :
  - Elles prennent une ou plusieurs fonctions en entrée.
  - Elles renvoient une fonction.
- En mathématiques, on les appelle des opérateurs ou des fonctionnelles. La dérivée en calcul infinitésimal est un exemple, car elle associe une fonction (la dérivée) à une autre fonction (la fonction dont on cherche la dérivée).
- Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Fonctions curryfiées

- En OCaml, toutes les fonctions ayant plus d'un paramètre (plus d'une « flèche » dans leurs types) sont d'ordre supérieur.
- Une fonction de type $\tau_1 \to \tau_2 \to \tau_3$ peut être vue comme une fonction qui prend un paramètre de type τ_1 et rend une fonction de type $\tau_2 \to \tau_3$.
- De telles fonctions sont dites curryfiées.
- La curryfication est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments. L'opération inverse est possible et s'appelle la décurryfication.
- Le terme vient du nom du mathématicien américain Haskell Curry, bien que cette opération ait été introduite pour la première fois par Moses Schönfinkel.
- Les fonctions sous forme curryfiée permettent en particulier de faire de l'application partielle (on ne donne pas tous les arguments à la fonction).

Fonctions curryfiées et application partielle

```
# let add x y = x + y;;
val add : int → int → int = <fun>

# add 1 1;;
- : int = 2

# add 1;;
- : int → int = <fun>

# let succ = add 1;;
val succ : int → int = <fun>

# succ 1;;
- : int = 2
```

Composition de fonctions

- Par exemple, on peut écrire la fonction qui compose (au sens mathématique du terme) deux fonctions :

```
# let comp f g x = f (g x);;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

- Le type de `comp` est bien celui d'une fonction prenant deux fonctions en arguments de type `'a -> 'b` et `'c -> 'a`, où le `'a` montre l'adéquation de l'image de `g` avec le domaine de `f`.

Composition de fonctions

- On peut alors appliquer `comp` à deux fonctions :

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

# let sqr x = x * x;;
val sqr : int → int = <fun>

# let sqr_succ = comp sqr succ;;
val sqr_succ : int → int = <fun>

# sqr_succ 1;;
- : int = 4
```

Tri d'une liste paramétré par l'ordre

- On peut écrire le tri par insertion sur une liste en paramétrant la fonction par l'ordre sur les éléments.
- On écrit d'abord la fonction d'insertion (dans une liste triée) :

```
# let rec insert ord x = function
  | [] → [x]
  | e :: tl →
    if ord x e then x :: e :: tl
    else e :: (insert ord x tl);;
val insert : ('a → 'a → bool) → 'a → 'a list → 'a list = <fun>

# let l = [1; 3; 4; 5];;
val l : int list = [1; 3; 4; 5]

# insert (<=) 2 l;;
- : int list = [1; 2; 3; 4; 5]
```

Tri d'une liste paramétré par l'ordre

- On écrit ensuite la fonction de tri elle-même :

```
# let rec insertion_sort ord = function
    | [] → []
    | e :: tl → insert ord e (insertion_sort ord tl);;
val insertion_sort : ('a → 'a → bool) → 'a list →
'a list = <fun>

# let l = [5; 4; 3; 2; 1];;
val l : int list = [5; 4; 3; 2; 1]

# insertion_sort (<=) l;;
- : int list = [1; 2; 3; 4; 5]
```

Itérateur `map` sur les listes

- Un exemple très courant de fonction d'ordre supérieur est celui d'un itérateur. Dès que l'on a une structure de données de type collection (ensemble, dictionnaire, file, etc.), on l'équipe naturellement d'une manière d'itérer sur tous ses éléments (par exemple pour les afficher ou les compter).
- Sur les listes, on peut écrire la fonction `map`, qui étant données une fonction `f` et une liste `[a1; ...; an]`, rend la liste `[f a1; ...; f an]` :

```
# let rec map f = function
    | [] → []
    | e :: tl → (f e) :: (map f tl);;
val map : ('a → 'b) → 'a list → 'b list = <fun>

# map succ [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

Itérateurs sur les listes

Notion d'itérateur

- La plupart des structures de données OCaml fournissent des itérateurs, souvent avec plusieurs variantes, y compris les structures impératives usuelles telles que les tableaux, les files, les tables de hachage, etc.
- Pour cette raison en particulier, il est fréquent d'utiliser des fonctions anonymes que l'on passe en paramètre à ces itérateurs (comme le `map`).
- Certains itérateurs rendent un objet de même type que l'objet sur lequel on itère (comme le `map`), mais d'autres rendent des valeurs d'un type différent.
- La bibliothèque des listes contient de nombreux itérateurs, qu'il convient de savoir manipuler afin de pouvoir écrire des fonctions assez puissantes en très peu de lignes de code.
- Tous les itérateurs de la bibliothèque `List` sont décrits à l'adresse suivante : `https://ocaml.org/api/List.html`.

Itérateurs sur les listes

Itérateur `map`

- Étant données une fonction `f` et une liste `[a1; ...; an]`, la fonction `List.map` rend la liste `[f a1; ...; f an]` :

