

Programmation fonctionnelle

Samuele Giraud

Université Gustave Eiffel

LIGM, bureau 4B162

`samuele.giraud@u-pem.fr`

`http://igm.univ-mlv.fr/~giraud/`

Prélude n°1

LES ARBRES BINAIRES DE RECHERCHE EN PROGRAMMATION FONCTIONNELLE

Arbres binaires

Un arbre binaire est

Arbres binaires

Un **arbre binaire** est

1. soit une feuille



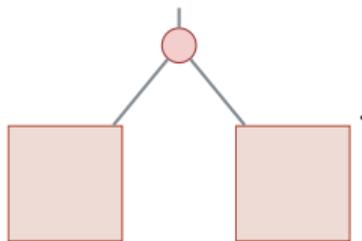
Arbres binaires

Un **arbre binaire** est

1. soit une feuille



2. soit un nœud attaché à deux arbres binaires



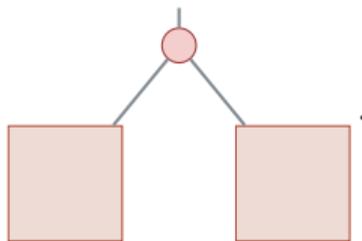
Arbres binaires

Un **arbre binaire** est

1. soit une feuille



2. soit un nœud attaché à deux arbres binaires



C'est une définition **récursive** car la définition de la notion fait référence à la notion elle-même.

Arbres binaires en CAML

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

Arbres binaires en CAML

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

Définition du type arbre binaire en CAML :

```
type arbre_b =  
  |Feuille  
  |Noeud of arbre_b * int * arbre_b
```

Arbres binaires en CAML

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

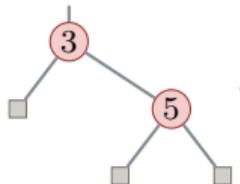
Définition du type arbre binaire en CAML :

```
type arbre_b =  
  |Feuille  
  |Noeud of arbre_b * int * arbre_b
```

Construction d'un arbre binaire en CAML :

```
Noeud (Feuille, 3, Noeud (Feuille, 5, Feuille))
```

Cette expression désigne l'arbre binaire



Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Cet arbre se définit de la manière suivante :

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Cet arbre se définit de la manière suivante :

1. si $T = \square$, alors



Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

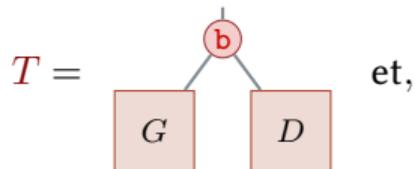
On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Cet arbre se définit de la manière suivante :

1. si $T = \square$, alors



2. sinon, T est de la forme



Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

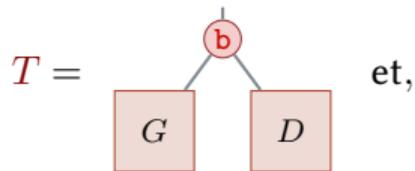
On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Cet arbre se définit de la manière suivante :

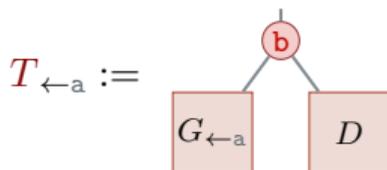
1. si $T = \square$, alors



2. sinon, T est de la forme



- si $a \leq b$, alors



Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

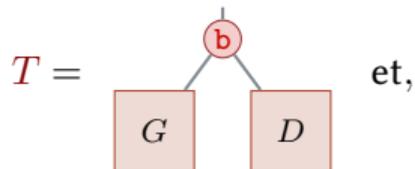
On note $T_{\leftarrow a}$ l'arbre binaire obtenu après **insertion** de la lettre a dans T .

Cet arbre se définit de la manière suivante :

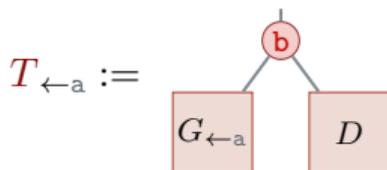
1. si $T = \square$, alors



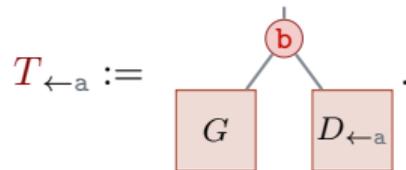
2. sinon, T est de la forme



- si $a \leq b$, alors



- sinon $a > b$ et



Insertion dans un arbre binaire de recherche en CAML

Définition de la fonction (récursive) d'insertion dans un arbre binaire de recherche en CAML :

```
let rec inserer t a =  
  match t with  
  |Feuille -> Noeud (Feuille, a, Feuille)  
  |Noeud (g, b, d) when a <= b -> Noeud (inserer g a, b, d)  
  |Noeud (g, b, d) -> Noeud (g, b, inserer d a)
```

Insertion dans un arbre binaire de recherche en CAML

Définition de la fonction (récursive) d'insertion dans un arbre binaire de recherche en CAML :

```
let rec inserer t a =  
  match t with  
  |Feuille -> Noeud (Feuille, a, Feuille)  
  |Noeud (g, b, d) when a <= b -> Noeud (inserer g a, b, d)  
  |Noeud (g, b, d) -> Noeud (g, b, inserer d a)
```

Construction d'un arbre binaire de recherche par insertions successives :

```
let t1 = inserer Feuille 3 in  
let t2 = inserer t1 4 in  
let t3 = inserer t2 2 in  
inserer t3 1
```

Prélude n°2

INSTRUCTIONS VS EXPRESSIONS

Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le paradigme impératif, on propose en PYTHON la fonction ci-contre :

```
def f(lst) :  
    res = []  
    for i in range(len(lst)) :  
        if i % 2 == 0 :  
            res.append(lst[i])  
    return res
```

Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le paradigme impératif, on propose en PYTHON la fonction ci-contre :

```
def f(lst) :  
    res = []  
    for i in range(len(lst)) :  
        if i % 2 == 0 :  
            res.append(lst[i])  
    return res
```

Le calcul de `f([0, 1, 2, 3, 4])` utilise de la mémoire :

- ▶ pour construire le résultat `res` ;
- ▶ pour maintenir la variable `i` ;
- ▶ éventuellement pour les fonctions appelées (`range`, `append`, *etc.*);
- ▶ pour enregistrer l'état de la machine (adresse de l'instruction exécutée).

Les variables `res` et `i` sont lues et modifiées au cours du temps.

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

```
f([0; 1; 2; 3; 4])
```

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])` → `0 :: f([2; 3; 4])`

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])` → `0 :: f([2; 3; 4])` → `0 :: 2 :: f([4])`

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])` → `0 :: f([2; 3; 4])` → `0 :: 2 :: f([4])` → `0 :: 2 :: [4]`

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])` → `0 :: f([2; 3; 4])` → `0 :: 2 :: f([4])` → `0 :: 2 :: [4]` \rightsquigarrow `[0; 2; 4]`

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'expression `f([0; 1; 2; 3; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])` → `0 :: f([2; 3; 4])` → `0 :: 2 :: f([4])` → `0 :: 2 :: [4]` \rightsquigarrow `[0; 2; 4]`

Dans ce cas, il n'y a

- ▶ ni utilisation externe de la mémoire;
- ▶ ni état d'avancement du calcul qui dépend du temps.

Le calcul se fait en **réécrivant** l'expression tant que possible. La définition de **fonctions** permet d'expliquer comment réaliser les réécritures.

Prélude n°3

PRINCIPES GÉNÉRAUX

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.

~> le programmeur se concentre sur des aspects moins techniques et plus importants.

- ▶ Programmer = **penser**.

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- ▶ Programmer = **penser**.

↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.
↪ le programmeur se concentre sur des aspects moins techniques et plus importants.
- ▶ Programmer = **penser**.
↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.
- ▶ Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

Quelques caractéristiques du CAML

- ▶ **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- ▶ Programmer = **penser**.

↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

- ▶ Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

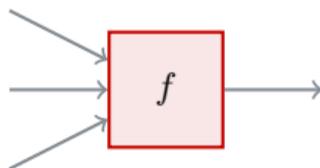
↪ en pratique :

*« Le compilateur CAML n'accepte presque jamais mon code
mais
quand il l'accepte, ça marche comme je le voulais. »*

*« Le compilateur **X** accepte presque toujours mon code
mais
ça ne marche presque jamais comme je le voulais. »*

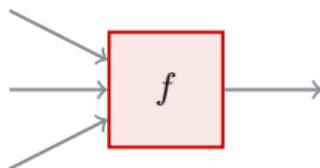
Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



Principes de la programmation fonctionnelle

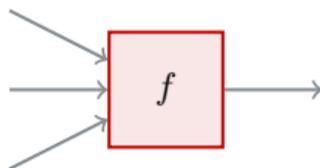
La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :

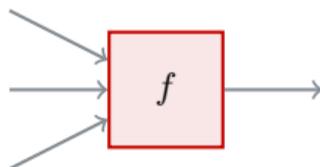


On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

On **évite les séquences d'instructions impératives** (et donc, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

On **évite les séquences d'instructions impératives** (et donc, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

On utilise en revanche l'**application de fonctions** et la **récurtivité**.

Pré-requis

Ce cours demande les pré-réquis suivants :

- ▶ des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets);

Pré-requis

Ce cours demande les pré-réquis suivants :

- ▶ des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets);
- ▶ des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord);

Pré-requis

Ce cours demande les pré-réquis suivants :

- ▶ des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets);
- ▶ des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord);
- ▶ des connaissances solides du **langage C** ou du **langage PYTHON**;

Pré-requis

Ce cours demande les pré-réquis suivants :

- ▶ des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets);
- ▶ des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord);
- ▶ des connaissances solides du **langage C** ou du **langage PYTHON**;
- ▶ des connaissances de base en **algorithmique** (récursivité, manipulation de listes, d'arbres).

Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1^{re} approche du **paradigme de programmation fonctionnelle**;
2. d'apprendre les bases de **programmation en CAML** en mode fonctionnel.

Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1^{re} approche du **paradigme de programmation fonctionnelle**;
2. d'apprendre les bases de **programmation en CAML** en mode fonctionnel.

Celui-ci est organisé en trois axes.

1. **Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1^{re} approche du **paradigme de programmation fonctionnelle**;
2. d'apprendre les bases de **programmation en CAML** en mode fonctionnel.

Celui-ci est organisé en trois axes.

1. **Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

2. **Axe 2 : concepts premiers.**

Programmation en CAML, liaisons, fonctions, entrées / sorties, compilation, types.

Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1^{re} approche du **paradigme de programmation fonctionnelle**;
2. d'apprendre les bases de **programmation en CAML** en mode fonctionnel.

Celui-ci est organisé en trois axes.

1. **Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

2. **Axe 2 : concepts premiers.**

Programmation en CAML, liaisons, fonctions, entrées / sorties, compilation, types.

3. **Axe 3 : concepts avancés.**

Récursivité terminale, filtrage, fonctions d'ordre supérieur, polymorphisme, stratégies d'évaluation, opérations sur les listes, non mutabilité, λ -calcul.

Contenu du cours

Axe 1.

Théorie

Caractéristiques

Axe 2.

Programmation

Pratique

Types

Axe 3.

Notions

Listes

λ -calcul

Bibliographie

Bibliographie non exhaustive :

- ▶ X. Leroy, P. Weis, *Le langage Caml*, Dunod, 2^e édition, 2009.
Lien : <http://caml.inria.fr/distrib/books/llc.pdf>
- ▶ E. Chailloux, P. Manoury, B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly, 2000.
- ▶ X. Leroy *et al.*, The OCaml system release 4.11, August 19, 2020.
Lien : <http://caml.inria.fr/pub/docs/manual-ocaml/>
- ▶ G. Dowek, J.-J. Lévy, *Introduction à la théorie des langages de programmation*, École Polytechnique, 2006.
- ▶ C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1999.
- ▶ S. L. Peyton Jones, D. Lester, *Implementing Functional Languages*, Prentice Hall, 1992.
- ▶ P. Hudak, D. Quick, *The Haskell School of Music : From Signals to Symphonies*, Cambridge University Press, 2018.

Axe 1 : bases théoriques

Théorie

Caractéristiques

Plan

Théorie

Machines de Turing

Décidabilité et indécidabilité

λ -calcul

Plan

Théorie

Machines de Turing

Décidabilité et indécidabilité

λ -calcul

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- ▶ 1936 : Church introduit le **λ -calcul**.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- ▶ 1936 : Church introduit le **λ -calcul**.
- ▶ 1936 : Turing invente la **machine de Turing**.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- ▶ 1936 : Church introduit le **λ -calcul**.
- ▶ 1936 : Turing invente la **machine de Turing**.
- ▶ Années 1940 : programmation en assembleur et en langage machine.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- ▶ 1936 : Church introduit le **λ -calcul**.
- ▶ 1936 : Turing invente la **machine de Turing**.
- ▶ Années 1940 : programmation en assembleur et en langage machine.
- ▶ Années 1950-1960 : 1^{ers} vrais **langages de programmation** : FORTRAN, COBOL et LISP.

Brève chronologie de la programmation

- ▶ 1801 : 1^{re} machine programmable : le **métier à tisser** de Jackard.
- ▶ 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- ▶ 1936 : Church introduit le **λ -calcul**.
- ▶ 1936 : Turing invente la **machine de Turing**.
- ▶ Années 1940 : programmation en assembleur et en langage machine.
- ▶ Années 1950-1960 : 1^{ers} vrais **langages de programmation** : FORTRAN, COBOL et LISP.
- ▶ Années 1960-1970 : **paradigmes de programmation** : impératif, fonctionnel, orienté objet, logique.

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

1. E est un ensemble fini d'états;

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

1. E est un ensemble fini d'états;
2. $i \in E$ est l'état initial;

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

1. E est un ensemble fini d'états;
2. $i \in E$ est l'état initial;
3. $t \in E$ est l'état terminal;

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

1. E est un ensemble fini d'états;
2. $i \in E$ est l'état initial;
3. $t \in E$ est l'état terminal;
4. $\Delta : E \times \{\cdot, 0, 1\} \rightarrow E \times \{\cdot, 0, 1\} \times \{G, D\}$ est une fonction de transition.

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

L'**exécution** de M sur u consiste à :

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

L'**exécution** de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le **ruban** :

| | | | | | | |
|-------|-------|---------|-----------|---|---|---------|
| u_1 | u_2 | \dots | $u_{ u }$ | . | . | \dots |
|-------|-------|---------|-----------|---|---|---------|

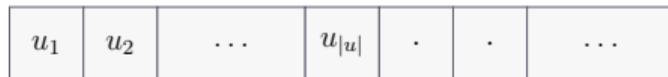
Les cases à droite de u sont remplies jusqu'à l'infini de .

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

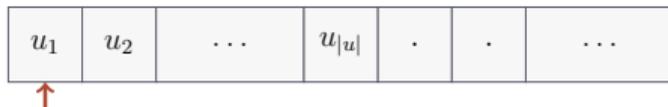
L'**exécution** de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de u sont remplies jusqu'à l'infini de .

2. Placer la **tête de lecture / écriture** sur la 1^{re} case du ruban :



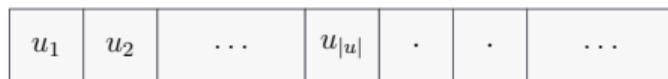
On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture / écriture à un instant donné.

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

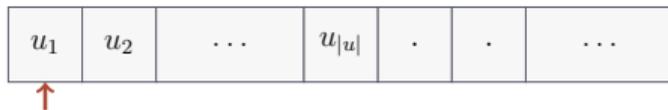
L'**exécution** de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de u sont remplies jusqu'à l'infini de .

2. Placer la **tête de lecture / écriture** sur la 1^{re} case du ruban :



On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture / écriture à un instant donné.

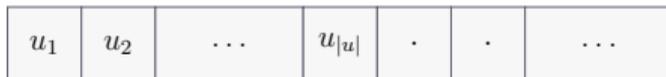
3. Affecter au **registre d'état** e la valeur i (l'état initial).

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

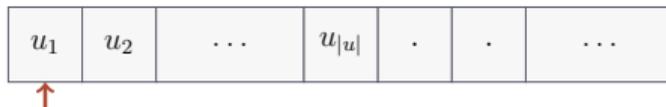
L'**exécution** de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de u sont remplies jusqu'à l'infini de .

2. Placer la **tête de lecture / écriture** sur la 1^{re} case du ruban :



On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture / écriture à un instant donné.

3. Affecter au **registre d'état** e la valeur i (l'état initial).
4. Réaliser les actions dictées par Δ , le **programme**.

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

(1) calculer $(e', a', s') := \Delta(e, a)$.

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

- (1) calculer $(e', a', s') := \Delta(e, a)$.
- (2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

- (1) calculer $(e', a', s') := \Delta(e, a)$.
- (2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.
- (3) Affecter au registre d'état e l'état e' .

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

- (1) calculer $(e', a', s') := \Delta(e, a)$.
- (2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.
- (3) Affecter au registre d'état e l'état e' .
- (4) Si $s' = D$, déplacer la tête de lecture / écriture d'un pas vers la droite; sinon, la déplacer si possible d'un pas vers la gauche.

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

- (1) calculer $(e', a', s') := \Delta(e, a)$.
- (2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.
- (3) Affecter au registre d'état e l'état e' .
- (4) Si $s' = D$, déplacer la tête de lecture / écriture d'un pas vers la droite; sinon, la déplacer si possible d'un pas vers la gauche.
- (5) Si $e = t$, alors l'exécution est terminée; sinon, revenir en (1).

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M , on procède séquentiellement comme suit :

- (1) calculer $(e', a', s') := \Delta(e, a)$.
- (2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.
- (3) Affecter au registre d'état e l'état e' .
- (4) Si $s' = D$, déplacer la tête de lecture / écriture d'un pas vers la droite; sinon, la déplacer si possible d'un pas vers la gauche.
- (5) Si $e = t$, alors l'exécution est terminée; sinon, revenir en (1).

Résultat $M(u)$ de l'exécution de M sur u : plus court préfixe du ruban qui contient tous ses 0 et ses 1.

Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, 0)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, 0)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, 1)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

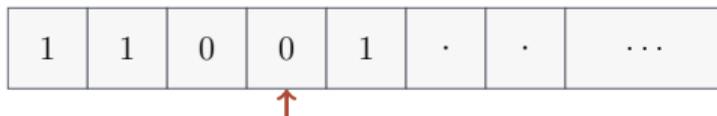
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, 0)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, 1)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

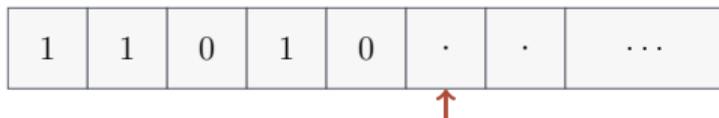
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_1, \cdot)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

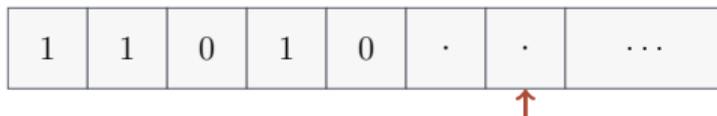
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_2, \cdot)$$



Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | | |
|---|---|---|---|---|---|---|-----|
| 1 | 1 | 0 | 1 | 0 | . | . | ... |
|---|---|---|---|---|---|---|-----|

L'exécution de M sur u fournit ainsi le résultat 11010.

Exemple : complémentaire d'un mot

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de $M(00101)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | | |
|---|---|---|---|---|---|---|-----|
| 1 | 1 | 0 | 1 | 0 | . | . | ... |
|---|---|---|---|---|---|---|-----|

L'exécution de M sur u fournit ainsi le résultat 11010.

Ici, ce programme Δ permet de calculer le complémentaire de tout mot $u \in \{0, 1\}^*$ en entrée.

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

Exécutions qui ne se terminent pas

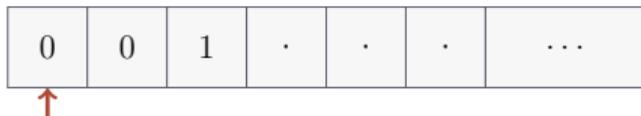
Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_1, 0)$$



Exécutions qui ne se terminent pas

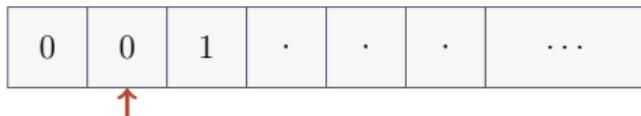
Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_1, 0)$$



Exécutions qui ne se terminent pas

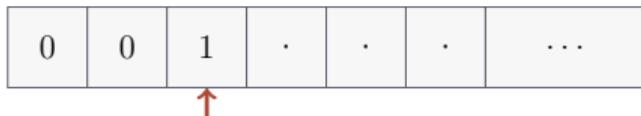
Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_1, 1)$$



Exécutions qui ne se terminent pas

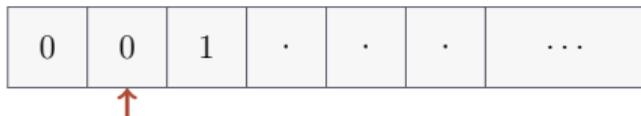
Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, 0)$$



Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 1 | . | . | . | ... |
|---|---|---|---|---|---|-----|

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 1 | . | . | . | ... |
|---|---|---|---|---|---|-----|

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

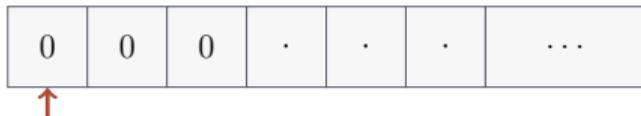
$$(e, a) = (e_2, \cdot)$$



On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, 0)$$



Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$



On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, 0)$$



Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 1 | . | . | . | ... |
|---|---|---|---|---|---|-----|

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, 0)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 0 | . | . | . | ... |
|---|---|---|---|---|---|-----|

↑

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

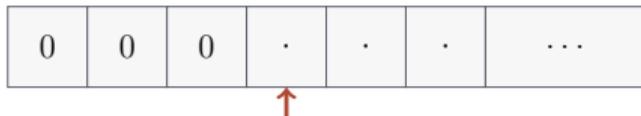
$$(e, a) = (e_2, \cdot)$$



On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, \cdot)$$



Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$



On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, \cdot)$$



Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 1 | . | . | . | ... |
|---|---|---|---|---|---|-----|

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 0 | . | . | . | ... |
|---|---|---|---|---|---|-----|

↑

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u , l'exécution de M sur u **ne se termine pas**.

Soit par exemple la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de $M(001)$:

$$(e, a) = (e_2, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 1 | . | . | . | ... |
|---|---|---|---|---|---|-----|

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

$$(e, a) = (e_1, \cdot)$$

| | | | | | | |
|---|---|---|---|---|---|-----|
| 0 | 0 | 0 | . | . | . | ... |
|---|---|---|---|---|---|-----|

La tête de lecture / écriture part vers infini, l'exécution ne se termine pas.

Coder une machine de Turing par un entier naturel

On associe à toute machine de Turing M un entier naturel $\text{code}(M)$ de la manière suivante :

Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing** M un **entier naturel** $\text{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII;

Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing** M un **entier naturel** $\text{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII;
2. on considère la suite de caractères m ainsi obtenue qui code M ;

Coder une machine de Turing par un entier naturel

On associe à toute machine de Turing M un entier naturel $\text{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII;
2. on considère la suite de caractères m ainsi obtenue qui code M ;
3. on considère la suite de bits u obtenue en remplaçant chaque caractère de m par sa représentation binaire;

Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing** M un **entier naturel** $\text{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII;
2. on considère la suite de caractères m ainsi obtenue qui code M ;
3. on considère la suite de bits u obtenue en remplaçant chaque caractère de m par sa représentation binaire;
4. on obtient finalement l'entier naturel $\text{code}(M)$ en considérant l'entier dont u est la représentation binaire.

Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

M se code en caractères ASCII en le texte m suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

M se code en caractères ASCII en le texte m suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

On en déduit la représentation binaire

$$u = 0110010101110100 \dots 00101001$$

et de celle-ci, l'entier naturel $\text{code}(M)$.

Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

M se code en caractères ASCII en le texte m suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

On en déduit la représentation binaire

$$u = 0110010101110100 \dots 00101001$$

et de celle-ci, l'entier naturel $\text{code}(M)$.

En décimal :

$$\text{code}(M) = 119527348352246394769818842856657399486293905629080176290359813575 \ 7140294873881005500260610437207957403372$$
$$269791891457737724736737165168419101329132422751418637824051118841 \ 640585439990747361814064299553649044252$$
$$299751665450294668928324126341232682416673453539993886044581503368 \ 944491857205586247218648808828981756969 \dots$$

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang** $\text{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment.

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang** $\text{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment.

L'application `rang` fournit une bijection entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang** $\text{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment.

L'application rang fournit une bijection entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

Conclusion : un programme (une machine de Turing) est un entier et réciproquement.

Plan

Théorie

Machines de Turing

Décidabilité et indécidabilité

λ -calcul

Problèmes de décision

Problème de décision : question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond oui ou non.

Problèmes de décision

Problème de décision : question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond oui ou non.

Par exemple :

- ▶ P : le mot est-il un palindrome? $P(010010) = \text{oui}$, $P(011) = \text{non}$;

Problèmes de décision

Problème de décision : question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond oui ou non.

Par exemple :

- ▶ P : le mot est-il un palindrome? $P(010010) = \text{oui}$, $P(011) = \text{non}$;
- ▶ P : la longueur du mot est-elle paire? $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;

Problèmes de décision

Problème de décision : question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond oui ou non.

Par exemple :

- ▶ P : le mot est-il un palindrome? $P(010010) = \text{oui}$, $P(011) = \text{non}$;
- ▶ P : la longueur du mot est-elle paire? $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;
- ▶ P : l'entier en base deux codé par le mot est-il premier? $P(111) = \text{oui}$, $P(100) = \text{non}$;

Problèmes de décision

Problème de décision : question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond oui ou non.

Par exemple :

- ▶ P : le mot est-il un palindrome? $P(010010) = \text{oui}$, $P(011) = \text{non}$;
- ▶ P : la longueur du mot est-elle paire? $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;
- ▶ P : l'entier en base deux codé par le mot est-il premier? $P(111) = \text{oui}$, $P(100) = \text{non}$;
- ▶ P : le mot est-il le codage binaire d'un programme C accepté à la compilation par `gcc` avec l'option `-ansi` ?

Décidabilité et indécidabilité

Problème de décision décidable : problème de décision P tel qu'il existe une machine de Turing M_P telle que pour toute entrée $u \in \{0, 1\}^*$, l'exécution de M_P sur u se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui,} \\ 0 & \text{sinon.} \end{cases}$$

Décidabilité et indécidabilité

Problème de décision décidable : problème de décision P tel qu'il existe une machine de Turing M_P telle que pour toute entrée $u \in \{0, 1\}^*$, l'exécution de M_P sur u se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui,} \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing, P est dit **indécidable**.

Décidabilité et indécidabilité

Problème de décision décidable : problème de décision P tel qu'il existe une machine de Turing M_P telle que pour toute entrée $u \in \{0, 1\}^*$, l'exécution de M_P sur u se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oùi,} \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing, P est dit **indécidable**.

Intuitivement, un problème de décision P est décidable s'il est possible d'écrire, dans un langage suffisamment complet, une fonction f paramétrée par un objet u renvoyant `true` si $P(u) = \text{oùi}$ et `false` sinon.

Le problème de l'arrêt

Problème de l'arrêt : problème de décision **Arr** prenant en entrée le codage binaire u d'un programme f et renvoyant oui si l'exécution du programme f se termine et non sinon.

Le problème de l'arrêt

Problème de l'arrêt : problème de décision **Arr** prenant en entrée le codage binaire u d'un programme f et renvoyant oui si l'exécution du programme f se termine et non sinon.

Le problème de l'arrêt est **indécidable**.

Le problème de l'arrêt

Problème de l'arrêt : problème de décision **Arr** prenant en entrée le codage binaire u d'un programme f et renvoyant oui si l'exécution du programme f se termine et non sinon.

Le problème de l'arrêt est **indécidable**.

Intuitivement, cela dit qu'il est impossible de concevoir un programme qui accepte en entrée un autre programme f et qui teste si l'exécution de f se termine.

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde ,
 - (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde ,
 - (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, absurde est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que absurde .

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde ,
 - (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, absurde est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que absurde .

On vérifie facilement que l'exécution de absurde se termine.

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde ,
 - (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, absurde est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que absurde .

On vérifie facilement que l'exécution de absurde se termine.

Ainsi, la valeur de retour d' absurde figure dans E (étape (a)).

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde ,
 - (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, absurde est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que absurde .

On vérifie facilement que l'exécution de absurde se termine.

Ainsi, la valeur de retour d' absurde figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par absurde ne figure pas dans E (étape (3)).

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr} .

On se base sur cette existence pour construire l'entier positif $absurde$ défini par les instructions :

- (1) soit E l'ensemble vide;
- (2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant $absurde$,
 - (a) si $M_{Arr}(u) = 1$, où u est le codage binaire de f :
 - (i) ajouter à E la valeur calculée par f , exprimée par un entier.
- (3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, $absurde$ est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que $absurde$.

On vérifie facilement que l'exécution de $absurde$ se termine.

Ainsi, la valeur de retour d' $absurde$ figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par $absurde$ ne figure pas dans E (étape (3)).

Ceci est absurde : M_{Arr} n'existe pas et Arr est donc indécidable.

Plan

Théorie

Machines de Turing

Décidabilité et indécidabilité

λ -calcul

Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Par exemple,

- ▶ $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$;
- ▶ $f : n \mapsto 1$ si n est premier, 0 sinon;
- ▶ $f : n \mapsto 1$ si $n \leq 1$, $n \times f(n - 1)$ sinon;

sont des fonctions récursives.

Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Par exemple,

- ▶ $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$;
- ▶ $f : n \mapsto 1$ si n est premier, 0 sinon;
- ▶ $f : n \mapsto 1$ si $n \leq 1$, $n \times f(n-1)$ sinon;

sont des fonctions récursives.

En revanche, la fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ définie par

$$f(n) := \begin{cases} 1 & \text{si Arr}(g) = \text{oui où } g \text{ est le prog. tq. rang}(g) = n, \\ 0 & \text{sinon,} \end{cases}$$

n'est pas une fonction récursive (si elle était calculable, le problème de l'arrêt serait décidable).

λ -calcul

Le λ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

λ -calcul

Le λ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En λ -calcul, la notion première est celle d'*expression*. Les *fonctions* sont des expressions particulières.

λ -calcul

Le λ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En λ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du λ -calcul est

1. soit une **variable**, notée x, y, z, \dots ;
2. soit l'**application** d'une expression f à une expression g , notée fg ;
3. soit l'**abstraction** d'une expression f , notée $\lambda x.f$ où x est une variable.

