Programmation fonctionnelle

Samuele Giraudo

Université Paris-Est Marne-la-Vallée
LIGM, bureau 4B055
samuele.giraudo@univ-mlv.fr
http://igm.univ-mlv.fr/~giraudo/

Contenu du cours

Objectifs et bibliographie

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1re approche du paradigme de programmation fonctionnelle ;
2. d'apprendre les bases de programmation en Caml en mode fonctionnel.

Objectifs et bibliographie

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1re approche du paradigme de programmation fonctionnelle ;
2. d'apprendre les bases de programmation en Caml en mode fonctionnel.

Quelques éléments bibliographiques :

- X. Leroy, P. Weis, *Le langage Caml*, Dunod, 2e édition, 1999.
  Lien : `http://caml.inria.fr/distrib/books/llc.pdf`
- E. Chailloux, P. Manoury, B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly, 2000.
  Lien : `http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-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.

Plan

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.
- 1936 : Turing invente la machine de Turing.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 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 : 1re machine programmable : le métier à tisser de Jackard.
- 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 : 1ers vrais langages de programmation : FORTRAN, COBOL et LISP.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 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 : 1ers vrais langages de programmation : FORTRAN, COBOL et LISP.
- Années 1960-1970 : paradigmes de programmation : impératif, fonctionnel, orienté objet, logique.

Plan

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. $\Delta : E \times \{\cdot, 0, 1\} \to E \times \{\cdot, 0, 1\} \times \{\mathrm{G}, \mathrm{D}\}$ 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	$\cdots$	$u_{\lvert u \rvert}$	$\cdot$	$\cdot$	$\cdots$

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

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	$\cdots$	$u_{\|u\|}$	$\cdot$	$\cdot$	$\cdots$

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

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

u_1	u_2	$\cdots$	$u_{\|u\|}$	$\cdot$	$\cdot$	$\cdots$

   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** :

u_1	u_2	$\cdots$	$u_{\lvert u \rvert}$	$\cdot$	$\cdot$	$\cdots$

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

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

u_1	u_2	$\cdots$	$u_{\lvert u \rvert}$	$\cdot$	$\cdot$	$\cdots$
$\uparrow$						

   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 (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** :

u_1	u_2	$\cdots$	$u_{\lvert u \rvert}$	$\cdot$	$\cdot$	$\cdots$

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

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

u_1	u_2	$\cdots$	$u_{\lvert u \rvert}$	$\cdot$	$\cdot$	$\cdots$
$\uparrow$						

   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 (état initial).

4. Réaliser les actions dictées par Δ, le **programme**.

Fonctionnement des machines de Turing

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

Fonctionnement des machines de Turing

Pour réaliser les action 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 action 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 action 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 action 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 $s' = \mathrm{D}$, 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 action 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 $s' = \mathrm{D}$, 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 action 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 $s' = \mathrm{D}$, 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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

0	0	1	0	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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Calcul de $M(00101)$:

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

1	0	1	0	1	$\cdot$	$\cdot$	$\cdots$

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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	1	0	1	$\cdot$	$\cdot$	$\cdots$

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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	0	0	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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	0	1	1	$\cdot$	$\cdot$	$\cdots$

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

$$\begin{aligned}
(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}), \\
(e_1, 1) &\mapsto (e_1, 0, \mathrm{D}), \\
(e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).
\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	0	1	0	$\cdot$	$\cdot$	$\cdots$

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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	0	1	0	$\cdot$	$\cdot$	$\cdots$

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

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\ (e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\ (e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).\end{aligned}$$

Calcul de $M(00101)$:

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

1	1	0	1	0	$\cdot$	$\cdot$	$\cdots$

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

$$
\begin{aligned}
(e_1, 0) &\mapsto (e_1, 1, \mathrm{D}),\\
(e_1, 1) &\mapsto (e_1, 0, \mathrm{D}),\\
(e_1, \cdot) &\mapsto (e_2, \cdot, \mathrm{D}).
\end{aligned}
$$

Calcul de $M(00101)$:

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

1	1	0	1	0	$\cdot$	$\cdot$	$\cdots$

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

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

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

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

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

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

Calcul de $M(001)$:

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

0	0	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

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

Calcul de $M(001)$:

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

0	0	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

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

Calcul de $M(001)$:

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

0	0	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

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

Calcul de $M(001)$:

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

0	0	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

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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

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

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

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

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	·	·	·	...

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

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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	$\cdot$	$\cdot$	$\cdot$	$\cdots$

$\uparrow$

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

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

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	·	·	·	...

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

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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

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

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

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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

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

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 $\mathrm{code}(M)$ de la manière suivante :

Coder une machine de Turing par un entier naturel

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

1. on fixe des conventions 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions 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 $\mathrm{code}(M)$ 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

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

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

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

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

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

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

$$u = 011001010111010 0 \dots 0010100100001101$$

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

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).

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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

1. Théorie
   - Machines de Turing
   - Décidabilité et indécidabilité
   - λ-calcul
   - Impératif *vs* fonctionnel
   - Caractéristiques des langages

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 ? $P(010010) = \mathrm{oui}$, $P(011) = \mathrm{non}$;
- P : la longueur du mot est-elle paire ? $P(\epsilon) = \mathrm{oui}$, $P(1) = \mathrm{non}$;

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

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

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

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

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

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

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

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 $P(u) = \mathrm{oui}$ 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}.

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

   (a) si $M_{\mathrm{Arr}}(u) = 1$, où u est le codage binaire de f, 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}.

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, où u est le codage binaire de f, ajouter à E la valeur calculée par f, exprimée par un entier.

(3) Renvoyer $\min \mathbb{N} \setminus E$.

Comme l'exécution d'$\mathrm{absurde}$ 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}.

Soit 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 absurde,

   (a) si $M_{\mathrm{Arr}}(u) = 1$, où u est le codage binaire de f, ajouter à E la valeur calculée par f, exprimée par un entier.

(3) Renvoyer $\min \mathbb{N} \setminus E$.

Comme l'exécution d'absurde se termine, 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}.

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

   (a) si $M_{\mathrm{Arr}}(u) = 1$, où u est le codage binaire de f, ajouter à E la valeur calculée par f, exprimée par un entier.

(3) Renvoyer $\min \mathbb{N} \setminus E$.

Comme l'exécution d'$\mathrm{absurde}$ se termine, la valeur de retour d'$\mathrm{absurde}$ figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par $\mathrm{absurde}$ 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}.

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,
    (a) si $M_{\mathrm{Arr}}(u) = 1$, où u est le codage binaire de f, ajouter à E la valeur calculée par f, exprimée par un entier.
(3) Renvoyer $\min \mathbb{N} \setminus E$.

Comme l'exécution d'$\mathrm{absurde}$ se termine, la valeur de retour d'$\mathrm{absurde}$ figure dans E (étape (a)).

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

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

Plan

1. Théorie
    - Machines de Turing
    - Décidabilité et indécidabilité
    - λ-calcul
    - Impératif *vs* fonctionnel
    - Caractéristiques des langages

Fonctions récursives

Une fonction récursive est une fonction

$$f : \mathbb{N}^k \to \mathbb{N}$$

qui est intuitivement calculable.

Fonctions récursives

Une fonction récursive est une fonction

$$f : \mathbb{N}^k \to \mathbb{N}$$

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 $n \leqslant 1, n \times f(n-1)$ sinon ;

sont des fonctions récursives.

Fonctions récursives

Une fonction récursive est une fonction

$$f : \mathbb{N}^k \to \mathbb{N}$$

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 $n \leqslant 1, n \times f(n-1)$ sinon ;

sont des fonctions récursives.

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

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

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 de fonction.

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 de fonction.

Une expression du λ-calcul est

1. soit une **variable**, notée x, y, z, ...;
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 de fonction.

Une expression du λ-calcul est

1. soit une **variable**, notée x, y, z, $\ldots$;
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 de fonction.

Une expression du λ-calcul est

1. soit une **variable**, notée x, y, z, $\ldots$;
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

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$^{\text{er}}$ se base sur la machine de Turing, le 2$^{\text{e}}$ 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 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 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 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.

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).

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.

Plan

Typage 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 ».

Typage 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** (typage) :

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

Typage 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** (typage) :

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

Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :     # Test de parité sur ‘n‘.
    print n / 2     # Calcul arithm. sur ‘n‘ et affichage.
else :
    print n[1]      # Affichage de la 1re case de ‘n‘.
```

Typage dynamique

Considérons le programme Python

```
n = int(input())   # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur ‘n‘.
    print n / 2    # Calcul arithm. sur ‘n‘ et affichage.
else :
    print n[1]     # Affichage de la 1re case de ‘n‘.
```

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 ;

Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :     # Test de parité sur ‘n‘.
    print n / 2     # Calcul arithm. sur ‘n‘ et affichage.
else :
    print n[1]      # Affichage de la 1re case de ‘n‘.
```

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.

Typage dynamique

Considérons le programme Python

```
n = int(input())   # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur ‘n‘.
    print n / 2    # Calcul arithm. sur ‘n‘ et affichage.
else :
    print n[1]     # Affichage de la 1re case de ‘n‘.
```

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.

Typage dynamique

Considérons le programme Python

```
n = int(input())   # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur ‘n‘.
    print n / 2    # Calcul arithm. sur ‘n‘ et affichage.
else :
    print n[1]     # Affichage de la 1re case de ‘n‘.
```

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__'
```

Typage 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;
}
```

Typage 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 *`.

Typage 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

Typage dynamique.

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

Avantages et inconvénients

Typage 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

Typage 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).

Typage 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

Typage 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).

Typage 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.

Typage explicite *vs* implicite

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

Typage 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. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;

Typage 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. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;
2. typage implicite, où les **types** des variables ne sont **pas mentionnés** dans le programme. Ils sont devinés lors de la compilation (si typage statique) ou lors de l'exécution (si typage dynamique) en fonction du contexte.

Typage 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. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;
2. typage implicite, où les **types** des variables ne sont **pas mentionnés** dans le programme. Ils sont devinés lors de la compilation (si typage statique) ou lors de l'exécution (si typage dynamique) en fonction du contexte.

   Ce mécanisme s'appelle l'inférence des types.