```
# List.map succ [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

- On peut bien sûr utiliser une fonction anonyme en paramètre :

```
# List.map (fun e -> e + 1) [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]
```

Itérateurs sur les listes

Itérateur `fold_left`

- On pourra également trouver la fonction `List.fold_left`, qui étant données une fonction `f`, une expression `a` et une liste `[b1; ...; bn]`, rend l'expression `f (... (f (f a b1) b2) ...) bn`.
- Elle permet, par exemple, de sommer les éléments d'une liste d'entiers :

```
# List.fold_left (fun a e -> a + e) 0 [1; 2; 3; 4; 5];;
- : int = 15
```

- Si on note `f` la fonction `(fun a e -> a + e)`, cette expression est équivalente à :
  - `f (f (f (f (f 0 1) 2) 3) 4) 5`
  - `f (f (f (f 1 2) 3) 4) 5`
  - `f (f (f 3 3) 4) 5`
  - `f (f 6 4) 5`
  - `f 10 5`
  - `15`

Itérateurs sur les listes

Itérateur `fold_right`

- La fonction `List.fold_right` se comporte comme la fonction `List.fold_left` mais en commençant par la fin de la liste.
- La fonction passée en paramètre prend l'élément de la liste (sur laquelle on effectue l'itération), puis l'accumulateur (ordre inversé par rapport à la fonction `List.fold_left`).
- Étant données une fonction `f`, une liste `[b1; ...; bn]` et une expression `a`, `List.fold_right` rend l'expression `f b1 (f b2 (... (f bn a) ...))`.
- Si la fonction `f` est associative (comme pour la fonction + par exemple), alors le résultat de `List.fold_right` est le même que celui de `List.fold_left`.

Itérateurs sur les listes

Itérateur `fold_right`

```
# List.fold_right (fun e a → e + a) [1; 2; 3; 4; 5] 0;;
- : int = 15

# List.fold_left (fun a e → a + e) 0 [1; 2; 3; 4; 5];;
- : int = 15

# List.fold_right (-) [1; 2; 3; 4; 5] 0;;
- : int = 3

# List.fold_left (-) 0 [1; 2; 3; 4; 5];;
- : int = -15
```

Itérateurs sur les listes

Itérateur `filter`

- Il y a aussi la fonction `List.filter`, qui étant données une fonction `p` (un prédicat) et une liste `l`, rend tous les éléments de la liste `l` qui vérifient le prédicat `p`.
- On peut, par exemple, sélectionner uniquement les éléments non négatifs d'une liste d'entiers :

```
# List.filter (fun e -> e >= 0) [1; 0; -2; 3; -4];;
- : int list = [1; 0; 3]
```

Itérateurs sur les listes

Itérateur `for_all`

- Il y a également des itérateurs qui combinent des opérateurs logiques.
- Par exemple, on pourra utiliser la fonction `List.for_all`, qui étant données une fonction `p` (un prédicat) et une liste `[a1; ...; an]`, vérifie que le prédicat `p` est vérifié pour tous les éléments `ai` de la liste et qui rend donc l'expression `(f a1) && (f a2) && ... && (f an)`.
- Si la liste passée en paramètre est vide, la fonction rend `true`.

```
# List.for_all (fun e -> e >= 0) [1; 0; 2; 3; 4];;
- : bool = true
```