λ -calcul

Le λ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En λ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du λ -calcul est

1. soit une **variable**, notée x, y, z, \dots ;
2. soit l'**application** d'une expression f à une expression g , notée fg ;
3. soit l'**abstraction** d'une expression f , notée $\lambda x.f$ où x est une variable.

Par exemple, $(\lambda z.z)((\lambda x.x)(\lambda y.((\lambda x.x)y)))$ est une expression.

λ -calcul

Le λ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En λ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du λ -calcul est

1. soit une **variable**, notée x, y, z, \dots ;
2. soit l'**application** d'une expression f à une expression g , notée fg ;
3. soit l'**abstraction** d'une expression f , notée $\lambda x.f$ où x est une variable.

Par exemple, $(\lambda z.z)((\lambda x.x)(\lambda y.((\lambda x.x)y)))$ est une expression.

La β -**substitution** est le mécanisme qui permet de simplifier (calculer) une expression. Il consiste, étant donnée une expression de la forme $(\lambda x.f)g$ à la simplifier en substituant g aux occurrences libres de x dans f .

Plan

Caractéristiques

Impératif vs fonctionnel

Caractéristiques des langages

Plan

Caractéristiques

Impératif vs fonctionnel

Caractéristiques des langages

Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

1. le **paradigme impératif**;
2. le **paradigme fonctionnel**.

Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

1. le **paradigme impératif**;
2. le **paradigme fonctionnel**.

Le 1^{er} se base sur la machine de Turing, le 2^e sur le λ -calcul.

Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

1. les instructions d'affectation;
2. les instructions de branchement;
3. les instruction de boucle;
4. les structures de données mutables.

Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

1. les instructions d'affectation;
2. les instructions de branchement;
3. les instruction de boucle;
4. les structures de données mutables.

Des instructions peuvent **modifier l'état de la machine** en altérant sa mémoire.

Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Les éléments suivants sont constitutifs de ce paradigme :

1. la liaison d'un nom à une valeur ;
2. la récursivité ;
3. les instructions de branchement ;
4. les structures de données non mutables.

Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Les éléments suivants sont constitutifs de ce paradigme :

1. la liaison d'un nom à une valeur ;
2. la récursivité ;
3. les instructions de branchement ;
4. les structures de données non mutables.

Il n'y a pas de notion d'état de la machine car celui-ci ne peut pas être modifié.

Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {
    printf("a");
    return n;
}

int g(int n) {
    return n;
}
...
g(f(1));
```

Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}  
  
int g(int n) {  
    return n;  
}  
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche `a` mais pas `g(1)`.

Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}  
  
int g(int n) {  
    return n;  
}  
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche `a` mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}  
  
int g(int n) {  
    return n;  
}  
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche `a` mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

Règle : en programmation fonctionnelle, le principe de transparence référentielle s'applique.

Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

1. d'utiliser des variables (et donc aussi des affectations);
2. d'utiliser des instructions de boucle;
3. de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

1. d'utiliser des variables (et donc aussi des affectations);
2. d'utiliser des instructions de boucle;
3. de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

1. les **fonctions récursives**;
2. les **fonctions locales**.

Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

1. d'utiliser des variables (et donc aussi des affectations);
2. d'utiliser des instructions de boucle;
3. de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

1. les **fonctions récursives**;
2. les **fonctions locales**.

Une variable peut-être vue comme une **fonction d'arité zéro** (c.-à-d. une fonction qui ne prend pas d'entrée).

Plan

Caractéristiques

Impératif vs fonctionnel

Caractéristiques des langages

Vérification de types dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Vérification de types dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Il y a deux stratégies pour **vérifier les types** (typer) :

1. **vérification dynamique**, où les types sont vérifiés lors de l'**exécution** du programme;

Vérification de types dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Il y a deux stratégies pour **vérifier les types** (typer) :

1. **vérification dynamique**, où les types sont vérifiés lors de l'**exécution** du programme;
2. **vérification statique**, où les types sont vérifiés lors de la **compilation** du programme.

Vérification de types dynamique

Considérons le programme Python

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Vérification de types dynamique

Considérons le programme Python

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- ▶ si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée; sinon, elle ne l'est pas;

Vérification de types dynamique

Considérons le programme Python

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- ▶ si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée; sinon, elle ne l'est pas;
- ▶ si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée; sinon, c'est l'expression `n[1]` qui est évaluée.

Vérification de types dynamique

Considérons le programme Python

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- ▶ si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée; sinon, elle ne l'est pas;
- ▶ si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

Vérification de types dynamique

Considérons le programme Python

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- ▶ si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée; sinon, elle ne l'est pas;
- ▶ si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

Cette information n'est donnée que lors de l'exécution :

```
Traceback (most recent call last):
  File "Prog.py", line 5, in <module>
    print(n[1])
TypeError: 'int' object has no attribute '__getitem__'
```

Vérification de types statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Vérification de types statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa **compilation**, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

Vérification de types statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa **compilation**, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

Cette information est donnée lors de la compilation :

```
Prog.c: In function 'main':
Prog.c:8:25: error: subscripted value is neither array nor pointer nor vector
    printf("%d\n", n[1]);
                    ^
```

Avantages et inconvénients

Vérification de types dynamique.

- ▶ **Avantage** : grande flexibilité dans l'écriture des programmes.

Avantages et inconvénients

Vérification de types dynamique.

- ▶ **Avantage** : grande flexibilité dans l'écriture des programmes.
- ▶ **Inconvénients** : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

Avantages et inconvénients

Vérification de types dynamique.

- ▶ **Avantage** : grande flexibilité dans l'écriture des programmes.
- ▶ **Inconvénients** : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

Vérification de types statique.

- ▶ **Avantages** : sécurité lors de l'écriture du programme (les erreurs les plus courantes sont détectées à la compilation), bonne efficacité lors de l'exécution.

Avantages et inconvénients

Vérification de types dynamique.

- ▶ **Avantage** : grande flexibilité dans l'écriture des programmes.
- ▶ **Inconvénients** : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

Vérification de types statique.

- ▶ **Avantages** : sécurité lors de l'écriture du programme (les erreurs les plus courantes sont détectées à la compilation), bonne efficacité lors de l'exécution.
- ▶ **Inconvénient** : moins de flexibilité dans l'écriture des programmes.

Attribution de types explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Attribution de types explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. **attribution explicite** (nommé parfois *Church-style*), où les **types** des variables sont **mentionnés** dans le programme ;

Attribution de types explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. **attribution explicite** (nommé parfois *Church-style*), où les **types** des variables sont **mentionnés** dans le programme ;
2. **attribution implicite** (nommé parfois *Curry-style*), où les **types** des variables ne sont **pas mentionnés** dans le programme.

Ils sont devinés lors de la compilation (si vérification statique) ou lors de l'exécution (si vérification dynamique) en fonction du contexte.

Attribution de types explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. **attribution explicite** (nommé parfois *Church-style*), où les **types** des variables sont **mentionnés** dans le programme ;
2. **attribution implicite** (nommé parfois *Curry-style*), où les **types** des variables ne sont **pas mentionnés** dans le programme.

Ils sont devinés lors de la compilation (si vérification statique) ou lors de l'exécution (si vérification dynamique) en fonction du contexte.

Ce mécanisme s'appelle l'**inférence des types**.

Attribution explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)",
           p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Attribution explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)",
           p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Dans la fonction `main`, on a demandé explicitement que la variable `p` soit de type `Point3D`. La fonction `afficher` demande l'existence des champs `x` et `y` de son argument. Ces champs existent dans `p` et cependant, ce code ne compile pas.

Attribution explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)",
           p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Dans la fonction `main`, on a demandé explicitement que la variable `p` soit de type `Point3D`. La fonction `afficher` demande l'existence des champs `x` et `y` de son argument. Ces champs existent dans `p` et cependant, ce code ne compile pas.

En effet, même si une variable de type `Point3D` semble pouvoir être utilisée comme une variable de type `Point2D`, ceci est impossible en C qui est un langage à attribution de types explicite : le type de `p` a été fixé lors de la déclaration de `afficher`.

Attribution implicite

Considérons le programme Caml

```
let suivant x = x + 1 in  
(print_int (suivant 5))
```

Attribution implicite

Considérons le programme Caml

```
let suivant x = x + 1 in  
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

Attribution implicite

Considérons le programme Caml

```
let suivant x = x + 1 in  
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

De cette manière, le mécanisme d'inférence des types suggère au compilateur que `suivant` est une fonction qui accepte un entier et qui renvoie un entier.

Attribution implicite

Considérons le programme Caml

```
let suivant x = x + 1 in  
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

De cette manière, le mécanisme d'inférence des types suggère au compilateur que `suivant` est une fonction qui accepte un entier et qui renvoie un entier.

Cette valeur renvoyée peut donc être traitée par la fonction `print_int` qui accepte en entrée des valeurs entières.

Avantages et inconvénients

Attribution explicite.

- ▶ Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.

Avantages et inconvénients

Attribution explicite.

- ▶ Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- ▶ Inconvénients : programmes verbeux.

Avantages et inconvénients

Attribution explicite.

- ▶ Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- ▶ Inconvénients : programmes verbeux.

Attribution implicite.

- ▶ Avantages : programmes plus concis, le programmeur ne se préoccupe pas d'indiquer les types : le compilateur se charge de trouver ceux qui sont les plus adaptés.

Avantages et inconvénients

Attribution explicite.

- ▶ Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- ▶ Inconvénients : programmes verbeux.

Attribution implicite.

- ▶ Avantages : programmes plus concis, le programmeur ne se préoccupe pas d'indiquer les types : le compilateur se charge de trouver ceux qui sont les plus adaptés.
- ▶ Inconvénients : programmes moins lisibles. Si la vérification de types est statique, la compilation peut être plus longue (il faut deviner les types). Si la vérification de types est dynamique, l'exécution peut être moins efficace.

Portée statique vs dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Portée statique vs dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Les identificateurs dans un langage de programmation peuvent être de deux principales catégories :

1. de **portée statique**, où chaque occurrence d'un identificateur doit être mis en relation avec sa définition préalable;

Portée statique vs dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Les identificateurs dans un langage de programmation peuvent être de deux principales catégories :

1. de **portée statique**, où chaque occurrence d'un identificateur doit être mis en relation avec sa définition préalable ;
2. de **portée dynamique**, où ce que représente un identificateur peut dépendre de l'exécution du programme (dans certains *mauvais* cas, il peut même ne rien représenter du tout).

Portée statique

Considérons le programme Caml

```
let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))  
in  
(fact 3)
```

Portée statique

Considérons le programme Caml

```
let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))  
in  
(fact 3)
```

Lors de sa compilation, une erreur se produit : l'identificateur `fact`, utilisé en l. 5 est encore non défini.

On obtient le message suivant du compilateur :

```
File "Prog.ml", line 5, characters 12-16:  
Error: Unbound value fact
```

Portée dynamique

Considérons le programme Python

```
n = int(input())
if n == 0 :
    res = "a"
elif n == 1 :
    res = [1, 2, 3]
print(res)
```

Portée dynamique

Considérons le programme Python

```
n = int(input())
if n == 0 :
    res = "a"
elif n == 1 :
    res = [1, 2, 3]
print(res)
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- ▶ si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;

Portée dynamique

Considérons le programme Python

```
n = int(input())
if n == 0 :
    res = "a"
elif n == 1 :
    res = [1, 2, 3]
print(res)
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- ▶ si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;
- ▶ si l'utilisateur saisit `1`, `res` est défini comme étant la liste contenant successivement les éléments `1`, `2` et `3` ;

Portée dynamique

Considérons le programme Python

```
n = int(input())
if n == 0 :
    res = "a"
elif n == 1 :
    res = [1, 2, 3]
print(res)
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- ▶ si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;
- ▶ si l'utilisateur saisit `1`, `res` est défini comme étant la liste contenant successivement les éléments `1`, `2` et `3` ;
- ▶ dans tous les autres cas, `res` est un identificateur non défini.

Cette information est donnée, **lors de l'exécution**, par

```
Traceback (most recent call last):
  File "Prog.py", line 6, in <module>
    print(res)
NameError: name 'res' is not defined
```

Langages fonctionnels purs vs impurs

Il y a deux sortes de langages fonctionnels :

1. les langages **fonctionnels purs**, où tout effet de bord est interdit. Des problèmes se posent notamment lors de la gestion des entrées sorties (mais qui sont résolus élégamment au moyen des *monades*).

Langages fonctionnels purs vs impurs

Il y a deux sortes de langages fonctionnels :

1. les langages **fonctionnels purs**, où tout effet de bord est interdit. Des problèmes se posent notamment lors de la gestion des entrées sorties (mais qui sont résolus élégamment au moyen des *monades*).
2. les langages **fonctionnels impurs**, où certaines particularités des langages impératifs sont utilisables, comme la gestion classique des entrées / sorties, les affectations ou encore les instructions de boucle.

Caractéristiques des principaux langages

| Langage | Vérif. dyn. | Vérif. stat. | Attr. expl. | Attr. impl. | Imp. | Fonc. |
|---------|-------------|--------------|-------------|-------------|------|-------|
| C | Non | Oui | Oui | Non | Oui | Non |
| PYTHON | Oui | Non | Non | Oui | Oui | Impur |
| CAML | Non | Oui | <i>Non</i> | Oui | Oui | Impur |
| HASKELL | Non | Oui | Non | Oui | Non | Pur |

Caractéristiques des principaux langages

| Langage | Vérif. dyn. | Vérif. stat. | Attr. expl. | Attr. impl. | Imp. | Fonc. |
|---------|-------------|--------------|-------------|-------------|------------|--------------|
| C | Non | Oui | Oui | Non | Oui | Non |
| PYTHON | Oui | Non | Non | Oui | Oui | Impur |
| CAML | Non | Oui | Non | OUI | Non | Impur |
| HASKELL | Non | Oui | Non | Oui | Non | Pur |

Axe 2 : concepts premiers

Programmation

Pratique

Types

Plan

Programmation

- Interpréteur CAML

- Liaisons

- Types de base

- Fonctions

La 1^{re} chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *)
```

La 1^{re} chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *)
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

La 1^{re} chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *)
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire (* imbriquée. *) *)
```

En CAML, ceci fonctionne.

La 1^{re} chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *) .
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire (* imbriquée. *) *)
```

En CAML, ceci fonctionne.

```
/* Ceci est un commentaire /* imbriquée. */ */
```

En C, ceci ne fonctionne pas.

Plan

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode**;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode**;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif**;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode**;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif**;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode**;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif**;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

La 3^e est intéressante car elle permet de faire du **développement incrémental**, c.-à-d. l'écriture et le test pas à pas des fonctions nécessaires à la résolution d'un problème.

Dans ce cas, le programme n'est pas exécuté mais est **interprété**.

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
```

```
#
```

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
```

```
#
```

Celle-ci lance une **boucle d'interaction** qui se comporte, de manière répétée, de la façon suivante :

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
```

```
#
```

Celle-ci lance une **boucle d'interaction** qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;

Lire

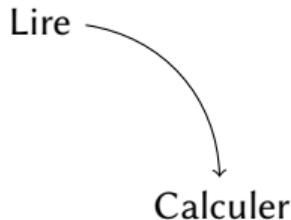
Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
#
```

Celle-ci lance une **boucle d'interaction** qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;



Interpréteur

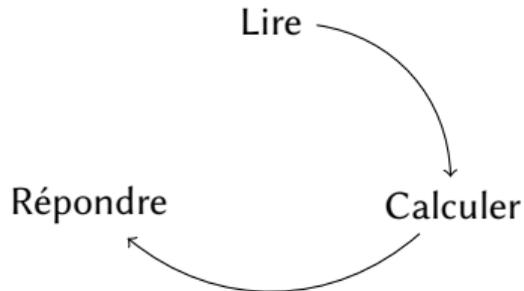
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
```

```
#
```

Celle-ci lance une **boucle d'interaction** qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Interpréteur

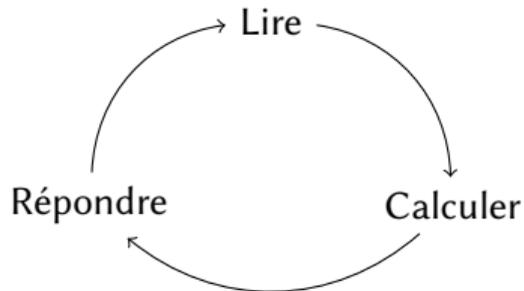
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
Objective Caml version 4.11.1
```

```
#
```

Celle-ci lance une **boucle d'interaction** qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Et ainsi de suite.

Phrases et réponses

Une **phrase** est une **expression** terminée par **::** (marqueur de fin de phrase).

```
# 1 + 1;;
```

Phrases et réponses

Une **phrase** est une **expression** terminée par `::` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1  
+ 1;;
```

Phrases et réponses

Une **phrase** est une **expression** terminée par `::` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1  
+ 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
```

```
- : int = 2
```

Phrases et réponses

Une **phrase** est une **expression** terminée par `::` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1  
+ 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
```

```
- : int = 2
```

Explications :

- ▶ le signe `-` signifie qu'une **valeur** a été calculée;

Phrases et réponses

Une **phrase** est une **expression** terminée par `::` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1  
+ 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
```

```
- : int = 2
```

Explications :

- ▶ le signe `-` signifie qu'une **valeur** a été calculée;
- ▶ `: int` signifie que cette valeur est de **type** `int` ;

Phrases et réponses

Une **phrase** est une **expression** terminée par `::` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1  
+ 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
```

```
- : int = 2
```

Explications :

- ▶ le signe `-` signifie qu'une **valeur** a été calculée;
- ▶ `: int` signifie que cette valeur est de **type** `int` ;
- ▶ `= 2` signifie que cette valeur **est** `2`.

Plan

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« *Soit n l'entier 5.* »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« Soit n l'entier 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une **définition**, ou encore la **liaison d'un nom à une valeur**.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« Soit n l'entier 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une **définition**, ou encore la **liaison d'un nom à une valeur**.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où **ID** est un **nom** (identificateur) et **VAL** est une expression.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« Soit n l'entier 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une **définition**, ou encore la **liaison d'un nom à une valeur**.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où **ID** est un **nom** (identificateur) et **VAL** est une expression.

```
# let n = 5;;
```

```
val n : int = 5
```

La 1^{re} phrase lie au nom **n** la valeur **5**.

L'interpréteur **le signale** en commençant sa réponse par **val n**.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« Soit n l'entier 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une **définition**, ou encore la **liaison d'un nom à une valeur**.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où **ID** est un **nom** (identificateur) et **VAL** est une expression.

```
# let n = 5;;
```

```
val n : int = 5
```

```
# n;;
```

```
- : int = 5
```

La 1^{re} phrase lie au nom **n** la valeur **5**.

L'interpréteur **le signale** en commençant sa réponse par `val n`.

La 2^e phrase donne la valeur à laquelle **n** est liée.

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où **ID** est un nom, **VAL** est une expression et **EXP** est une expression.

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où **ID** est un nom, **VAL** est une expression et **EXP** est une expression.

Très important : cette expression **possède comme valeur** celle de **EXP** dans laquelle les occurrences libres de **ID** sont remplacées par **VAL** .

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où **ID** est un nom, **VAL** est une expression et **EXP** est une expression.

Très important : cette expression **possède comme valeur** celle de **EXP** dans laquelle les occurrences libres de **ID** sont remplacées par **VAL** .

```
# let n = 5 in n + 1;;
```

```
- : int = 6
```

La 2^e occurrence de **n** a pour valeur **5** à cause de la liaison locale précédente. Ainsi, **n + 1** a pour valeur **6** .

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où **ID** est un nom, **VAL** est une expression et **EXP** est une expression.

Très important : cette expression **possède comme valeur** celle de **EXP** dans laquelle les occurrences libres de **ID** sont remplacées par **VAL**.

```
# let n = 5 in n + 1;;
```

```
- : int = 6
```

La 2^e occurrence de **n** a pour valeur **5** à cause de la liaison locale précédente. Ainsi, **n + 1** a pour valeur **6**.

```
# let n = 3;;
```

```
val n : int = 3
```

```
# let n = 4 in 2 * n;;
```

```
- : int = 8
```

La liaison de **n** à **4** dans la 2^e phrase est locale : le nom global **n** défini en 1^{re} phrase reste inchangé.

Liaisons locales

Il est possible de définir des noms **localement** à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où **ID** est un nom, **VAL** est une expression et **EXP** est une expression.

Très important : cette expression **possède comme valeur** celle de **EXP** dans laquelle les occurrences libres de **ID** sont remplacées par **VAL** .

```
# let n = 5 in n + 1;;
```

```
- : int = 6
```

La 2^e occurrence de **n** a pour valeur **5** à cause de la liaison locale précédente. Ainsi, **n + 1** a pour valeur **6** .

```
# let n = 3;;
```

```
val n : int = 3
```

```
# let n = 4 in 2 * n;;
```

```
- : int = 8
```

```
# n;;
```

```
- : int = 3
```

La liaison de **n** à **4** dans la 2^e phrase est locale : le nom global **n** défini en 1^{re} phrase reste inchangé.

Ceci renseigne sur la valeur du nom **n** de la 3^e phrase.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom **x** à sa définition.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
  let s = s * s in
    let s = s * s in
      s + 4;;
```

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
  let s = s * s in
    let s = s * s in
      s + 4;;
```

```
- : int = 20
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
  let s = s * s in
    let s = s * s in
      s + 4;;
```

```
- : int = 20
```

```
# let s = let x = 3 in
  x * x;;
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
  let s = s * s in
    let s = s * s in
      s + 4;;
```

```
- : int = 20
```

```
# let s = let x = 3 in
  x * x;;
```

```
val s : int = 9
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Le nom `s` est lié à la valeur `9`.

En effet, la sous-expression `let x = 3 in x * x` a pour valeur `9`.

Liaisons locales — exemples

```
# let x =  
  let y = 2 in  
    let z = 3 in  
      y + let z = 8 in  
        z * z;;
```

Liaisons locales — exemples

```
# let x =  
  let y = 2 in  
    let z = 3 in  
      y + let z = 8 in  
          z * z;;
```

```
Warning 26: unused variable z.  
val x : int = 66
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =  
  let y = 2 in  
    let z = 3 in  
      y + let z = 8 in  
          z * z;;
```

```
Warning 26: unused variable z.  
val x : int = 66
```

```
# let x = let y = 2 in  
  x + y;;
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =  
  let y = 2 in  
    let z = 3 in  
      y + let z = 8 in  
          z * z;;
```

```
Warning 26: unused variable z.  
val x : int = 66
```

```
# let x = let y = 2 in  
  x + y;;
```

```
Error: Unbound value x
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales **simultanément** dans une même phrase.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales **simultanément** dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où **ID1** et **ID2** sont des identificateurs, et **VAL1**, **VAL2** et **EXP** sont des expressions.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales **simultanément** dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où **ID1** et **ID2** sont des identificateurs, et **VAL1**, **VAL2** et **EXP** sont des expressions.

```
# let x = 1 and y = 2;;
```

```
val x : int = 1  
val y : int = 2
```

Cette phrase lie simultanément au nom **x** la valeur **1** et au nom **y** la valeur **2**.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales **simultanément** dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où **ID1** et **ID2** sont des identificateurs, et **VAL1**, **VAL2** et **EXP** sont des expressions.

```
# let x = 1 and y = 2;;
```

```
val x : int = 1  
val y : int = 2
```

```
# let x = 1 in  
  let y = 3  
  and z = 4 in  
    x + y + z;;
```

```
- : int = 8
```

Cette phrase lie simultanément au nom **x** la valeur **1** et au nom **y** la valeur **2**.

Il est possible d'imbriquer les définitions locales et simultanées.

On notera l'indentation différente impliquée par les **in** et les **and**.

Liaisons simultanées — exemples

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

Liaisons simultanées — exemples

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

```
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

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

```
- : int = 6
```

```
# let x = 1 in
  let x = 2 in
    let y = x + 3 in
      x + y;;
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

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

```
- : int = 6
```

```
# let x = 1 in
  let x = 2 in
    let y = x + 3 in
      x + y;;
```

```
Warning 26: unused variable x.
```

```
- : int = 7
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Liaisons simultanées — exemples

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

```
- : int = 6
```

```
# let x = 1 in
  let x = 2 in
    let y = x + 3 in
      x + y;;
```

```
Warning 26: unused variable x.
```

```
- : int = 7
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Le résultat est différent du précédent : dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 2.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
    x * x * x;;
```

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
```

```
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
           x * x * x;;
```

```
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
           x * x * x;;
```

```
val x : int = 1
val y : int = 2
val z : int = 4096
```

```
# let x = 1
  and y = x + 1;;
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
```

```
val x : int = 1
val y : int = 2
val z : int = 4096
```

```
# let x = 1
  and y = x + 1;;
```

```
Error: Unbound value x
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- ▶ une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- ▶ une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;  
val x : int = 3  
...  
# x + 1;;
```

```
- : int = 4
```

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- ▶ une liaison `let ID = VAL` permet de former une phrase qui **n'a pas de valeur** et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;  
val x : int = 3  
...  
# x + 1;;
```

```
- : int = 4
```

- ▶ Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui **a une valeur** et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- ▶ une liaison `let ID = VAL` permet de former une phrase qui **n'a pas de valeur** et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;  
val x : int = 3  
...  
# x + 1;;
```

```
- : int = 4
```

- ▶ Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui **a une valeur** et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`. P.ex.,

```
# let x = 3 in x + 1;;
```

```
- : int = 4
```

Plan

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

Les six types de base

| Nom du type | Utilisation |
|---------------------|--|
| <code>int</code> | Représentation des entiers signés |
| <code>float</code> | Représentation des nombres à virgule signés |
| <code>char</code> | Représentation des caractères |
| <code>string</code> | Représentation des chaînes de caractères |
| <code>bool</code> | Représentation des booléens |
| <code>unit</code> | Type contenant une unique valeur |

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

| Opérateur | Arité | Rôle |
|-------------------------|-------|-------------|
| <code>not</code> | 1 | Non logique |
| <code>&&</code> | 2 | Et logique |
| <code> </code> | 2 | Ou logique |

Ces opérateurs produisent des valeurs de type `bool`.

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

| Opérateur | Arité | Rôle |
|-------------------------|-------|-------------|
| <code>not</code> | 1 | Non logique |
| <code>&&</code> | 2 | Et logique |
| <code> </code> | 2 | Ou logique |

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (false || (not false)) && (not (true || false));;
```

```
- : bool = false
```

```
# not true && false;;
```

```
- : bool = false
```

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

| Opérateur | Arité | Rôle |
|-------------------------|-------|-------------|
| <code>not</code> | 1 | Non logique |
| <code>&&</code> | 2 | Et logique |
| <code> </code> | 2 | Ou logique |

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (false || (not false)) && (not (true || false));;
```

```
- : bool = false
```

```
# not true && false;;
```

```
- : bool = false
```

Règle : ne jamais hésiter à introduire des parenthèses (sans exagérer) pour gagner en lisibilité.

Le type `int`

Une valeur de type `int` peut s'écrire en

- ▶ décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);

Le type `int`

Une valeur de type `int` peut s'écrire en

- ▶ décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- ▶ hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);

Le type `int`

Une valeur de type `int` peut s'écrire en

- ▶ décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- ▶ hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- ▶ binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

Le type `int`

Une valeur de type `int` peut s'écrire en

- ▶ décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- ▶ hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- ▶ binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

La plage des `int` s'étend de `min_int` à `max_int`.

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904$$

à

$$2^{62} - 1 = 4611686018427387903.$$

Le type `int`

Une valeur de type `int` peut s'écrire en

- ▶ décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- ▶ hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- ▶ binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

La plage des `int` s'étend de `min_int` à `max_int`.

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904$$

à

$$2^{62} - 1 = 4611686018427387903.$$

La plage ne s'étend pas de -2^{n-1} à $2^{n-1} - 1$: utilisation d'un bit pour la gestion automatique de la mémoire.

Opérations arithmétiques sur les `int`

| Opérateur | Arité | Rôle |
|---------------------------------|-------|--------------------------|
| <code>-</code> , <code>+</code> | 1 | Moins, Plus (signe) |
| <code>-</code> , <code>+</code> | 2 | Soustraction, Addition |
| <code>/</code> , <code>*</code> | 2 | Division, Multiplication |
| <code>succ</code> | 1 | Successeur |
| <code>pred</code> | 1 | Prédécesseur |
| <code>abs</code> | 1 | Valeur absolue |
| <code>mod</code> | 2 | Modulo |

Ces opérateurs produisent des valeurs de type `int`.

Opérations arithmétiques sur les `int`

| Opérateur | Arité | Rôle |
|---------------------------------|-------|--------------------------|
| <code>-</code> , <code>+</code> | 1 | Moins, Plus (signe) |
| <code>-</code> , <code>+</code> | 2 | Soustraction, Addition |
| <code>/</code> , <code>*</code> | 2 | Division, Multiplication |
| <code>succ</code> | 1 | Successeur |
| <code>pred</code> | 1 | Prédécesseur |
| <code>abs</code> | 1 | Valeur absolue |
| <code>mod</code> | 2 | Modulo |

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `pred` et `abs` sont des fonctions.

Opérations arithmétiques sur les `int`

| Opérateur | Arité | Rôle |
|---------------------------------|-------|--------------------------|
| <code>-</code> , <code>+</code> | 1 | Moins, Plus (signe) |
| <code>-</code> , <code>+</code> | 2 | Soustraction, Addition |
| <code>/</code> , <code>*</code> | 2 | Division, Multiplication |
| <code>succ</code> | 1 | Successeur |
| <code>pred</code> | 1 | Prédécesseur |
| <code>abs</code> | 1 | Valeur absolue |
| <code>mod</code> | 2 | Modulo |

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `pred` et `abs` sont des fonctions.

`mod` n'est pas une fonction, c'est un opérateur.

Opérations relationnelles sur les `int`

| Opérateur | Arité | Rôle |
|---|-------|---------------------|
| <code>=</code> , <code><></code> | 2 | Égalité, Différence |
| <code><</code> , <code>></code> | 2 | Comparaison stricte |
| <code><=</code> , <code>>=</code> | 2 | Comparaison large |

Ces opérateurs produisent des valeurs de type `bool`.

Opérations relationnelles sur les `int`

| Opérateur | Arité | Rôle |
|---|-------|---------------------|
| <code>=</code> , <code><></code> | 2 | Égalité, Différence |
| <code><</code> , <code>></code> | 2 | Comparaison stricte |
| <code><=</code> , <code>>=</code> | 2 | Comparaison large |

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (2 = 1) || (32 <= 64);;
```

```
- : bool = true
```

Opérations bit à bit sur les `int`

| Opérateur | Arité | Rôle |
|-------------------------------------|-------|---|
| <code>lnot</code> | 1 | Non bit à bit |
| <code>land</code> | 2 | Et bit à bit |
| <code>lor</code> | 2 | Ou bit à bit |
| <code>lxor</code> | 2 | Ou exclusif bit à bit |
| <code>lsl</code> , <code>lsr</code> | 2 | Décalage à gauche, droite bit à bit |
| <code>asr</code> | 2 | Décalage à droite avec respect du signe bit à bit |

Ces opérateurs produisent des valeurs de type `int` .