Typage 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;
}
```

Typage 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.

Typage 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 typage explicite : le type de `p` a été fixé lors de la déclaration de `afficher`.

Typage implicite

Considérons le programme Caml

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

Typage 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.

Typage 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.

Typage 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

Typage explicite.

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

Avantages et inconvénients

Typage explicite.

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

Avantages et inconvénients

Typage explicite.

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

Typage 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

Typage explicite.

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

Typage 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 le typage est statique, la compilation peut être plus longue (il faut deviner les types). Si le typage 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 peut 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 peut 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())      # Lecture d’un entier.
if n == 0 :
    res = "a"         # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]   # ‘res‘ est une liste.
print res             # Tentative d’affichage de ‘res‘.
```

Portée dynamique

Considérons le programme Python

```
n = int(input())     # Lecture d'un entier.
if n == 0 :
    res = "a"        # `res` est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]  # `res` est une liste.
print res            # Tentative d'affichage de `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())     # Lecture d'un entier.
if n == 0 :
    res = "a"        # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]  # ‘res‘ est une liste.
print res            # Tentative d'affichage de ‘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())     # Lecture d'un entier.
if n == 0 :
    res = "a"        # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]  # ‘res‘ est une liste.
print res            # Tentative d'affichage de ‘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	T. dyn.	T. stat.	T. expl.	T. impl.	Imp.	Fonc.
C	Non	Oui	Oui	Non	Oui	Non
Python	Oui	Non	Non	Oui	Oui	Impur
Caml	Non	Oui	*Non*	Oui	Oui	Impur
Haskell	Non	Oui	Non	Oui	Non	Pur

Caractéristiques des principaux langages

Langage	T. dyn.	T. stat.	T. expl.	T. 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

Plan

2 Programmation
- Interpréteur Caml
- Liaisons
- Types de base
- Fonctions
- Entrées et sorties
- Compilation

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

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

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

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

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 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

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
     (* imbrique. *) *)
```

En Caml, ceci fonctionne.

La 1^{re} chose à apprendre

« La 1^{re} chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

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
   (* imbrique. *) *)
```

En Caml, ceci fonctionne.

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

En C, ceci ne fonctionne pas.

Plan

2 Programmation

- Interpréteur Caml
- Liaisons
- Types de base
- Fonctions
- Entrées et sorties
- Compilation

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 4.01.0

#
```

Interpréteur

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

```
        Objective Caml version 4.01.0

#
```

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 4.01.0

#
```

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

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

Lecture

Interpréteur

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

```
        Objective Caml version 4.01.0

#
```

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** ;

$$\begin{tikzcd}
\text{Lecture} \arrow[dr, bend left] & \\
& \text{Calcul}
\end{tikzcd}$$

Interpréteur

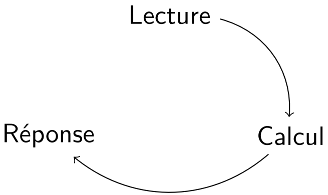
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.01.0

#
```

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

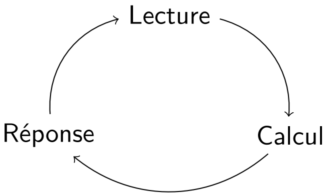
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.01.0

#
```

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

2 Programmation

- Interpréteur Caml
- Liaisons
- Types de base
- Fonctions
- Entrées et sorties
- Compilation

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 = EXP
```

où `ID` est un nom (identificateur) et `EXP` 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 = EXP`

où `ID` est un nom (identificateur) et `EXP` est une expression.

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

# n;;
- : int = 5
```

La 1re 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 = EXP`

où `ID` est un nom (identificateur) et `EXP` est une expression.

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

# n;;
- : int = 5
```

La 1re phrase lie au nom `n` la valeur `5`. L'interpréteur le signale en commençant sa réponse par `val n`.

La 2e 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 valeur 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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison 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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2^e^ occurrence de `n` a pour valeur `5` à cause de la liaison 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.

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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison 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 2e phrase est locale : le nom global `n` défini en 1re phrase reste inchangé.

Ceci explique la valeur du nom `n` de la 3e 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.

Dessiner les arbres syntaxiques.

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;;
```

Dessiner les arbres syntaxiques.

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.

Dessiner les arbres syntaxiques.

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.

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

Dessiner les arbres syntaxiques.

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.

```
# let s = let x = 3 in
      x * x;;
val s : int = 9
```

Le nom `s` est lié à la valeur `9`. En effet, l'expression `let x = 3 in x * x` a pour valeur `9`.

Dessiner les arbres syntaxiques.

Liaisons locales — exemples

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

Dessiner les arbres syntaxiques.

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.

Dessiner les arbres syntaxiques.

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.

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

Dessiner les arbres syntaxiques.

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.

```
# let x = let y = 2 in
      x + y;;
Error: Unbound value x
```

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.

Dessiner les arbres syntaxiques.

Liaisons simultanées

Il est possible de définir des noms simultanément dans une même phrase.

Liaisons simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

Liaisons simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

```
# 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 simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

```
# 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`.

```
# let x = 1 in
      let y = 3
      and z = 4 in
          x + y + z;;
- : int = 8
```

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;;
```

Dessiner les arbres syntaxiques.

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.

Dessiner les arbres syntaxiques.

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.

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

Dessiner les arbres syntaxiques.

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.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Dessiner les arbres syntaxiques.

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.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

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.

Dessiner les arbres syntaxiques.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
      x * x * x;;
```

Dessiner les arbres syntaxiques.

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.

Dessiner les arbres syntaxiques.

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.

Dessiner les arbres syntaxiques.

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.

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

Dessiner les arbres syntaxiques.

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.

```
# let x = 1
  and y = x + 1;;
Error: Unbound value x
```

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.

Dessiner les arbres syntaxiques.

Plan

Les six types de base

Nom du type	Utilisation
`int`	Représentation des **entiers signés**
`float`	Représentation des **nombres à virgule signés**
`char`	Représentation des **caractères**
`string`	Représentation des **chaînes de caractères**
`bool`	Représentation des **booléens**
`unit`	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
`not`	1	Non logique
`&&`	2	Et logique
`||`	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
`not`	1	Non logique
`&&`	2	Et logique
`||`	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
`not`	1	Non logique
`&&`	2	Et logique
`||`	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
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`prec`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

Opérations arithmétiques sur les `int`

Opérateur	Arité	Rôle
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`prec`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `prec` et `abs` sont des fonctions.

Opérations arithmétiques sur les `int`

Opérateur	Arité	Rôle
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`prec`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `prec` et `abs` sont des fonctions.

`mod` n'est pas une fonction, c'est un opérateur.

Opérations relationnelles sur les `int`

Opérateur	Arité	Rôle
`=`, `<>`	2	Égalité, Différence
`<`, `>`	2	Comparaison stricte
`<=`, `>=`	2	Comparaison large

Ces opérateurs produisent des valeurs de type `bool`.

Opérations relationnelles sur les `int`

Opérateur	Arité	Rôle
`=`, `<>`	2	Égalité, Différence
`<`, `>`	2	Comparaison stricte
`<=`, `>=`	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
`lnot`	1	Non bit à bit
`land`	2	Et bit à bit
`lor`	2	Ou bit à bit
`lxor`	2	Ou exclusif bit à bit
`lsl`, `lsr`	2	Décalage à gauche, droite bit à bit
`asr`	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
`lnot`	1	Non bit à bit
`land`	2	Et bit à bit
`lor`	2	Ou bit à bit
`lxor`	2	Ou exclusif bit à bit
`lsl`, `lsr`	2	Décalage à gauche, droite bit à bit
`asr`	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

$$-\texttt{max\_float} = -1.79769313486231571 \times 10^{308}$$

à

$$\texttt{max\_float} = 1.79769313486231571 \times 10^{308}.$$

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
abs_float	1	Valeur absolue
floor, ceil	1	Partie entière inf., sup.
sqrt	1	Racine carrée
log, exp	1	Logarithme népérien, exponentielle
cos, sin, tan	1	Fonctions trigonométriques
**	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
`abs_float`	1	Valeur absolue
`floor`, `ceil`	1	Partie entière inf., sup.
`sqrt`	1	Racine carrée
`log`, `exp`	1	Logarithme népérien, exponentielle
`cos`, `sin`, `tan`	1	Fonctions trigonométriques
`**`	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"
```

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`.

```
# 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"
```

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`.

```
# let u = "ab"
  and v = "ba" in
      u ^ v ^ u ^ v;;
- : string = "abbaabba"
```

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 `string`. On peut ainsi totalement comparer des chaînes de caractères.

Les opérateurs de comparaison considèrent 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 string. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;
- : bool = false
```

Les opérateurs de comparaison considèrent 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 `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 considèrent 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 `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 considèrent 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 `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
```

```
# "abc" < "ad";;
- : bool = true
```

Les opérateurs de comparaison considèrent 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

2 Programmation

- Interpréteur Caml
- Liaisons
- Types de base
- Fonctions
- Entrées et sorties
- Compilation

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

$$f : E_1 \times E_2 \times \cdots \times E_n \rightarrow S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

On appelle `n` l'arité de f (qui est son nombre d'entrées).

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

On appelle `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

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

On appelle `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 **valeur** 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 **valeur** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x, y) := x + y$.

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 **valeur** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x, y) := x + y$.

Les noms x et y 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 **valeur** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x, y) := x + y$.

Les noms x et y sont les paramètres de f.Lors de l'appel $f(2, 6)$, f est appelée avec les arguments 2 et 6.

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 **valeur** qui vient se substituer à un paramètre lors de l'appel à une fonction.

P.ex., soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x, y) := x + y$.

Les noms x et y sont les paramètres de f.Lors de l'appel $f(2, 6)$, f est appelée avec les arguments 2 et 6.

Lors d'un appel à une fonction f avec les arguments e_1, ..., e_n, on dit que l'on applique f à e_1, ..., e_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.

La fonction `ID` renvoie ainsi la valeur de `EXP`.

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.

La fonction `ID` renvoie ainsi la valeur de `EXP`.

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.

La fonction `ID` renvoie ainsi la valeur de `EXP`.

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.

La fonction `ID` renvoie ainsi la valeur de `EXP`.

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.

La fonction `ID` renvoie ainsi la valeur de `EXP`.

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 concat u =
      let aux u =
          u ^ "a" ^ u
      in
      (aux u) ^ (aux ("b" ^ u));;
val concat : 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 concat u =
      let aux u =
          u ^ "a" ^ u
      in
      (aux u) ^ (aux ("b" ^ u));;
val concat : string -> string = <fun>
```

```
# (concat "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 concat u =
      let aux u =
          u ^ "a" ^ u
      in
      (aux u) ^ (aux ("b" ^ u));;
val concat : string -> string = <fun>
```

```
# (concat "cd");;
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) simultanées :

```
# let f x =
      let g y =
          y + 1 = x
      and h x =
          2 * x
      in
      (g (h x));;
val f : int -> bool = <fun>
```

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.

Dessiner l'arbre syntaxique.