Itérateurs sur les listes

Itérateur `exists`

- De même, on pourra trouver la fonction `List.exists`, qui étant données une fonction `p` (un prédicat) et une liste `[a1; ...; an]`, vérifie que le prédicat `p` est vérifié pour au moins un des éléments `ai` de la liste et qui rend donc l'expression `(f a1) || (f a2) || ... || (f an)`.
- Si la liste passée en paramètre est vide, la fonction rend `false`.

```
# List.exists (fun e -> e < 0) [1; 0; -2; 3; 4];;
- : bool = true
```

Itérateurs sur les listes

Itérateur `iter`

- Il est également possible d'itérer des fonctions qui font des effets de bord (elles retournent le type `unit`).
- Par exemple, il y a la fonction `List.iter`, qui étant données une fonction `f` (qui fait un effet de bord) et une liste `[a1; ...; an]`, itère la fonction `f` sur chaque élément `ai` de la liste séquentiellement, ce qui équivaut à rendre l'expression `f a1; f a2; ...; f an; ()`.

```
# List.iter (fun e -> print_int e; print_newline ())
    [1; 2; 3; 4; 5];;
1
2
3
4
5
- : unit = ()
```

Encodage de types de données dans du fonctionnel pur

Booléens

- On n'a que les fonctions et rien d'autre (pas de types de données).
- On encode *true* et *false* de la façon suivante :

```
# let trueb a b = a;;
val trueb : 'a → 'b → 'a = <fun>

# let falseb a b = b;;
val falseb : 'a → 'b → 'b = <fun>

# trueb 1 2;;
- : int = 1

# falseb 1 2;;
- : int = 2
```

Encodage de types de données dans du fonctionnel pur

Booléens

- On peut programmer la conditionnelle comme suit :

```
# let ifthenelse c a b = c a b;;
val ifthenelse : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# ifthenelse trueb 1 2;;
- : int = 1

# ifthenelse falseb 1 2;;
- : int = 2
```

Encodage de types de données dans du fonctionnel pur

Booléens

- On peut programmer les connecteurs booléens :

```
# let andb a b = ifthenelse a b falseb;;
val andb : ('a → ('b → 'c → 'c) → 'd) → 'a → 'd = <fun>

# let res_andb_1 = andb trueb trueb;;
# let res_andb_2 = andb trueb falseb;;

# res_andb_1 1 2;;
- : int = 1

# res_andb_2 1 2;;
- : int = 2
```

Encodage de types de données dans du fonctionnel pur

Booléens

- On peut programmer les connecteurs booléens :

```
# let notb a = ifthenelse a falseb trueb;;

# let res_notb_1 = notb trueb;;
# let res_notb_2 = notb falseb;;

# res_notb_1 1 2;;
# res_notb_2 1 2;;
```

Plan du cours

Optimiser les appels terminaux

Appels de fonctions

- On utilise une zone de la mémoire appelée la pile.
- On empile plusieurs choses :
  - Les paramètres effectifs de la fonction.
  - L'adresse de retour de la fonction.
  - Les variables locales à la fonction.
- Tout appel de fonction crée une trame de pile.
- C'est complètement transparent pour le développeur OCaml.
- C'est le compilateur qui se charge de générer le code correspondant.

Optimiser les appels terminaux

Exemple d'appel de fonction

```
let sqr x = x * x;;
let succ_sqr x = 1 + (sqr x);;
let main = succ_sqr 2;;
```

Optimiser les appels terminaux

Compilation correspondante en assembleur MIPS

```
sqr :       mul $v0, $a0, $a0
            jr $ra

succ_sqr :  addiu $sp, $sp, -4
            sw $ra, 0($sp)
            jal sqr
            addiu $v0, $v0, 1
            lw $ra, 0($sp)
            addiu $sp, $sp, 4
            jr $ra

main :      li $a0, 2
            jal succ_sqr
```

Optimiser les appels terminaux

Appels terminaux

- Un appel g dans une fonction h est dit terminal si cet appel est la dernière opération effectuée dans h.
- En particulier, dans ce cas là, on voit que le résultat de g devient celui de h, donc h peut « passer la main » à g.
- Une fonction est terminale si tous ses appels de fonctions (dans son corps) sont terminaux.
- On cherche à optimiser les appels terminaux.