Opérations bit à bit sur les `int`

| Opérateur | Arité | Rôle |
|-------------------------------------|-------|---|
| <code>lnot</code> | 1 | Non bit à bit |
| <code>land</code> | 2 | Et bit à bit |
| <code>lor</code> | 2 | Ou bit à bit |
| <code>lxor</code> | 2 | Ou exclusif bit à bit |
| <code>lsl</code> , <code>lsr</code> | 2 | Décalage à gauche, droite bit à bit |
| <code>asr</code> | 2 | Décalage à droite avec respect du signe bit à bit |

Ces opérateurs produisent des valeurs de type `int` .

Par exemple :

```
# 1 lsl 10;;
```

```
- : int = 1024
```

```
# (lnot 0) lsr 1;;
```

```
- : int = 4611686018427387903
```

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ». P.ex., `4.52` est l'écriture du nombre 4.52.

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ». P.ex., `4.52` est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.`.

```
# 0;;
```

```
- : int = 0
```

```
# 0.;;
```

```
- : float = 0.
```

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ». P.ex., `4.52` est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.`.

```
# 0;;
```

```
- : int = 0
```

```
# 0.;;
```

```
- : float = 0.
```

La plage des `float` s'étend de `-max_float` à `max_float`.

Sur un système `64` bits, ceci va de

$$-\text{max_float} = -1.79769313486231571 \times 10^{308}$$

à

$$\text{max_float} = 1.79769313486231571 \times 10^{308}.$$

Opérations sur les float

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.` , `/.` , `*.` .

```
# 1. +. 1.;;
```

```
- : float = 2.
```

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.` , `/.`, `*.`.

```
# 1. +. 1.;;
```

```
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
```

```
Error: This expression has type int but an expression was
      expected of type float
```

Opérations sur les float

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.` , `/.` , `*.` .

```
# 1. +. 1.;;
```

```
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
```

```
Error: This expression has type int but an expression was  
expected of type float
```

On peut convertir un `int` en `float` par la fonction `float_of_int` :

```
# (float_of_int 32);;
```

```
- : float = 32.
```

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.` , `/.` , `*.` .

```
# 1. +. 1.;;
```

```
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
```

```
Error: This expression has type int but an expression was
expected of type float
```

On peut convertir un `int` en `float` par la fonction `float_of_int` :

```
# (float_of_int 32);;
```

```
- : float = 32.
```

et un `float` en `int` par la fonction `int_of_float` (troncature) :

```
# (int_of_float 21.9);;
```

```
- : int = 21
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
```

```
- : bool = true
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
```

```
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
```

```
Error: This expression has type int but an expression was  
expected of type float
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
```

```
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
```

```
Error: This expression has type int but an expression was  
       expected of type float
```

Il faut en revanche écrire

```
# 67.67 = (float_of_int 8);;
```

```
- : bool = false
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
```

```
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
```

```
Error: This expression has type int but an expression was
      expected of type float
```

Il faut en revanche écrire

```
# 67.67 = (float_of_int 8);;
```

```
- : bool = false
```

pour demander explicitement la conversion d'un `int` en un `float`.

Opérations sur les `float`

Il existe des opérateurs spécifiques aux `float`. Entre autres :

| Opérateur | Arité | Rôle |
|--|-------|------------------------------------|
| <code>abs_float</code> | 1 | Valeur absolue |
| <code>floor</code> , <code>ceil</code> | 1 | Partie entière inf., sup. |
| <code>sqrt</code> | 1 | Racine carrée |
| <code>log</code> , <code>exp</code> | 1 | Logarithme népérien, exponentielle |
| <code>cos</code> , <code>sin</code> , <code>tan</code> | 1 | Fonctions trigonométriques |
| <code>**</code> | 2 | Exponentiation |

Ces opérateurs produisent des valeurs de type `float`.

Opérations sur les `float`

Il existe des opérateurs spécifiques aux `float`. Entre autres :

| Opérateur | Arité | Rôle |
|--|-------|------------------------------------|
| <code>abs_float</code> | 1 | Valeur absolue |
| <code>floor</code> , <code>ceil</code> | 1 | Partie entière inf., sup. |
| <code>sqrt</code> | 1 | Racine carrée |
| <code>log</code> , <code>exp</code> | 1 | Logarithme népérien, exponentielle |
| <code>cos</code> , <code>sin</code> , <code>tan</code> | 1 | Fonctions trigonométriques |
| <code>**</code> | 2 | Exponentiation |

Ces opérateurs produisent des valeurs de type `float`.

Hormis `**` qui est bien un opérateur du langage, les autres sont en réalité des fonctions prédéfinies.

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- ▶ par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- ▶ par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- ▶ par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- ▶ par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

La fonction `int_of_char` calcule le code ASCII d'un caractère.

```
# (int_of_char 'G');
```

```
- : int = 71
```

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- ▶ par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- ▶ par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

La fonction `int_of_char` calcule le code ASCII d'un caractère.

```
# (int_of_char 'G');
```

```
- : int = 71
```

La fonction `char_of_int` calcule le caractère de code ASCII spécifié.

```
# (char_of_int 40);;
```

```
- : char = '('
```

```
# (char_of_int 2);;
```

```
- : char = '\002'
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`,
`"\100\101f"`.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
```

```
- : string = "abcdef"
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
```

```
- : string = "abcdef"
```

```
# let u = "ab"  
  and v = "ba" in  
  u ^ v ^ u ^ v;;
```

```
- : string = "abbaabba"
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;  
- : string = "abcdef"
```

```
# let u = "ab"  
  and v = "ba" in  
  u ^ v ^ u ^ v;;  
- : string = "abbaabba"
```

Plus précisément, si `u` et `v` sont des noms liés à des chaînes de caractères, `u ^ v` est une expression dont la valeur est la concaténation de `u` et de `v`.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;  
- : string = "abcdef"
```

```
# let u = "ab"  
  and v = "ba" in  
  u ^ v ^ u ^ v;;  
- : string = "abbaabba"
```

Plus précisément, si `u` et `v` sont des noms liés à des chaînes de caractères, `u ^ v` est une expression dont la valeur est la concaténation de `u` et de `v`.

Il n'y a pas d'effet de bord : les chaînes `u` et `v` ne sont pas modifiées lors de leur concaténation.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;  
- : bool = false
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";  
- : bool = false
```

```
# let u = "ab" and v = "ab" in  
  u = v;;  
- : bool = true
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";  
- : bool = false
```

```
# let u = "ab" and v = "ab" in  
  u = v;;  
- : bool = true
```

```
# "abc" <= "aaaa";;  
- : bool = false
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- ▶ `string_of_bool` et `bool_of_string` ;
- ▶ `string_of_int` et `int_of_string` ;
- ▶ `string_of_float` et `float_of_string` .

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";  
- : bool = false
```

```
# "abc" <= "aaaa";  
- : bool = false
```

```
# let u = "ab" and v = "ab" in  
  u = v;;  
- : bool = true
```

```
# "abc" < "ad";;  
- : bool = true
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `unit`

Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

Le type `unit`

Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

```
# ();;
```

```
- : unit = ()
```

Le type `unit`

Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

```
# ();;
```

```
- : unit = ()
```

```
# let a = () in let b = a in  
  b;;
```

```
- : unit = ()
```

Le type `unit`

Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

```
# ();;
```

```
- : unit = ()
```

```
# let a = () in let b = a in  
  b;;
```

```
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

Le type `unit`

Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

```
# ();;
```

```
- : unit = ()
```

```
# let a = () in let b = a in  
  b;;
```

```
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

Dans la suite, nous verrons que ce type sert à rendre les fonctions homogènes au sens où toute fonction doit renvoyer une valeur.

En effet, une fonction qui n'est pas sensée renvoyer de valeur va renvoyer `()`.

Plan

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;

La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;

La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**.

La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction** ;
2. un programme est une collection de définitions de fonctions ;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**.

Cette valeur calculée est précisément la **valeur de retour** de la fonction principale du programme.

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$ telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$ telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

D'usage, la propriété $(x_1, x_2, \dots, x_n, y) \in f$ est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$ telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

D'usage, la propriété $(x_1, x_2, \dots, x_n, y) \in f$ est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier n l'arité de f (qui est son nombre d'entrées).

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$ telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

D'usage, la propriété $(x_1, x_2, \dots, x_n, y) \in f$ est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier n l'arité de f (qui est son nombre d'entrées).

Distinction à faire entre

- ▶ **application** : fonction définie en tout point;

La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien $T_1 \times T_2 \times \cdots \times T_n \times S$ telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

D'usage, la propriété $(x_1, x_2, \dots, x_n, y) \in f$ est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier n l'arité de f (qui est son nombre d'entrées).

Distinction à faire entre

- ▶ **application** : fonction définie en tout point ;
- ▶ **fonction** : application partielle (pas de résultat pour certaines entrées).

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les paramètres de f .

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les paramètres de f .

Lors de l'appel $f(2, 6 - 3)$, f est appelée avec les arguments 2 et $6 - 3$.

La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les paramètres de f .

Lors de l'appel $f(2, 6 - 3)$, f est appelée avec les arguments 2 et $6 - 3$.

Lors d'un appel à une fonction f avec les arguments a_1, \dots, a_n , on dit que l'on **applique** f à a_1, \dots, a_n .

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences (libres) de ses paramètres sont remplacées par les arguments.

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences (libres) de ses paramètres sont remplacées par les arguments.

P.ex.,

```
# let oppose x = -x;;
```

produit l'évaluation

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

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences (libres) de ses paramètres sont remplacées par les arguments.

P.ex.,

```
# let oppose x = -x;;
```

produit l'évaluation

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

Explications :

- ▶ `val oppose` informe qu'on a lié au nom `oppose` une valeur;

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences (libres) de ses paramètres sont remplacées par les arguments.

P.ex.,

```
# let oppose x = -x;;
```

produit l'évaluation

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

Explications :

- ▶ `val oppose` informe qu'on a lié au nom `oppose` une valeur;
- ▶ `: int -> int` informe que cette valeur est de type `int -> int` ;

Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences (libres) de ses paramètres sont remplacées par les arguments.

P.ex.,

```
# let oppose x = -x;;
```

produit l'évaluation

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

Explications :

- ▶ `val oppose` informe qu'on a lié au nom `oppose` une valeur;
- ▶ `: int -> int` informe que cette valeur est de type `int -> int` ;
- ▶ `= <fun>` est un affichage générique la fonction.

Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
  let n2 = n * n in  
  let n4 = n2 * n2 in  
  n4 * n4;;
```

```
val puissance_8 : int -> int = <fun>
```

Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
  let n2 = n * n in  
  let n4 = n2 * n2 in  
  n4 * n4;;
```

```
val puissance_8 : int -> int = <fun>
```

Pour appliquer `puissance_8` à l'argument `2`, on écrit

```
# (puissance_8 2);;
```

```
- : int = 256
```

On notera l'utilisation inhabituelle des parenthèses.

Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
  let n2 = n * n in  
  let n4 = n2 * n2 in  
  n4 * n4;;
```

```
val puissance_8 : int -> int = <fun>
```

Pour appliquer `puissance_8` à l'argument `2`, on écrit

```
# (puissance_8 2);;
```

```
- : int = 256
```

On notera l'utilisation inhabituelle des parenthèses.

Il est possible d'appeler une fonction dans une autre :

```
# let puissance_16 n =  
  let n8 = (puissance_8 n) in  
  n8 * n8;;
```

```
val puissance_16 : int -> int = <fun>
```

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let f u =  
  let aux u =  
    u ^ "a" ^ u  
  in  
  (aux u) ^ (aux ("b" ^ u));;
```

```
val f : string -> string = <fun>
```

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let f u =  
  let aux u =  
    u ^ "a" ^ u  
  in  
  (aux u) ^ (aux ("b" ^ u));;
```

```
val f : string -> string = <fun>
```

```
# (f "");;
```

```
- : string = "abab"
```

```
# (f "cd");;
```

```
- : string = "cdacdbcdabcd"
```

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let f u =  
  let aux u =  
    u ^ "a" ^ u  
  in  
    (aux u) ^ (aux ("b" ^ u));;
```

```
val f : string -> string = <fun>
```

```
# (f "");;
```

```
- : string = "abab"
```

```
# (f "cd");;
```

```
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) simultanées.

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let f u =  
  let aux u =  
    u ^ "a" ^ u  
  in  
    (aux u) ^ (aux ("b" ^ u));;
```

```
val f : string -> string = <fun>
```

```
# (f "");;
```

```
- : string = "abab"
```

```
# (f "cd");;
```

```
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) simultanées.

```
# let f x =  
  let g y =  
    y - 2 = x  
  and h x =  
    2 * x  
  in  
    (g (h x));;
```

```
val f : int -> bool = <fun>
```

Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

```
# let f u =  
  let aux u =  
    u ^ "a" ^ u  
  in  
    (aux u) ^ (aux ("b" ^ u));;
```

```
val f : string -> string = <fun>
```

```
# (f "");;
```

```
- : string = "abab"
```

```
# (f "cd");;
```

```
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) simultanées.

```
# let f x =  
  let g y =  
    y - 2 = x  
  and h x =  
    2 * x  
  in  
    (g (h x));;
```

```
val f : int -> bool = <fun>
```

```
# (f 1);;
```

```
- : bool = false
```

```
# (f 2);;
```

```
- : bool = true
```

Expressions conditionnelles

Une **expression conditionnelle** est une expression de la forme

```
if COND then EXP1 else EXP2
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP1** et **EXP2** sont deux expressions dont les valeurs sont **toutes deux d'un même type**.

Expressions conditionnelles

Une **expression conditionnelle** est une expression de la forme

```
if COND then EXP1 else EXP2
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP1** et **EXP2** sont deux expressions dont les valeurs sont **toutes deux d'un même type**.

```
# if (3 >= 2) || ("aab" <= "aa") then  
  "ABC"  
else  
  "CDE";;
```

```
- : string = "ABC"
```

Expressions conditionnelles

Une **expression conditionnelle** est une expression de la forme

```
if COND then EXP1 else EXP2
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP1** et **EXP2** sont deux expressions dont les valeurs sont **toutes deux d'un même type**.

```
# if (3 >= 2) || ("aab" <= "aa") then
  "ABC"
else
  "CDE";;
```

```
- : string = "ABC"
```

Toute expression conditionnelle **possède une valeur** :

1. lorsque **COND** s'évalue en **true**, l'expression conditionnelle à pour valeur **EXP1** ;
2. lorsque **COND** s'évalue en **false**, l'expression conditionnelle à pour valeur **EXP2** .

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

```
# if true then
  if 2 = 3 then
    'A'
  else
    'B'
else
  'C';;
```

```
- : char = 'B'
```

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

```
# if true then
  if 2 = 3 then
    'A'
  else
    'B'
else
  'C';;
```

```
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

```
# if true then
  if 2 = 3 then
    'A'
  else
    'B'
else
  'C';;
```

```
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur CAML la comprend sans ambiguïté.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

```
# if true then
  if 2 = 3 then
    'A'
  else
    'B'
else
  'C';;
```

```
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur CAML la comprend sans ambiguïté.

Il n'a pas besoin de marqueur de fin (comme le `}` de certains langages).

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24  
→ if true then 21 else 24 → 21 ,
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

 →

```
21
```

 ,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

 →

```
21
```

 ,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

→

```
if if true then 1 = 0 else true then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

 →

```
21
```

 ,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

→

```
if if true then 1 = 0 else true then 28 else 21
```

→

```
if 1 = 0 then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

 →

```
21
```

 ,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

→

```
if if true then 1 = 0 else true then 28 else 21
```

→

```
if 1 = 0 then 28 else 21
```

→

```
if false then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21 ,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

```
→ if if true then 1 = 0 else true then 28 else 21
```

```
→ if 1 = 0 then 28 else 21
```

```
→ if false then 28 else 21 → 21 .
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

→

```
if true then 21 else 24
```

 → **21** ,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

→

```
if if true then 1 = 0 else true then 28 else 21
```

→

```
if 1 = 0 then 28 else 21
```

→

```
if false then 28 else 21
```

 → **21** .

Grâce au **principe de transparence référentielle**, ces deux expressions peuvent être remplacées par leurs valeurs, **21** , dans tout programme les employant sans modifier la valeur qu'il calcule.

Demi-expressions conditionnelles

Une **demi-expression conditionnelle** est une expression de la forme

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP** est une expression.

Demi-expressions conditionnelles

Une **demi-expression conditionnelle** est une expression de la forme

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP** est une expression.

Le système complète implicitement et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

```
if COND then EXP else ()
```

Demi-expressions conditionnelles

Une **demi-expression conditionnelle** est une expression de la forme

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP** est une expression.

Le système complète implicitement et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

```
if COND then EXP else ()
```

En conséquence, la valeur de **EXP** doit être de type **unit**.

Demi-expressions conditionnelles

Une **demi-expression conditionnelle** est une expression de la forme

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type **bool** et **EXP** est une expression.

Le système complète implicitement et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

```
if COND then EXP else ()
```

En conséquence, la valeur de **EXP** doit être de type **unit**.

```
let f x =  
  if x >= 9 then  
    ()
```

est équivalent à

```
let f x =  
  if x >= 9 then  
    ()  
  else  
    ()
```

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où E_1 , ..., E_n et S sont des types.

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où E_1 , ..., E_n et S sont des types.

Plus précisément,

- ▶ pour tout $1 \leq i \leq n$, E_i est le type attendu du i^{e} argument de F ;

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où E_1 , ..., E_n et S sont des types.

Plus précisément,

- ▶ pour tout $1 \leq i \leq n$, E_i est le type attendu du i^{e} argument de F ;
- ▶ S est le type de retour de F .

Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où E_1 , ..., E_n et S sont des types.

Plus précisément,

- ▶ pour tout $1 \leq i \leq n$, E_i est le type attendu du i^{e} argument de F ;
- ▶ S est le type de retour de F .

$E_1 -> \dots -> E_n -> S$ est un type particulier, dit **type fonction**.

Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;
```

```
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;
```

```
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;
```

```
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Cependant, lorsque l'on en met pas, la convention stipule que le parenthésage est fait **de droite à gauche**.

Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;
```

```
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Cependant, lorsque l'on en met pas, la convention stipule que le parenthésage est fait **de droite à gauche**.

Ainsi, le type `int -> char -> int` est équivalent au type dénoté par l'expression totalement parenthésée `(int -> (char -> int))`.

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types E_1 et E_2 et à type de retour S .

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types E_1 et E_2 et à type de retour S .

Deux faits évidents :

1. la fonction f est de type $E_1 \rightarrow E_2 \rightarrow S$;

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types $E1$ et $E2$ et à type de retour S .

Deux faits évidents :

1. la fonction f est de type $E1 \rightarrow E2 \rightarrow S$;
2. si $e1$ et $e2$ sont des valeurs de types respectifs $E1$ et $E2$, l'application de f à $e1$ et $e2$ par $(f\ e1\ e2)$ renvoie une valeur de type S .

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types $E1$ et $E2$ et à type de retour S .

Deux faits évidents :

1. la fonction f est de type $E1 \rightarrow E2 \rightarrow S$;
2. si $e1$ et $e2$ sont des valeurs de types respectifs $E1$ et $E2$, l'application de f à $e1$ et $e2$ par $(f\ e1\ e2)$ renvoie une valeur de type S .

Il est cependant possible d'**appliquer** f à **seulement** $e1$ par $(f\ e1)$. On obtient ainsi une valeur de type $E2 \rightarrow S$ qui est donc une **fonction**.

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types $E1$ et $E2$ et à type de retour S .

Deux faits évidents :

1. la fonction f est de type $E1 \rightarrow E2 \rightarrow S$;
2. si $e1$ et $e2$ sont des valeurs de types respectifs $E1$ et $E2$, l'application de f à $e1$ et $e2$ par $(f\ e1\ e2)$ renvoie une valeur de type S .

Il est cependant possible d'**appliquer** f à **seulement** $e1$ par $(f\ e1)$. On obtient ainsi une valeur de type $E2 \rightarrow S$ qui est donc une **fonction**.

La fonction $(f\ e1)$ est ainsi une fonction qui se comporte comme f lorsque son 1^{er} paramètre est fixé à la valeur $e1$.

Applications partielles de fonctions

Considérons une fonction f à deux paramètres de types E_1 et E_2 et à type de retour S .

Deux faits évidents :

1. la fonction f est de type $E_1 \rightarrow E_2 \rightarrow S$;
2. si e_1 et e_2 sont des valeurs de types respectifs E_1 et E_2 , l'application de f à e_1 et e_2 par $(f\ e_1\ e_2)$ renvoie une valeur de type S .

Il est cependant possible d'**appliquer** f à **seulement** e_1 par $(f\ e_1)$. On obtient ainsi une valeur de type $E_2 \rightarrow S$ qui est donc une **fonction**.

La fonction $(f\ e_1)$ est ainsi une fonction qui se comporte comme f lorsque son 1^{er} paramètre est fixé à la valeur e_1 .

On dit que $(f\ e_1)$ est une **application partielle** de f à des arguments.

Applications partielles de fonctions

Il y a donc équivalence entre l'appel

```
(f e1 e2)
```

et l'appel

```
((f e1) e2) .
```

Applications partielles de fonctions

Il y a donc équivalence entre l'appel

```
(f e1 e2)
```

et l'appel

```
((f e1) e2) .
```

Plus généralement, si f est une fonction de type

```
E1 -> ... -> En -> S
```

et e_1, \dots, e_k sont des valeurs de types respectifs E_1, \dots, E_k avec $k \leq n$,

Applications partielles de fonctions

Il y a donc équivalence entre l'appel

```
(f e1 e2)
```

et l'appel

```
((f e1) e2) .
```

Plus généralement, si `f` est une fonction de type

```
E1 -> ... -> En -> S
```

et `e1`, ..., `ek` sont des valeurs de types respectifs `E1`, ..., `Ek` avec $k \leq n$, l'application partielle

```
(f e1 ... ek)
```

est une fonction de type

```
Ek+1 -> ... -> En -> S
```

Applications partielles de fonctions — exemple

```
# let distr a b c =  
  a * b + a * c;;
```

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

Cette fonction représente $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$ vérifiant
 $(a, b, c) \mapsto ab + ac$.

Applications partielles de fonctions — exemple

```
# let distr a b c =  
  a * b + a * c;;
```

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

```
# let distr_3 = distr 3;;
```

```
val distr_3 : int -> int -> int = <fun>
```

Cette fonction représente $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$ vérifiant
 $(a, b, c) \mapsto ab + ac$.

Cette fonction représente $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ vérifiant
 $(b, c) \mapsto 3b + 3c$.

Applications partielles de fonctions — exemple

```
# let distr a b c =  
  a * b + a * c;;
```

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

```
# let distr_3 = distr 3;;
```

```
val distr_3 : int -> int -> int = <fun>
```

```
# let distr_3_5 = distr_3 5;;
```

```
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$ vérifiant
 $(a, b, c) \mapsto ab + ac$.

Cette fonction représente $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ vérifiant
 $(b, c) \mapsto 3b + 3c$.

Cette fonction représente $f_{3,5} : \mathbb{Z} \rightarrow \mathbb{Z}$ vérifiant
 $c \mapsto 15 + 3c$.

Applications partielles de fonctions — exemple

```
# let distr a b c =  
  a * b + a * c;;
```

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

```
# let distr_3 = distr 3;;
```

```
val distr_3 : int -> int -> int = <fun>
```

```
# let distr_3_5 = distr_3 5;;
```

```
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

Cette fonction représente $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ vérifiant $(b, c) \mapsto 3b + 3c$.

Cette fonction représente $f_{3,5} : \mathbb{Z} \rightarrow \mathbb{Z}$ vérifiant $c \mapsto 15 + 3c$.

La fonction `distr_3_5` peut aussi être définie directement par l'une ou l'autre des deux manières suivantes :

```
let distr_3_5 c = distr 3 5 c
```

```
let distr_3_5 = distr 3 5
```

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une **fonction anonyme**.

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une **fonction anonyme**.

La construction syntaxique

```
fun P1 ... Pn -> EXP
```

où **P1**, ..., **Pn** sont des paramètres et **EXP** est une expression permet de définir une fonction anonyme.

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une **fonction anonyme**.

La construction syntaxique

```
fun P1 ... Pn -> EXP
```

où **P1**, ..., **Pn** sont des paramètres et **EXP** est une expression permet de définir une fonction anonyme.

```
# ((fun a b -> (a + b) * a) 4 3);;
```

```
- : int = 28
```

Définit une fonction anonyme appliquée aux arguments

4 et **3**.

Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une **fonction anonyme**.

La construction syntaxique

```
fun P1 ... Pn -> EXP
```

où **P1**, ..., **Pn** sont des paramètres et **EXP** est une expression permet de définir une fonction anonyme.

```
# ((fun a b -> (a + b) * a) 4 3);;
```

```
- : int = 28
```

```
# let produit k =  
  fun x -> x * k;;
```

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

Définit une fonction anonyme appliquée aux arguments

4 et **3**.

L'expression **produit 10** renvoie une fonction de type

int -> int qui multiplie par **10** son argument.

Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then z
```

Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit .
```

Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit .
```

```
let etrange x y =  
  "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```

Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit .
```

```
let etrange x y =  
  "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```

Cette fonction est de type

```
(char -> int) -> (int -> int -> string) -> string .
```

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float)  
->(bool -> int -> string)  
-> bool -> int -> float
```

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float)  
->(bool -> int -> string)  
-> bool -> int -> float
```

```
let etrange y =  
  ((fun x -> x + 1) 2) + (y (string_of_int 3))
```

Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float)  
->(bool -> int -> string)  
-> bool -> int -> float
```

```
let etrange y =  
  ((fun x -> x + 1) 2) + (y (string_of_int 3))
```

Cette fonction est de type

```
(string -> int) -> int
```

Plan

Pratique

Entrées et sorties

Compilation

Plan

Pratique

Entrées et sorties

Compilation

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des **fonctions d'entrée / sortie** qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des **fonctions d'entrée / sortie** qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée / sortie utilisent le type `unit` et son unique valeur `()`.

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des **fonctions d'entrée / sortie** qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée / sortie utilisent le type `unit` et son unique valeur `()`.

Rappel : l'utilisation d'entrées / sorties fait que l'on **sort du paradigme de programmation fonctionnelle pure** car elles produisent un effet (affichage ou bien attente d'une action de l'utilisateur).

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, toute fonction d'écriture renvoie `()`.

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, toute fonction d'écriture renvoie `()`.

Les fonctions d'écriture principales sont

```
val print_int : int -> unit
```

```
val print_float : float -> unit
```

```
val print_char : char -> unit
```

```
val print_string : string -> unit
```

```
val print_newline : unit -> unit
```

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

Les fonctions de lecture principales sont

```
val read_int : unit -> int
```

```
val read_float : unit -> float
```

```
val read_line : unit -> string
```

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'**opérateur de séquence** `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'**opérateur de séquence** `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'**opérateur de séquence** `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'**opérateur de séquence** `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

De ce fait, `E1`, ..., `En-1` doivent être de type `unit`.

Séquences — exemples

```
# let add x y =  
  print_string "Appel de add";  
  print_int x;  
  print_int y;  
  x + y;;  
val add : int -> int -> int = <fun>
```

Séquences — exemples

```
# let add x y =  
  print_string "Appel de add";  
  print_int x;  
  print_int y;  
  x + y;;  
val add : int -> int -> int = <fun>
```

```
# let test_div n =  
  if n mod 2 = 0 then  
    print_string "pair\n";  
  if n mod 3 = 0 then  
    print_string "multiple de 3\n";;  
val test_div : int -> unit = <fun>
```

Séquences et blocs

Considérons la fonction

```
# let div_decr x =  
  if x mod 2 = 0 then  
    print_string "pair";  
    x / 2  
  else  
    print_string "impair";  
    x - 1;;  
Error: Syntax error
```

Séquences et blocs

Considérons la fonction

```
# let div_decr x =  
  if x mod 2 = 0 then  
    print_string "pair";  
    x / 2  
  else  
    print_string "impair";  
    x - 1;;  
Error: Syntax error
```

L'opérateur de séquence est moins prioritaire que la conditionnelle.

Ainsi, cette fonction est comprise en

```
# let div_decr x =  
  if x mod 2 = 0 then  
    print_string "pair";  
  x / 2  
  else  
    print_string "impair";  
  x - 1;;
```

Ceci explique l'**erreur de syntaxe**.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de **bloc**. Un bloc est une expression de la forme

```
begin EXP end
```

où **EXP** est une expression.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de **bloc**. Un bloc est une expression de la forme

```
begin EXP end
```

où **EXP** est une expression. La valeur de `begin EXP end` est celle de **EXP**.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de **bloc**. Un bloc est une expression de la forme

```
begin EXP end
```

où **EXP** est une expression. La valeur de `begin EXP end` est celle de **EXP**.

La fonction précédente devient ainsi correcte en écrivant

```
let div_decr x =  
  if x mod 2 = 0 then begin  
    print_string "pair";  
    x / 2  
  end  
  else begin  
    print_string "impair";  
    x - 1  
  end  
end
```

Plan

Pratique

Entrées et sorties

Compilation

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur CAML, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour **obtenir un exécutable**.

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur CAML, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour **obtenir un exécutable**.

Pour cela, on écrit le programme dans un fichier d'extension `.ml`. On le compile par la commande

```
ocamlc -o Prog Prog.ml
```

Ceci invoque le **compilateur bytecode**.

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur CAML, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour **obtenir un exécutable**.

Pour cela, on écrit le programme dans un fichier d'extension `.ml`. On le compile par la commande

```
ocamlc -o Prog Prog.ml
```

Ceci invoque le **compilateur bytecode**.

Le **compilateur natif** produit des exécutables souvent plus performants. On y fait appel par la commande

```
ocamlopt -o Prog Prog.ml
```

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur CAML, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour **obtenir un exécutable**.

Pour cela, on écrit le programme dans un fichier d'extension `.ml`. On le compile par la commande

```
ocamlc -o Prog Prog.ml
```

Ceci invoque le **compilateur bytecode**.

Le **compilateur natif** produit des exécutables souvent plus performants. On y fait appel par la commande

```
ocamlopt -o Prog Prog.ml
```

Dans les deux cas, l'exécutable se lance par

```
./Prog
```

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

1. on construit un **fichier objet** (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

```
ocamlc -c F.ml
```

ou bien

```
ocamlopt -c F.ml
```

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

1. on construit un **fichier objet** (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

```
ocamlc -c F.ml
```

ou bien

```
ocamlopt -c F.ml
```

2. on appelle l'**éditeur de liens** par la commande

```
ocamlc -o Prog F1.cmo ... Fn.cmo
```

ou bien

```
ocamlopt -o Prog F1.cmx ... Fn.cmx
```

où les `F1.cm*`, ..., `Fn.cm*` sont les fichiers objet du projet.