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'
```

Dessiner l'arbre syntaxique.

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.

Dessiner l'arbre syntaxique.

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.

Dessiner l'arbre syntaxique.

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).

Dessiner l'arbre syntaxique.

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
          ();;
val f : int -> unit = <fun>
```

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
          ();;
val f : int -> unit = <fun>
```

est équivalent à

```
# let f x =
      if x >= 9 then
          ()
      else
          ();;
val f : int -> unit = <fun>
```

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 `E1`, ..., `En` 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 `E1`, ..., `En` et `S` sont des types.

Plus précisément,

- pour tout `1` $\leqslant$ `i` $\leqslant$ `n`, `Ei` 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 `E1`, ..., `En` et `S` sont des types.

Plus précisément,

- pour tout `1` $\leqslant$ `i` $\leqslant$ `n`, `Ei` 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 `E1`, ..., `En` et `S` sont des types.

Plus précisément,

- pour tout $1 \leqslant i \leqslant n$, `Ei` est le type attendu du `i`ᵉ argument de `F` ;
- `S` est le type de retour de `F`.

`E1 -> ... -> En -> 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 `E1` et `E2` et à type de retour `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 -> E2 -> 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 -> E2 -> 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 -> E2 -> 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 -> 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 -> E2 -> 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 -> S` qui est donc une **fonction**.

La fonction `(f e1)` est ainsi une fonction qui se comporte comme `f` lorsque son 1er paramètre est fixé à la valeur `e1`.

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 -> E2 -> 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 -> S` qui est donc une **fonction**.

La fonction `(f e1)` est ainsi une fonction qui se comporte comme `f` lorsque son 1er paramètre est fixé à la valeur `e1`.

On dit que `(f e1)` 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 `e1`, ..., `ek` sont des valeurs de types respectifs `E1`, ..., `Ek` avec $\texttt{k} \leqslant \texttt{n}$,

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 $k \leqslant n$, 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 $f : \mathbb{Z}^3 \to \mathbb{Z}$ 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>
```

Cette fonction représente $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

```
# let distr_3 = (distr 3);;
val distr_3 : int -> int
    -> int = <fun>
```

Cette fonction représente $f_3 : \mathbb{Z}^2 \to \mathbb{Z}$ 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>
```

Cette fonction représente $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

```
# let distr_3 = (distr 3);;
val distr_3 : int -> int
    -> int = <fun>
```

Cette fonction représente $f_3 : \mathbb{Z}^2 \to \mathbb{Z}$ vérifiant $(b, c) \mapsto 3b + 3c$.

```
# let distr_3_5 = (distr_3 5);;
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente $f_{3,5} : \mathbb{Z} \to \mathbb{Z}$ 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>
```

Cette fonction représente $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

```
# let distr_3 = (distr 3);;
val distr_3 : int -> int
    -> int = <fun>
```

Cette fonction représente $f_3 : \mathbb{Z}^2 \to \mathbb{Z}$ vérifiant $(b, c) \mapsto 3b + 3c$.

```
# let distr_3_5 = (distr_3 5);;
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente $f_{3,5} : \mathbb{Z} \to \mathbb{Z}$ 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);;
val distr_3_5 : int -> int = <fun>
```

```
# let distr_3_5 = (distr 3 5);;
val distr_3_5 : int -> int = <fun>
```

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
```

Ceci définit une fonction anonyme appliquée d'emblée 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
```

Ceci définit une fonction anonyme appliquée d'emblée aux arguments `4` et `3`.

```
# let produit k =
      fun x -> x * k;;
val produit : int -> int -> int
    = <fun>
```

`(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

2 Programmation

- Interpréteur Caml
- Liaisons
- Types de base
- Fonctions
- **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 entrée** 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 entrée** par l'utilisateur, est leur seul intérêt.

En pratique, comme toute fonction possède au moins un paramètre, par convention, toute fonction de lecture accepte `()` en entrée. Une fonction qui ne possède pas de paramètre est une constante (ce qui n'est pas la nature d'une fonction de lecture).

Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur entrée** par l'utilisateur, est leur seul intérêt.

En pratique, comme toute fonction possède au moins un paramètre, par convention, toute fonction de lecture accepte `()` en entrée. Une fonction qui ne possède pas de paramètre est une constante (ce qui n'est pas la nature d'une fonction de lecture).

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;;
val div_decr : int -> int = <fun>
```

Plan

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
```

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.

Espaces de noms

Un espace de nom est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition.

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.

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.

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.

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.

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.

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.

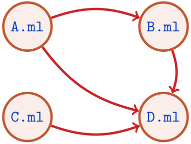
Ordre de compilation

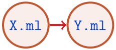
À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

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 :



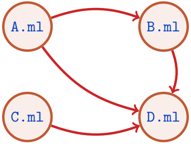
Toute flèche 

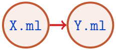
signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

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`.

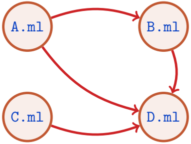
Dans cet exemple, on a les trois ordres suivants possibles :

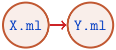
- `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

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`.

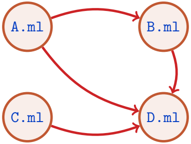
Calcul des dépendances automatisé

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`.

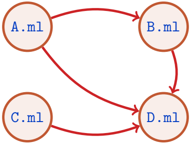
Reprenons le graphe d'inclusions



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



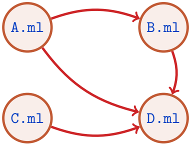
La commande `ocamldep *.ml` affiche

```
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

Plan

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
`->`	2	Flèche
`*`	$\geqslant 2$	Produit cartésien
`{}`	$\geqslant 1$	Produit nommé
`|`	$\geqslant 1$	Somme

Plan

Produit cartésien

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

désigne le type **produit cartésien** de `T1` et `T2`.

Produit cartésien

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

désigne le type **produit cartésien** de `T1` et `T2`.

Il contient pour valeurs les couples `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

Produit cartésien

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien** 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;;
type point = int * int
```

Le type `point` contient tous les couples à coordonnées entières.

Produit cartésien

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

désigne le type **produit cartésien** 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;;
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

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien** 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;;
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)
```

Produit cartésien

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type **produit cartésien** 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;;
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)
```

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

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1re coordonnée par `(fst c)` et à sa 2e coordonnée par `(snd c)`.

Accès aux coordonnées d'un couple

Si `c` est un couple, on accède à sa 1re coordonnée par `(fst c)` et à sa 2e 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 1re coordonnée par `(fst c)` et à sa 2e 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 1re coordonnée par `(fst c)` et à sa 2e 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 1re coordonnée par `(fst c)` et à sa 2e 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 ...
```

```
# 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>
# (conc_21 (("a", "b"), "c"));;
- : string = "acb"
```

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"
```

et

```
# 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>
# (conc_21 (("a", "b"), "c"));;
- : string = "acb"
```

et

```
# 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"
```

et

```
# 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"
```

et

```
# 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 `T1`, ..., `Tn`,

```
T1 * ... * Tn
```

désigne le type **produit cartésien généralisé** de `T1`, ..., `Tn`.

n-uplets

Étant donnés des types `T1`, ..., `Tn`,

```
T1 * ... * Tn
```

désigne le type **produit cartésien généralisé** de `T1`, ..., `Tn`.

Il contient pour valeurs les *n*-uplets `(e1, ..., en)` où pour tout `1` $\leqslant$ `i` $\leqslant$ `n`, `ei` est de type `Ti`.

n-uplets

Étant donnés des types `T1`, ..., `Tn`,

`T1 * ... * Tn`

désigne le type **produit cartésien généralisé** de `T1`, ..., `Tn`.

Il contient pour valeurs les *n-uplets* `(e1, ..., en)` où pour tout $1 \leqslant i \leqslant n$, `ei` est de type `Ti`.

```
# type c = int * unit * (int -> char);;
type c = int * unit * (int -> char)
```

n-uplets

Étant donnés des types `T1`, ..., `Tn`,

`T1 * ... * Tn`

désigne le type **produit cartésien généralisé** de `T1`, ..., `Tn`.

Il contient pour valeurs les *n*-uplets `(e1, ..., en)` où pour tout `1` $\leqslant$ `i` $\leqslant$ `n`, `ei` est de type `Ti`.

```
# type c = int * unit * (int -> char);;
type c = int * unit * (int -> char)
```

Un *n*-uplet s'écrit

`(e1, ..., en)`

Les parenthèses sont facultatives.

n-uplets

Étant donnés des types `T1`, ..., `Tn`,

`T1 * ... * Tn`

désigne le type **produit cartésien généralisé** de `T1`, ..., `Tn`.

Il contient pour valeurs les n-uplets `(e1, ..., en)` où pour tout $1 \leqslant i \leqslant n$, `ei` est de type `Ti`.

```
# type c = int * unit * (int -> char);;
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 `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type **produit nommé** de `T1`, ..., `Tn`.

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`.

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};;
type personne = { nom : string; age : int; }
```

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};;
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`.

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};;
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`

```
# 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
```

```
# 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 » de 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 » de 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 » de 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 » de 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};;
type point = { a : int; b : int; c : int; }
```

« Modification » de 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};;
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 » de 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};;
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 » de 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};;
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};;
# type t2 = {a : int ; c : char};;
type t2 = { a : int; c : char; }
# let f x = x.a;;
val f : t2 -> int = <fun>
```

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};;
# 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'}
# {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 de `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 de `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 de `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 de `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

3 Types

- L'algèbre des types
- Types produit
- Types somme
- Types paramétrés

Somme

Étant donnés des identificateurs `Id1`, …, `Idn` dont les **1res lettres sont des majuscules**,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, …, `Idn`.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les **1res lettres sont des majuscules**,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les **1res lettres sont des majuscules**,

```
Id1 | ... | Idn
```

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `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 `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
type numero = Un | Deux | Trois
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les **1res lettres sont des majuscules**,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
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 **1res lettres sont des majuscules**,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
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 **1res lettres sont des majuscules**,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
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 **1res lettres sont des majuscules**,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
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
```

Somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les **1res lettres sont des majuscules**,

`Id1 | ... | Idn`

désigne le type **somme** de `Id1`, ..., `Idn`.

Il contient `n` valeurs : `Id1`, ..., `Idn`.

On appelle `Id1`, ..., `Idn` des constructeurs.

```
# type numero = Un | Deux | Trois;;
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;;
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;;
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;;
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;;
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;;
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 — 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 — 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 **récursif** suivant :

```
# type liste_int = Vide | Cellule of int * liste_int;;
type liste_int = Vide | Cellule of int * liste_int
```

Exemple — 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 **récursif** suivant :

```
# type liste_int = Vide | Cellule of int * liste_int;;
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 — 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 **récursif** suivant :

```
# type liste_int = Vide | Cellule of int * liste_int;;
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 — 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 **récursif** suivant :

```
# type liste_int = Vide | Cellule of int * liste_int;;
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 — 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 **récursif** suivant :

```
# type liste_int = Vide | Cellule of int * liste_int;;
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 — 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 — 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 **récursif** suivant :

```
# type arbre_b_int =
    Vide |
    Noeud of arbre_b_int * int * arbre_b_int;;
type arbre_b_int = Vide | Noeud of arbre_b_int * int * arbre_b_int
```

Exemple — 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 **récursif** suivant :

```
# type arbre_b_int =
    Vide |
    Noeud of arbre_b_int * int * arbre_b_int;;
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 — 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 **récursif** suivant :

```
# type arbre_b_int =
    Vide |
    Noeud of arbre_b_int * int * arbre_b_int;;
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 — 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 **récursif** suivant :

```
# type arbre_b_int =
    Vide |
    Noeud of arbre_b_int * int * arbre_b_int;;
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));;
  Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide)), 5,
   Noeud (Vide, 1, Vide))
```

Exemple — 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.

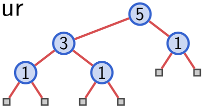
Cette définition se traduit en le type somme **récursif** suivant :

```
# type arbre_b_int =
    Vide |
    Noeud of arbre_b_int * int * arbre_b_int;;
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));;
  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 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- l'arbre vide ou bien

Exemple — 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 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

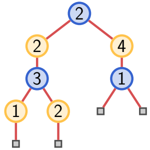
- l'arbre vide ou bien
- un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 ou bien
- un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

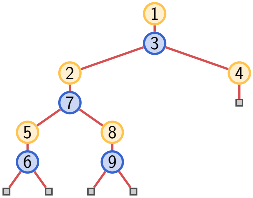
Exemple — arbres unaires binaires d'entiers

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 — arbres unaires binaires d'entiers

La définition de ces objets se traduit en les définitions de types suivantes.