Optimiser les appels terminaux

Exemple de fonction terminale

```
let sqr x = x * x;;
let call_sqr x = sqr x;;
let main = succ_sqr 2;;
```

Optimiser les appels terminaux

Compilation correspondante en assembleur MIPS

```
sqr :       mul $v0, $a0, $a0
            jr $ra

call_sqr :  addiu $sp, $sp, -4
            sw $ra, 0($sp)
            jal sqr
            lw $ra, 0($sp)
            addiu $sp, $sp, 4
            jr $ra

main :      li $a0, 2
            jal call_sqr
```

Optimisation des appels terminaux

Principe général

- Après le retour d'un appel terminal, l'appelant se contentera de détruire sa trame puis de passer la main à son propre appelant.
- Par conséquent, la trame de l'appelant n'a plus lieu d'être avant même l'appel, et il vaut mieux que l'appelé rende directement la main à l'appelant de l'appelant.

Optimisation des appels terminaux

Concrètement en MIPS

Supposons que l'on ait un appel terminal à g dans h. Celui-ci sera compilé de la manière optimisée suivante :

- la valeur initiale de $ra, qui contient donc l'adresse de retour dans l'appelant de h, est restaurée.
- La trame de h est désallouée.
- Les arguments de g sont passés comme d'habitude, les quatre premiers dans les registres $a0 à $a3, les autres sur la pile.
- Le contrôle est transféré à g par un simple saut (j au lieu de jal).

Du point de vue de l'appelé g, tout se passe comme s'il était appelé par l'appelant de h.

Si tous les appels sont terminaux dans h, il sera inutile d'empiler $ra et il n'y aura peut-être même aucune trame de pile !

Optimiser les appels terminaux

Compilation optimisée en assembleur MIPS

```
sqr :       mul $v0, $a0, $a0
            jr $ra

call_sqr :  addiu $sp, $sp, -4
            sw $ra, 0($sp)
            lw $ra, 0($sp)
            addiu $sp, $sp, 4
            j sqr

main :      li $a0, 2
            jal call_sqr
```

Optimiser les appels terminaux

Compilation optimisée en assembleur MIPS

```
sqr :       mul $v0, $a0, $a0
            jr $ra

call_sqr :  j sqr

main :      li $a0, 2
            jal call_sqr
```

La fonction `call_sqr` n'est qu'une indirection !
L'optimisation est transparente pour l'utilisateur, c'est le compilateur qui la réalise.

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

Quelle optimisation pour les fonctions récursives ?

Compilation correspondante en assembleur MIPS

```
fact:  addiu $sp, $sp, -8
       sw $ra, 4($sp)
       sw $s0, 0($sp)
       move $s0, $a0
       blez $s0, base
       addiu $a0, $s0, -1
       jal fact
       mul $v0, $s0, $v0
end:   lw $ra, 4($sp)
       lw $s0, 0($sp)
       addiu $sp, $sp, 8
       jr $ra
base:  li $v0, 1
       j end
```

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- fact 3 (une trame) : 3 * (fact 2)
- fact 2 (une trame) : 2 * (fact 1)
- fact 1 (une trame) : 1 * (fact 0)
- fact 0 (une trame) : 1

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2)`
- `fact 2` (une trame) : `2 * (fact 1)`
- `fact 1` (une trame) : `1 * (fact 0)`
- `fact 0` (une trame) : `1`

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2)`
- `fact 2` (une trame) : `2 * (fact 1)`
- `fact 1` (une trame) : `1 * (fact 0)  = 1`
- `fact 0` (une trame) : `1`

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2)`
- `fact 2` (une trame) : `2 * (fact 1)  = 2`
- `fact 1` (une trame) : `1 * (fact 0)  = 1`
- `fact 0` (une trame) : `1`

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2) = 6`
- `fact 2` (une trame) : `2 * (fact 1) = 2`
- `fact 1` (une trame) : `1 * (fact 0) = 1`
- `fact 0` (une trame) : `1`

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2)  = 6`
- `fact 2` (une trame) : `2 * (fact 1)  = 2`
- `fact 1` (une trame) : `1 * (fact 0)  = 1`
- `fact 0` (une trame) : `1`

Donc, `(fact n)` crée `n` + 1 trames de pile.