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. **inclure** `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

2. Utiliser `f` aux endroits désirés dans `B.ml`.

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. **inclure** `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

2. Utiliser `f` aux endroits désirés dans `B.ml`.

Dans `B.ml`, le nom de `f` devient

```
A.f
```

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. **inclure** `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

2. Utiliser `f` aux endroits désirés dans `B.ml`.

Dans `B.ml`, le nom de `f` devient

```
A.f
```

Exemple :

```
(* A.ml *)  
  
let double x =  
  2 * x
```

```
(* B.ml *)  
  
open A  
  
let quadruple x =  
  2 * (A.double x)
```

Dans ce projet, `B.ml` inclut `A.ml`. Ainsi, la fonction `double` de `A.ml` est visible dans `B.ml` par l'identificateur `A.double`.

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple :

```
(* A.ml *)  
let calcul x =  
...
```

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple :

```
(* A.ml *)  
let calcul x =  
...
```

```
(* B.ml *)  
let calcul x y =  
...
```

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple :

```
(* A.ml *)  
let calcul x =  
...
```

```
(* B.ml *)  
let calcul x y =  
...
```

```
(* C.ml *)  
open A  
open B  
...  
A.calcul 1  
...  
B.calcul 'a' 2  
...
```

Espaces de noms

Un **espace de nom** est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de restreindre la visibilité de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple :

```
(* A.ml *)  
let calcul x =  
...
```

```
(* B.ml *)  
let calcul x y =  
...
```

```
(* C.ml *)  
open A  
open B  
...  
A.calcul 1  
...  
B.calcul 'a' 2  
...
```

Dans ce projet, deux fonctions nommées `calcul` sont définies. Leur nom absolu n'est en revanche pas le même (`A.calcul` et `B.calcul`). Il n'y a ainsi aucune ambiguïté dans `C.ml` qui inclut les deux fichiers précédents.

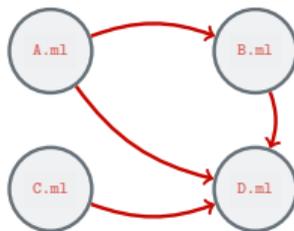
Ordre de compilation

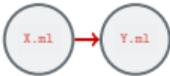
À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

Ordre de compilation

À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

Considérons le graphe d'inclusions suivant :

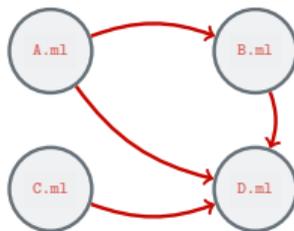


Toute flèche  signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

Ordre de compilation

À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

Considérons le graphe d'inclusions suivant :



Toute flèche  signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

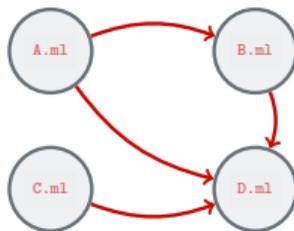
Dans cet exemple, on a les trois ordres suivants possibles :

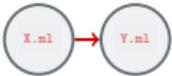
- ▶ `D.ml` , `C.ml` , `B.ml` , `A.ml` ;
- ▶ `D.ml` , `B.ml` , `C.ml` , `A.ml` ;
- ▶ `D.ml` , `B.ml` , `A.ml` , `C.ml` .

Ordre de compilation

À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

Considérons le graphe d'inclusions suivant :



Toute flèche  signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

Il faut donc compiler le projet dans l'ordre dicté par un **tri topologique** du graphe dual du graphe d'inclusions du projet.

Dans cet exemple, on a les trois ordres suivants possibles :

- ▶ `D.ml` , `C.ml` , `B.ml` , `A.ml` ;
- ▶ `D.ml` , `B.ml` , `C.ml` , `A.ml` ;
- ▶ `D.ml` , `B.ml` , `A.ml` , `C.ml` .

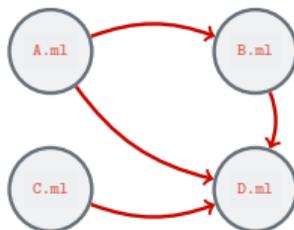
Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

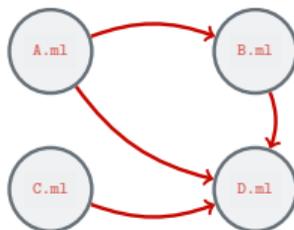
Reprenons le graphe d'inclusions



Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Reprenons le graphe d'inclusions



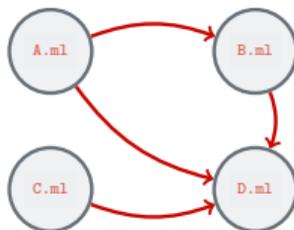
La commande `ocamldep *.ml` affiche

```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
D.cmo:
D.cmx:
```

Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Reprenons le graphe d'inclusions



La commande `ocamldep *.ml` affiche

```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
D.cmo:
D.cmx:
```

Il est ainsi possible d'écrire des `Makefile` génériques en appelant, à l'intérieur, `ocamldep`.

Plan

Types

- L'algèbre des types

- Types produit

- Types somme

- Types paramétrés

Plan

Types

L'algèbre des types

Types produit

Types somme

Types paramétrés

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

Dire qu'un identificateur x est de type T signifie que la valeur de x est dans T .

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

Dire qu'un identificateur `x` est de type `T` signifie que la valeur de `x` est dans `T`.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

Dire qu'un identificateur `x` est de type `T` signifie que la valeur de `x` est dans `T`.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les **types construits**, qui sont des constructions faisant intervenir des types scalaires ou construits et des **opérateurs de types**.

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

Dire qu'un identificateur `x` est de type `T` signifie que la valeur de `x` est dans `T`.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les **types construits**, qui sont des constructions faisant intervenir des types scalaires ou construits et des **opérateurs de types**.

Les types peuvent avoir des relations entre eux (p.ex., un type `T` peut être un sous-ensemble d'un type `S`).

L'algèbre des types

En programmation (fonctionnelle), un **type** est un ensemble de valeurs.

Dire qu'un identificateur `x` est de type `T` signifie que la valeur de `x` est dans `T`.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les **types construits**, qui sont des constructions faisant intervenir des types scalaires ou construits et des **opérateurs de types**.

Les types peuvent avoir des relations entre eux (p.ex., un type `T` peut être un sous-ensemble d'un type `S`).

L'ensemble des types d'un langage et des opérateurs de types est son **algèbre des types**.

L'algèbre des types

La définition d'un nouveau type `ID` se fait par

```
type ID = OP
```

où `OP` fait intervenir des types et des opérateurs de types.

L'algèbre des types

La définition d'un nouveau type `ID` se fait par

```
type ID = OP
```

où `OP` fait intervenir des types et des opérateurs de types.

On rappelle que les types scalaires dont nous disposons sont

```
int , float , char , string , bool , unit .
```

L'algèbre des types

La définition d'un nouveau type `ID` se fait par

```
type ID = OP
```

où `OP` fait intervenir des types et des opérateurs de types.

On rappelle que les types scalaires dont nous disposons sont

```
int , float , char , string , bool , unit .
```

Voici les opérateurs de types que l'on va considérer :

| Opérateur | Arité | Nom |
|--------------------|----------|----------------------------|
| <code>-></code> | 2 | Flèche |
| <code>*</code> | 2 | Produit cartésien binaire |
| <code>*</code> | ≥ 2 | Produit cartésien multiple |
| <code>{ }</code> | ≥ 1 | Produit nommé |
| <code> </code> | ≥ 1 | Somme |

Plan

Types

L'algèbre des types

Types produit

Types somme

Types paramétrés

Produit cartésien binaire

Étant donnés deux types T_1 et T_2 ,

$T_1 * T_2$

désigne le type **produit cartésien binaire** de T_1 et T_2 .

Produit cartésien binaire

Étant donnés deux types T_1 et T_2 ,

$T_1 * T_2$

désigne le type **produit cartésien binaire** de T_1 et T_2 .

Il contient pour valeurs les **couples** (e_1, e_2) où e_1 (resp. e_2) est de type T_1 (resp. T_2).

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien binaire** de `T1` et `T2`.

Il contient pour valeurs les **couples** `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

```
type point = int * int
```

Le type `point` contient tous les couples à coordonnées entières.

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien binaire** de `T1` et `T2`.

Il contient pour valeurs les **couples** `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

```
type point = int * int
```

Le type `point` contient tous les couples à coordonnées entières.

Un couple s'écrit

`(e1, e2)`

Les parenthèses sont facultatives.

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien binaire** de `T1` et `T2`.

Il contient pour valeurs les **couples** `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

```
type point = int * int
```

Le type `point` contient tous les couples à coordonnées entières.

Un couple s'écrit

`(e1, e2)`

Les parenthèses sont facultatives.

```
# (3.5, 21);;
```

```
- : float * int = (3.5, 21)
```

```
# 3.5, 21;;
```

```
- : float * int = (3.5, 21)
```

```
# (1, (2, 3));;
```

```
- : int * (int * int) = (1, (2, 3))
```

```
# ((1, 2), 3);;
```

```
- : (int * int) * int = ((1, 2), 3)
```

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1^{re} coordonnée par `fst c` et à sa 2^e coordonnée par `snd c`.

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1^{re} coordonnée par `fst c` et à sa 2^e coordonnée par `snd c`.

```
# let add c =  
    (fst c) + (snd c);;
```

```
val add : int * int -> int = <fun>
```

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1^{re} coordonnée par `fst c` et à sa 2^e coordonnée par `snd c`.

```
# let add c =  
    (fst c) + (snd c);;
```

```
val add : int * int -> int = <fun>
```

```
# add (2, 4);;
```

```
- : int = 6
```

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1^{re} coordonnée par `fst c` et à sa 2^e coordonnée par `snd c`.

```
# let add c =  
    (fst c) + (snd c);;
```

```
val add : int * int -> int = <fun>
```

```
# add (2, 4);;
```

```
- : int = 6
```

L'opérateur de types `*` est prioritaire face à `->`.

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1^{re} coordonnée par `fst c` et à sa 2^e coordonnée par `snd c`.

```
# let add c =  
    (fst c) + (snd c);;
```

```
val add : int * int -> int = <fun>
```

```
# add (2, 4);;
```

```
- : int = 6
```

L'opérateur de types `*` est prioritaire face à `->`.

Il existe un moyen plus élégant (et plus général) pour accéder aux coordonnées de `c` au moyen de

```
let (c1, c2) = c in ...
```

P.ex.,

```
# let add c =  
    let (c1, c2) = c in  
    c1 + c2;;
```

```
val add : int * int -> int = <fun>
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

```
# let conc_12 x =  
  let (x1, x2) = x in  
  let (x21, x22) = x2 in  
  x21 ^ x1 ^ x22;;
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

```
# let conc_12 x =  
  let (x1, x2) = x in  
  let (x21, x22) = x2 in  
  x21 ^ x1 ^ x22;;
```

```
val conc_12 : string * (string * string)  
-> string = <fun>
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

```
# (conc_21 (("a", "b"), "c"));;
```

```
- : string = "acb"
```

```
# let conc_12 x =  
  let (x1, x2) = x in  
  let (x21, x22) = x2 in  
  x21 ^ x1 ^ x22;;
```

```
val conc_12 : string * (string * string)  
-> string = <fun>
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

```
# (conc_21 (("a", "b"), "c"));;
```

```
- : string = "acb"
```

```
# let conc_12 x =  
  let (x1, x2) = x in  
  let (x21, x22) = x2 in  
  x21 ^ x1 ^ x22;;
```

```
val conc_12 : string * (string * string)  
-> string = <fun>
```

```
# (conc_12 ("a", ("b", "c")));;
```

```
- : string = "bac"
```

Non associativité du produit cartésien

Considérons les fonctions

```
# let conc_21 x =  
  let (x1, x2) = x in  
  let (x11, x12) = x1 in  
  x11 ^ x2 ^ x12;;
```

```
val conc_21 : (string * string) * string  
-> string = <fun>
```

```
# (conc_21 (("a", "b"), "c"));;
```

```
- : string = "acb"
```

```
# let conc_12 x =  
  let (x1, x2) = x in  
  let (x21, x22) = x2 in  
  x21 ^ x1 ^ x22;;
```

```
val conc_12 : string * (string * string)  
-> string = <fun>
```

```
# (conc_12 ("a", ("b", "c")));;
```

```
- : string = "bac"
```

Ces deux fonctions ne sont pas du même type car l'opérateur de types `*` est **non associatif**. En effet, les types `(string * string) * string` et `string * (string * string)` sont différents.

n -uplets

Étant donnés des types T_1, \dots, T_n ,

$T_1 * \dots * T_n$

désigne le type **produit cartésien multiple** de T_1, \dots, T_n .

n -uplets

Étant donnés des types T_1, \dots, T_n ,

$T_1 * \dots * T_n$

désigne le type **produit cartésien multiple** de T_1, \dots, T_n .

Il contient pour valeurs les n -uplets (e_1, \dots, e_n) où pour tout $1 \leq i \leq n$, e_i est de type T_i .

n -uplets

Étant donnés des types T_1, \dots, T_n ,

```
T1 * ... * Tn
```

désigne le type **produit cartésien multiple** de T_1, \dots, T_n .

Il contient pour valeurs les n -uplets (e_1, \dots, e_n) où pour tout $1 \leq i \leq n$, e_i est de type T_i .

```
type c = int * unit * (int -> char)
```

n -uplets

Étant donnés des types `T1`, ..., `Tn`,

```
T1 * ... * Tn
```

désigne le type **produit cartésien multiple** de `T1`, ..., `Tn`.

Il contient pour valeurs les n -uplets `(e1, ..., en)` où pour tout $1 \leq i \leq n$, `ei` est de type `Ti`.

```
type c = int * unit * (int -> char)
```

Un n -uplet s'écrit

```
(e1, ..., en)
```

Les parenthèses sont facultatives.

n -uplets

Étant donnés des types T_1, \dots, T_n ,

```
T1 * ... * Tn
```

désigne le type **produit cartésien multiple** de T_1, \dots, T_n .

Il contient pour valeurs les n -uplets (e_1, \dots, e_n) où pour tout $1 \leq i \leq n$, e_i est de type T_i .

```
type c = int * unit * (int -> char)
```

Un n -uplet s'écrit

```
(e1, ..., en)
```

Les parenthèses sont facultatives.

```
# (0., 1, "abc", 'v');
```

```
- : float * int * string * char = (0., 1, "abc", 'v')
```

Accès aux coordonnées d'un n -uplet

Il n'est pas possible d'utiliser `fst` ni `snd` pour accéder aux coordonnées d'un n -uplet.

Accès aux coordonnées d'un n -uplet

Il n'est pas possible d'utiliser `fst` ni `snd` pour accéder aux coordonnées d'un n -uplet.

Si `c` est un n -uplet, on accède à ses coordonnées au moyen de

```
let (e1, ..., en) = c in ...
```

Accès aux coordonnées d'un n -uplet

Il n'est pas possible d'utiliser `fst` ni `snd` pour accéder aux coordonnées d'un n -uplet.

Si `c` est un n -uplet, on accède à ses coordonnées au moyen de

```
let (e1, ..., en) = c in ...
```

Il est possible de ne recueillir que la k^{e} de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = c in ...
```

Le symbole `_` est un **joker**. Il ne peut pas être lié à une valeur mais accepte néanmoins syntaxiquement d'être défini.

Accès aux coordonnées d'un n -uplet

Il n'est pas possible d'utiliser `fst` ni `snd` pour accéder aux coordonnées d'un n -uplet.

Si `c` est un n -uplet, on accède à ses coordonnées au moyen de

```
let (e1, ..., en) = c in ...
```

Il est possible de ne recueillir que la k^e de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = c in ...
```

```
# let (_, _, x, _, _) = (1, 2, 3, 4, 5) in x + 10;;
```

Le symbole `_` est un **joker**. Il ne peut pas être lié à une valeur mais accepte néanmoins syntaxiquement d'être défini.

Accès aux coordonnées d'un n -uplet

Il n'est pas possible d'utiliser `fst` ni `snd` pour accéder aux coordonnées d'un n -uplet.

Si `c` est un n -uplet, on accède à ses coordonnées au moyen de

```
let (e1, ..., en) = c in ...
```

Il est possible de ne recueillir que la k^e de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = c in ...
```

```
# let (_, _, x, _, _) = (1, 2, 3, 4, 5) in x + 10;;
```

```
- : int = 13
```

Le symbole `_` est un **joker**. Il ne peut pas être lié à une valeur mais accepte néanmoins syntaxiquement d'être défini.

Produit nommé

Étant donnés des types T_1, \dots, T_n et des identificateurs ID_1, \dots, ID_n ,

$\{ID_1 : T_1 ; \dots ; ID_n : T_n\}$

désigne le type **produit nommé** de T_1, \dots, T_n .

Produit nommé

Étant donnés des types T_1, \dots, T_n et des identificateurs ID_1, \dots, ID_n ,

$$\{ID_1 : T_1 ; \dots ; ID_n : T_n\}$$

désigne le type **produit nommé** de T_1, \dots, T_n .

Il contient pour valeurs les **enregistrements** dont les **champs** sont ID_1, \dots, ID_n de types respectifs T_1, \dots, T_n .

Produit nommé

Étant donnés des types T_1, \dots, T_n et des identificateurs ID_1, \dots, ID_n ,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type **produit nommé** de T_1, \dots, T_n .

Il contient pour valeurs les **enregistrements** dont les **champs** sont ID_1, \dots, ID_n de types respectifs T_1, \dots, T_n .

```
type personne = {nom : string ; age : int}
```

Produit nommé

Étant donnés des types T_1, \dots, T_n et des identificateurs ID_1, \dots, ID_n ,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type **produit nommé** de T_1, \dots, T_n .

Il contient pour valeurs les **enregistrements** dont les **champs** sont ID_1, \dots, ID_n de types respectifs T_1, \dots, T_n .

```
type personne = {nom : string ; age : int}
```

Un enregistrement s'écrit

```
{ID1 = V1 ; ... ; IDn = Vn}
```

où V_1, \dots, V_n sont des valeurs de types respectifs T_1, \dots, T_n .

Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type **produit nommé** de `T1`, ..., `Tn`.

Il contient pour valeurs les **enregistrements** dont les **champs** sont `ID1`, ..., `IDn` de types respectifs `T1`, ..., `Tn`.

```
type personne = {nom : string ; age : int}
```

Un enregistrement s'écrit

```
{ID1 = V1 ; ... ; IDn = Vn}
```

où `V1`, ..., `Vn` sont des valeurs de types respectifs `T1`, ..., `Tn`.

```
# {nom = "Haskell Curry" ; age = 81};;
```

```
- : personne = {nom = "Haskell Curry"; age = 81}
```

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

`e.c`

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

`e.c`

P.ex.,

```
# let p = {nom = "Alan Turing" ; age = 41} in p.nom;;  
- : string = "Alan Turing"
```

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

```
e.c
```

P.ex.,

```
# let p = {nom = "Alan Turing" ; age = 41} in p.nom;;
```

```
- : string = "Alan Turing"
```

```
# let nom_du_plus_age p1 p2 =  
  if p1.age > p2.age then  
    p1.nom  
  else if p1.age < p2.age then  
    p2.nom  
  else  
    "";;
```

```
val nom_du_plus_age : personne -> personne -> string = <fun>
```

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

```
type point = {a : int ; b : int ; c : int}
```

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

```
type point = {a : int ; b : int ; c : int}
```

```
# let p1 = {a = 1 ; b = 2 ; c = 3};;
```

```
val p1 : point = {a = 1; b = 2; c = 3}
```

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

```
type point = {a : int ; b : int ; c : int}
```

```
# let p1 = {a = 1 ; b = 2 ; c = 3};;
```

```
val p1 : point = {a = 1; b = 2; c = 3}
```

```
# let p2 = {p1 with a = -1};;
```

```
val p2 : point = {a = -1; b = 2; c = 3}
```

« Modification » des champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

```
type point = {a : int ; b : int ; c : int}
```

```
# let p1 = {a = 1 ; b = 2 ; c = 3};;
```

```
val p1 : point = {a = 1; b = 2; c = 3}
```

```
# let p2 = {p1 with a = -1};;
```

```
val p2 : point = {a = -1; b = 2; c = 3}
```

```
# let p3 = {p1 with b = 3 ; c = 4};;
```

```
val p3 : point = {a = 1; b = 3; c = 4}
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne).

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# type t1 = {a : int ; b : int};;  
type t1 = {a : int ; b : int; };;
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}
type t2 = {a : int ; c : char}
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# type t1 = {a : int ; b : int};;
type t1 = {a : int ; b : int; };;
```

```
# type t2 = {a : int ; c : char};;
type t2 = { a : int; c : char; }
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# type t1 = {a : int ; b : int};;
```

```
type t1 = {a : int ; b : int; };;
```

```
# let f x = x.a;;
```

```
val f : t2 -> int = <fun>
```

```
# type t2 = {a : int ; c : char};;
```

```
type t2 = { a : int; c : char; }
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# type t1 = {a : int ; b : int};;  
type t1 = {a : int ; b : int; };
```

```
# type t2 = {a : int ; c : char};;  
type t2 = { a : int; c : char; }
```

```
# let f x = x.a;;  
val f : t2 -> int = <fun>
```

```
# {a = 2 ; c = 'y'};;  
- : t2 = {a = 2; c = 'y'}
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
type t1 = {a : int ; b : int}  
type t2 = {a : int ; c : char}  
let f x = x.a
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# type t1 = {a : int ; b : int};;  
type t1 = {a : int ; b : int; };;
```

```
# let f x = x.a;;  
val f : t2 -> int = <fun>
```

```
# {a = 2 ; c = 'y'};;  
- : t2 = {a = 2; c = 'y'}
```

```
# type t2 = {a : int ; c : char};;  
type t2 = { a : int; c : char; }
```

```
# {a = 2 ; b = 3};;
```

```
Error: The record field label b belongs to the type t1  
but is mixed here with labels of type t2
```

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par **A.** s'ils sont d'un type défini dans un fichier nommé **A.ml** .

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par **A.** s'ils sont d'un type défini dans un fichier nommé **A.ml** .

Par exemple :

```
(* A.ml *)  
type a = {a : int ; b : int}
```

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par **A.** s'ils sont d'un type défini dans un fichier nommé **A.ml** .

Par exemple :

```
(* A.ml *)
```

```
type a = {a : int ; b : int}
```

```
(* B.ml *)
```

```
type b = {a : int ; c : char}
```

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par **A.** s'ils sont d'un type défini dans un fichier nommé **A.ml** .

Par exemple :

```
(* A.ml *)
```

```
type a = {a : int ; b : int}
```

```
(* B.ml *)
```

```
type b = {a : int ; c : char}
```

```
(* C.ml *)
```

```
open A
```

```
open B
```

```
let c1 = {B.a = 2 ; B.c = 'y'}
```

```
and c2 = {A.a = 2 ; A.b = 3}
```

Plan

Types

L'algèbre des types

Types produit

Types somme

Types paramétrés

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Une **valeur** d'un type somme s'écrit via son constructeur.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
```

```
- : numero = Deux
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
```

```
- : numero = Deux
```

```
# let plusieurs n =  
    n = Deux || n = Trois;;
```

```
val plusieurs : numero -> bool = <fun>
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
```

```
- : numero = Deux
```

```
# plusieurs Un;;
```

```
- : bool = false
```

```
# let plusieurs n =  
    n = Deux || n = Trois;;
```

```
val plusieurs : numero -> bool = <fun>
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1^{res} lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des **constructeurs**.

```
type numero = Un | Deux | Trois
```

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
```

```
- : numero = Deux
```

```
# let plusieurs n =  
    n = Deux || n = Trois;;
```

```
val plusieurs : numero -> bool = <fun>
```

```
# plusieurs Un;;
```

```
- : bool = false
```

```
# plusieurs Trois;;
```

```
- : bool = true
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# Entier 13;;
```

```
- : nombre = Entier 13
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# Entier 13;;
```

```
- : nombre = Entier 13
```

```
# Rationnel (2, 3);;
```

```
- : nombre = Rationnel (2, 3)
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un **argument**. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ... | Idk of T | ... | Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# Entier 13;;
```

```
- : nombre = Entier 13
```

```
# Rationnel (2, 3);;
```

```
- : nombre = Rationnel (2, 3)
```

```
# Infini;;
```

```
- : nombre = Infini
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récurif** suivant :

```
type liste_int =  
  |Vide  
  |Cellule of int * liste_int
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récurif** suivant :

```
type liste_int =  
  |Vide  
  |Cellule of int * liste_int
```

On construit des listes d'entiers de la manière suivante :

```
# let e1 = Cellule (1, Vide);; val e1 : liste_int = Cellule (1, Vide)
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récuratif** suivant :

```
type liste_int =  
  |Vide  
  |Cellule of int * liste_int
```

On construit des listes d'entiers de la manière suivante :

```
# let e1 = Cellule (1, Vide);; val e1 : liste_int = Cellule (1, Vide)
```

```
# let e2 = Cellule (2, e1);; val e2 : liste_int = Cellule (2, Cellule (1, Vide))
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récuratif** suivant :

```
type liste_int =  
  | Vide  
  | Cellule of int * liste_int
```

On construit des listes d'entiers de la manière suivante :

```
# let e1 = Cellule (1, Vide);; val e1 : liste_int = Cellule (1, Vide)
```

```
# let e2 = Cellule (2, e1);; val e2 : liste_int = Cellule (2, Cellule (1, Vide))
```

```
# let e3 = Cellule (3, e2);; val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récuratif** suivant :

```
type liste_int =  
  |Vide  
  |Cellule of int * liste_int
```

On construit des listes d'entiers de la manière suivante :

```
# let e1 = Cellule (1, Vide);; val e1 : liste_int = Cellule (1, Vide)
```

```
# let e2 = Cellule (2, e1);; val e2 : liste_int = Cellule (2, Cellule (1, Vide))
```

```
# let e3 = Cellule (3, e2);; val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom `e3` est lié à la liste de valeur

| | | |
|---|---|---|
| 3 | 2 | 1 |
|---|---|---|

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récuratif** ci-contre :

```
type arbre_b_int =  
  |Vide  
  |Noeud of arbre_b_int * int * arbre_b_int
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récuratif** ci-contre :

```
type arbre_b_int =  
  |Vide  
  |Noeud of arbre_b_int * int * arbre_b_int
```

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
```

```
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récuratif** ci-contre :

```
type arbre_b_int =  
  |Vide  
  |Noeud of arbre_b_int * int * arbre_b_int
```

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
```

```
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

```
# let a2 = (Noeud (a1, 3, a1));;
```

```
val a2 : arbre_b_int = Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récuratif** ci-contre :

```
type arbre_b_int =  
  |Vide  
  |Noeud of arbre_b_int * int * arbre_b_int
```

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
```

```
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

```
# let a2 = (Noeud (a1, 3, a1));;
```

```
val a2 : arbre_b_int = Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
```

```
# let a3 = (Noeud (a2, 5, a1));;
```

```
val a3 : arbre_b_int = Noeud (Noeud (Noeud (Vide,1,Vide), 3, Noeud (Vide,1,Vide)), 5, Noeud (Vide,1,Vide))
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récuratif** ci-contre :

```
type arbre_b_int =  
  |Vide  
  |Noeud of arbre_b_int * int * arbre_b_int
```

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
```

```
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

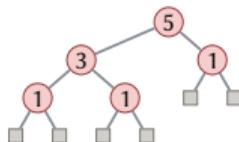
```
# let a2 = (Noeud (a1, 3, a1));;
```

```
val a2 : arbre_b_int = Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
```

```
# let a3 = (Noeud (a2, 5, a1));;
```

```
val a3 : arbre_b_int = Noeud (Noeud (Noeud (Vide,1,Vide), 3, Noeud (Vide,1,Vide)), 5, Noeud (Vide,1,Vide))
```

Le nom **a3** est lié l'arbre binaire de valeur



Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- ▶ l'arbre vide **ou bien**

Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- ▶ l'arbre vide **ou bien**
- ▶ un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 **ou bien**

Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

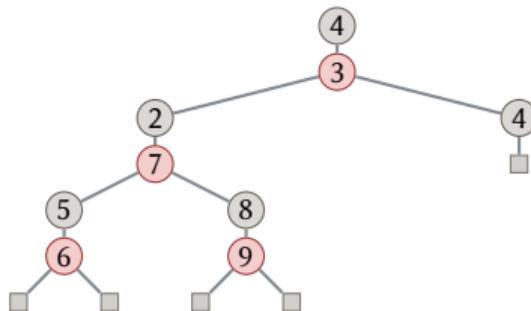
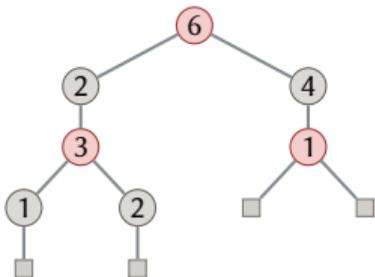
- ▶ l'arbre vide **ou bien**
- ▶ un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 **ou bien**
- ▶ un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

Exemple 3 – arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- ▶ l'arbre vide **ou bien**
- ▶ un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 **ou bien**
- ▶ un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

Voici p.ex. deux arbres unaires binaires, respectivement dont la racine est d'arité 2 et dont la racine est d'arité 1 :



Exemple 3 — arbres unaires binaires d'entiers

La définition de ces objets se traduit en les définitions de types suivantes.

Exemple 3 — arbres unaires binaires d'entiers

La définition de ces objets se traduit en les définitions de types suivantes.

On commence par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

```
type arbre_1 =  
  |Vide1  
  |Noeud1 of int * arbre_2  
and arbre_2 =  
  |Vide2  
  |Noeud2 of arbre_1 * int * arbre_1
```

Exemple 3 — arbres unaires binaires d'entiers

La définition de ces objets se traduit en les définitions de types suivantes.

On commence par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

```
type arbre_1 =  
  |Vide1  
  |Noeud1 of int * arbre_2  
and arbre_2 =  
  |Vide2  
  |Noeud2 of arbre_1 * int * arbre_1
```

Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet **mutuellement récursifs**.

Exemple 3 — arbres unaires binaires d'entiers

La définition de ces objets se traduit en les définitions de types suivantes.

On commence par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

```
type arbre_1 =  
  |Vide1  
  |Noeud1 of int * arbre_2  
and arbre_2 =  
  |Vide2  
  |Noeud2 of arbre_1 * int * arbre_1
```

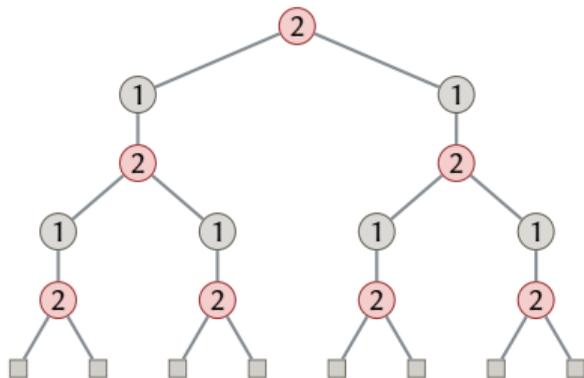
Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet **mutuellement récursifs**.

On se base sur la définition des arbres unaires binaires pour construire finalement le type recherché :