Exemple — 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;;
type arbre_1 = Vide1 | Noeud1 of int * arbre_2
and arbre_2 = Vide2 | Noeud2 of arbre_1 * int * arbre_1
```

Exemple — 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;;
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 — 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;;
type arbre_1 = Vide1 | Noeud1 of int * arbre_2
and arbre_2 = Vide2 | Noeud2 of arbre_1 * int * arbre_1
```

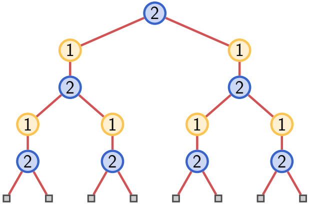
Le mot-clé `and` permet de réaliser des définitions de types simultanées : les types `arbre_1` et `arbre_2` sont en effet mutuellement récursifs.

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;;
type arbre_12 = Vide | Arbre1 of arbre_1 | Arbre2 of arbre_2
```

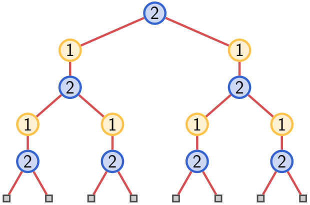
Exemple — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



Exemple — arbres unaires binaires d'entiers

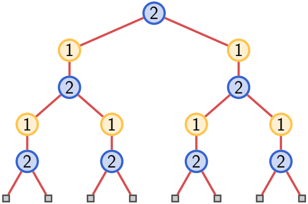
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
```

Exemple — arbres unaires binaires d'entiers

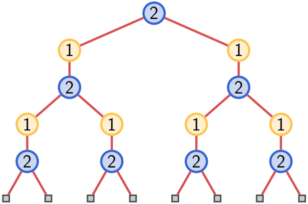
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
```

Exemple — arbres unaires binaires d'entiers

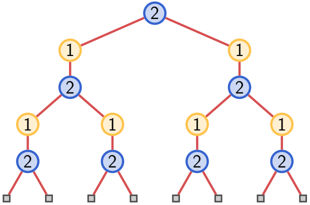
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
          let a22 = (Noeud2 (a1, 2, a1)) in
```

Exemple — arbres unaires binaires d'entiers

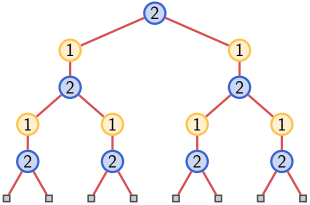
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# 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 — arbres unaires binaires d'entiers

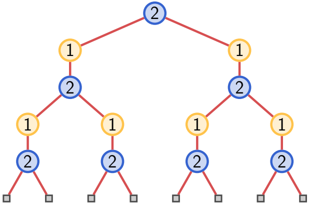
Écrivons une expression de type arbre_12 qui représente l'arbre ci-contre :




```
# 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 — 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

Paramètres dans les types

Exactement de la même manière qu'il est possible de définir des fonctions à plusieurs paramètres, il est possible de définir des types avec des paramètres.

Paramètres dans les types

Exactement de la même manière qu'il est possible de définir des fonctions à plusieurs paramètres, il est possible de définir des types avec des paramètres.

On parle alors de types paramétrés.

Paramètres dans les types

Exactement de la même manière qu'il est possible de définir des fonctions à plusieurs paramètres, il est possible de définir des types avec des paramètres.

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

Exactement de la même manière qu'il est possible de définir des fonctions à plusieurs paramètres, il est possible de définir des types avec des paramètres.

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

Exactement de la même manière qu'il est possible de définir des fonctions à plusieurs paramètres, il est possible de définir des types avec des paramètres.

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 `'P1`, ..., `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, 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 `'P1`, ..., `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, 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 `'P1`, . . ., `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, 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 `'P1`, ..., `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, 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`, ..., `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, 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 — listes génériques

Le type à un paramètre

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

Exemple — listes génériques

Le type à un paramètre

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a 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 — listes génériques

Le type à un paramètre

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a 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 — listes génériques

Le type à un paramètre

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a 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 — listes génériques

Le type à un paramètre

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a 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 — 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 — 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};;
type 'a couple_autre = { x : 'a; y : 'a; }
```

dans lequel les deux coordonnées doivent être d'un même type.

Exemple — 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};;
type 'a couple_autre = { x : 'a; y : 'a; }
```

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 — 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};;
type 'a couple_autre = { x : 'a; y : 'a; }
```

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 — 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};;
type 'a couple_autre = { x : 'a; y : 'a; }
```

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 — 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};;
type 'a couple_autre = { x : 'a; y : 'a; }
```

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 — 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 — 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 — 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.

```
# type ('u, 'x, 'y) arbre_1 =
    Vide1 |
    Noeud1 of 'u * ('u, 'x, 'y) arbre_2
```

Exemple — 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.

```
# 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 — 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.

```
# 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 ('a, 'b, 'c) arbre_1 = Vide1 | Noeud1 of 'a * ('a, 'b, 'c) arbre_2
and ('a, 'b, 'c) arbre_2 =
    Vide2
  | Noeud2 of ('a, 'b, 'c) arbre_1 * ('b * 'c) * ('a, 'b, 'c) arbre_1
```

Exemple — 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.

```
# 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 ('a, 'b, 'c) arbre_1 = Vide1 | Noeud1 of 'a * ('a, 'b, 'c) arbre_2
and ('a, 'b, 'c) arbre_2 =
    Vide2
  | Noeud2 of ('a, 'b, 'c) arbre_1 * ('b * 'c) * ('a, 'b, 'c) arbre_1

# type ('u, 'x, 'y) arbre_12 =
    Vide |
    Arbre1 of ('u, 'x, 'y) arbre_1 |
    Arbre2 of ('u, 'x, 'y) arbre_2;;
```

Exemple — 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.

```
# 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 ('a, 'b, 'c) arbre_1 = Vide1 | Noeud1 of 'a * ('a, 'b, 'c) arbre_2
and ('a, 'b, 'c) arbre_2 =
    Vide2
  | Noeud2 of ('a, 'b, 'c) arbre_1 * ('b * 'c) * ('a, 'b, 'c) arbre_1

# type ('u, 'x, 'y) arbre_12 =
    Vide |
    Arbre1 of ('u, 'x, 'y) arbre_1 |
    Arbre2 of ('u, 'x, 'y) arbre_2;;
type ('a, 'b, 'c) arbre_12 =
    Vide
  | Arbre1 of ('a, 'b, 'c) arbre_1
  | Arbre2 of ('a, 'b, 'c) arbre_2
```

Plan

4 Notions
- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

Plan

4 Notions
- Récursivité
- 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 rec x =
      let x = 20 in
          x + x;;
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
- : int = 30
```

```
# 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;;
val x : int = 40
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
- : int = 30
```

```
# 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;;
val x : int = 40
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
- : int = 30
```

```
# 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 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 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));;
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 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));;
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 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));;
val f : int -> int = <fun>
# (f 3);;
- : int = 9
```

```
# 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 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 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 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 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 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 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 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 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 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) $\rightarrow$ (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) $\rightarrow$ (un 3) $\rightarrow$ (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)` :

$$\texttt{(zero 4)} \rightarrow \texttt{(un 3)} \rightarrow \texttt{(deux 2)} \rightarrow \texttt{(zero 1)}$$

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)` :

$$\begin{aligned}(\texttt{zero } 4) &\to (\texttt{un } 3) \to (\texttt{deux } 2) \\ &\to (\texttt{zero } 1) \to (\texttt{un } 0)\end{aligned}$$

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)` :

$$\begin{aligned}(\texttt{zero } 4) &\rightarrow (\texttt{un } 3) \rightarrow (\texttt{deux } 2)\\ &\rightarrow (\texttt{zero } 1) \rightarrow (\texttt{un } 0)\\ &\rightarrow \texttt{"un"}.\end{aligned}$$

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 **:**
    $\mathbf{I}$
Fin

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 **:**
    I
Fin

Fonction `rec` f **:**
    Si C **:**
        I
        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 = 0 then
            0
        else
            i + (aux (i - 1))
    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 = 0 then
            0
        else
            i + (aux (i - 1))
    in
    (aux n);;
```

Boucle `for` simple :

Fonction C

```
int somme_pairs(int n) {
    int i, res;
    res = 0;
    for (i = 0 ; i <= n ; i += 2) {
        res += i;
    }
    return res;
}
```

Fonction Caml

```
let somme_pairs n =
    let rec aux i =
        if i > n then
            0
        else
            i + (aux (i + 2))
    in
    (aux 0);;
```

Récursivité 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);;
```

Récursivité 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

Récursivité 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)`

Récursivité 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

$$(\texttt{fact}\ 4) \quad \rightarrow \quad 4 * (\texttt{fact}\ 3)$$

Récursivité 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

$$(\texttt{fact}\ 4) \quad \to \quad 4 * (\texttt{fact}\ 3) \quad \to \quad 4 * 3 * (\texttt{fact}\ 2)$$

Récursivité 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

$$
\begin{aligned}
(\texttt{fact}\ 4) \quad &\to \quad 4 * (\texttt{fact}\ 3) \quad \to \quad 4 * 3 * (\texttt{fact}\ 2) \\
&\to \quad 4 * 3 * 2 * (\texttt{fact}\ 1)
\end{aligned}
$$

Récursivité 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

$$\begin{array}{lllll} (\texttt{fact}\ 4) & \to & 4 * (\texttt{fact}\ 3) & \to & 4 * 3 * (\texttt{fact}\ 2) \\ & \to & 4 * 3 * 2 * (\texttt{fact}\ 1) & \to & 4 * 3 * 2 * 1 \end{array}$$

Récursivité 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

$$
\begin{aligned}
(\texttt{fact}\ 4) \quad & \to \quad 4 * (\texttt{fact}\ 3) \quad \to \quad 4 * 3 * (\texttt{fact}\ 2) \\
& \to \quad 4 * 3 * 2 * (\texttt{fact}\ 1) \quad \to \quad 4 * 3 * 2 * 1 \\
& \rightsquigarrow \quad 24
\end{aligned}
$$

Récursivité 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

$$\begin{aligned}(\mathtt{fact}\ 4) &\rightarrow 4 * (\mathtt{fact}\ 3) \rightarrow 4 * 3 * (\mathtt{fact}\ 2) \\ &\rightarrow 4 * 3 * 2 * (\mathtt{fact}\ 1) \rightarrow 4 * 3 * 2 * 1 \\ &\rightsquigarrow 24\end{aligned}$$

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.

Récursivité 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);;
```