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

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

let main = fact 3;;
```

Combien de trames allouées ?

- `fact 3` (une trame) : `3 * (fact 2)  = 6`
- `fact 2` (une trame) : `2 * (fact 1)  = 2`
- `fact 1` (une trame) : `1 * (fact 0)  = 1`
- `fact 0` (une trame) : `1`

Donc, `(fact n)` crée `n` + 1 trames de pile.
On souhaite éliminer des trames par des appels terminaux.

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;
```

On appelle la fonction avec `(fact n 1)`.
Le paramètre `acc` est un accumulateur qui stocke les résultats intermédiaires, qui sont calculés progressivement et non après coup comme avant.

L'appel à `fact` est bien terminal.
On peut appliquer notre optimisation sur les appels terminaux.

Quelle optimisation pour les fonctions récursives ?

Compilation correspondante en assembleur MIPS

```
fact:  addiu $sp, $sp, -4
       sw $ra, 4($sp)
       blez $a0, base
       mul $a1, $a0, $a1
       addiu $a0, $a0, -1
       lw $ra, 0($sp)
       addiu $sp, $sp, 4
       j fact
end:   lw $ra, 0($sp)
       addiu $sp, $sp, 4
       jr $ra
base:  move $v0, $a1
       j end
```

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, (`fact n 1`) crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : `6`

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, `(fact n 1)` crée `n + 1` trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, (`fact n 1`) crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, (`fact n 1`) crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, `(fact n 1)` crée `n` + 1 trames de pile.

Mais pas simultanément sur la pile ; la taille de la pile est constante.

La pile contient toujours l'adresse de retour de l'appelant (le `main`).

On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, `(fact n 1)` crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, `(fact n 1)` crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Fonction factorielle non terminale

```
let rec fact n acc =
  if n = 0 then acc
  else fact (n - 1) (n * acc);;

let main = fact 3 1;;
```

Combien de trames allouées ?

- `fact 3 1` (une trame allouée puis désallouée) : `fact 2 (3 * 1)`
- `fact 2 3` (une trame allouée puis désallouée) : `fact 1 (3 * 2)`
- `fact 1 6` (une trame allouée puis désallouée) : `fact 0 (6 * 1)`
- `fact 0 6` (une trame allouée puis désallouée) : 6

Donc, `(fact n 1)` crée `n` + 1 trames de pile.
Mais pas simultanément sur la pile ; la taille de la pile est constante.
La pile contient toujours l'adresse de retour de l'appelant (le `main`).
On peut donc optimiser encore en ne l'empilant pas du tout puisque tous les appels dans `fact` sont terminaux !

Quelle optimisation pour les fonctions récursives ?

Compilation encore plus optimisée en assembleur MIPS

```
fact: blez $s0, base
      mul $a1, $a0, $a1
      addiu $a0, $a0, -1
      j fact
end:  jr $ra
base: move $v0, $a1
      j end
```

Il n'y a plus aucune trame de pile dans ce cas.
La fonction récursive ressemble à sa version impérative.
Elle a été dérécursivée.

Quelle optimisation pour les fonctions récursives ?

Fonctions récursives terminales

```
(* Version non terminale *)
let rec sum_list = function
| [] → 0
| e::tl → e + (sum_list tl);;

(* Version terminale *)
let rec sum_list_aux acc = function
| [] → acc
| e::tl → sum_list (acc + e) tl;;

let sum_list = sum_list_aux 0;;
```

Écrire les fonctions récursives terminales que nous avons déjà vues dans ce cours est plutôt un exercice facile.
Généralement, nous n'avons besoin que d'un seul accumulateur.
Cela se complique si nous avons besoin de plusieurs accumulateurs.

Quelle optimisation pour les fonctions récursives ?

Fonction de Fibonacci

```
let rec fib n =
  if n <= 1 then n
  else (fib (n - 1)) + (fib (n - 2));;
```

Le nombre de calculs est environ le nombre de nœuds dans un arbre binaire de hauteur n, soit 2^n (exponentiel).
Pour écrire sa version terminale, le mieux est d'écrire sa version impérative (dans un autre langage ou un pseudo-langage) afin de connaître le nombre de variables (accumulateurs) nécessaires.

Quelle optimisation pour les fonctions récursives ?

Fonction de Fibonacci (version impérative)

```
function fib(n : integer) return integer is
   first  : integer := 1;
   second : integer := 0;
   tmp    : integer;
   begin
     for i in 1..n loop
       tmp    := first + second;
       second := first;
       first  := tmp;
     end loop;
   return second;
end fib;
```

On a besoin de deux variables (*first* et *second*).