```
type arbre_12 =  
  |Vide  
  |Arbre1 of arbre_1  
  |Arbre2 of arbre_2
```

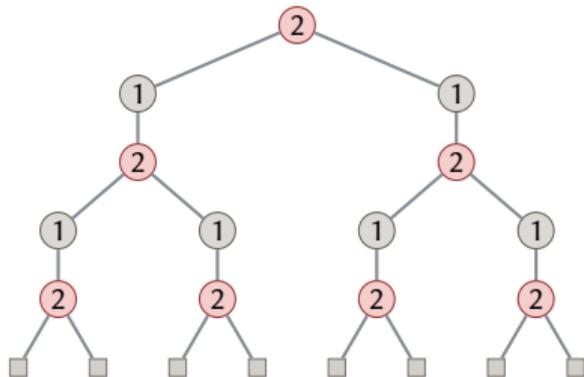
Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



Exemple 3 — arbres unaires binaires d'entiers

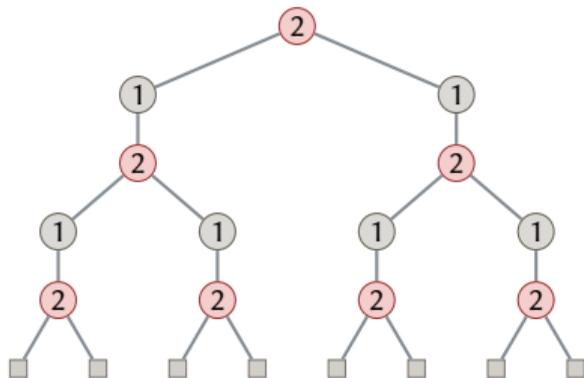
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
```

Exemple 3 — arbres unaires binaires d'entiers

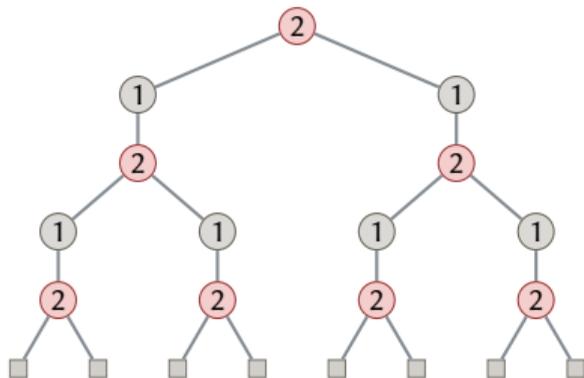
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in  
  let a1 = Noeud1 (1, a2) in
```

Exemple 3 — arbres unaires binaires d'entiers

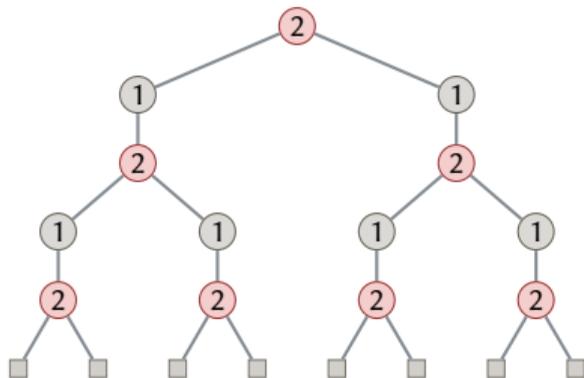
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
  let a1 = Noeud1 (1, a2) in
    let a22 = Noeud2 (a1, 2, a1) in
```

Exemple 3 — arbres unaires binaires d'entiers

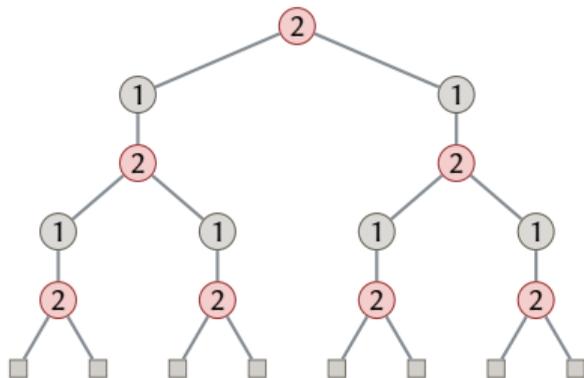
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
  let a1 = Noeud1 (1, a2) in
    let a22 = Noeud2 (a1, 2, a1) in
      let a11 = Noeud1 (1, a22) in
```

Exemple 3 — arbres unaires binaires d'entiers

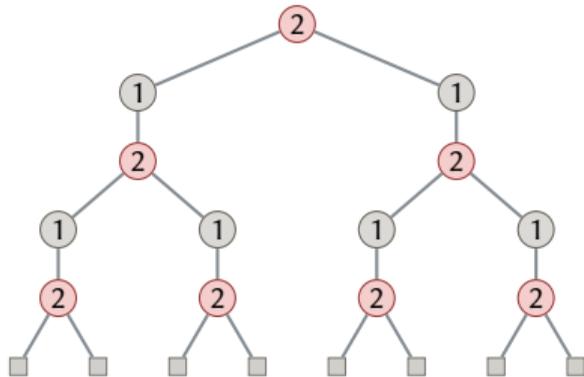
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
  let a1 = Noeud1 (1, a2) in
    let a22 = Noeud2 (a1, 2, a1) in
      let a11 = Noeud1 (1, a22) in
        Arbre2 (Noeud2 (a11, 2, a11));;
```

Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
  let a1 = Noeud1 (1, a2) in
    let a22 = Noeud2 (a1, 2, a1) in
      let a11 = Noeud1 (1, a22) in
        Arbre2 (Noeud2 (a11, 2, a11));;
```

```
- : arbre_12 =
Arbre2
(Noeud2
 (Noeud1 (1,
  Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
  Noeud1 (1, Noeud2 (Vide1, 2, Vide1))))) ,
 2,
Noeud1 (1,
  Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
  Noeud1 (1, Noeud2 (Vide1, 2, Vide1)))))
```

Plan

Types

L'algèbre des types

Types produit

Types somme

Types paramétrés

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

La définition d'un nouveau type paramétré **ID** se fait par

```
type ('P1, ..., 'Pn) ID = OP
```

où **P1**, ..., **Pn** sont des identificateurs et **OP** fait intervenir des types, des opérateurs de types et **'P1**, ..., **'Pn**.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

La définition d'un nouveau type paramétré **ID** se fait par

```
type ('P1, ..., 'Pn) ID = OP
```

où **P1**, ..., **Pn** sont des identificateurs et **OP** fait intervenir des types, des opérateurs de types et **'P1**, ..., **'Pn**.

Les **'P1**, ..., **'Pn** sont des **paramètres de types**.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

La définition d'un nouveau type paramétré `ID` se fait par

```
type ('P1, ..., 'Pn) ID = OP
```

où `P1`, ..., `Pn` sont des identificateurs et `OP` fait intervenir des types, des opérateurs de types et `'P1`, ..., `'Pn`.

Les `'P1`, ..., `'Pn` sont des **paramètres de types**.

Lorsque `n = 1`, la définition se fait simplement (sans parenthèses) par

```
type 'P1 ID = OP
```

Rôle des paramètres de type

Supposons que `T` soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

Rôle des paramètres de type

Supposons que `T` soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que `T` est paramétré par `'P1`, ..., `'Pn`.

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que T est paramétré par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que T est paramétré par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que T est paramétré par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que T est paramétré par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);
2. des paramètres de types (vus comme **des variables**);

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que T est paramétré par $'P1, \dots, 'Pn$.

Dans la définition de T , les occurrences de $'Pi$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);
2. des paramètres de types (vus comme **des variables**);
3. des opérateurs de types.

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste =  
  |Vide  
  |Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

Exemple 1 – listes génériques

Le type à un paramètre

```
type 'e liste =  
  |Vide  
  |Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Exemple 1 – listes génériques

Le type à un paramètre

```
type 'e liste =  
  | Vide  
  | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# Cellule ('a', (Cellule ('b', Vide))); - : char liste = Cellule ('a', Cellule ('b', Vide))
```

Exemple 1 – listes génériques

Le type à un paramètre

```
type 'e liste =  
  | Vide  
  | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# Cellule ('a', (Cellule ('b', Vide))); - : char liste = Cellule ('a', Cellule ('b', Vide))
```

Une liste de listes d'entiers :

```
# Cellule ((Cellule (1, Vide)), Vide);  
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

Exemple 1 – listes génériques

Le type à un paramètre

```
type 'e liste =  
  | Vide  
  | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# Cellule ('a', (Cellule ('b', Vide))); - : char liste = Cellule ('a', Cellule ('b', Vide))
```

Une liste de listes d'entiers :

```
# Cellule ((Cellule (1, Vide)), Vide);  
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

Une liste dont le type des éléments n'est pas déterminé : `# Vide;;` - : 'a liste = Vide

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en `'a`, `'b`, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en `'a`, `'b`, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};; val c : (int, char) couple = {x = 4; y = 'd'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en `'a`, `'b`, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};; val c : (int, char) couple = {x = 4; y = 'd'}
```

```
# {c with y = 'e'};; - : (int, char) couple = {x = 4; y = 'e'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
```

```
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en `'a`, `'b`, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};; val c : (int, char) couple = {x = 4; y = 'd'}
```

```
# {c with y = 'e'};; - : (int, char) couple = {x = 4; y = 'e'}
```

```
# {c with y = 2.2};; - : (int, float) couple = {x = 4; y = 2.2}
```

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : `'u` pour le type des valeurs des nœuds unaires, et `'x` et `'y` pour ceux des nœuds binaires.

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : `'u` pour le type des valeurs des nœuds unaires, et `'x` et `'y` pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  |Vide1  
  |Noeud1 of 'u * ('u, 'x, 'y) arbre_2
```

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : `'u` pour le type des valeurs des nœuds unaires, et `'x` et `'y` pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  |Vide1  
  |Noeud1 of 'u * ('u, 'x, 'y) arbre_2  
and ('u, 'x, 'y) arbre_2 =  
  |Vide2  
  |Noeud2 of ('u, 'x, 'y) arbre_1 * ('x * 'y) * ('u, 'x, 'y) arbre_1
```

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : `'u` pour le type des valeurs des nœuds unaires, et `'x` et `'y` pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  |Vide1  
  |Noeud1 of 'u * ('u, 'x, 'y) arbre_2  
and ('u, 'x, 'y) arbre_2 =  
  |Vide2  
  |Noeud2 of ('u, 'x, 'y) arbre_1 * ('x * 'y) * ('u, 'x, 'y) arbre_1
```

```
type ('u, 'x, 'y) arbre_12 =  
  |Vide  
  |Arbre1 of ('u, 'x, 'y) arbre_1  
  |Arbre2 of ('u, 'x, 'y) arbre_2
```

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u :=$ **baacba** vérifie $u_1 =$ **b** et $u_4 =$ **c**.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u :=$ `baacba` vérifie $u_1 =$ `b` et $u_4 =$ `c`.

Usuellement, en **programmation impérative** (et donc dans un style mutable), un mot u est représenté par un tableau `tab` qui contient ses lettres ($u_i =$ `tab[i - 1]`). Ceci supporte les opérations de lecture et modification de lettres, ainsi que de concaténation.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u :=$ `baacba` vérifie $u_1 =$ `b` et $u_4 =$ `c`.

Usuellement, en **programmation impérative** (et donc dans un style mutable), un mot u est représenté par un tableau `tab` qui contient ses lettres ($u_i =$ `tab[i - 1]`). Ceci supporte les opérations de lecture et modification de lettres, ainsi que de concaténation.

En **programmation fonctionnelle** (et donc dans un style non mutable), il est possible d'utiliser une liste simplement chaînée pour représenter les lettres du mot. Il est cependant possible de faire mieux.

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot ;

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
  lettres : int -> 'a;  
  longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot ;
- ▶ le champ `lettres` est la fonction dont le rôle a été décrit plus haut ;

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
  lettres : int -> 'a;  
  longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot ;
- ▶ le champ `lettres` est la fonction dont le rôle a été décrit plus haut ;
- ▶ le champ `longueur` contient la longueur du mot (ceci est nécessaire car on ne peut pas déduire la longueur du mot uniquement depuis la fonction `lettres`).

Exemple 4 — Mots

```
# let mot_1 = {  
  lettres =  
    (fun i ->  
      if i = 2 || i = 3 then  
        'a'  
      else  
        'b'  
    );  
  longueur = 5  
};;
```

```
val mot_1 : char mot = {lettres = <fun>; longueur = 5}
```

Ceci lie au nom `mot_1` le mot `baabb`.

Exemple 4 — Mots

```
# let mot_1 = {  
  lettres =  
    (fun i ->  
      if i = 2 || i = 3 then  
        'a'  
      else  
        'b'  
    );  
  longueur = 5  
};;
```

```
# let mot_2 = {  
  lettres = (fun i -> i mod 2 = 0);  
  longueur = 4294967296  
};;
```

```
val mot_2 : bool mot = {lettres = <fun>;  
  longueur = 4294967296}
```

```
val mot_1 : char mot = {lettres = <fun>; longueur = 5}
```

Ceci lie au nom `mot_1` le mot `baabb`.

Ceci lie au nom `mot_2` le mot `FTFTF...` de longueur $4294967296 = 2^{32}$ (où `F` désigne `false` et `T` désigne `true`).

La quantité de mémoire utilisée est négligeable devant la longueur du mot.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
type point = int * int
```

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
type point = int * int

type 'a image = {
  contenu_pixels : point -> 'a;
  largeur : int;
  hauteur : int
}
```

Exemple 5 — Images

Avec de plus la définition du type

```
type couleur = {rouge : int; vert : int; bleu : int}
```

toute valeur de type `couleur image` fait le travail.

Exemple 5 — Images

Avec de plus la définition du type

```
type couleur = {rouge : int; vert : int; bleu : int}
```

toute valeur de type `couleur image` fait le travail.

P.ex.,

```
# let im_1 = {
  contenus_pixels =
    (fun p ->
      let (x, y) = p in
      if x = y then
        {rouge = 127; vert = 127; bleu = 127}
      else
        {rouge = 0; vert = 0; bleu = 0}
    );
  largeur = 16;
  hauteur = 16
}
```

```
val im_1 : couleur image = {contenus_pixels = <fun>; largeur = 16; hauteur = 16}
```

Ceci lie au nom `im_1` une image de définition 16×16 dont les pixels sur la diagonale sont gris et les autres noirs.

Exemple 5 — Images

Voici un autre exemple :

```
# let im_2 = {
  contenus_pixels =
    (fun p ->
      let (x, y) = p in
      if (sqrt ((float_of_int x) ** 2.
                +. (float_of_int y) ** 2.)) <= 1024. then
        {rouge = 0; vert = 0; bleu = 0}
      else
        {rouge = 255; vert = 255; bleu = 255}
    );
  largeur = 1048576;
  hauteur = 1048576
}
```

```
val im_2 : couleur image = {contenus_pixels = <fun>; largeur = 1048576; hauteur = 1048576}
```

Ceci lie au nom `im_2` une image (de très haute définition) d'un disque noir sur un fond blanc.

Axe 3 : concepts avancés

Notions

Listes

λ -calcul

Plan

Notions

Récursivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

```
let rec ID P1 ... Pn = EXP
```

où `ID` est un identificateur, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

```
Error: Unbound value fact
```

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

```
let rec ID P1 ... Pn = EXP
```

où `ID` est un identificateur, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

```
# let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

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

Ceci définit bien la fonction factorielle.

```
# fact 7;;  
- : int = 5040
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
   x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
```

Warning 26: unused variable x.

```
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
   x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
```

Warning 26: unused variable x.

```
- : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
```

```
- : int = 30
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
Error: This kind of expression is
not allowed as right-hand side of
'let rec'
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
Error: Unbound value f
```

```
# let rec f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
```

```
# let rec f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
Error: Unbound value f
```

```
# let rec f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
```

```
# let rec f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# f 3;;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

Définitions récursives

```
# let f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
Error: Unbound value f
```

```
# let rec f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
Stack overflow during evaluation
(looping recursion?).
```

```
# let rec f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
```

Définitions récursives

```
# let f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
Error: Unbound value f
```

```
# let rec f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
Stack overflow during evaluation
(looping recursion?).
```

```
# let rec f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
Stack overflow during evaluation
(looping recursion?).
```

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
  if x = 0 then  
    "zero"  
  else  
    un (x - 1)  
and un x =  
  if x = 0 then  
    "un"  
  else  
    deux (x - 1)  
and deux x =  
  if x = 0 then  
    "deux"  
  else  
    zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    un (x - 1)
and un x =
  if x = 0 then
    "un"
  else
    deux (x - 1)
and deux x =
  if x = 0 then
    "deux"
  else
    zero (x - 1);;
```

```
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
    if x = 0 then  
        "zero"  
    else  
        un (x - 1)  
and un x =  
    if x = 0 then  
        "un"  
    else  
        deux (x - 1)  
and deux x =  
    if x = 0 then  
        "deux"  
    else  
        zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    un (x - 1)
and un x =
  if x = 0 then
    "un"
  else
    deux (x - 1)
and deux x =
  if x = 0 then
    "deux"
  else
    zero (x - 1);;
```

```
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

```
(zero 4)
```

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    un (x - 1)
and un x =
  if x = 0 then
    "un"
  else
    deux (x - 1)
and deux x =
  if x = 0 then
    "deux"
  else
    zero (x - 1);;
```

```
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

`(zero 4)` → `(un 3)`

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
  if x = 0 then  
    "zero"  
  else  
    un (x - 1)  
and un x =  
  if x = 0 then  
    "un"  
  else  
    deux (x - 1)  
and deux x =  
  if x = 0 then  
    "deux"  
  else  
    zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

`(zero 4)` → `(un 3)` → `(deux 2)`

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
  if x = 0 then  
    "zero"  
  else  
    un (x - 1)  
and un x =  
  if x = 0 then  
    "un"  
  else  
    deux (x - 1)  
and deux x =  
  if x = 0 then  
    "deux"  
  else  
    zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

```
(zero 4) → (un 3) → (deux 2)  
          → (zero 1)
```

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
  if x = 0 then  
    "zero"  
  else  
    un (x - 1)  
and un x =  
  if x = 0 then  
    "un"  
  else  
    deux (x - 1)  
and deux x =  
  if x = 0 then  
    "deux"  
  else  
    zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

```
(zero 4) → (un 3) → (deux 2)  
          → (zero 1) → (un 0)
```

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =  
  if x = 0 then  
    "zero"  
  else  
    un (x - 1)  
and un x =  
  if x = 0 then  
    "un"  
  else  
    deux (x - 1)  
and deux x =  
  if x = 0 then  
    "deux"  
  else  
    zero (x - 1);;
```

```
val zero : int -> string = <fun>  
val un : int -> string = <fun>  
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

`(zero 4)` → `(un 3)` → `(deux 2)`

→ `(zero 1)` → `(un 0)`

→ `"un"`.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

Ici, **C** est une condition et **I** est une expression.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que C :  
  /  
Fin
```

Ici, **C** est une condition et **/** est une expression.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que C :  
  /  
Fin
```

```
Fonction rec f :  
  Si C :  
    /  
    Appel à f  
  Fin  
Fin  
Appel à f
```

Ici, *C* est une condition et *I* est une expression.

Simulation des instructions de boucle — exemples

Boucle `while` simple :

Fonction C

```
int triangle(int n) {
  int i, res;
  res = 0;
  i = n;
  while (i >= 1) {
    res += i;
    i -= 1;
  }
  return res;
}
```

Fonction CAML

```
let triangle n =
  let rec aux i =
    if i >= 1 then
      i + (aux (i - 1))
    else
      0
  in
  aux n
```

Simulation des instructions de boucle — exemples

Boucle `while` **simple** :

Fonction C

```
int triangle(int n) {
  int i, res;
  res = 0;
  i = n;
  while (i >= 1) {
    res += i;
    i -= 1;
  }
  return res;
}
```

Fonction CAML

```
let triangle n =
  let rec aux i =
    if i >= 1 then
      i + (aux (i - 1))
    else
      0
  in
  aux n
```

Boucle `for` **simple** :

Fonction C

```
int somme_paires(int n) {
  int i, res;
  res = 0;
  for (i = 0 ; i <= n ; i += 2) {
    res += i;
  }
  return res;
}
```

Fonction CAML

```
let somme_paires n =
  let rec aux i =
    if i <= n then
      i + (aux (i + 2))
    else
      0
  in
  aux 0
```

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

```
fact 4 .
```

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

```
fact 4 .
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

```
fact 4 .
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

```
fact 4
```

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

```
fact 4 .
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
fact 4
```

→

```
4 * (fact 3)
```

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

```
fact 4 .
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
fact 4
```

→

```
4 * (fact 3)
```

→

```
4 * 3 * (fact 2)
```

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

`fact 4` .

Cette dernière expression s'**évalue** au fil des appels récursifs en

`fact 4` → `4 * (fact 3)` → `4 * 3 * (fact 2)`
→ `4 * 3 * 2 * (fact 1)`

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

`fact 4` .

Cette dernière expression s'*évalue* au fil des appels récursifs en

`fact 4` → `4 * (fact 3)` → `4 * 3 * (fact 2)`
→ `4 * 3 * 2 * (fact 1)` → `4 * 3 * 2 * 1`

Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

`fact 4` .

Cette dernière expression s'évalue au fil des appels récurtifs en

`fact 4` → `4 * (fact 3)` → `4 * 3 * (fact 2)`
→ `4 * 3 * 2 * (fact 1)` → `4 * 3 * 2 * 1`
↪ `24`

Réversivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et l'appel

`fact 4` .

Cette dernière expression s'**évalue** au fil des appels récursifs en

```
fact 4 → 4 * (fact 3) → 4 * 3 * (fact 2)  
→ 4 * 3 * 2 * (fact 1) → 4 * 3 * 2 * 1  
↪ 24
```

Ce calcul, pour être mené à bien, a dû **garder en mémoire l'expression**

```
4 * 3 * 2 * 1
```

qui fait intervenir quatre (**= n**) opérandes et trois (**= n - 1**) opérateurs.

Réversivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récursifs en

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
fact 4 1
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en

```
fact 4 1 → fact 3 (4 * 1)
```

Réversivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
fact 4 1 → fact 3 (4 * 1) ∼ fact 3 4
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en

```
fact 4 1 → fact 3 (4 * 1)  ~→ fact 3 4  
          → fact 2 (3 * 4)
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en

```
fact 4 1 → fact 3 (4 * 1) ~→ fact 3 4  
→ fact 2 (3 * 4) ~→ fact 2 12
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en

```
fact 4 1 → fact 3 (4 * 1) ~→ fact 3 4  
→ fact 2 (3 * 4) ~→ fact 2 12  
→ fact 1 (2 * 12)
```

Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en

```
fact 4 1 → fact 3 (4 * 1) ~→ fact 3 4  
→ fact 2 (3 * 4) ~→ fact 2 12  
→ fact 1 (2 * 12) ~→ fact 1 24
```

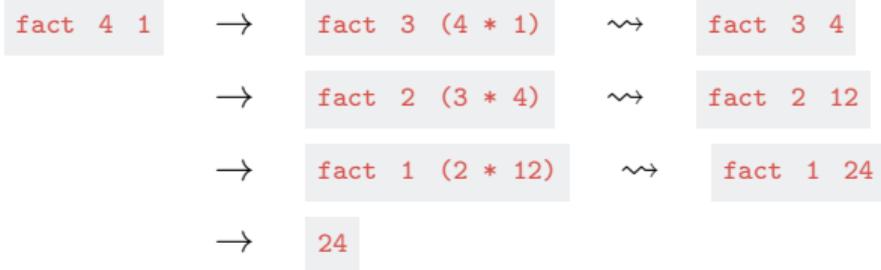
Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en



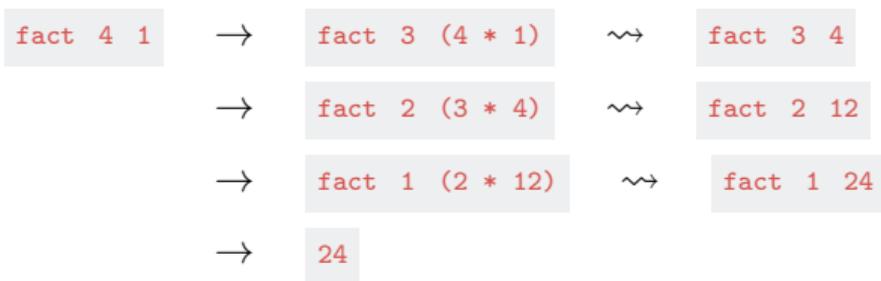
Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    fact (n - 1) (n * acc)
```

Considérons la fonction ci-contre et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récurtifs en



Ce calcul, pour être mené à bien, a dû **garder en mémoire des expressions** faisant intervenir au plus deux opérandes et un opérateur, en plus de l'appel de fonction.

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

Réversité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

Réversité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
fact (n - 1) (n * acc) .
```

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
fact (n - 1) (n * acc) .
```

Le calcul de la 1^{re} version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2^e ne travaille que sur une expression de taille $\Theta(1)$.

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel réursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
fact (n - 1) (n * acc) .
```

Le calcul de la 1^{re} version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2^e ne travaille que sur une expression de taille $\Theta(1)$.

Les fonctions réursives terminales utilisent **moins de mémoire** que leurs analogues non réursives terminales. Elles sont donc à préférer.

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

devient

```
let fact n =  
  let rec aux n acc =  
    if n <= 1 then  
      acc  
    else  
      (aux (n - 1) (n * acc))  
  in  
  (aux n 1)
```

Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

C'est la version non récursive terminale de la fonction calculant le n^e nombre de Fibonacci.

Réversivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

C'est la version non récursive terminale de la fonction calculant le n^e nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      aux (n - 1) (acc1 + acc2) acc1  
  in  
  aux n 1 0
```

C'est la version non récursive terminale de la fonction calculant le n^{e} nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version récursive terminale de la fonction précédente.

Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      aux (n - 1) (acc1 + acc2) acc1  
  in  
  aux n 1 0
```

C'est la version non récursive terminale de la fonction calculant le n^{e} nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version récursive terminale de la fonction précédente.

Elle utilise deux accumulateurs (à cause du double appel récursif précédent). `acc1` contient la valeur du $n - 1^{\text{e}}$ nombre de Fibonacci et `acc2` contient la valeur du $n - 2^{\text{e}}$ nombre de Fibonacci.

Récurtivité terminale — forme générale

Une fonction récursive terminale a pour **forme générale**

Fonction $\text{rec } f(x_1, \dots, x_n, \text{acc}_1, \dots, \text{acc}_m) :$

Si $C :$

$f(\text{maje}(x_1, \dots, x_n), \text{maj}_s(\text{acc}_1, \dots, \text{acc}_m))$

Sinon :

R

Fin

Fin

où

- ▶ x_1, \dots, x_n sont les paramètres (entrées);
- ▶ $\text{acc}_1, \dots, \text{acc}_m$ sont les accumulateurs (sorties);
- ▶ C est une expression booléenne dépendant des entrées et / ou des sorties;
- ▶ R est une expression résultat obtenue à partir des accumulateurs;
- ▶ maje indique la mise à jour des x_1, \dots, x_n lors de l'appel récursif;
- ▶ maj_s indique la mise à jour des $\text{acc}_1, \dots, \text{acc}_m$ lors de l'appel récursif.

Récurtivité terminale — dérécursivation

La **dérécursivation** est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Les notations sont ici les mêmes que celles utilisées précédemment.

Récurtivité terminale – dérécursivation

La **dérécursivation** est un procédé qui permet de transformer toute fonction réursive terminale en une fonction itérative.

Voici une fonction réursive terminale dans sa forme générale et sa version dérécursivée :

```
Fonction rec  $f(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m)$  :  
  Si  $C$  :  
     $f(\text{maje}(x_1, \dots, x_n),$   
       $\text{majs}(acc_1, \dots, acc_m))$   
  Sinon :  
     $R$   
Fin  
Fin
```

```
Fonction it  $g(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m)$  :  
  Tant que  $C$  :  
     $(x_1, \dots, x_n)$   
       $:= \text{maje}(x_1, \dots, x_n)$   
     $(acc_1, \dots, acc_m)$   
       $:= \text{majs}(acc_1, \dots, acc_m)$   
  Fin  
   $R$   
Fin
```

Les notations sont ici les mêmes que celles utilisées précédemment.

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      aux (n - 1) (acc1 + acc2) acc1  
  in  
  aux n 1 0
```

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      aux (n - 1) (acc1 + acc2) acc1  
  in  
  aux n 1 0
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    fibo (n - 1) (acc1 + acc2) acc1  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1
```

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      aux (n - 1) (acc1 + acc2) acc1  
  in  
  aux n 1 0
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    fibo (n - 1) (acc1 + acc2) acc1  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1
```

Version dérécursivée en C :

```
int fibo(int n, int acc1, int acc2) {  
  while (n >= 2) {  
    n = n - 1;  
    acc1 = acc1 + acc2;  
    acc2 = acc1 - acc2;  
  }  
  if (n == 0) return acc2;  
  else return acc1;  
}
```

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |Droite x -> Droite (-x)  
  |Plan (x, y) -> Plan (-x, -y)  
  |Espace (x, y, z) -> Espace (-x, -y, -z)
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |Droite x -> Droite (-x)  
  |Plan (x, y) -> Plan (-x, -y)  
  |Espace (x, y, z) -> Espace (-x, -y, -z)  
  
# oppose (Plan (3,1));;  
- : point = Plan (-3, -1)
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |Droite x -> Droite (-x)  
  |Plan (x, y) -> Plan (-x, -y)  
  |Espace (x, y, z) -> Espace (-x, -y, -z)
```

```
# oppose (Plan (3,1));;  
- : point = Plan (-3, -1)
```

```
# oppose (Espace(1, 0, -1));;  
- : point = Espace (-1, 0, 1)
```

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
```

```
|MOTIF1 -> EXP1
```

```
...
```

```
|MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
```

```
|MOTIF1 -> EXP1
```

```
...
```

```
|MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

Syntaxe et évaluation

La construction syntaxique

```
match EXP with  
  |MOTIF1 -> EXP1  
  ...  
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. `EXP` est évaluée;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. `EXP` est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. `EXP` est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas;
3. si un motif `MOTIFi` filtre la valeur de `EXP`, la valeur de toute l'expression est celle de `EXPi` ;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. `EXP` est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas;
3. si un motif `MOTIFi` filtre la valeur de `EXP`, la valeur de toute l'expression est celle de `EXPi` ;
4. si aucun motif ne filtre la valeur de `EXP`, une erreur est signalée (à l'exécution).

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =  
  match p with  
  |Droite x -> 1  
  |Plan (x, y) -> 2  
  |Espace (x, y, z) -> 3
```

renvoie le nombre de coordonnées du point

`p`.