Récursivité 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

Récursivité 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)

Récursivité 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

$$(\texttt{fact}\ 4\ 1) \quad \rightarrow \quad (\texttt{fact}\ 3\ (4 * 1))$$

Récursivité 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

$$(\texttt{fact}\ 4\ 1) \quad \rightarrow \quad (\texttt{fact}\ 3\ (4 * 1)) \quad \rightsquigarrow \quad (\texttt{fact}\ 3\ 4)$$

Récursivité 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

$$
\begin{array}{lcccc}
(\mathtt{fact}\ 4\ 1) & \to & (\mathtt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\mathtt{fact}\ 3\ 4) \\
 & \to & (\mathtt{fact}\ 2\ (3 * 4)) & &
\end{array}
$$

Récursivité 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

$$
\begin{array}{lllll}
(\texttt{fact}\ 4\ 1) & \rightarrow & (\texttt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\texttt{fact}\ 3\ 4) \\
 & \rightarrow & (\texttt{fact}\ 2\ (3 * 4)) & \rightsquigarrow & (\texttt{fact}\ 2\ 12)
\end{array}
$$

Récursivité 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

$$\begin{array}{lllll}
(\texttt{fact}\ 4\ 1) & \rightarrow & (\texttt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\texttt{fact}\ 3\ 4) \\
 & \rightarrow & (\texttt{fact}\ 2\ (3 * 4)) & \rightsquigarrow & (\texttt{fact}\ 2\ 12) \\
 & \rightarrow & (\texttt{fact}\ 1\ (2 * 12)) & &
\end{array}$$

Récursivité 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

$$
\begin{array}{lclcl}
(\texttt{fact}\ 4\ 1) & \rightarrow & (\texttt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\texttt{fact}\ 3\ 4) \\
 & \rightarrow & (\texttt{fact}\ 2\ (3 * 4)) & \rightsquigarrow & (\texttt{fact}\ 2\ 12) \\
 & \rightarrow & (\texttt{fact}\ 1\ (2 * 12)) & \rightsquigarrow & (\texttt{fact}\ 1\ 24)
\end{array}
$$

Récursivité 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

$$
\begin{array}{lllll}
(\texttt{fact}\ 4\ 1) & \rightarrow & (\texttt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\texttt{fact}\ 3\ 4) \\
 & \rightarrow & (\texttt{fact}\ 2\ (3 * 4)) & \rightsquigarrow & (\texttt{fact}\ 2\ 12) \\
 & \rightarrow & (\texttt{fact}\ 1\ (2 * 12)) & \rightsquigarrow & (\texttt{fact}\ 1\ 24) \\
 & \rightarrow & 24 & &
\end{array}
$$

Récursivité 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

$$
\begin{array}{lllll}
(\texttt{fact}\ 4\ 1) & \to & (\texttt{fact}\ 3\ (4 * 1)) & \rightsquigarrow & (\texttt{fact}\ 3\ 4) \\
 & \to & (\texttt{fact}\ 2\ (3 * 4)) & \rightsquigarrow & (\texttt{fact}\ 2\ 12) \\
 & \to & (\texttt{fact}\ 1\ (2 * 12)) & \rightsquigarrow & (\texttt{fact}\ 1\ 24) \\
 & \to & 24 & &
\end{array}
$$

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.

Récursivité terminale

La 2e version de la fonction `fact` possède la propriété d'être récursive terminale : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

Récursivité terminale

La 2ᵉ version de la fonction `fact` possède la propriété d'être récursive terminale : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1ʳᵉ version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

Récursivité terminale

La 2e version de la fonction `fact` possède la propriété d'être récursive terminale : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1re version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Récursivité terminale

La 2e version de la fonction `fact` possède la propriété d'être récursive terminale : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1re version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Le calcul de la 1re version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2e ne travaille que sur une expression de taille $\Theta(1)$.

Récursivité terminale

La 2e version de la fonction `fact` possède la propriété d'être récursive terminale : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1re version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Le calcul de la 1re version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2e ne travaille que sur une expression de taille $\Theta(1)$.

Les fonctions récursives terminales utilisent **moins de mémoire** que leurs analogues non récursives terminales. Elles sont donc à préférer.

Récursivité 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.

Récursivité 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.

Récursivité 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, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

Récursivité 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, 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));;
```

Récursivité 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, 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);;
```

Récursivité 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.

Récursivité 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).

Récursivité 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 $\mathtt{n}^{\text{e}}$ nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

```
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 récursive terminale de la fonction précédente.

Récursivité 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).

```
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 récursive terminale de la fonction précédente.

Elle utilise deux accumulateurs (à cause du double appel récursif précédent). `acc1` contient la valeur du `n - 1`^{e} nombre de Fibonacci et `acc2` contient la valeur du `n - 2`^{e} nombre de Fibonacci.

Récursivité terminale — forme générale

Une fonction récursive terminale a pour forme générale

Fonction `rec` *f*(*x1*, ..., *xn*, *acc1*, ..., *accm*) **:**
    Si *C* **:**
        f(`maje`(*x1*, ..., *xn*), `majs`(*acc1*, ..., *accm*))
    Sinon :
        R
    Fin
Fin

où

- *x1*, ..., *xn* sont les paramètres (entrées) ;
- *acc1*, ..., *accm* sont les accumulateurs (sorties) ;
- *R* est une expression résultat obtenue à partir des accumulateurs ;
- `maje` indique la mise à jour des *x1*, ..., *xn* lors de l'appel récursif ;
- `majs` indique la mise à jour des *acc1*, ..., *accm* lors de l'appel récursif.

Récursivité 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.

Récursivité terminale — dérécursivation

La dérécursivation est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Voici une fonction récursive terminale dans sa forme générale et sa version dérécursivée :

```
Fonction rec  f( x1, ..., xn,
       acc1, ..., accm)  :
    Si  C  :
        f(maje( x1, ..., xn),
              majs( acc1, ..., accm))
    Sinon :
        R
    Fin
Fin
```

```
Fonction it  g( x1, ..., xn,
       acc1, ..., accm)  :
    Tant que  C  :
        ( x1, ..., xn)
            := maje( x1, ..., xn)
        ( acc1, ..., accm)
            := majs( acc1, ..., accm)
    Fin
    R
Fin
```

Les notations sont ici les mêmes que celles utilisées précédemment.

Récursivité 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);;
```

Récursivité 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;;
```

Récursivité 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

4 Notions

- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

Exemple introductif

Considérons le type

```
type point =
    |Seg of int
    |Plan of int * int
    |Espace of int * int * int;;
```

Exemple introductif

Considérons le type

```
type point =
    |Seg 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 =
    |Seg 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
        |(Seg x) -> (Seg (-x))
        |(Plan (x, y)) -> (Plan (-x, -y))
        |(Espace (x, y, z)) -> (Espace (-x, -y, -z));;
```

Exemple introductif

Considérons le type

```
type point =
    |Seg 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
        |(Seg x) -> (Seg (-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 =
    |Seg 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
        |(Seg x) -> (Seg (-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 1re 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$^{\text{re}}$ 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 1re 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
        |(Seg 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 1re 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
           |(Seg 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 1re 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
        |(Seg 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 premiere_coordonnee p =
    match p with
        |(Seg 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
          |(Seg 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
          |(Seg 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
          |(Seg 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 une clause

`MOTIF -> EXP`.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à une clause

`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 une clause

`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 une clause

`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
        |(Seg _) -> 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
        |(Seg 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
        |(Seg x) -> x
        |(Plan (x, y)) -> x + y
        |(Espace (x, y, z)) ->
            x + y + z;;
```

Attention : ici l'occurrence de `n` dans le 2ᵉ 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
        |(Seg 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
        |(Seg 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
        |(Seg 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

$$f := (\neg P) \land (P \lor R)$$

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

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

`E1 -> E2 -> ... -> En -> S`,

toute application partielle

`(f e1 e2 ... ek)`

avec $1 \leqslant k \leqslant n - 1$ produit une valeur de type

`Ek+1 -> ... -> En -> S`

qui est donc une fonction.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

```
E1 -> E2 -> ... -> En -> S,
```

toute application partielle

```
(f e1 e2 ... ek)
```

avec $1 \leqslant k \leqslant n - 1$ produit une valeur de type

```
Ek+1 -> ... -> En -> 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**.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

`E1 -> E2 -> ... -> En -> S`,

toute application partielle

`(f e1 e2 ... ek)`

avec $1 \leqslant k \leqslant n - 1$ produit une valeur de type

`Ek+1 -> ... -> En -> 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**.

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 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

$$\texttt{(int -> int) -> int -> int -> int},$$

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`$^{\text{e}}$ de `f` sur l'entier `x`, c.-à-d.,

$$(\underbrace{\mathtt{f} \circ \cdots \circ \mathtt{f}}_{\mathtt{n} \text{ fois}})(\mathtt{x})$$

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

$$\texttt{(int -> int) -> int -> int -> int},$$

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`$^{\text{e}}$ de `f` sur l'entier `x`, c.-à-d.,

$$(\underbrace{\mathtt{f} \circ \cdots \circ \mathtt{f}}_{\mathtt{n}\text{ fois}})(\mathtt{x})$$

```
# (appli_repetee (fun x -> 2 * x) 3 4);;
- : int = 48
```

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"
```

Exemple complet : 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 : 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

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Exemple complet : 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

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

Exemple complet : 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

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Exemple complet : 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

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \ldots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Question : comment représenter des séries génératrices ?

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

Exemple complet : 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 : 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 : 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 : 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 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\ ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k\ ;$$

3. **produit** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}t^k.$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k \;;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k \;;$$

3. **produit** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}t^k.$$

Il est possible de les implanter simplement en utilisant les fonctions d'ordre supérieur.

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

Exemple complet : 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 : 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 : 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 : 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 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction 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 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction 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 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction 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 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction 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

4 Notions

- Récursivité
- 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 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re 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 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
`snd : 'a * 'b -> 'b`	Renvoie la 2e 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 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e 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 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e 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$^{\text{er}}$ paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2$^{\text{e}}$ paramètre de type `'e` est l'unité 1 ;
- le 3$^{\text{e}}$ paramètre de type `'e` est l'élément `x` ;
- le 4$^{\text{e}}$ 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 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n`.