Quelle optimisation pour les fonctions récursives ?

Fonction de Fibonacci (version récursive terminale)

```
let rec fib_aux n first second =
  if n = 0 then second
  else fib_aux (n - 1) second (first + second);;

let fib n = fib_aux n 1 0;;
```

Quelle optimisation pour les fonctions récursives ?

Soit la fonction suivante

$$f(n) = \begin{cases} n - 10, \text{ si } n > 100 \\ f(f(n+11)), \text{ sinon} \end{cases}$$

Que calcule cette fonction pour $n \leq 101$?

$$\begin{aligned} f(99) &= f(f(110)) \text{ car } 99 \leq 100 \\ &= f(100) \text{ car } 110 > 100 \\ &= f(f(111)) \text{ car } 100 \leq 100 \\ &= f(101) \text{ car } 111 > 100 \\ &= 91 \text{ car } 101 > 100 \end{aligned}$$

Cette fonction rend toujours 91 pour $n \leq 101$.
C'est la fonction 91 de John McCarthy (prix Turing en 1971).

Quelle serait la version terminale de la fonction de McCarthy ?

Quelle optimisation pour les fonctions récursives ?

Soit la fonction suivante

$$f(n) = \begin{cases} n - 10, \text{ si } n > 100 \\ f(f(n+11)), \text{ sinon} \end{cases}$$

Que calcule cette fonction pour $n \leq 101$?

$$\begin{aligned} f(99) &= f(f(110)) \text{ car } 99 \leq 100 \\ &= f(100) \text{ car } 110 > 100 \\ &= f(f(111)) \text{ car } 100 \leq 100 \\ &= f(101) \text{ car } 111 > 100 \\ &= 91 \text{ car } 101 > 100 \end{aligned}$$

Cette fonction rend toujours 91 pour $n \leq 101$.
C'est la fonction 91 de John McCarthy (prix Turing en 1971).

Quelle serait la version terminale de la fonction de McCarthy ?

Quelle optimisation pour les fonctions récursives ?

Soit la fonction suivante

$$f(n) = \begin{cases} n - 10, \text{ si } n > 100 \\ f(f(n+11)), \text{ sinon} \end{cases}$$

Que calcule cette fonction pour $n \leq 101$?

$$\begin{aligned} f(99) &= f(f(110)) \text{ car } 99 \leq 100 \\ &= f(100) \text{ car } 110 > 100 \\ &= f(f(111)) \text{ car } 100 \leq 100 \\ &= f(101) \text{ car } 111 > 100 \\ &= 91 \text{ car } 101 > 100 \end{aligned}$$

Cette fonction rend toujours 91 pour $n \leq 101$.
C'est la fonction 91 de John McCarthy (prix Turing en 1971).

Quelle serait la version terminale de la fonction de McCarthy ?

Quelle optimisation pour les fonctions récursives ?

Version récursive terminale de la fonction de McCarthy

$$g(n,c) = \begin{cases} n, \text{ si } c = 0 \\ g(n-10, c-1), \text{ si } n > 100 \text{ et } c \neq 0 \\ g(n+11, c+1), \text{ si } n \leq 100 \text{ et } c \neq 0 \end{cases}$$

$$f(n) = g(n,1)$$

Il n'y a pas vraiment de méthodes pour passer des fonctions récursives générales à des fonctions récursives terminales.
On peut cependant citer la transformation CPS (Continuation-Passing Style) qui élimine la notion de pile mais la représente en réalité sous la forme d'une clôture.
Donc, cette transformation ne résout pas forcément le problème.

Quelle optimisation pour les fonctions récursives ?

Version récursive terminale de la fonction de McCarthy

$$g(n,c) = \begin{cases} n, \text{ si } c = 0 \\ g(n-10, c-1), \text{ si } n > 100 \text{ et } c \neq 0 \\ g(n+11, c+1), \text{ si } n \leq 100 \text{ et } c \neq 0 \end{cases}$$

$$f(n) = g(n,1)$$

Il n'y a pas vraiment de méthodes pour passer des fonctions récursives générales à des fonctions récursives terminales.
On peut cependant citer la transformation CPS (Continuation-Passing Style) qui élimine la notion de pile mais la représente en réalité sous la forme d'une clôture.
Donc, cette transformation ne résout pas forcément le problème.