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =  
  match p with  
    |Droite x -> 1  
    |Plan (x, y) -> 2  
    |Espace (x, y, z) -> 3
```

renvoie le nombre de coordonnées du point

`p`.

2. lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =  
  match p with  
    |Droite x -> 1  
    |Plan (x, y) -> 2  
    |Espace (x, y, z) -> 3
```

renvoie le nombre de coordonnées du point

`p`.

2. lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

```
let projection_x p =  
  match p with  
    |Droite x -> x  
    |Plan (x, y) -> x  
    |Espace (x, y, z) -> x
```

renvoie la première coordonnée du point `p`.

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |Droite x -> 1  
    |Espace (x, y, z) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |Droite x -> 1  
    |Espace (x, y, z) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |Droite x -> 1  
    |Espace (x, y, z) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

```
# let entier_vers_chaine n =  
  match n with  
    |0 -> "zero"  
    |1 -> "un"  
    |2 -> "deux"  
    |_ -> "autre";;  
val entier_vers_chaine : int -> string = <fun>
```

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

```
MOTIF -> EXP .
```

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

```
MOTIF -> EXP .
```

On utilise pour cela la syntaxe

```
MOTIF when TEST -> EXP
```

où **TEST** est une expression de type **bool** appelée **garde**.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

```
MOTIF -> EXP .
```

On utilise pour cela la syntaxe

```
MOTIF when TEST -> EXP
```

où `TEST` est une expression de type `bool` appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de `TEST` soit `true`.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

```
MOTIF -> EXP .
```

On utilise pour cela la syntaxe

```
MOTIF when TEST -> EXP
```

où **TEST** est une expression de type **bool** appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de **TEST** soit **true**.

P.ex.,

```
let est_dans_quart_de_plan p =  
  match p with  
  |Droite _ -> false  
  |Plan (x, y) when x >= 0 && y >= 0 -> true  
  |Plan (_, _) -> false  
  |Espace (_, _, _) -> false
```

teste si l'argument est un point du plan à coordonnées positives.

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0` , `true` , `()` , `'a'` , `"abc"` , *etc.*);

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0` , `true` , `()` , `'a'` , `"abc"` , *etc.*);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0` , `true` , `()` , `'a'` , `"abc"` , *etc.*);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |Droite x -> x  
  |Plan (x, y) -> x + y  
  |Espace (x, y, z) -> x + y + z
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |Droite x -> x  
  |Plan (x, y) -> x + y  
  |Espace (x, y, z) -> x + y + z
```

Attention : ici l'occurrence de `n` dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |Droite x -> x  
  |Plan (x, y) -> x + y  
  |Espace (x, y, z) -> x + y + z
```

Attention : ici l'occurrence de `n` dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1  
# f 0;;  
- : int = 0
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |Droite x -> x  
  |Plan (x, y) -> x + y  
  |Espace (x, y, z) -> x + y + z
```

Attention : ici l'occurrence de `n` dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1  
# f 0;;  
- : int = 0  
# f 3;;  
- : int = -1
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |Droite x -> x  
  |Plan (x, y) -> x + y  
  |Espace (x, y, z) -> x + y + z
```

Attention : ici l'occurrence de `n` dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1
```

```
# f 0;;  
- : int = 0  
# f 3;;  
- : int = -1
```

Ainsi, le motif `n` filtre toutes les valeurs.

Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une formule est une donnée récursive :

1. c'est un atome P ;
2. ou bien est la négation d'une formule ($\neg F$);
3. ou bien est la conjonction de deux formules ($F \wedge G$);
4. ou bien est la disjonction de deux formules ($F \vee G$).

Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une formule est une donnée récursive :

1. c'est un atome P ;
2. ou bien est la négation d'une formule ($\neg F$);
3. ou bien est la conjonction de deux formules ($F \wedge G$);
4. ou bien est la disjonction de deux formules ($F \vee G$).

On en déduit la définition de type (somme à paramètres et récursive) suivante :

```
type formule =  
  |Atome of char  
  |Non of formule  
  |Et of formule * formule  
  |Ou of formule * formule
```

Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool
```

Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool
```

La fonction d'**évaluation** d'une formule sous une valuation s'écrit très simplement au moyen d'un filtrage :

Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool
```

La fonction d'**évaluation** d'une formule sous une valuation s'écrit très simplement au moyen d'un filtrage :

```
let rec evaluer form valu =  
  match form with  
  |Atome c -> valu c  
  |Non f -> not (evaluer f valu)  
  |Et (f, g) -> (evaluer f valu) && (evaluer g valu)  
  |Ou (f, g) -> (evaluer f valu) || (evaluer g valu)
```

Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation v

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation v

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela, f est codée par

```
let f =  
  Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R')))
```

Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation v

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela, f est codée par

```
let f =  
  Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R')))
```

et v par

```
let v c =  
  match c with  
  | 'P' -> false  
  | 'R' -> true  
  | _ -> false
```

Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation v

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela, f est codée par

```
let f =  
  Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R')))
```

et v par

```
let v c =  
  match c with  
  | 'P' -> false  
  | 'R' -> true  
  | _ -> false
```

L'évaluation de f sous v donne

```
# evaluer f v;;  
- : bool = true
```

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Définition

Une **fonction d'ordre supérieur** est une fonction f qui vérifie au moins l'une des deux conditions suivantes :

1. f possède un paramètre de type fonction;
2. f renvoie une fonction.

Définition

Une **fonction d'ordre supérieur** est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un paramètre de type fonction;
2. `f` renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Définition

Une **fonction d'ordre supérieur** est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un paramètre de type fonction;
2. `f` renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Le fait de pouvoir renvoyer une fonction est un procédé très puissant en programmation fonctionnelle. Le programmeur n'est plus le seul concepteur de fonctions : l'exécution/l'interprétation peut en créer à la volée et en appeler.

Fonctions curryfiées

Rappelons que si f est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec $1 \leq k \leq n - 1$ produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

Fonctions curryfiées

Rappelons que si f est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec $1 \leq k \leq n - 1$ produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont **curryfiées** en CAML : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions** (donc de type

$$E_1 \rightarrow (E_2 \rightarrow \dots \rightarrow E_n \rightarrow S)).$$

Fonctions curryfiées

Rappelons que si f est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec $1 \leq k \leq n - 1$ produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont **curryfiées** en CAML : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions** (donc de type

$$E_1 \rightarrow (E_2 \rightarrow \dots \rightarrow E_n \rightarrow S)).$$

On peut donc voir toute fonction à deux paramètres ou plus comme une fonction d'ordre supérieur car son application partielle renvoie une fonction.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w)
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
    (fun v -> u ^ v ^ w)
```

On infère le type

```
string -> string -> string -> string .
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w)
```

On infère le type

```
string -> string -> string -> string .
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string) ,
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w)
```

On infère le type

```
string -> string -> string -> string .
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string) ,
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w)
```

On infère le type

```
string -> string -> string -> string .
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string) ,
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = encadrer "aa" "bb";;  
val f : string -> string = <fun>
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w)
```

On infère le type

```
string -> string -> string -> string .
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string) ,
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = encadrer "aa" "bb";;  
val f : string -> string = <fun>  
# f "bab";;  
- : string = "aababbb"
```

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    f (appli_repetee f x (n - 1))
```

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    f (appli_repetee f x (n - 1))
```

On infère le type

```
(int -> int) -> int -> int -> int ,
```

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    f (appli_repetee f x (n - 1))
```

On infère le type

```
(int -> int) -> int -> int -> int ,
```

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée n^e de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(f \circ \dots \circ f)}_{n \text{ fois}}(x)$$

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    f (appli_repetee f x (n - 1))
```

On infère le type

```
(int -> int) -> int -> int -> int ,
```

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée n^e de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(f \circ \dots \circ f)}_{n \text{ fois}}(x)$$

```
# appli_repetee (fun x -> x + 1) 3 4;; - : int = 7
```

```
# appli_repetee (fun x -> 2 * x) 3 4;; - : int = 48
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =  
  fun x -> 0 + f (0 + (g (f (g x))))
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =  
  fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

```
(int -> int) -> (int -> int) -> int -> int ,
```

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =  
  fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

```
(int -> int) -> (int -> int) -> int -> int ,
```

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

```
f (g (f (g (x)))).
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =  
  fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

```
(int -> int) -> (int -> int) -> int -> int ,
```

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

```
f (g (f (g (x)))).
```

```
# let h = entrelacer (fun x -> x * 2) (fun x -> x + 1);;  
val h : int -> int = <fun>
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =  
  fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

```
(int -> int) -> (int -> int) -> int -> int ,
```

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

```
f (g (f (g (x)))).
```

```
# let h = entrelacer (fun x -> x * 2) (fun x -> x + 1);;  
val h : int -> int = <fun>  
# h 3;;  
- : int = 18
```

Exemple complet 1 : mots fonctionnels

Reprenons la façon fonctionnelle de représenter les mots vue précédemment :

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

On rappelle que si `u` est un mot, l'expression `u.lettres i` a pour valeur la `i`^e lettre de `u`.

Exemple complet 1 : mots fonctionnels

Reprenons la façon fonctionnelle de représenter les mots vue précédemment :

```
type 'a mot = {  
  lettres : int -> 'a;  
  longueur : int  
}
```

On rappelle que si `u` est un mot, l'expression `u.lettres i` a pour valeur la `i`^e lettre de `u`.

Par exemple,

```
let mot_3 = {  
  lettres =  
    (fun i ->  
      match i with  
      | 1 -> 'a'  
      | 2 -> 'b'  
      | _ -> 'b'  
    );  
  longueur = 3  
}
```

représente le mot (de lettres de type `char`) `abb`.

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
  in  
  aux 1
```

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
    in  
    aux 1
```

Quelques explications :

- ▶ Cette fonction est de type `'a mot -> ('a -> char) -> string ;`

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
    in  
  aux 1
```

Quelques explications :

- ▶ Cette fonction est de type `'a mot -> ('a -> char) -> string` ;
- ▶ `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char` ;

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
    in  
  aux 1
```

Quelques explications :

- ▶ Cette fonction est de type `'a mot -> ('a -> char) -> string` ;
- ▶ `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char` ;
- ▶ l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
    in  
    aux 1
```

Quelques explications :

- ▶ Cette fonction est de type `'a mot -> ('a -> char) -> string` ;
- ▶ `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char` ;
- ▶ l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

P.ex.,

```
# string_of_mot mot_3 (fun x -> x);;  
- : string = "abb"
```

Exemple complet 1 : mots fonctionnels

Voici une fonction qui **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =  
  let rec aux i =  
    if i = u.longueur + 1 then  
      ""  
    else  
      let ch = String.make 1 (lettre_vers_char (u.lettres i)) in  
      ch ^ (aux (i + 1))  
    in  
  aux 1
```

Quelques explications :

- ▶ Cette fonction est de type `'a mot -> ('a -> char) -> string` ;
- ▶ `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char` ;
- ▶ l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

P.ex.,

```
# string_of_mot mot_3 (fun x -> x);;  
- : string = "abb"  
  
# string_of_mot {lettres = (fun i -> true); longueur = 4}  
  (fun b -> if b then "T" else "F");;  
- : string = "TTT"
```

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de **concaténation** de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =  
  let lettres i =  
    if i <= u.longueur then  
      u.lettres i  
    else  
      v.lettres (i - u.longueur)  
  in  
  {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de **concaténation** de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =  
  let lettres i =  
    if i <= u.longueur then  
      u.lettres i  
    else  
      v.lettres (i - u.longueur)  
  in  
  {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- ▶ son type est `'a mot -> 'a mot -> 'a mot ;`

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de **concaténation** de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =  
  let lettres i =  
    if i <= u.longueur then  
      u.lettres i  
    else  
      v.lettres (i - u.longueur)  
  in  
  {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- ▶ son type est `'a mot -> 'a mot -> 'a mot ;` ;
- ▶ pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leq i \leq |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de **concaténation** de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =  
  let lettres i =  
    if i <= u.longueur then  
      u.lettres i  
    else  
      v.lettres (i - u.longueur)  
  in  
  {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- ▶ son type est `'a mot -> 'a mot -> 'a mot ;` ;
- ▶ pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leq i \leq |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

P.ex.,

```
# let mot_4 = concatener mot_3 mot_3 in  
string_of_mot mot_4 (fun x -> x);;  
- : string = "abbabb"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))

# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))

# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))

# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))

# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geq 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
  if n = 0 then
    {lettres = (fun i -> 'b'); longueur = 1}
  else if n = 1 then
    {lettres = (fun i -> 'a'); longueur = 1}
  else
    concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))

# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
# let w = mot_fibo 5 in string_of_mot w (fun x -> x);;
- : string = "abaababa"
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
  contenus_pixels : point -> 'a;
  largeur : int;
  hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
  contenus_pixels : point -> 'a;
  largeur : int;
  hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
  contenus_pixels =
    (fun p ->
      let (x, y) = p in
      if x = y then
        {rouge = 127; vert = 127;
         bleu = 127}
      else
        {rouge = 255; vert = 255;
         bleu = 255}
    );
  largeur = 24;
  hauteur = 24
}
```

Exemple complet 2 : images fonctionnelles

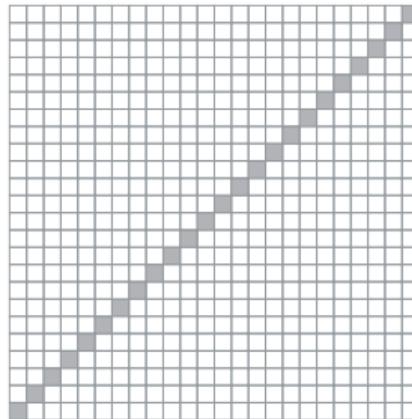
Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
  contenus_pixels : point -> 'a;
  largeur : int;
  hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
  contenus_pixels =
    (fun p ->
      let (x, y) = p in
      if x = y then
        {rouge = 127; vert = 127;
         bleu = 127}
      else
        {rouge = 255; vert = 255;
         bleu = 255}
    );
  largeur = 24;
  hauteur = 24
}
```

représente l'image



Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en **complémentant les couleurs** des pixels d'une image en entrée :

```
let complements im =  
  let contenus_pixels p =  
    let (x, y) = p in  
    {rouge = 255 - (im.contenus_pixels p).rouge;  
     vert = 255 - (im.contenus_pixels p).vert;  
     bleu = 255 - (im.contenus_pixels p).bleu}  
  in  
  {contenus_pixels = contenus_pixels;  
   largeur = im.largeur;  
   hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en **complémentant les couleurs** des pixels d'une image en entrée :

```
let complements im =  
  let contenus_pixels p =  
    let (x, y) = p in  
    {rouge = 255 - (im.contenus_pixels p).rouge;  
     vert = 255 - (im.contenus_pixels p).vert;  
     bleu = 255 - (im.contenus_pixels p).bleu}  
  in  
  {contenus_pixels = contenus_pixels;  
   largeur = im.largeur;  
   hauteur = im.hauteur}
```

Voici une fonction qui calcule l'**image miroir** d'une image en entrée :

```
let miroir im =  
  let contenus_pixels p =  
    let (x, y) = p in  
    im.contenus_pixels (im.largeur - x + 1, y)  
  in  
  {contenus_pixels = contenus_pixels;  
   largeur = im.largeur;  
   hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en **complémentant les couleurs** des pixels d'une image en entrée :

```
let complements im =
  let contenus_pixels p =
    let (x, y) = p in
    {rouge = 255 - (im.contenus_pixels p).rouge;
     vert = 255 - (im.contenus_pixels p).vert;
     bleu = 255 - (im.contenus_pixels p).bleu}
  in
  {contenus_pixels = contenus_pixels;
   largeur = im.largeur;
   hauteur = im.hauteur}
```

Voici une fonction qui calcule l'**image miroir** d'une image en entrée :

```
let miroir im =
  let contenus_pixels p =
    let (x, y) = p in
    im.contenus_pixels (im.largeur - x + 1, y)
  in
  {contenus_pixels = contenus_pixels;
   largeur = im.largeur;
   hauteur = im.hauteur}
```

Toutes ces fonctions s'évaluent en temps $\Theta(1)$. Cette complexité ne dépend donc pas de la taille de l'image!

Exemple complet 3 : séries génératrices

On souhaite représenter des *séries génératrices*. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers.

Exemple complet 3 : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les α_i sont des entiers.

Exemple complet 3 : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

Exemple complet 3 : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geq 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \dots$$

Exemple complet 3 : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geq 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \dots$$

Question : comment représenter des séries génératrices ?

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n .

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n .

C'est le type

```
type serie_g = int -> int;;
```

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Par exemple, la série génératrice des puissances de 2 est ainsi codée par

```
let puissances_2 =  
  fun n -> (int_of_float (2. ** (float_of_int n)))
```

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) + \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) + \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) \cdot \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) + \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) \bullet \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

3. **produit** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) \cdot \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \sum_{i=0}^k \alpha_i \beta_{k-i} t^k .$$

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) + \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) \bullet \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

3. **produit** :

$$\left(\sum_{n \geq 0} \alpha_n t^n \right) \cdot \left(\sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \sum_{i=0}^k \alpha_i \beta_{k-i} t^k .$$

Il est possible de les implanter simplement en utilisant les fonctions d'ordre supérieur.

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k))
```

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k))
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res
```

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k))
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res  
  
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>
```

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k))
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res  
  
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>  
# s3 3;;  
- : int = 16
```

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k))
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res  
  
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>  
# s3 3;;  
- : int = 16
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =  
  (fun k -> (s1 k) * (s2 k))
```

Exemple complet 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =  
  let resultat k =  
    let rec aux i =  
      if i > k then  
        0  
      else  
        (aux (i + 1)) + (s1 i) * (s2 (k - i))  
    in  
    aux 0  
  in  
  resultat;;
```

Exemple complet 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    aux 0
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>
```

Exemple complet 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    aux 0
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = produit sg_un sg_un;;
val sg_un_carre : int -> int = <fun>
```

Exemple complet 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    aux 0
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = produit sg_un sg_un;;
val sg_un_carre : int -> int = <fun>

# (sg_un_carre 0), (sg_un_carre 1), (sg_un_carre 2), (sg_un_carre 3);;
- : int * int * int * int = (1, 2, 3, 4)
```

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Objets polymorphes

Un objet est dit **polymorphe** s'il n'est pas d'un type fixé.

Objets polymorphes

Un objet est dit **polymorphe** s'il n'est pas d'un type fixé.

- ▶ Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;  
val vrai : 'a -> bool = <fun>
```

Objets polymorphes

Un objet est dit **polymorphe** s'il n'est pas d'un type fixé.

- ▶ Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;  
val vrai : 'a -> bool = <fun>
```

- ▶ Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;  
type 'a liste = Vide | Cellule of 'a * 'a liste
```

Objets polymorphes

Un objet est dit **polymorphe** s'il n'est pas d'un type fixé.

- ▶ Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;  
val vrai : 'a -> bool = <fun>
```

- ▶ Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;  
type 'a liste = Vide | Cellule of 'a * 'a liste
```

- ▶ Une **valeur polymorphe** est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé. P.ex.,

```
# Vide;;  
- : 'a liste = Vide
```

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type **t** bien défini, soit de tous les types possibles **'a** (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre **x** d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de **x** dans la fonction pour tenter de déterminer son type en fonction de son utilisation;

Polymorphisme paramétrique

En CAML, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type t bien défini, soit de tous les types possibles $'a$ (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre x d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de x dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de x), alors x est du type le plus général $'a$.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`(=) : 'a -> 'a -> bool`

Test d'égalité entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`(=) : 'a -> 'a -> bool`

Test d'égalité entre deux valeurs d'un même type.

`(<>) : 'a -> 'a -> bool`

Test de différence entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`(=) : 'a -> 'a -> bool`

Test d'égalité entre deux valeurs d'un même type.

`(<>) : 'a -> 'a -> bool`

Test de différence entre deux valeurs d'un même type.

`compare : 'a -> 'a -> int`

Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1^{re} est strict. inf. (resp. sup.) à la 2^e et `0` sinon.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`(=) : 'a -> 'a -> bool`

Test d'égalité entre deux valeurs d'un même type.

`(<>) : 'a -> 'a -> bool`

Test de différence entre deux valeurs d'un même type.

`compare : 'a -> 'a -> int`

Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1^{re} est strict. inf. (resp. sup.) à la 2^e et `0` sinon.

`fst : 'a * 'b -> 'a`

Renvoie la 1^{re} coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`(=) : 'a -> 'a -> bool`

Test d'égalité entre deux valeurs d'un même type.

`(<>) : 'a -> 'a -> bool`

Test de différence entre deux valeurs d'un même type.

`compare : 'a -> 'a -> int`

Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1^{re} est strict. inf. (resp. sup.) à la 2^e et `0` sinon.

`fst : 'a * 'b -> 'a`

Renvoie la 1^{re} coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

`snd : 'a * 'b -> 'b`

Renvoie la 2^e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Exemple : exponentiation rapide

Le calcul de x^n , où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Exemple : exponentiation rapide

Le calcul de x^n , où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
  if n = 0 then
    1
  else
    let tmp = (puiss x (n / 2)) in
    if n mod 2 = 0 then
      tmp * tmp
    else
      tmp * tmp * x
```

Exemple : exponentiation rapide

Le calcul de x^n , où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
  if n = 0 then
    1
  else
    let tmp = (puiss x (n / 2)) in
    if n mod 2 = 0 then
      tmp * tmp
    else
      tmp * tmp * x
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;
- ▶ le 2^e paramètre de type `'e` est l'unité **1** ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;
- ▶ le 2^e paramètre de type `'e` est l'unité **1** ;
- ▶ le 3^e paramètre de type `'e` est l'élément `x` ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;
- ▶ le 2^e paramètre de type `'e` est l'unité **1** ;
- ▶ le 3^e paramètre de type `'e` est l'élément `x` ;
- ▶ le 4^e paramètre de type `int` est l'entier `n` ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;
- ▶ le 2^e paramètre de type `'e` est l'unité **1** ;
- ▶ le 3^e paramètre de type `'e` est l'élément `x` ;
- ▶ le 4^e paramètre de type `int` est l'entier `n` ;
- ▶ Le type de retour est `'e`.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- ▶ le 1^{er} paramètre de type `('e -> 'e -> 'e)` est une fonction codant \times ;
- ▶ le 2^e paramètre de type `'e` est l'unité **1** ;
- ▶ le 3^e paramètre de type `'e` est l'élément `x` ;
- ▶ le 4^e paramètre de type `int` est l'entier `n` ;
- ▶ Le type de retour est `'e`.

La valeur renvoyée est `xn`.

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =  
  if n = 0 then  
    unite  
  else  
    let tmp = (puiss_poly op unite x (n / 2)) in  
    if n mod 2 = 0 then  
      op tmp tmp  
    else  
      op (op tmp tmp) x
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =  
  if n = 0 then  
    unite  
  else  
    let tmp = (puiss_poly op unite x (n / 2)) in  
    if n mod 2 = 0 then  
      op tmp tmp  
    else  
      op (op tmp tmp) x
```

Exemples d'utilisation :

```
# puiss_poly (fun a b -> a + b) 0 1 6;;  
- : int = 6
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
  if n = 0 then
    unite
  else
    let tmp = (puiss_poly op unite x (n / 2)) in
    if n mod 2 = 0 then
      op tmp tmp
    else
      op (op tmp tmp) x
```

Exemples d'utilisation :

```
# puiss_poly (fun a b -> a + b) 0 1 6;;
- : int = 6
# puiss_poly (fun a b -> a * b) 1 2 10;;
- : int = 1024
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
  if n = 0 then
    unite
  else
    let tmp = (puiss_poly op unite x (n / 2)) in
    if n mod 2 = 0 then
      op tmp tmp
    else
      op (op tmp tmp) x
```

Exemples d'utilisation :

```
# puiss_poly (fun a b -> a + b) 0 1 6;;
- : int = 6
# puiss_poly (fun a b -> a * b) 1 2 10;;
- : int = 1024
# puiss_poly (fun u v -> u ^ v) "" "abb" 4;;
- : string = "abbabbabbabb"
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
  if n = 0 then
    unite
  else
    let tmp = (puiss_poly op unite x (n / 2)) in
    if n mod 2 = 0 then
      op tmp tmp
    else
      op (op tmp tmp) x
```

Exemples d'utilisation :

```
# puiss_poly (fun a b -> a + b) 0 1 6;;
- : int = 6
# puiss_poly (fun a b -> a * b) 1 2 10;;
- : int = 1024
# puiss_poly (fun u v -> u ^ v) "" "abb" 4;;
- : string = "abbabbabbabb"
# puiss_poly (fun a b -> a || b) false false 293898273;;
- : bool = false
```

Exemple : exponentiation rapide

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Exemple : exponentiation rapide

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Pour cela, on représente les monoïdes au moyen du type enregistrement

```
type 'a monoïde = {  
  op : 'a -> 'a -> 'a;  
  unite : 'a  
}
```

Exemple : exponentiation rapide

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Pour cela, on représente les monoïdes au moyen du type enregistrement

```
type 'a monoide = {  
  op : 'a -> 'a -> 'a;  
  unite : 'a  
}
```

La fonction précédente devient

```
let rec puiss_poly monoide x n =  
  if n = 0 then  
    monoide.unite  
  else  
    let tmp = (puiss_poly monoide x (n / 2)) in  
    if n mod 2 = 0 then  
      monoide.op tmp tmp  
    else  
      monoide.op (monoide.op tmp tmp) x
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2^e paramètre dans le 1^{er}.

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2^e paramètre dans le 1^{er}.

```
let rec appartient_liste lst x =  
  match lst with  
  |Vide -> false  
  |Cellule (y, _) when y = x -> true  
  |Cellule (_, reste) -> appartient_liste reste x
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2^e paramètre dans le 1^{er}.

```
let rec appartient_liste lst x =  
  match lst with  
  |Vide -> false  
  |Cellule (y, _) when y = x -> true  
  |Cellule (_, reste) -> appartient_liste reste x
```

Ceci se base sur le fait que test d'égalité `=` est polymorphe (de type `'a -> 'a -> 'a` ; il n'est donc pas à fournir en paramètre à la fonction).

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- ▶ le 1^{er} paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant l'opération « max » d'une relation d'ordre totale.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- ▶ le 1^{er} paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant l'opération « max » d'une relation d'ordre totale.
- ▶ Le 2^e paramètre est la liste générique dans laquelle la recherche est effectuée.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- ▶ le 1^{er} paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant l'opération « max » d'une relation d'ordre totale.
- ▶ Le 2^e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- ▶ le 1^{er} paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant l'opération « max » d'une relation d'ordre totale.
- ▶ Le 2^e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

```
let maximum_liste mmax lst =  
  let rec aux lst max_prefixe =  
    match lst with  
    | Vide -> max_prefixe  
    | Cellule (x, reste) -> aux reste (mmax max_prefixe x)  
  in  
  match lst with  
  | Vide -> failwith "liste vide"  
  | Cellule (x, reste) -> aux reste x
```

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :  
  Si x = 0 :  
    0  
  Sinon :  
    x + f(x - 1)  
Fin
```

```
Fin  
  
Fonction g(x, y) :  
  Si y est pair :  
    y  
  Sinon :
```

```
      x  
      Fin  
Fin  
Début :  
  g(f(-1), 0)  
Fin
```

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :  
  Si x = 0 :  
    0  
  Sinon :  
    x + f(x - 1)  
Fin
```

```
Fin  
  
Fonction g(x, y) :  
  Si y est pair :  
    y  
  Sinon :  
    x  
Fin
```

```
Fin  
Fin  
Début :  
  g(f(-1), 0)  
Fin
```

Réponse : tout dépend de la **stratégie d'évaluation** du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :  
  Si x = 0 :  
    0  
  Sinon :  
    x + f(x - 1)  
Fin
```

```
Fin  
x  
Fin  
Fin  
Fonction g(x, y) :  
  Si y est pair :  
    y  
  Sinon :  
    Début :  
      g(f(-1), 0)  
    Fin
```

Réponse : tout dépend de la **stratégie d'évaluation** du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :                               Fin
  Si x = 0 :                                       Fin
    0                                               Fin
  Sinon :                                          Fin
    x + f(x - 1)
  Fin
Fonction g(x, y) :
  Si y est pair :
    y
  Sinon :
    x
Fin
Début :
  g(f(-1), 0)
Fin
```

Réponse : tout dépend de la **stratégie d'évaluation** du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison;

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :                               Fin
  Si x = 0 :                                       Fin
    0                                               Fin
  Sinon :                                          Fin
    x + f(x - 1)
  Fin
Fonction g(x, y) :
  Si y est pair :
    y
  Sinon :
    x
```

Réponse : tout dépend de la **stratégie d'évaluation** du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution termine dans ce cas.

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

```
~> (f 1 (f 2 1 4) 12)
```

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

```
~> (f 1 (f 2 1 4) 12)
```

```
~> (f 1 6 12)
```

Appel par valeur

La stratégie d'évaluation en **appel par valeur** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =  
  x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

```
↪ (f 1 (f 2 1 4) 12)
```

```
↪ (f 1 6 12)
```

```
↪ 13
```

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent);

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent);
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent);
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont le CAML.

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction f à des expressions a_1, \dots, a_n , à **substituer** les a_i **sans les évaluer** aux occurrences des paramètres de f correspondants.

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =  
    x + z
```

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =  
    x + z
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =  
    x + z
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =  
    x + z
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

```
↪ (1 * 1) + (3 * 4)
```

Appel par nom

La stratégie d'évaluation en **appel par nom** consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =  
    x + z
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

```
↪ (1 * 1) + (3 * 4)
```

```
↪ 13
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

l'expression

```
(f (4 * 3) (2 * 1))
```

s'évalue en appel par nom au moyen des étapes suivantes :

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

l'expression

```
(f (4 * 3) (2 * 1))
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

l'expression

```
(f (4 * 3) (2 * 1))
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
```

```
↪ (4 * 3) * (4 * 3) + (2 * 1)
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

l'expression

```
(f (4 * 3) (2 * 1))
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
```

```
↪ (4 * 3) * (4 * 3) + (2 * 1)
```

```
↪ 146
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont copiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =  
    x * x + y
```

l'expression

```
(f (4 * 3) (2 * 1))
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3) (2 * 1))
```

```
↪ (4 * 3) * (4 * 3) + (2 * 1)
```

```
↪ 146
```

L'argument `(4 * 3)` est évalué ainsi deux fois (au lieu d'une seule que ferait un appel par valeur).

Appel par nécessité

La stratégie en **appel par nécessité** est une **version mémorisée de l'appel par nom**.

Une fonction est **mémorisée** si, à chaque premier appel pour un jeu d'arguments donnés, la valeur qu'elle renvoie est enregistrée dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas une réévaluation.

Appel par nécessité

La stratégie en **appel par nécessité** est une **version mémorisée de l'appel par nom**.