Le type de retour est `'e`.

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 : 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 2e paramètre dans le 1er.

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 2e paramètre dans le 1er.

```
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 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant 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 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e 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 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e 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 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e 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

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
```

```
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
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    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.

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) :
    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
```

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) :
    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
```

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))
```

$\leadsto$ `(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 `a1`, ..., `an`, à **substituer** les `ai` **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 recopiés 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 recopiés 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 recopiés 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 recopiés 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 recopiés 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 recopiés 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 recopiés 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 mémoïsée de l'appel par nom**.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

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.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

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

Plan

5 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 infixe d'arité deux.

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
```

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 1er 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 1er 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 1er 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$^{\text{er}}$ 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$^{\text{er}}$ 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 standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes.

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire 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. **reconstruire** une liste en une autre.

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. **[reconstruction]** tri d'une liste, image mirroir 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. **[reconstruction]** tri d'une liste, image mirroir 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'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
2. d'une opération binaire associative $\times$;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
2. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
2. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `lst` est une liste d'entiers et que $\times$ 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'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
2. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `lst` est une liste d'entiers et que $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `lst` est une liste de chaînes de caractères et $\times$ 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'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
2. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

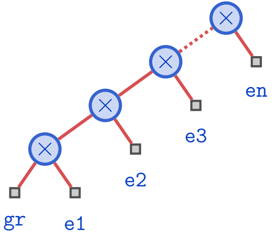
$$\texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `lst` est une liste d'entiers et que $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;
- lorsque `lst` est une liste dont les éléments sont comparables et $\times$ est la fonction qui renvoie **le plus grand** de ses deux arguments, ceci calcule la plus grande valeur de `lst`.

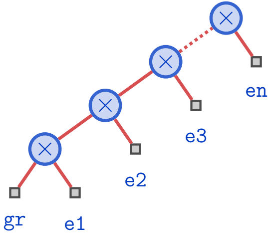
Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Pliage à gauche

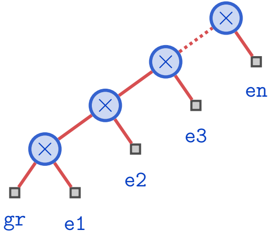
En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



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 $\times$, on souhaite évaluer l'arbre syntaxique



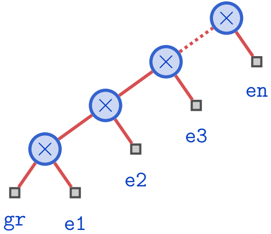
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



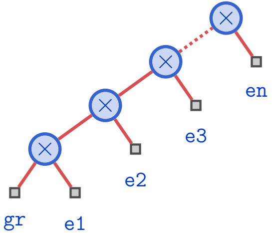
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;
2. la graine `gr` ;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, 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 $\times$;
2. la graine `gr` ;
3. la liste `lst` des opérandes.

Pliage à gauche

L'opération $\times$ 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 $\times$ 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 $\times$ 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 $\times$ 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 $\times$ 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 $\times$ 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 :

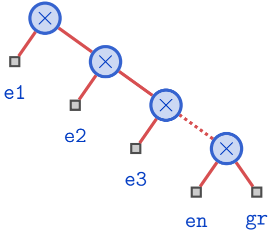
```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11
```

```
# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5
```

```
# (pliage_gauche (^) "" ["re" ; "la" ; "re" ; "sol" ; "si" ; "mi"]);;
- : string = "relaresolsimi"
```

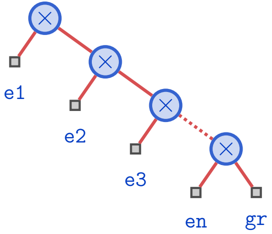
Pliage à droite

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) :

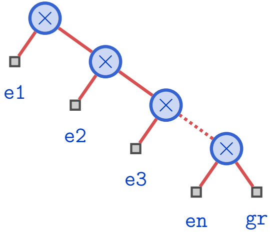


L'opération de pliage à droite est ainsi de type

```
('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 $\times$ 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 $\times$ 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 **récursif terminal** alors que le pliage à droite ne l'est pas.

Plan

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 d'avantage 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 d'avantage 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 d'avantage 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 d'avantage 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 1re créé une liste (trois cellules) en mémoire :

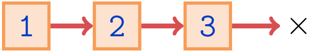
Le cas des listes

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 1re créé une liste (trois cellules) en mémoire :



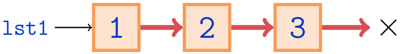
Le cas des listes

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$^{\text{re}}$ créé une liste (trois cellules) en mémoire :



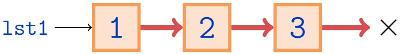
Le cas des listes

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} créé une liste (trois cellules) en mémoire :



La 2^{e} ne créé qu'une seule cellule et partage les trois précédentes :

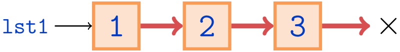
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

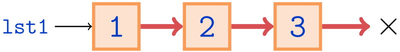
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :



Le cas des listes

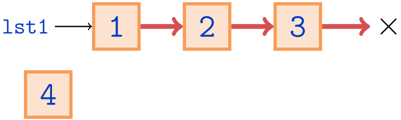
Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :

La 2e ne créé qu'une seule cellule et partage les trois précédentes :



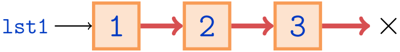
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

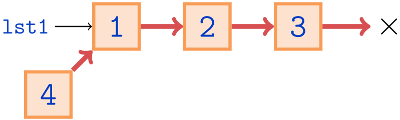
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :



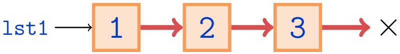
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

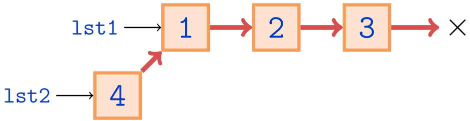
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé 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

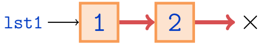
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

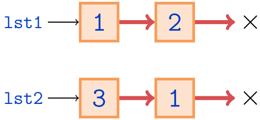
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

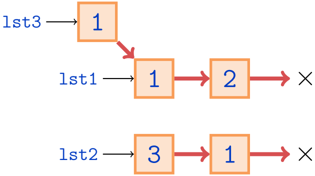
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

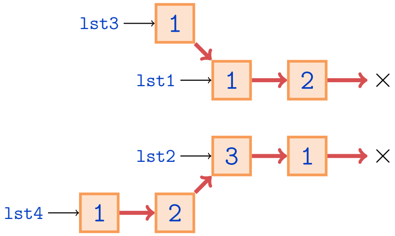
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

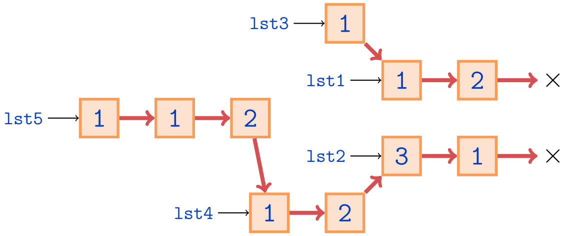
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

Files

On souhaite implanter les files (piles First In, First Out) dont les éléments sont d'un type quelconque.

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` ;

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 (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 ;

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 ;

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 renvoyer et supprimer 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 renvoyer et supprimer 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 renvoie/supprime la tête de out.

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`		
`ajout(2)`		
`ajout(3)`		
`ancien()`		
`ajout(4)`		
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`		
`ajout(3)`		
`ancien()`		
`ajout(4)`		
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`		
`ancien()`		
`ajout(4)`		
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`		
`ajout(4)`		
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	[]	[]
`ajout(1)`	[1]	[]
`ajout(2)`	[2 ; 1]	[]
`ajout(3)`	[3 ; 2 ; 1]	[]
`ancien()`	[]	[1 ; 2 ; 3]
`ajout(4)`		
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`		
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`		
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`	`[5 ; 4]`	`[2 ; 3]`
`ancien()`		
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`	`[5 ; 4]`	`[2 ; 3]`
`ancien()`	`[5 ; 4]`	`[2 ; 3]`
`supprimer()`		
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	[]	[]
`ajout(1)`	[1]	[]
`ajout(2)`	[2 ; 1]	[]
`ajout(3)`	[3 ; 2 ; 1]	[]
`ancien()`	[]	[1 ; 2 ; 3]
`ajout(4)`	[4]	[1 ; 2 ; 3]
`ajout(5)`	[5 ; 4]	[1 ; 2 ; 3]
`supprimer()`	[5 ; 4]	[2 ; 3]
`ancien()`	[5 ; 4]	[2 ; 3]
`supprimer()`	[5 ; 4]	[3]
`ajouter(6)`		
`supprimer()`		
`supprimer()`		

Implantation astucieuse — exemple

Opération	in	out
`vide`	`[]`	`[]`
`ajout(1)`	`[1]`	`[]`
`ajout(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`	`[5 ; 4]`	`[2 ; 3]`
`ancien()`	`[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(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`	`[5 ; 4]`	`[2 ; 3]`
`ancien()`	`[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(2)`	`[2 ; 1]`	`[]`
`ajout(3)`	`[3 ; 2 ; 1]`	`[]`
`ancien()`	`[]`	`[1 ; 2 ; 3]`
`ajout(4)`	`[4]`	`[1 ; 2 ; 3]`
`ajout(5)`	`[5 ; 4]`	`[1 ; 2 ; 3]`
`supprimer()`	`[5 ; 4]`	`[2 ; 3]`
`ancien()`	`[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

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 celle d'ensemble).

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 celle d'ensemble).

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 celle d'ensemble).

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 celle d'ensemble).

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 celle d'ensemble).

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

6 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, *etc.*,

λ-termes

Un λ-terme est

1. une **variable** x, y, z, t, *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, *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** $s\,t$ où s et t sont des λ-termes.

λ-termes

Un λ-terme est

1. une **variable** x, y, z, t, *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** $s\,t$ 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, *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** $s\,t$ 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, *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** $s\,t$ 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

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''.

P.ex.,

- l'arbre syntaxique de $(\lambda x.((x\, y)\, x))\, (\lambda y.\lambda x.y)$ est

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''.

P.ex.,

- l'arbre syntaxique de $(\lambda x.((x\,y)\,x))\,(\lambda y.\lambda x.y)$ est *dessiner*,

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''.

P.ex.,

- l'arbre syntaxique de $(\lambda x.((x\,y)\,x))\,(\lambda y.\lambda x.y)$ est *dessiner*,
- l'arbre syntaxique de $\lambda x.((y\,x)\,(\lambda y.(x\,y)))$ est

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''.

P.ex.,

- l'arbre syntaxique de $(\lambda x.((x\,y)\,x))\,(\lambda y.\lambda x.y)$ est *dessiner*,
- l'arbre syntaxique de $\lambda x.((y\,x)\,(\lambda y.(x\,y)))$ est *dessiner*.

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 ;

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 ;
- 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 ;
- sinon, on s'arrête.

Finalement, si f est liée à une 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

Voici les propriétés libres / liées des occurrences des variables de

$$\lambda x.\lambda x.((x\,(\lambda x.x))\,(y\,(\lambda y.y))).$$

Dessiner l'arbre syntaxique.

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.,

- $\lambda z.(x\, y) \;\star_x\; z\, z$

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.,

- $\lambda z.(x\, y) \star_x z\, z = \lambda z.((z\, z)\, y)$,

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.,

- $\lambda z.(x\, y) \star_x z\, z = \lambda z.((z\, z)\, y)$,
- $\lambda x.(x\, y) \star_x z\, z$

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.,

- $\lambda z.(x\, y) \ \star_x \ z\, z = \lambda z.((z\, z)\, y)$,
- $\lambda x.(x\, y) \ \star_x \ z\, z = \lambda x.(x\, y)$,

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.,

- $\lambda z.(x\, y) \;\star_x\; z\, z = \lambda z.((z\, z)\, y)$,
- $\lambda x.(x\, y) \;\star_x\; z\, z = \lambda x.(x\, y)$,
- $(z\, (\lambda z.(y\, z)))\, z \;\star_z\; \lambda x.(x\, x)$

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.,

- $\lambda z.(x\, y) \ \star_x \ z\, z = \lambda z.((z\, z)\, y)$,
- $\lambda x.(x\, y) \ \star_x \ z\, z = \lambda x.(x\, y)$,
- $(z\,(\lambda z.(y\, z)))\, z \ \star_z \ \lambda x.(x\, x) = ((\lambda x.(x\, x))\,(\lambda z.(y\, z)))\,(\lambda x.(x\, x))$.

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 x, on doit α-renommer x en une variable x' dans u, où x' est une 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 x, on doit α-renommer x en une variable x' dans u, où x' est une variable qui n'admet pas d'occurrence libre dans v.

On note $t \rightarrow_{\beta'} t'$ 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

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \to_{\beta'} z\,y$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \to_{\beta'} z\,y$

- $(\lambda x.\lambda y.((x\,x)\,(\lambda x.x)))\,(\lambda y.(y\,x))$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \to_{\beta'} z\,y$

- $(\lambda x.\lambda y.((x\,x)\,(\lambda x.x)))\,(\lambda y.(y\,x)) \to_{\beta'} \lambda y.(((\lambda y.(y\,x))\,(\lambda y(y\,x)))\,(\lambda x.x))$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \to_{\beta'} z\,y$

- $(\lambda x.\lambda y.((x\,x)\,(\lambda x.x)))\,(\lambda y.(y\,x)) \to_{\beta'} \lambda y.(((\lambda y.(y\,x))\,(\lambda y(y\,x)))\,(\lambda x.x))$

- $(\lambda x.\lambda y.x)\,(z\,y)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\, z)) \to_{\beta'} \lambda x.(x\, z)$

- $(\lambda x.(x\, y))\, z \to_{\beta'} z\, y$

- $(\lambda x.\lambda y.((x\, x)\,(\lambda x.x)))\,(\lambda y.(y\, x)) \to_{\beta'} \lambda y.(((\lambda y.(y\, x))\,(\lambda y(y\, x)))\,(\lambda x.x))$

- $(\lambda x.\lambda y.x)\,(z\, y) \not\to_{\beta'} \lambda y.(z\, y)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \to_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \to_{\beta'} z\,y$

- $(\lambda x.\lambda y.((x\,x)\,(\lambda x.x)))\,(\lambda y.(y\,x)) \to_{\beta'} \lambda y.(((\lambda y.(y\,x))\,(\lambda y(y\,x)))\,(\lambda x.x))$

- $(\lambda x.\lambda y.x)\,(z\,y) \not\to_{\beta'} \lambda y.(z\,y)$

  Ici, il l'occurrence libre de y dans $(z\,y)$ serait capturée par le λy dans le « résultat ».

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\,z)) \rightarrow_{\beta'} \lambda x.(x\,z)$

- $(\lambda x.(x\,y))\,z \rightarrow_{\beta'} z\,y$

- $(\lambda x.\lambda y.((x\,x)\,(\lambda x.x)))\,(\lambda y.(y\,x)) \rightarrow_{\beta'} \lambda y.(((\lambda y.(y\,x))\,(\lambda y(y\,x)))\,(\lambda x.x))$

- $(\lambda x.\lambda y.x)\,(z\,y) \not\rightarrow_{\beta'} \lambda y.(z\,y)$

  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

$$(\lambda x.\lambda y.x)\,(z\,y) = (\lambda x.\lambda y'.x)\,(z\,y) \rightarrow_{\beta'} \lambda y'.(z\,y).$$

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

$$x, \quad y,$$

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

$$x,\quad y,\quad y\,y,$$

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

$$x, \quad y, \quad y\,y, \quad \lambda y.(y\,y), \quad \lambda x.\lambda y.(y\,y), \quad x\,(\lambda x.\lambda y.(y\,y)).$$

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 $t \rightarrow_{\beta^*} t'$ 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 :

$$((\lambda x.(x\, y))\, y)\, ((\lambda y.y)\, ((\lambda x.x)\, x)) \to_\beta (y\, y)\, ((\lambda y.y)\, ((\lambda x.x)\, x))$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x)
\end{aligned}
$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\, y))\, y)\, ((\lambda y.y)\, ((\lambda x.x)\, x)) &\to_\beta (y\, y)\, ((\lambda y.y)\, ((\lambda x.x)\, x)) \\
&\to_\beta (y\, y)\, ((\lambda y.y)\, x) \\
&\to_\beta (y\, y)\, x.
\end{aligned}
$$

- 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 :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- 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 :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$(\lambda y.y)\,((\lambda x.x)\,x) \to_\beta (\lambda y.y)\,x$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda y.y)\,x \\
&\to_\beta x,
\end{aligned}
$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda y.y)\,x \\
&\to_\beta x,
\end{aligned}
$$

$$(\lambda y.y)\,((\lambda x.x)\,x)$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda y.y)\,x \\
&\to_\beta x,
\end{aligned}
$$

$$
(\lambda y.y)\,((\lambda x.x)\,x) \to_\beta (\lambda x.x)\,x
$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda y.y)\,x \\
&\to_\beta x,
\end{aligned}
$$

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda x.x)\,x \\
&\to_\beta x.
\end{aligned}
$$

β-réductions — exemples

- Voici une suite de β-réductions :

$$
\begin{aligned}
((\lambda x.(x\,y))\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) &\to_\beta (y\,y)\,((\lambda y.y)\,((\lambda x.x)\,x)) \\
&\to_\beta (y\,y)\,((\lambda y.y)\,x) \\
&\to_\beta (y\,y)\,x.
\end{aligned}
$$

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda y.y)\,x \\
&\to_\beta x,
\end{aligned}
$$

$$
\begin{aligned}
(\lambda y.y)\,((\lambda x.x)\,x) &\to_\beta (\lambda x.x)\,x \\
&\to_\beta x.
\end{aligned}
$$

Ainsi dans ces deux cas, $(\lambda y.y)\,((\lambda x.x)\,x) \to_{\beta^*} x$.

Règle de calcul

La relation de β-réduction $\to_\beta$ est une règle de calcul.

Règle de calcul

La relation de β-réduction $\to_\beta$ 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 $\rightarrow_\beta$ 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 $\rightarrow_\beta$ 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 $\rightarrow_\beta$ 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 $\rightarrow_\beta$ 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 $\rightarrow_\beta$ 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

$$\begin{aligned}\Delta\,\Delta &= (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \\ &\to_\beta (\lambda x.(x\,x))\,(\lambda x.(x\,x))\end{aligned}$$

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

$$\begin{aligned}\Delta\,\Delta &= (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \\ &\to_\beta (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \\ &\to_\beta \cdots\end{aligned}$$

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

$$\begin{aligned}\Delta\,\Delta &= (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \\ &\rightarrow_\beta (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \\ &\rightarrow_\beta \cdots\end{aligned}$$

Ainsi, l'évaluation de $\Delta\,\Delta$ ne termine pas. La réponse est donc non.

Plan

6 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 :

$$\mathrm{vrai} := \lambda x.\lambda y.x,$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x,$$
$$\mathrm{faux} := \lambda x.\lambda y.y,$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x,$$
$$\mathrm{faux} := \lambda x.\lambda y.y,$$

Nous avons, si u et v sont deux λ-termes,

$$(\mathrm{vrai}\, u)\, v$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned}\mathrm{vrai} &:= \lambda x.\lambda y.x,\\ \mathrm{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$(\mathrm{vrai}\, u)\, v = ((\lambda x.\lambda y.x)\, u)\, v$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned} \mathrm{vrai} &:= \lambda x.\lambda y.x, \\ \mathrm{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned} \mathrm{vrai} &:= \lambda x.\lambda y.x, \\ \mathrm{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\ u)\ v &= ((\lambda x.\lambda y.x)\ u)\ v \\ &\rightarrow_\beta (\lambda y.u)\ v \\ &\rightarrow_\beta u. \end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x, \qquad \text{renvoie son } 1^{\text{er}} \text{ argument},$$
$$\mathrm{faux} := \lambda x.\lambda y.y,$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}(\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u.\end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned} \mathrm{vrai} &:= \lambda x.\lambda y.x, \qquad \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathrm{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u. \end{aligned}$$

De même,

$$(\mathrm{faux}\, u)\, v$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x, \qquad \text{renvoie son 1}^{\text{er}} \text{ argument,}$$
$$\mathrm{faux} := \lambda x.\lambda y.y,$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}(\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u.\end{aligned}$$

De même,

$$(\mathrm{faux}\, u)\, v = ((\lambda x.\lambda y.y)\, u)\, v$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned} \mathrm{vrai} &:= \lambda x.\lambda y.x, \qquad \text{renvoie son } 1^{\text{er}} \text{ argument,} \\ \mathrm{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u. \end{aligned}$$

De même,

$$\begin{aligned} (\mathrm{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\begin{aligned} \mathrm{vrai} &:= \lambda x.\lambda y.x, &\quad& \text{renvoie son } 1^{\text{er}} \text{ argument,} \\ \mathrm{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u. \end{aligned}$$

De même,

$$\begin{aligned} (\mathrm{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \\ &\to_\beta v. \end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x, \qquad \text{renvoie son 1}^{\text{er}} \text{ argument,}$$
$$\mathrm{faux} := \lambda x.\lambda y.y, \qquad \text{renvoie son 2}^{\text{e}} \text{ argument.}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}(\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\to_\beta (\lambda y.u)\, v \\ &\to_\beta u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathrm{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \\ &\to_\beta v.\end{aligned}$$