Une fonction est **mémorisée** si, à chaque premier appel pour un jeu d'arguments donnés, la valeur qu'elle renvoie est enregistrée dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas une réévaluation.

Lors de l'application d'une fonction f à des expressions a_1, \dots, a_n , chaque a_i n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

Appel par nécessité

La stratégie en **appel par nécessité** est une **version mémorisée de l'appel par nom**.

Une fonction est **mémorisée** si, à chaque premier appel pour un jeu d'arguments donnés, la valeur qu'elle renvoie est enregistrée dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas une réévaluation.

Lors de l'application d'une fonction **f** à des expressions **a1**, ..., **an**, chaque **ai** n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un **ai**, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de **f** correspondant à un **ai** ne redemande pas d'être réévaluée.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage HASKELL utilise cette stratégie.

Plan

Listes

- Opérations

- Non-mutabilité

- Files

Plan

Listes

- Opérations

- Non-mutabilité

- Files

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;
```

```
- : int list = [2; 4; 8; 16]
```

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;  
- : int list = [2; 4; 8; 16]
```

La **liste vide** est notée `[]`.

```
# [];;  
- : 'a list = []
```

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les **listes** (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, ..., `en`.

```
# [2 ; 4 ; 8 ; 16];;  
- : int list = [2; 4; 8; 16]
```

La **liste vide** est notée `[]`.

```
# [];;  
- : 'a list = []
```

C'est une valeur polymorphe.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixé d'arité deux.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixé d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1^{er} élément et ceux de `lst` ensuite.

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1^{er} élément et ceux de `lst` ensuite.

P.ex.,

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

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1^{er} élément et ceux de `lst` ensuite.

P.ex.,

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

Il est **associatif de droite à gauche**.

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

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1^{er} élément et ceux de `lst` ensuite.

P.ex.,

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

Il est **associatif de droite à gauche**.

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

L'expression `1 :: 2 :: 3 :: []` désigne en effet l'expression totalement parenthésée `(1 :: (2 :: (3 :: [])))`.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =  
  match lst with  
  | [] -> (failwith "liste vide")  
  | e :: _ -> e;;  
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1^{er} élément.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =  
  match lst with  
    | [] -> (failwith "liste vide")  
    | e :: _ -> e;;  
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1^{er} élément.

```
# let rec un_sur_deux lst =  
  match lst with  
    | [] -> []  
    | [e] -> [e]  
    | e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;  
val un_sur_deux : 'a list -> 'a list = <fun>
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =  
  match lst with  
  | [] -> (failwith "liste vide")  
  | e :: _ -> e;;  
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1^{er} élément.

```
# let rec un_sur_deux lst =  
  match lst with  
  | [] -> []  
  | [e] -> [e]  
  | e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;  
val un_sur_deux : 'a list -> 'a list = <fun>
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : ce sont des fonctions polymorphes.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|-----------------|-------------------------------|------------------|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|-----------------|------------------------------------|-------------------------|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|------------------|---|--|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |
| <code>nth</code> | <code>'a list -> int -> 'a</code> | L'élément de la liste à l'indice donné |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|---------------------|---|--|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |
| <code>nth</code> | <code>'a list -> int -> 'a</code> | L'élément de la liste à l'indice donné |
| <code>length</code> | <code>'a list -> int</code> | longueur de la liste |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|---------------------|--|--|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |
| <code>nth</code> | <code>'a list -> int -> 'a</code> | L'élément de la liste à l'indice donné |
| <code>length</code> | <code>'a list -> int</code> | longueur de la liste |
| <code>mem</code> | <code>'a -> 'a list -> bool</code> | Présence de l'élément dans la liste |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.
Parmi elles :

| Fonction | Type | Valeur renvoyée |
|---------------------|--|--|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |
| <code>nth</code> | <code>'a list -> int -> 'a</code> | L'élément de la liste à l'indice donné |
| <code>length</code> | <code>'a list -> int</code> | longueur de la liste |
| <code>mem</code> | <code>'a -> 'a list -> bool</code> | Présence de l'élément dans la liste |
| <code>rev</code> | <code>'a list -> 'a list</code> | Liste miroir |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La bibliothèque de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

| Fonction | Type | Valeur renvoyée |
|---------------------|--|--|
| <code>hd</code> | <code>'a list -> 'a</code> | Tête de la liste |
| <code>tl</code> | <code>'a list -> 'a list</code> | Liste privée de sa tête |
| <code>nth</code> | <code>'a list -> int -> 'a</code> | L'élément de la liste à l'indice donné |
| <code>length</code> | <code>'a list -> int</code> | longueur de la liste |
| <code>mem</code> | <code>'a -> 'a list -> bool</code> | Présence de l'élément dans la liste |
| <code>rev</code> | <code>'a list -> 'a list</code> | Liste miroir |
| <code>append</code> | <code>'a list -> 'a list -> 'a list</code> | Concaténation des deux listes |

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété;
4. **combiner** les éléments d'une liste pour calculer une valeur;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété;
4. **combiner** les éléments d'une liste pour calculer une valeur;
5. **permuter** les éléments d'une liste.

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants;

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants;
2. [**sélection**] obtenir la sous-liste des nombres pairs d'une liste d'entiers;

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants;
2. [**sélection**] obtenir la sous-liste des nombres pairs d'une liste d'entiers;
3. [**test**] tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers;

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants;
2. [**sélection**] obtenir la sous-liste des nombres pairs d'une liste d'entiers;
3. [**test**] tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers;
4. [**combinaison**] calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers;

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants;
2. [**sélection**] obtenir la sous-liste des nombres pairs d'une liste d'entiers;
3. [**test**] tester si toutes les chaînes de caractères contenues dans une liste commencent par 'a', tester la présence de l'élément 7 dans une liste d'entiers;
4. [**combinaison**] calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers;
5. [**permutation**] tri d'une liste, image miroir d'une liste.

Opérations habituelles sur les listes

Exemples :

1. [**transformation**] multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants;
2. [**sélection**] obtenir la sous-liste des nombres pairs d'une liste d'entiers;
3. [**test**] tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers;
4. [**combinaison**] calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers;
5. [**permutation**] tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de **fonctions d'ordre supérieur**.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
( 'a -> 'b ) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =  
  match lst with  
  | [] -> []  
  | e :: reste -> (tr e) :: (transformer tr reste);;
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =  
  match lst with  
  | [] -> []  
  | e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;  
- : int list = [3; 6; 9; 12; 15; 18]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =  
  match lst with  
  | [] -> []  
  | e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;  
- : int list = [3; 6; 9; 12; 15; 18]
```

```
# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;  
- : int list = [97; 98; 99; 49; 50; 51]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =  
  match lst with  
  | [] -> []  
  | e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;  
- : int list = [3; 6; 9; 12; 15; 18]
```

```
# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;  
- : int list = [97; 98; 99; 49; 50; 51]
```

Nous avons réimplanté ici la fonction `map` du module `List`.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner sel lst =  
  match lst with  
  | [] -> []  
  | e :: reste ->  
    let suite = (selectionner sel reste) in  
    if (sel e) then e :: suite  
    else suite;;
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner sel lst =  
  match lst with  
  | [] -> []  
  | e :: reste ->  
    let suite = (selectionner sel reste) in  
    if (sel e) then e :: suite  
    else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;  
- : int list = [8; 8; 6; 2]
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner sel lst =  
  match lst with  
  | [] -> []  
  | e :: reste ->  
    let suite = (selectionner sel reste) in  
    if (sel e) then e :: suite  
    else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;  
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `filter` du module `List`.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =  
  match lst with  
  | [] -> true  
  | x :: _ when (not (prop x)) -> false  
  | _ :: reste -> (tester_univ prop reste);;
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =  
  match lst with  
  | [] -> true  
  | x :: _ when (not (prop x)) -> false  
  | _ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a" ; "aba" ; "abacaba" ; "abacabadabacaba"]);;  
- : bool = true
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =  
  match lst with  
  | [] -> true  
  | x :: _ when (not (prop x)) -> false  
  | _ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a" ; "aba" ; "abacaba" ; "abacabadabacaba"]);;  
- : bool = true
```

Nous avons réimplanté ici la fonction `for_all` du module `List`.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist prop lst =  
  match lst with  
  | [] -> false  
  | x :: _ when (prop x) -> true  
  | _ :: reste -> (tester_exist prop reste);;
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist prop lst =  
  match lst with  
  | [] -> false  
  | x :: _ when (prop x) -> true  
  | _ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;  
- : bool = false
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist prop lst =  
  match lst with  
  | [] -> false  
  | x :: _ when (prop x) -> true  
  | _ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;  
- : bool = false
```

Nous avons réimplanté ici la fonction `exists` du module `List`.

Association des éléments d'une liste

Le problème est le suivant.

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément gr ;
2. d'une liste lst de la forme $[e1 ; e2 ; \dots ; en]$;
3. d'une opération binaire associative \times ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative `×` ;

et on souhaite calculer la valeur

$$\text{gr} \times e1 \times e2 \times \dots \times en .$$

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative `×` ;

et on souhaite calculer la valeur

$$gr \times e1 \times e2 \times \dots \times en .$$

Ce problème admet de nombreuses instances :

- ▶ lorsque `gr = 0`, `lst` est une liste d'entiers et `×` est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative `×` ;

et on souhaite calculer la valeur

$$gr \times e1 \times e2 \times \dots \times en .$$

Ce problème admet de nombreuses instances :

- ▶ lorsque `gr = 0`, `lst` est une liste d'entiers et `×` est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- ▶ lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et `×` est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative `×` ;

et on souhaite calculer la valeur

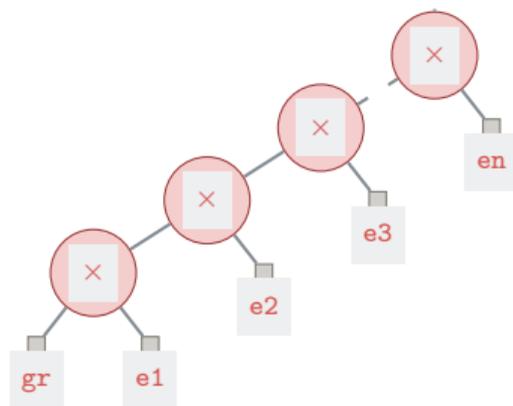
$$gr \times e1 \times e2 \times \dots \times en .$$

Ce problème admet de nombreuses instances :

- ▶ lorsque `gr = 0` , `lst` est une liste d'entiers et `×` est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- ▶ lorsque `gr = ""` , `lst` est une liste de chaînes de caractères et `×` est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;
- ▶ lorsque `gr = e1` , `lst` est une liste dont les éléments sont comparables et `×` est la fonction qui renvoie **le plus grand** de ses deux arguments, ceci calcule la plus grande valeur de `lst` .

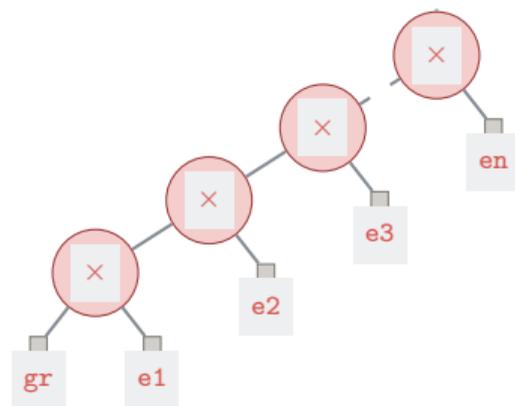
Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération `×`, on souhaite **évaluer** l'arbre syntaxique



Pliage à gauche

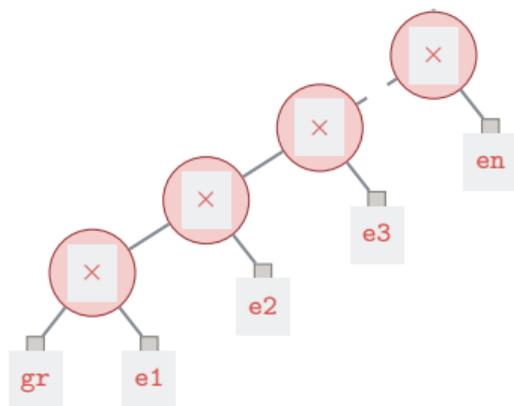
En d'autres termes, étant donnée une liste $[e_1 ; e_2 ; e_3 ; \dots ; e_n]$ et une opération \times , on souhaite **évaluer** l'arbre syntaxique



Ici, gr est un élément initial pour le calcul. On l'appelle **graine**.

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération `×`, on souhaite **évaluer** l'arbre syntaxique



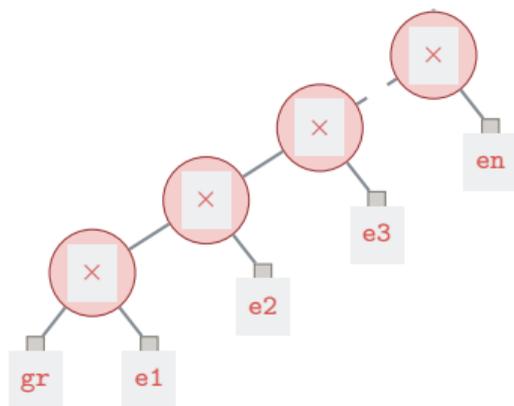
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée **pliage à gauche**, il est nécessaire de transmettre les informations suivantes :

1. l'opération `×` ;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération `×`, on souhaite **évaluer** l'arbre syntaxique



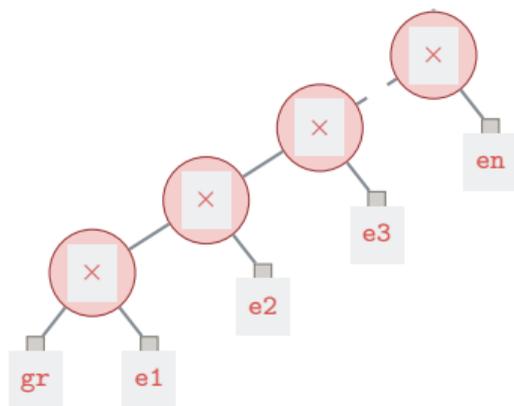
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée **pliage à gauche**, il est nécessaire de transmettre les informations suivantes :

1. l'opération `×` ;
2. la graine `gr` ;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération `×`, on souhaite **évaluer** l'arbre syntaxique



Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée **pliage à gauche**, il est nécessaire de transmettre les informations suivantes :

1. l'opération `×` ;
2. la graine `gr` ;
3. la liste `lst = [e1 ; e2 ; e3 ; ... ; en]` des opérandes.

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =  
  match lst with  
  | [] -> gr  
  | e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =  
  match lst with  
  | [] -> gr  
  | e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;  
- : int = 11
```

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =  
  match lst with  
  | [] -> gr  
  | e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;  
- : int = 11  
  
# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in  
  (pliage_gauche max (List.hd lst) (List.tl lst));;  
- : int = 5
```

Pliage à gauche

L'opération `x` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =  
  match lst with  
  | [] -> gr  
  | e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

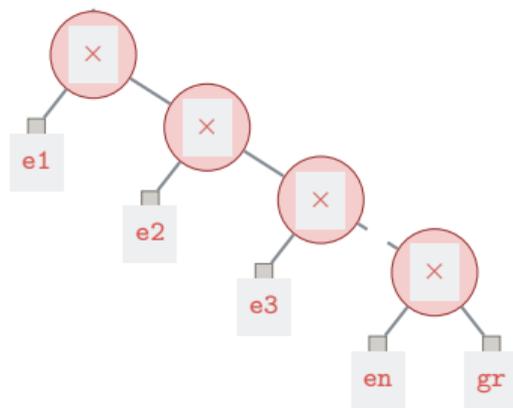
```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;  
- : int = 11
```

```
# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in  
  (pliage_gauche max (List.hd lst) (List.tl lst));;  
- : int = 5
```

```
# (pliage_gauche (^) "" ["mi" ; "la" ; "re" ; "sol" ; "do" ; "fa"]);;  
- : string = "milaresoldofa"
```

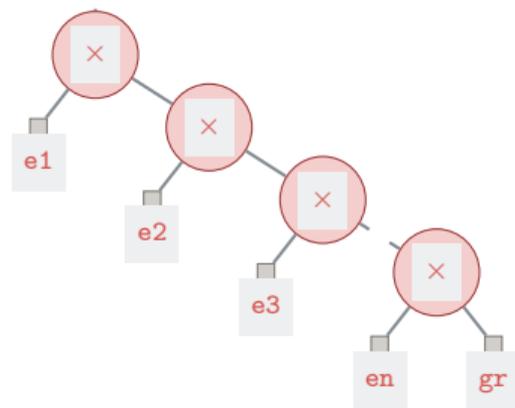
Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :



Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :

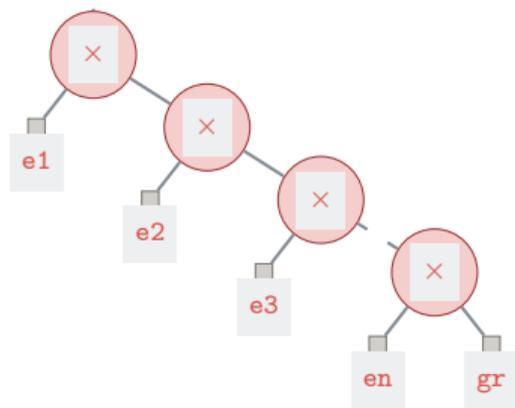


L'opération de **pliage à droite** est ainsi de type

```
('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :



L'opération de **pliage à droite** est ainsi de type

```
('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

et sa définition est

```
let rec pliage_droite op lst gr =  
  match lst with  
  | [] -> gr  
  | e :: reste -> (op e (pliage_droite op reste gr));;
```

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

D'un point de vue sémantique, lorsque l'opération `×` est **associative** et que la graine `gr` **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

D'un point de vue sémantique, lorsque l'opération `×` est **associative** et que la graine `gr` **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Il y a une différence d'efficacité : le pliage à gauche est **récurif terminal** alors que le pliage à droite ne l'est pas. En effet, dans le pliage à gauche, la graine est l'accumulateur.

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =  
  let rec aux lst acc =  
    match lst with  
    | [] -> acc  
    | e :: reste -> (aux reste (e :: acc))  
  in  
  (aux lst []);;
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =  
  let rec aux lst acc =  
    match lst with  
    | [] -> acc  
    | e :: reste -> (aux reste (e :: acc))  
  in  
  (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;  
- : char list = ['c'; 'b'; 'a']
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =  
  let rec aux lst acc =  
    match lst with  
    | [] -> acc  
    | e :: reste -> (aux reste (e :: acc))  
  in  
  (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;  
- : char list = ['c'; 'b'; 'a']
```

```
# (miroir [1 ; 1 ; 2 ; 2 ; 2 ; 1]);;  
- : int list = [1; 2; 2; 2; 1; 1]
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =  
  let rec aux lst acc =  
    match lst with  
    | [] -> acc  
    | e :: reste -> (aux reste (e :: acc))  
  in  
  (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;  
- : char list = ['c'; 'b'; 'a']
```

```
# (miroir [1 ; 1 ; 2 ; 2 ; 2 ; 1]);;  
- : int list = [1; 2; 2; 2; 1; 1]
```

Nous avons réimplanté ici la fonction `rev` du module `List`.

Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

| Fonction | Type |
|-------------------------|---|
| <code>map</code> | <code>('a -> 'b) -> 'a list -> 'b list</code> |
| <code>filter</code> | <code>('a -> bool) -> 'a list -> 'a list</code> |
| <code>for_all</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>exists</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>fold_left</code> | <code>('a -> 'b -> 'a) -> 'a -> 'b list -> 'a</code> |
| <code>fold_right</code> | <code>('a -> 'b -> 'b) -> 'a list -> 'b -> 'b</code> |

Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

| Fonction | Type |
|-------------------------|---|
| <code>map</code> | <code>('a -> 'b) -> 'a list -> 'b list</code> |
| <code>filter</code> | <code>('a -> bool) -> 'a list -> 'a list</code> |
| <code>for_all</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>exists</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>fold_left</code> | <code>('a -> 'b -> 'a) -> 'a -> 'b list -> 'a</code> |
| <code>fold_right</code> | <code>('a -> 'b -> 'b) -> 'a list -> 'b -> 'b</code> |

En pratique, on utilise ces fonctions sans les réimplanter.

Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

| Fonction | Type |
|-------------------------|---|
| <code>map</code> | <code>('a -> 'b) -> 'a list -> 'b list</code> |
| <code>filter</code> | <code>('a -> bool) -> 'a list -> 'a list</code> |
| <code>for_all</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>exists</code> | <code>('a -> bool) -> 'a list -> bool</code> |
| <code>fold_left</code> | <code>('a -> 'b -> 'a) -> 'a -> 'b list -> 'a</code> |
| <code>fold_right</code> | <code>('a -> 'b -> 'b) -> 'a list -> 'b -> 'b</code> |

En pratique, on utilise ces fonctions sans les réimplanter.

Il existe aussi la fonction `sort` de type

```
('a -> 'a -> int) -> 'a list -> 'a list
```

qui permet de renvoyer une version triée d'une liste d'éléments de type `'a` au moyen d'une fonction de comparaison (1^{er} paramètre).

Plan

Listes

Opérations

Non-mutabilité

Files

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet x , pour obtenir un objet x' calculé à partir de x , il faut **reconstruire** x' .

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet `x`, pour obtenir un objet `x'` calculé à partir de `x`, il faut **reconstruire** `x'`.

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet x , pour obtenir un objet x' calculé à partir de x , il faut **reconstruire** x' .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet x , pour obtenir un objet x' calculé à partir de x , il faut **reconstruire** x' .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;
3. gain de place mémoire.

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet `x`, pour obtenir un objet `x'` calculé à partir de `x`, il faut **reconstruire** `x'`.

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;
3. gain de place mémoire.

Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :

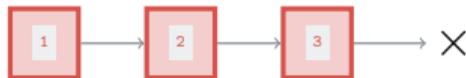
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



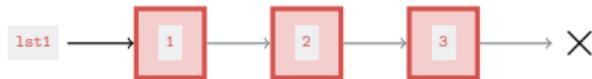
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :

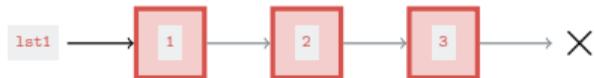
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

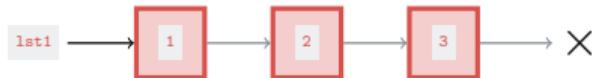
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :



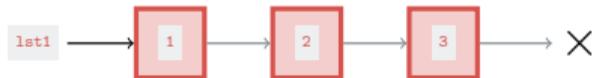
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

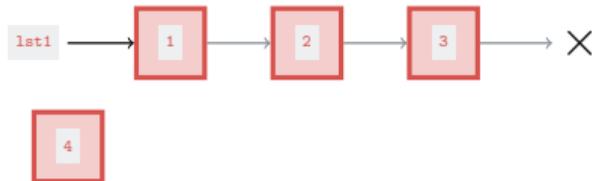
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :



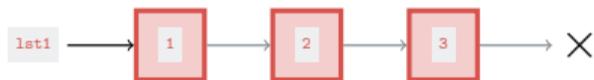
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

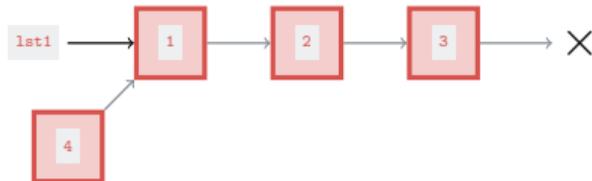
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :



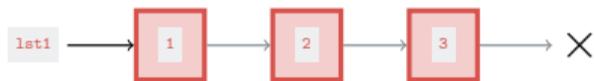
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

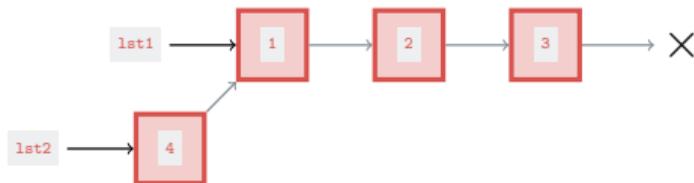
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1^{re} crée une liste (trois cellules) en mémoire :



La 2^e ne crée qu'une seule cellule et **partage** les trois précédentes :



Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;
```

On obtient en mémoire la configuration de partage suivante :



Le cas des listes

Considérons la fonction

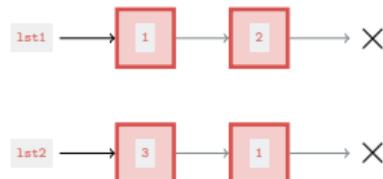
```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;  
# let lst2 = [3 ; 1];;
```

On obtient en mémoire la configuration de partage suivante :



Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

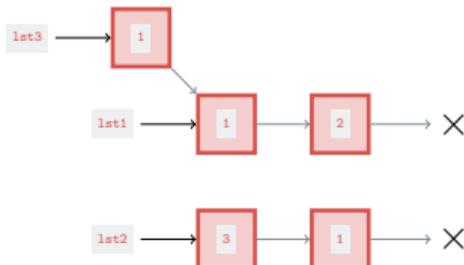
Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;  
# let lst2 = [3 ; 1];;
```

```
# let lst3 = 1 :: lst1;;
```

On obtient en mémoire la configuration de partage suivante :



Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

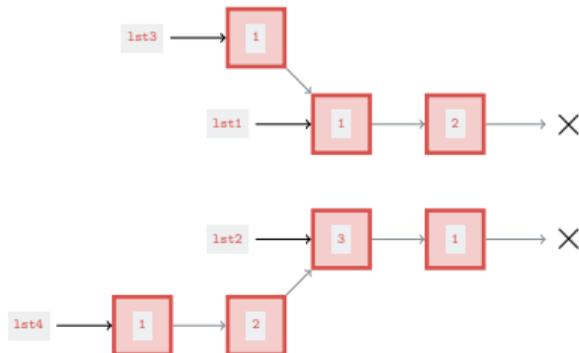
```
# let lst1 = [1 ; 2];;
```

```
# let lst2 = [3 ; 1];;
```

```
# let lst3 = 1 :: lst1;;
```

```
# let lst4 = (concatener lst1 lst2);;
```

On obtient en mémoire la configuration de partage suivante :



Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;
```

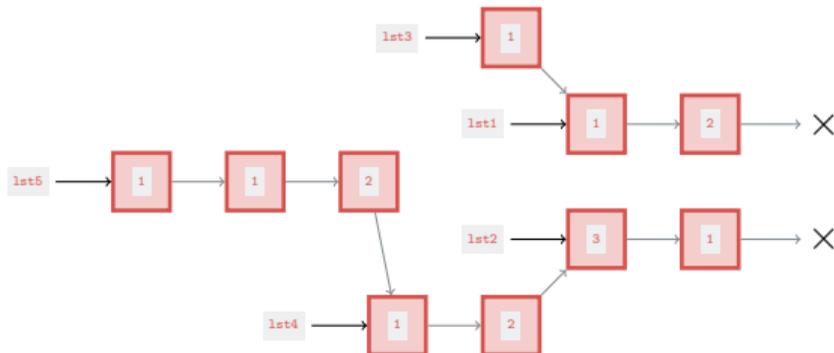
```
# let lst2 = [3 ; 1];;
```

```
# let lst3 = 1 :: lst1;;
```

```
# let lst4 = (concatener lst1 lst2);;
```

```
# let lst5 = (concatener lst3 lst4);;
```

On obtient en mémoire la configuration de partage suivante :



Plan

Listes

Opérations

Non-mutabilité

Files

Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque.

Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file` ;

Files

On souhaite implanter les **files** (piles First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file` ;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file` ;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file` ;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

4. définir une fonction

```
supprimer : 'a file -> 'a file
```

qui renvoie la file obtenue en supprimant son plus ancien élément;

Files

On souhaite implanter les **files** (piles First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file` ;

2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

4. définir une fonction

```
supprimer : 'a file -> 'a file
```

qui renvoie la file obtenue en supprimant son plus ancien élément;

5. définir une fonction

```
ajouter : 'a file -> 'a -> 'a file
```

qui renvoie une nouvelle file prenant en compte de l'ajout d'un élément.

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let rec ancien f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)
```

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

```
let rec ancien f =
```

```
  match f with
```

```
    | [] -> (failwith "file vide")
```

```
    | [e] -> e
```

```
    | e :: reste -> (ancien reste)
```

```
let rec supprimer f =
```

```
  match f with
```

```
    | [] -> (failwith "file vide")
```

```
    | [e] -> []
```

```
    | e :: reste -> e :: (supprimer reste)
```

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

```
let rec ancien f =
```

```
  match f with
```

```
    | [] -> (failwith "file vide")
```

```
    | [e] -> e
```

```
    | e :: reste -> (ancien reste)
```

```
let rec supprimer f =
```

```
  match f with
```

```
    | [] -> (failwith "file vide")
```

```
    | [e] -> []
```

```
    | e :: reste -> e :: (supprimer reste)
```

```
let ajouter f x =
```

```
  x :: f
```

Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let rec ancien f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)

let rec supprimer f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> []
  | e :: reste -> e :: (supprimer reste)

let ajouter f x =
  x :: f
```

En notant par n le nombre d'éléments de la file, on obtient les complexités

| Fonction | Complexité en temps |
|-----------|---------------------|
| ancien | $\Theta(n)$ |
| supprimer | $\Theta(n)$ |
| ajouter | $\Theta(1)$ |

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante.

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément e à la file consiste à positionner e en tête de **in**;

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément e à la file consiste à positionner e en tête de **in**;
- ▶ la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de **out** si elle est non vide.

Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

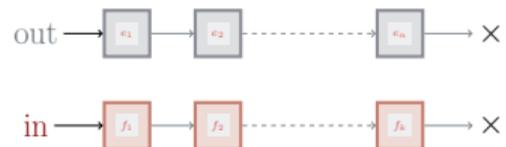
Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément e à la file consiste à positionner e en tête de **in**;
- ▶ la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de **out** si elle est non vide.

Si **out** est vide, on remplace **out** par le **miroir** de **in**, on vide **in** et on supprime/renvoie la tête de **out**.

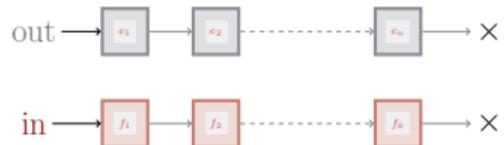
Implantation astucieuse

Voici une file dans une situation générique :

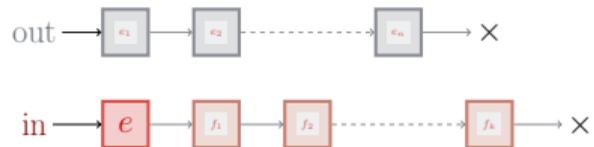


Implantation astucieuse

Voici une file dans une situation générique :

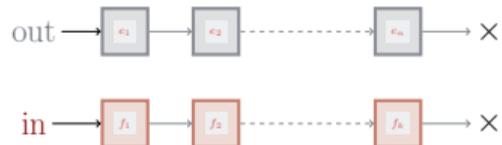


Après l'**ajout** d'un élément e , elle devient

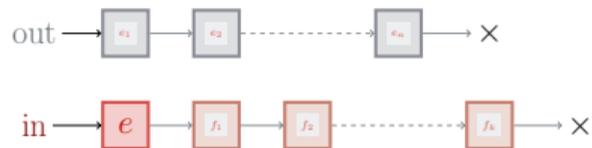


Implantation astucieuse

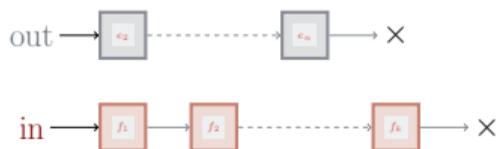
Voici une file dans une situation générique :



Après l'**ajout** d'un élément e , elle devient



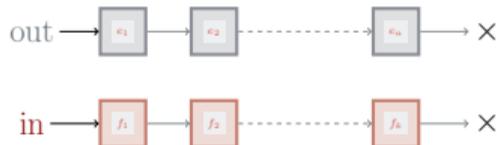
Après une **suppression** depuis la situation générique, elle devient



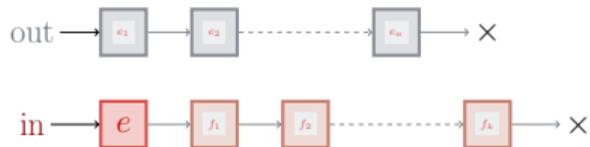
et renvoie e_1 si out n'est pas vide,

Implantation astucieuse

Voici une file dans une situation générique :



Après l'**ajout** d'un élément e , elle devient



Après une **suppression** depuis la situation générique, elle devient



et renvoie e_1 si out n'est pas vide,

et renvoie f_k si out est vide.

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|----|-----|
| vide | □ | □ |
| ajout(1) | | |
| ajout(1) | | |
| ajout(2) | | |
| ajout(3) | | |
| supprimer() | | |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----|-----|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | | |
| ajout(2) | | |
| ajout(3) | | |
| supprimer() | | |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|---------|-----|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | | |
| ajout(3) | | |
| supprimer() | | |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-------------|-----|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | | |
| supprimer() | | |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-----|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | | |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | | |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | | |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | | |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | [5 ; 4] | [2 ; 3] |
| supprimer() | | |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | [5 ; 4] | [2 ; 3] |
| supprimer() | [5 ; 4] | [3] |
| ajouter(6) | | |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | [5 ; 4] | [2 ; 3] |
| supprimer() | [5 ; 4] | [3] |
| ajouter(6) | [6 ; 5 ; 4] | [3] |
| supprimer() | | |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | [5 ; 4] | [2 ; 3] |
| supprimer() | [5 ; 4] | [3] |
| ajouter(6) | [6 ; 5 ; 4] | [3] |
| supprimer() | [6 ; 5 ; 4] | [] |
| supprimer() | | |

Implantation astucieuse — exemple

| Opération | in | out |
|-------------|-----------------|-------------|
| vide | [] | [] |
| ajout(1) | [1] | [] |
| ajout(1) | [1 ; 1] | [] |
| ajout(2) | [2 ; 1 ; 1] | [] |
| ajout(3) | [3 ; 2 ; 1 ; 1] | [] |
| supprimer() | [] | [1 ; 2 ; 3] |
| ajout(4) | [4] | [1 ; 2 ; 3] |
| ajout(5) | [5 ; 4] | [1 ; 2 ; 3] |
| supprimer() | [5 ; 4] | [2 ; 3] |
| supprimer() | [5 ; 4] | [3] |
| ajouter(6) | [6 ; 5 ; 4] | [3] |
| supprimer() | [6 ; 5 ; 4] | [] |
| supprimer() | [] | [5 ; 6] |

Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
   sortie : 'a list};;
```

Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
   sortie : 'a list};;  
  
let vide =  
  {entree = [] ; sortie = []};;
```

Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
   sortie : 'a list};;  
  
let vide =  
  {entree = [] ; sortie = []};;  
  
let ancien f =  
  match f.sortie with  
  | [] -> begin  
    match (List.rev f.entree) with  
    | [] -> (failwith  
             "file vide")  
    | e :: _ -> e  
  end  
  | e :: _ -> e;;
```

Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;
```

Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

Cette implantation est plus efficace que la précédente.

Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith
               "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

Cette implantation est plus efficace que la précédente.

Elle est même strictement plus efficace : les trois opérations ont une complexité en temps en $\Theta(1)$ sauf lorsque `out` est vide, ce qui demande pour `ancien` et `supprimer` une mise à jour en $\Theta(n)$.

Plan

λ -calcul

λ -termes

Codage

Implantation

Généralités

Le λ -calcul a été introduit par Church en 1936.

Généralités

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Généralités

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le λ -calcul, n'a pas un pouvoir d'expression suffisant.

Généralités

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le λ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Généralités

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le λ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Il offre ainsi un formalisme pour exprimer tout ce qui est calculable.

Généralités

Le λ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le λ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Il offre ainsi un formalisme pour exprimer tout ce qui est calculable.

Le λ -calcul constitue le cœur de tous les **langages de programmation fonctionnels**.

Plan

λ -calcul

λ -termes

Codage

Implantation

λ -termes

Un λ -terme est

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$,

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$, ou bien
2. une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme,

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$, ou bien
2. une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme, ou bien
3. une **application** st où s et t sont des λ -termes.

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$, ou bien
2. une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme, ou bien
3. une **application** st où s et t sont des λ -termes.

Remarque : il s'agit d'une définition récursive.

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$, ou bien
2. une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme, ou bien
3. une **application** st où s et t sont des λ -termes.

Remarque : il s'agit d'une définition récursive.

P.ex.,

$$(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$$

est un λ -terme.

λ -termes

Un λ -terme est

1. une **variable** $x, y, z, t, \text{etc.}$, ou bien
2. une **abstraction** $\lambda x.t$ où x est une variable et t est un λ -terme, ou bien
3. une **application** st où s et t sont des λ -termes.

Remarque : il s'agit d'une définition récursive.

P.ex.,

$$(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$$

est un λ -terme.

Attention à l'emploi de parenthèses pour éviter les ambiguïtés. Il existe diverses conventions pour réduire leur nombre (que nous n'emploierons pas ici).

Arbres syntaxiques

Tout λ -terme t peut se représenter par son **arbre syntaxique** de la manière suivante :

Arbres syntaxiques

Tout λ -terme t peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si t est une variable x , l'arbre syntaxique de t est la feuille étiquetée par x ;

Arbres syntaxiques

Tout λ -terme t peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si t est une variable x , l'arbre syntaxique de t est la feuille étiquetée par x ;
2. si t est une abstraction $\lambda x.t'$, l'arbre syntaxique de t est un nœud unaire étiqueté par λx qui possède comme fils l'arbre syntaxique de t' ;

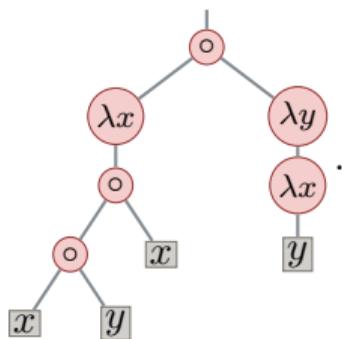
Arbres syntaxiques

Tout λ -terme t peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si t est une variable x , l'arbre syntaxique de t est la feuille étiquetée par x ;
2. si t est une abstraction $\lambda x.t'$, l'arbre syntaxique de t est un nœud unaire étiqueté par λx qui possède comme fils l'arbre syntaxique de t' ;
3. si t est une application $t' t''$, l'arbre syntaxique de t est un nœud binaire étiqueté par \circ qui possède comme fils gauche l'arbre syntaxique de t' et comme fils droit l'arbre syntaxique de t'' .

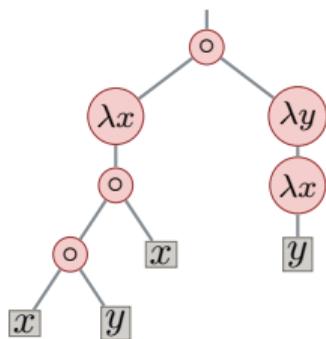
Arbres syntaxiques

L'arbre syntaxique de $(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$ est

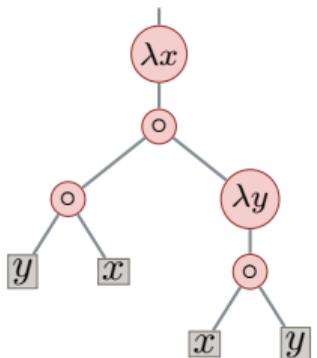


Arbres syntaxiques

L'arbre syntaxique de $(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$ est



L'arbre syntaxique de $\lambda x.((y x) (\lambda y.(x y)))$ est



Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

- ▶ si le nœud visité est étiqueté par λx , on s'arrête et on relie f à ce nœud;

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

- ▶ si le nœud visité est étiqueté par λx , on s'arrête et on relie f à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de a , on réitère ce processus depuis son parent;

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

- ▶ si le nœud visité est étiqueté par λx , on s'arrête et on relie f à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de a , on réitère ce processus depuis son parent;
- ▶ sinon, on s'arrête.

Variables libres/liées

Soit t un λ -terme et une occurrence d'une variable x y apparaissant.

Cette **occurrence** correspond à une **feuille** f de l'arbre syntaxique a de t .

Pour décider si cette occurrence de x dans t est libre ou liée, on considère le processus qui consiste à remonter depuis f vers la racine de a de sorte que

- ▶ si le nœud visité est étiqueté par λx , on s'arrête et on relie f à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de a , on réitère ce processus depuis son parent;
- ▶ sinon, on s'arrête.

Finalement, si f est liée à un nœud, on dit que l'occurrence de x est **liée** (ou encore **capturée** par le λx cible); sinon, on dit qu'elle est **libre**.

Variables libres/liées

Soit le λ -terme

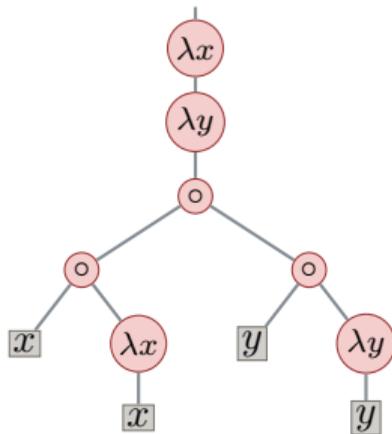
$$\lambda x. \lambda y. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Variables libres/liées

Soit le λ -terme

$$\lambda x. \lambda y. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Son arbre syntaxique est

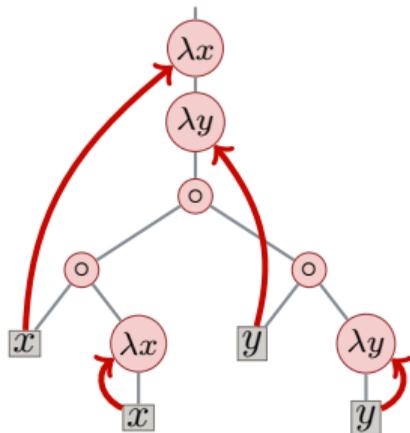


Variables libres/liées

Soit le λ -terme

$$\lambda x. \lambda y. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Son arbre syntaxique est



Les flèches connectent chaque occurrence de variable à son abstraction qui la capture.

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z$$

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z$$

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

$$\blacktriangleright (z (\lambda z.(y z))) z \star_z \lambda x.(x x)$$

Substitutions

Soit t un λ -terme, x une variable et s un autre λ -terme.

La **substitution** de s à x dans t consiste à remplacer chaque **occurrence libre** de x dans t par s .
On note $t \star_x s$ le λ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

$$\blacktriangleright (z (\lambda z.(y z))) z \star_z \lambda x.(x x) = ((\lambda x.(x x)) (\lambda z.(y z))) (\lambda x.(x x)).$$

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- ▶ l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- ▶ l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

Deux λ -termes t et t' sont α -équivalents s'il est possible de transformer t en t' par une suite d' α -renommages.

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- ▶ l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

Deux λ -termes t et t' sont α -équivalents s'il est possible de transformer t en t' par une suite d' α -renommages.

P.ex.,

- ▶ $\lambda x.x$ et $\lambda y.y$ sont α -équivalents;

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- ▶ l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

Deux λ -termes t et t' sont α -équivalents s'il est possible de transformer t en t' par une suite d' α -renommages.

P.ex.,

- ▶ $\lambda x.x$ et $\lambda y.y$ sont α -équivalents;
- ▶ $\lambda x.\lambda y.(x y)$ et $\lambda y.\lambda x.(y x)$ sont α -équivalents;

α -renommages et α -équivalence

Soient t un λ -terme, x une variable et y une variable qui n'admet aucune occurrence dans t .

L' α -renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y .

P.ex.,

- ▶ l' α -renommage de x en y du λ -terme $\lambda x.x$ est $\lambda y.y$;
- ▶ l' α -renommage de x en y du λ -terme $(x x) (\lambda x.(x z))$ est $(x x) (\lambda y.(y z))$.

Deux λ -termes t et t' sont α -équivalents s'il est possible de transformer t en t' par une suite d' α -renommages.

P.ex.,

- ▶ $\lambda x.x$ et $\lambda y.y$ sont α -équivalents;
- ▶ $\lambda x.\lambda y.(x y)$ et $\lambda y.\lambda x.(y x)$ sont α -équivalents;
- ▶ $\lambda x.(x y)$ et $\lambda x.(x z)$ ne sont pas α -équivalents.

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x.u) v.$$

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x.u) v.$$

La β -réduction en racine appliquée sur t consiste à transformer t en le λ -terme

$$u \star_x v.$$

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x.u) v.$$

La β -réduction en racine appliquée sur t consiste à transformer t en le λ -terme

$$u \star_x v.$$

Attention : aucune occurrence d'une variable libre de v ne doit être capturée dans $u \star_x v$.

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x.u) v.$$

La β -réduction en racine appliquée sur t consiste à transformer t en le λ -terme

$$u \star_x v.$$

Attention : aucune occurrence d'une variable libre de v ne doit être capturée dans $u \star_x v$.

De ce fait, dès que v admet une occurrence libre d'une variable y , on doit α -renommer y en y' dans u au préalable, où y' est une nouvelle variable qui n'admet pas d'occurrence libre dans v .

β -réductions en racine

Soit t un λ -terme de la forme

$$t := (\lambda x.u) v.$$

La β -réduction en racine appliquée sur t consiste à transformer t en le λ -terme

$$u \star_x v.$$

Attention : aucune occurrence d'une variable libre de v ne doit être capturée dans $u \star_x v$.

De ce fait, dès que v admet une occurrence libre d'une variable y , on doit α -renommer y en y' dans u au préalable, où y' est une nouvelle variable qui n'admet pas d'occurrence libre dans v .

On note $t \xrightarrow{\beta} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir du λ -terme t par une β -réduction en racine.

β -réductions en racine — exemples

► $(\lambda y.y)(\lambda x.(x z))$

β -réductions en racine — exemples

► $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶ $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x))$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶ $(\lambda x.\lambda y.((x x)(\lambda x.x)))(\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x))(\lambda y(y x))))(\lambda x.x)$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶ $(\lambda x.\lambda y.((x x)(\lambda x.x)))(\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x))(\lambda y(y x))))(\lambda x.x)$

▶ $(\lambda x.\lambda y.x)(z y)$

β -réductions en racine — exemples

$$\blacktriangleright (\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$$

$$\blacktriangleright (\lambda x.(x y)) z \xrightarrow{\beta} z y$$

$$\blacktriangleright (\lambda x.\lambda y.((x x)(\lambda x.x)))(\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x))(\lambda y(y x))))(\lambda x.x)$$

$$\blacktriangleright (\lambda x.\lambda y.x)(z y) \not\xrightarrow{\beta} \lambda y.(z y)$$

β -réductions en racine — exemples

▶ $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶ $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶ $(\lambda x.\lambda y.((x x)(\lambda x.x)))(\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x))(\lambda y(y x))))(\lambda x.x)$

▶ $(\lambda x.\lambda y.x)(z y) \not\xrightarrow{\beta} \lambda y.(z y)$

Ici, il l'occurrence libre de y dans $(z y)$ serait capturée par le λy dans le « résultat ».

β -réductions en racine — exemples

$$\blacktriangleright (\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$$

$$\blacktriangleright (\lambda x.(x y)) z \xrightarrow{\beta} z y$$

$$\blacktriangleright (\lambda x.\lambda y.((x x)(\lambda x.x)))(\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x))(\lambda y(y x))))(\lambda x.x)$$

$$\blacktriangleright (\lambda x.\lambda y.x)(z y) \not\xrightarrow{\beta} \lambda y.(z y)$$

Ici, il l'occurrence libre de y dans $(z y)$ serait capturée par le λy dans le « résultat ».

On α -renomme donc y en y' dans $(\lambda x.\lambda y.x)$ et on obtient

$$(\lambda x.\lambda y.x)(z y) = (\lambda x.\lambda y'.x)(z y) \xrightarrow{\beta} \lambda y'.(z y).$$

β -réductions

Soit t un λ -terme.

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

x ,

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$x, y,$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y,$$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y),$$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y),$$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La **β -réduction** appliquée sur t consiste à appliquer une β -réduction en racine sur l'un de ses sous-termes.

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La **β -réduction** appliquée sur t consiste à appliquer une β -réduction en racine sur l'un de ses sous-termes.

On note $t \rightarrow_{\beta} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir de t par une β -réduction.

β -réductions

Soit t un λ -terme.

On appelle **sous-terme** de t tout λ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t .

P.ex., le λ -terme $x (\lambda x. \lambda y. (y y))$ possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La **β -réduction** appliquée sur t consiste à appliquer une β -réduction en racine sur l'un de ses sous-termes.

On note $t \rightarrow_{\beta} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir de t par une β -réduction.

On note $t \xrightarrow{*}_{\beta} t'$ le fait qu'un λ -terme t' puisse être obtenu à partir de t par une suite de β -réductions.

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x))$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) \rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x))$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \end{aligned}$$

β -réductions — exemples

- Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$(\lambda y.y) ((\lambda x.x) x)$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$(\lambda y.y) ((\lambda x.x) x) \rightarrow_{\beta} (\lambda y.y) x$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

$$(\lambda y.y) ((\lambda x.x) x)$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned} (\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x, \end{aligned}$$

$$(\lambda y.y) ((\lambda x.x) x) \rightarrow_{\beta} (\lambda x.x) x$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda x.x) x \\ &\rightarrow_{\beta} x.\end{aligned}$$

β -réductions — exemples

- ▶ Voici une suite de β -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même λ -terme, on peut appliquer plusieurs suites de β -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda x.x) x \\ &\rightarrow_{\beta} x.\end{aligned}$$

Ainsi dans ces deux cas, $(\lambda y.y) ((\lambda x.x) x) \xrightarrow{*}_{\beta} x$.

Règle de calcul

La relation de β -réduction \rightarrow_β est une règle de calcul.

Règle de calcul

La relation de β -réduction \rightarrow_β est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

Règle de calcul

La relation de β -réduction \rightarrow_{β} est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

Règle de calcul

La relation de β -réduction \rightarrow_β est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Règle de calcul

La relation de β -réduction \rightarrow_{β} est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un λ -terme, plusieurs processus d'évaluation différents.

Règle de calcul

La relation de β -réduction \rightarrow_β est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un λ -terme, plusieurs processus d'évaluation différents.

Question : est-ce que les différentes évaluations d'un même λ -terme t aboutissent à la même valeur ?

Règle de calcul

La relation de β -réduction \rightarrow_β est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des β -réductions sur un λ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un λ -terme t complexe, d'obtenir un λ -terme t' plus simple que l'on ne peut plus β -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un λ -terme, plusieurs processus d'évaluation différents.

Question : est-ce que les différentes évaluations d'un même λ -terme t aboutissent à la même valeur ?

Réponse : oui, d'après le **théorème de Church-Rosser**.

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\Delta \Delta$$

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\Delta \Delta = (\lambda x.(x x)) (\lambda x.(x x))$$

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned}\Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x))\end{aligned}$$

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned}\Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} \dots\end{aligned}$$

Terminaison

Question : est-ce que l'évaluation de tout λ -terme aboutit toujours à une valeur ?

Réponse : considérons le λ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned}\Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} \dots\end{aligned}$$

Ainsi, l'évaluation de $\Delta \Delta$ ne termine pas. La réponse est donc non.

Plan

λ -calcul

λ -termes

Codage

Implantation

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de t correspond à l'**exécution** du programme qu'il code

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de t correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de t termine, le réduit obtenu est la valeur calculée par ce programme;

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de t correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de t termine, le réduit obtenu est la valeur calculée par ce programme;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de t correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de t termine, le réduit obtenu est la valeur calculée par ce programme;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

Il reste à savoir comment donner du **sens** à nos λ -termes.

Programmer en λ -calcul

Voici quelques correspondances entre le λ -calcul et la programmation (fonctionnelle).

Tout λ -terme t code un **programme**.

Étant donné que toutes les différentes évaluations d'un même λ -terme t aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de t correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de t termine, le réduit obtenu est la valeur calculée par ce programme;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

Il reste à savoir comment donner du **sens** à nos λ -termes.

La question est de savoir comment coder des **booléens**, des **entiers** et des **constructions conditionnelles**. Ceci offre un niveau d'expressivité raisonnable.

Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

$$\text{vrai} := \lambda x.\lambda y.x,$$

Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

`vrai` := $\lambda x.\lambda y.x$,

`faux` := $\lambda x.\lambda y.y$,

Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

$$\text{vrai} := \lambda x.\lambda y.x,$$
$$\text{faux} := \lambda x.\lambda y.y,$$

Nous avons, si u et v sont deux λ -termes,

$$(\text{vrai } u) v$$

Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

$$\text{vrai} := \lambda x.\lambda y.x,$$
$$\text{faux} := \lambda x.\lambda y.y,$$

Nous avons, si u et v sont deux λ -termes,

$$(\text{vrai } u) v = ((\lambda x.\lambda y.x) u) v$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x. \lambda y. x,$$

$$\mathbf{faux} := \lambda x. \lambda y. y,$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} (\mathbf{vrai} \ u) \ v &= ((\lambda x. \lambda y. x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. u) \ v \end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x. \lambda y. x,$$

$$\mathbf{faux} := \lambda x. \lambda y. y,$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} (\mathbf{vrai} \ u) \ v &= ((\lambda x. \lambda y. x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. u) \ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1^{er} argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned} \text{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \text{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} (\text{vrai } u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u. \end{aligned}$$

De même,

$$(\text{faux } u) v$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$(\mathbf{faux} u) v = ((\lambda x.\lambda y.y) u) v$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} u) v &= ((\lambda x.\lambda y.y) u) v \\ &\rightarrow_{\beta} (\lambda y.y) v\end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} u) v &= ((\lambda x.\lambda y.y) u) v \\ &\rightarrow_{\beta} (\lambda y.y) v \\ &\rightarrow_{\beta} v.\end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x. \lambda y. x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x. \lambda y. y, && \text{renvoie son 2}^{\text{e}} \text{ argument.}\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} \ u) \ v &= ((\lambda x. \lambda y. x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. u) \ v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} \ u) \ v &= ((\lambda x. \lambda y. y) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. y) \ v \\ &\rightarrow_{\beta} v.\end{aligned}$$

Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y, && \text{renvoie son 2}^{\text{e}} \text{ argument.}\end{aligned}$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} u) v &= ((\lambda x.\lambda y.y) u) v \\ &\rightarrow_{\beta} (\lambda y.y) v \\ &\rightarrow_{\beta} v.\end{aligned}$$

Attention : ce codage, tout comme ceux qui vont suivre, ne sont pas uniques. Ils ont simplement des propriétés qui font leur intérêt.

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\text{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\text{et} := \lambda b. \lambda c. ((b\ c)\ b).$$

Par exemple,

(**et vrai**) vrai

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\text{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$(\text{et vrai})\ \text{vrai} = ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x)$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\text{et} := \lambda b. \lambda c. ((b\ c)\ b).$$

Par exemple,

$$\begin{aligned} (\text{et vrai})\ \text{vrai} &= ((\lambda b. \lambda c. ((b\ c)\ b))(\lambda x. \lambda y. x))(\lambda x. \lambda y. x) \\ &\rightarrow_{\beta} (\lambda c. (((\lambda x. \lambda y. x)\ c)\ (\lambda x. \lambda y. x)))(\lambda x. \lambda y. x) \end{aligned}$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b. \lambda c. ((b\ c)\ b).$$

Par exemple,

$$\begin{aligned} (\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b. \lambda c. ((b\ c)\ b))(\lambda x. \lambda y. x))(\lambda x. \lambda y. x) \\ &\rightarrow_{\beta} (\lambda c. (((\lambda x. \lambda y. x)\ c)\ (\lambda x. \lambda y. x)))(\lambda x. \lambda y. x) \\ &\rightarrow_{\beta} (\lambda c. ((\lambda y. c)\ (\lambda x. \lambda y. x)))(\lambda x. \lambda y. x) \end{aligned}$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x)\end{aligned}$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} \lambda x.\lambda y.x\end{aligned}$$

Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} \lambda x.\lambda y.x \\ &= \mathbf{vrai}.\end{aligned}$$

Opérateurs booléens

On code le « ou » logique **ou** de la manière suivante :

$$\text{ou} := \lambda b. \lambda c. ((b\ b) c).$$

Opérateurs booléens

On code le « ou » logique **ou** de la manière suivante :

$$\text{ou} := \lambda b.\lambda c.((b b) c).$$

On code le « non » logique **non** de la manière suivante :

$$\text{non} := \lambda b.\lambda x.\lambda y.((b y) x).$$

Opérateurs booléens

On code le « ou » logique **ou** de la manière suivante :

$$\text{ou} := \lambda b.\lambda c.((b b) c).$$

On code le « non » logique **non** de la manière suivante :

$$\text{non} := \lambda b.\lambda x.\lambda y.((b y) x).$$

On code l'implication logique **imp** de la manière suivante :

$$\text{imp} := \lambda a.\lambda b.((\text{ou} (\text{non } a)) b).$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$((\text{sas}\ \text{vrai})\ u)\ v$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$((\text{sas}\ \text{vrai})\ u)\ v = (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.((\text{vrai}\ y)\ z))\ u)\ v \end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z.u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

De même, on obtient

$$((\text{sas}\ \text{faux})\ u)\ v \rightarrow_{\beta} v.$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si u et v sont deux λ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

De même, on obtient

$$((\text{sas}\ \text{faux})\ u)\ v \rightarrow_{\beta} v.$$

Ainsi, la valeur calculée par `sas`, appliquée à trois arguments, possède le comportement attendu.

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

$$\text{ent}(3) = \lambda f.\lambda x.(f (f (f x))).$$

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

$$\text{ent}(3) = \lambda f.\lambda x.(f (f (f x))).$$

Intuitivement, un entier n est une fonction à deux paramètres f et x qui applique f à x de manière itérée n fois.

Entiers naturels

On code tout **entier naturel** n par $\text{ent}(n)$ défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

$$\text{ent}(3) = \lambda f.\lambda x.(f (f (f x))).$$

Intuitivement, un entier n est une fonction à deux paramètres f et x qui applique f à x de manière itérée n fois.

Ce codage est connu sous le nom d'**entiers de Church**.

Arithmétique sur les entiers de Church

La fonction `succ` calcule le `successeur` d'un entier. Elle se code par

$$\text{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Arithmétique sur les entiers de Church

La fonction `succ` calcule le `successeur` d'un entier. Elle se code par

$$\text{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, `succ` est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

Arithmétique sur les entiers de Church

La fonction `succ` calcule le `successeur` d'un entier. Elle se code par

$$\text{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, `succ` est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\text{succ ent}(2)$$

Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\mathbf{succ} \text{ ent}(2) = (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x)))$$

Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \end{aligned}$$

Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \end{aligned}$$

Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (f (f x))) \end{aligned}$$

Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church n qui renvoie l'entier de Church obtenu en appliquant f à x une fois supplémentaire dans n .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (f (f x))) \\ &= \text{ent}(3). \end{aligned}$$

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers.

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers.

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

La fonction `pui` calcule l'**exponentiation** de deux entiers.

Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

La fonction `pui` calcule l'**exponentiation** de deux entiers. Elle se code par

$$\text{pui} := \lambda n. \lambda m. (m n).$$

Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry** ρ défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x)))).$$

Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry** ρ défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x))))).$$

Il vérifie la propriété fondamentale suivante : pour tout λ -terme u ,

$$\rho u \rightarrow_{\beta} u(\rho u).$$

Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry** ρ défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x))))).$$

Il vérifie la propriété fondamentale suivante : pour tout λ -terme u ,

$$\rho u \rightarrow_{\beta} u(\rho u).$$

Ceci implique

$$\rho u \rightarrow_{\beta} u(\rho u) \rightarrow_{\beta} u(u(\rho u)) \rightarrow_{\beta} u(u(u(\rho u))) \rightarrow_{\beta} \dots,$$

offrant un moyen d'appliquer de manière itérée le λ -terme (la fonction) u .

Plan

λ -calcul

λ -termes

Codage

Implantation

Représentation de λ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Représentation de λ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' α -renommage en particulier).

Représentation de λ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' α -renommage en particulier).

Le type pour représenter les λ -termes est un type somme :

```
type terme =  
  | Variable of variable  
  | Abstraction of variable * terme  
  | Application of terme * terme;;
```

Représentation de λ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' α -renommage en particulier).

Le type pour représenter les λ -termes est un type somme :

```
type terme =  
  | Variable of variable  
  | Abstraction of variable * terme  
  | Application of terme * terme;;
```

La définition de ce type `terme` est conforme à la définition des λ -termes (récursive et en trois parties).

Variables libres/liées

Cette fonction renvoie la liste des variables `x` qui admettent au moins une occurrence libre :

```
let rec variables_libres t =  
  match t with  
  |Variable v -> [v]  
  |Abstraction (v, t') ->  
    List.filter (fun x -> x <> v) (variables_libres t')  
  |Application (t', t'') ->  
    List.append (variables_libres t') (variables_libres t'')
```

Variables libres/liées

Cette fonction renvoie la liste des variables `x` qui admettent au moins une occurrence libre :

```
let rec variables_libres t =
  match t with
  |Variable v -> [v]
  |Abstraction (v, t') ->
    List.filter (fun x -> x <> v) (variables_libres t')
  |Application (t', t'') ->
    List.append (variables_libres t') (variables_libres t'')
```

P.ex., cette fonction appliquée au λ -terme

$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y)))$$

donne

```
# let t1 = Abstraction (x, Abstraction (x,
  Application (Application
    (Variable x, Abstraction (x, Variable x)),
  Application
    (Variable y, Abstraction (y, Variable y))))))
in
variables_libres t1;;
- : variable list = [nom = "y"; ref = 1]
```