Booléens

On code les deux booléens vrai et faux de la manière suivante :

$$\mathrm{vrai} := \lambda x.\lambda y.x, \qquad \text{renvoie son } 1^{\text{er}} \text{ argument,}$$
$$\mathrm{faux} := \lambda x.\lambda y.y, \qquad \text{renvoie son } 2^{\text{e}} \text{ argument.}$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned} (\mathrm{vrai}\, u)\, v &= ((\lambda x.\lambda y.x)\, u)\, v \\ &\rightarrow_\beta (\lambda y.u)\, v \\ &\rightarrow_\beta u. \end{aligned}$$

De même,

$$\begin{aligned} (\mathrm{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\rightarrow_\beta (\lambda y.y)\, v \\ &\rightarrow_\beta v. \end{aligned}$$

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 :

$$\mathrm{et} := \lambda b.\lambda c.((b\,c)\,b).$$

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 :

$$\mathrm{et} := \lambda b.\lambda c.((b\,c)\,b).$$

On code le « ou » logique ou de la manière suivante :

$$\mathrm{ou} := \lambda b.\lambda c.((b\,b)\,c).$$

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 :

$$\mathrm{et} := \lambda b.\lambda c.((b\,c)\,b).$$

On code le « ou » logique ou de la manière suivante :

$$\mathrm{ou} := \lambda b.\lambda c.((b\,b)\,c).$$

On code le « non » logique non de la manière suivante :

$$\mathrm{non} := \lambda b.\lambda x.\lambda y.((b\,y)\,x).$$

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 :

$$\mathrm{et} := \lambda b.\lambda c.((b\,c)\,b).$$

On code le « ou » logique ou de la manière suivante :

$$\mathrm{ou} := \lambda b.\lambda c.((b\,b)\,c).$$

On code le « non » logique non de la manière suivante :

$$\mathrm{non} := \lambda b.\lambda x.\lambda y.((b\,y)\,x).$$

On code l'implication logique imp de la manière suivante :

$$\mathrm{imp} := \lambda a.\lambda b.((\mathrm{ou}\,(\mathrm{non}\,a))\,b).$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\, y)\, z).$$

Nous avons, si u et v sont deux λ-termes,

$$((\mathrm{sas}\, \mathrm{vrai})\, u)\, v$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\, y)\, z).$$

Nous avons, si u et v sont deux λ-termes,

$$((\mathrm{sas}\,\mathrm{vrai})\, u)\, v = (((\lambda x.\lambda y.\lambda z.((x\, y)\, z))\, \mathrm{vrai})\, u)\, v$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}((\mathrm{sas}\,\mathrm{vrai})\,u)\,v &= (((\lambda x.\lambda y.\lambda z.((x\,y)\,z))\,\mathrm{vrai})\,u)\,v \\ &\to_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\,y)\,z))\,u)\,v\end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\, y)\, z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}((\mathrm{sas}\,\mathrm{vrai})\, u)\, v &= (((\lambda x.\lambda y.\lambda z.((x\, y)\, z))\,\mathrm{vrai})\, u)\, v \\ &\to_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\, y)\, z))\, u)\, v \\ &\to_\beta ((\lambda y.\lambda z.y)\, u)\, v\end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}((\mathrm{sas}\,\mathrm{vrai})\,u)\,v &= (((\lambda x.\lambda y.\lambda z.((x\,y)\,z))\,\mathrm{vrai})\,u)\,v \\ &\to_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\,y)\,z))\,u)\,v \\ &\to_\beta ((\lambda y.\lambda z.y)\,u)\,v \\ &\to_\beta (\lambda z.u)\,v\end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}
((\mathrm{sas}\,\mathrm{vrai})\,u)\,v &= (((\lambda x.\lambda y.\lambda z.((x\,y)\,z))\,\mathrm{vrai})\,u)\,v \\
&\to_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\,y)\,z))\,u)\,v \\
&\to_\beta ((\lambda y.\lambda z.y)\,u)\,v \\
&\to_\beta (\lambda z.u)\,v \\
&\to_\beta u.
\end{aligned}$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **si** **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}((\mathrm{sas}\,\mathrm{vrai})\,u)\,v &= (((\lambda x.\lambda y.\lambda z.((x\,y)\,z))\,\mathrm{vrai})\,u)\,v \\ &\to_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\,y)\,z))\,u)\,v \\ &\to_\beta ((\lambda y.\lambda z.y)\,u)\,v \\ &\to_\beta (\lambda z.u)\,v \\ &\to_\beta u.\end{aligned}$$

De même, on obtient

$$((\mathrm{sas}\,\mathrm{faux})\,u)\,v \to_\beta v.$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

$$\mathrm{sas} := \lambda x.\lambda y.\lambda z.((x\,y)\,z).$$

Nous avons, si u et v sont deux λ-termes,

$$\begin{aligned}((\mathrm{sas}\,\mathrm{vrai})\,u)\,v &= (((\lambda x.\lambda y.\lambda z.((x\,y)\,z))\,\mathrm{vrai})\,u)\,v \\ &\rightarrow_\beta ((\lambda y.\lambda z.((\mathrm{vrai}\,y)\,z))\,u)\,v \\ &\rightarrow_\beta ((\lambda y.\lambda z.y)\,u)\,v \\ &\rightarrow_\beta (\lambda z.u)\,v \\ &\rightarrow_\beta u.\end{aligned}$$

De même, on obtient

$$((\mathrm{sas}\,\mathrm{faux})\,u)\,v \rightarrow_\beta v.$$

Ainsi, la valeur calculée par sas, appliquée à trois arguments, possède le comportement attendu.

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1ers λ-termes codant les petits entiers :

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ-termes codant les petits entiers :

$$\mathrm{ent}(0) = \lambda f.\lambda x.x,$$

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\,x), \end{aligned}$$

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\,x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\,(f\,x)), \end{aligned}$$

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\,x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\,(f\,x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\,(f\,(f\,x))). \end{aligned}$$

Entiers naturels

On code tout entier naturel n par $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1ers λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\,x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\,(f\,x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\,(f\,(f\,x))). \end{aligned}$$

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 $\mathrm{ent}(n)$ défini de la manière suivante :

$$\mathrm{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f\,((\mathrm{ent}(n-1)\,f)\,x)) & \text{sinon.} \end{cases}$$

Voici les 1^{ers} λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\,x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\,(f\,x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\,(f\,(f\,x))). \end{aligned}$$

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

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\mathrm{succ}\,\mathrm{ent}(2)$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\mathrm{succ}\,\mathrm{ent}(2) = (\lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)))\,(\lambda f.\lambda x.(f\,(f\,x)))$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\begin{aligned}\mathrm{succ}\,\mathrm{ent}(2) &= (\lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)))\,(\lambda f.\lambda x.(f\,(f\,x))) \\ &\to_\beta \lambda f.\lambda x.(f\,(((\lambda f.\lambda x.(f\,(f\,x)))\,f)\,x))\end{aligned}$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\begin{aligned}
\mathrm{succ}\,\mathrm{ent}(2) &= (\lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)))\,(\lambda f.\lambda x.(f\,(f\,x))) \\
&\rightarrow_\beta \lambda f.\lambda x.(f\,(((\lambda f.\lambda x.(f\,(f\,x)))\,f)\,x)) \\
&\rightarrow_\beta \lambda f.\lambda x.(f\,(\lambda x.(f\,(f\,x))\,x))
\end{aligned}$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\begin{aligned}
\mathrm{succ}\,\mathrm{ent}(2) &= (\lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)))\,(\lambda f.\lambda x.(f\,(f\,x))) \\
&\to_\beta \lambda f.\lambda x.(f\,(((\lambda f.\lambda x.(f\,(f\,x)))\,f)\,x)) \\
&\to_\beta \lambda f.\lambda x.(f\,(\lambda x.(f\,(f\,x))\,x)) \\
&\to_\beta \lambda f.\lambda x.(f\,(f\,(f\,x)))
\end{aligned}$$

Arithmétique sur les entiers de Church

La fonction succ calcule le successeur d'un entier. Elle se code par

$$\mathrm{succ} := \lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)).$$

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

$$\begin{aligned}
\mathrm{succ}\,\mathrm{ent}(2) &= (\lambda n.\lambda f.\lambda x.(f\,((n\,f)\,x)))\,(\lambda f.\lambda x.(f\,(f\,x))) \\
&\rightarrow_\beta \lambda f.\lambda x.(f\,(((\lambda f.\lambda x.(f\,(f\,x)))\,f)\,x)) \\
&\rightarrow_\beta \lambda f.\lambda x.(f\,(\lambda x.(f\,(f\,x))\,x)) \\
&\rightarrow_\beta \lambda f.\lambda x.(f\,(f\,(f\,x))) \\
&= \mathrm{ent}(3).
\end{aligned}$$

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

$$\mathrm{add} := \lambda n.\lambda m.\lambda f.\lambda x.((n\,f)\,((m\,f)\,x)).$$

Arithmétique sur les entiers de Church

La fonction add calcule la somme de deux entiers. Elle se code par

$$\mathrm{add} := \lambda n.\lambda m.\lambda f.\lambda x.((n\,f)\,((m\,f)\,x)).$$

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

$$\mathrm{add} := \lambda n.\lambda m.\lambda f.\lambda x.((n\, f)\, ((m\, f)\, x)).$$

La fonction mul calcule le produit de deux entiers. Elle se code par

$$\mathrm{mul} := \lambda n.\lambda m.\lambda f.(n\, (m\, f)).$$

Arithmétique sur les entiers de Church

La fonction add calcule la somme de deux entiers. Elle se code par

$$\mathrm{add} := \lambda n.\lambda m.\lambda f.\lambda x.((n\,f)\,((m\,f)\,x)).$$

La fonction mul calcule le produit de deux entiers. Elle se code par

$$\mathrm{mul} := \lambda n.\lambda m.\lambda f.(n\,(m\,f)).$$

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

$$\mathrm{add} := \lambda n.\lambda m.\lambda f.\lambda x.((n\,f)\,((m\,f)\,x)).$$

La fonction mul calcule le produit de deux entiers. Elle se code par

$$\mathrm{mul} := \lambda n.\lambda m.\lambda f.(n\,(m\,f)).$$

La fonction pui calcule l'exponentiation de deux entiers. Elle se code par

$$\mathrm{pui} := \lambda n.\lambda m.(m\,n).$$

Récursivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

Récursivité

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

$$\rho := \lambda f.(\lambda x.((f\,(x\,x)))\,(\lambda x.(f\,(x\,x)))).$$

Récursivité

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,

$$\rho\, u \;\to_\beta\; u\,(\rho\, u).$$

Récursivité

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

$$\rho\, u \;\rightarrow_\beta\; u\,(\rho\, u) \;\rightarrow_\beta\; u\,(u\,(\rho\, u)) \;\rightarrow_\beta\; u\,(u\,(u\,(\rho\, u))) \;\rightarrow_\beta\; \cdots,$$

offrant un moyen d'appliquer de manière itérée le λ-terme (la fonction) u.

Plan

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).