Plan

Plan

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des fonctions d'entrée / sortie qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des fonctions d'entrée / sortie qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée / sortie utilisent le type `unit` et son unique valeur `()`.

Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins des fonctions d'entrée / sortie qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée / sortie utilisent le type `unit` et son unique valeur `()`.

Rappel : l'utilisation d'entrées / sorties fait que l'on **sort du paradigme de programmation fonctionnelle pure** car elles produisent un effet (affichage ou bien attente d'une action de l'utilisateur).

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, toute fonction d'écriture renvoie `()`.

Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, toute fonction d'écriture renvoie `()`.

Les fonctions d'écriture principales sont

```
val print_int : int -> unit
val print_float : float -> unit
val print_char : char -> unit
val print_string : string -> unit
val print_newline : unit -> unit
```

Fonctions de lecture

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

Fonctions de lecture

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

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Fonctions de lecture

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

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

Fonctions de lecture

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

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

Fonctions de lecture

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

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

Les fonctions de lecture principales sont

```
val read_int : unit -> int
val read_float : unit -> float
val read_line : unit -> string
```

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'opérateur de séquence `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'opérateur de séquence `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'opérateur de séquence `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'opérateur de séquence `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La valeur de `EXP1; EXP2` est celle de `EXP2`.

Le parenthésage d'une expression

```
E1; E2; ... ; En
```

est fait implicitement de gauche à droite en

```
((...(E1; E2); ...); En)
```

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

De ce fait, `E1`, ..., `En-1` doivent être de type `unit`.

Séquences — exemples

```
# let add x y =
      (print_string "Appel de add");
      (print_int x);
      (print_int y);
      x + y;;
val add : int -> int -> int = <fun>
```

Séquences — exemples

```
# let add x y =
      (print_string "Appel de add");
      (print_int x);
      (print_int y);
      x + y;;
val add : int -> int -> int = <fun>


# let test_div n =
      if n mod 2 = 0 then
          (print_string "pair\n");
      if n mod 3 = 0 then
          (print_string "multiple de 3\n");;
val test_div : int -> unit = <fun>
```

Séquences et blocs

Considérons la fonction

```
# let div_decr x =
      if x mod 2 = 0 then
          (print_string "pair");
          x / 2
      else
           (print_string "impair");
          x - 1;;
Error: Syntax error
```

Séquences et blocs

Considérons la fonction

```
# let div_decr x =
      if x mod 2 = 0 then
          (print_string "pair");
          x / 2
      else
           (print_string "impair");
          x - 1;;
Error: Syntax error
```

L'opérateur de séquence est moins prioritaire que la conditionnelle. Ainsi, cette fonction est comprise en

```
# let div_decr x =
      if x mod 2 = 0 then
          (print_string "pair");
      x / 2
      else
          (print_string "impair");
      x - 1;;
```

Ceci explique l'**erreur de syntaxe**.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de bloc. Un bloc est une expression de la forme

```
begin EXP end
```

où `EXP` est une expression.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de bloc. Un bloc est une expression de la forme

```
begin EXP end
```

où `EXP` est une expression. La valeur de `begin EXP end` est celle de `EXP`.

Séquences et blocs

Pour résoudre ce problème, on utilise la notion de bloc. Un bloc est une expression de la forme

`begin EXP end`

où `EXP` est une expression. La valeur de `begin EXP end` est celle de `EXP`.

La fonction précédente devient ainsi correcte en écrivant

```
let div_decr x =
    if x mod 2 = 0 then begin
        (print_string "pair");
        x / 2
    end
    else begin
        (print_string "impair");
        x - 1
    end;
```

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
```

Programmation dans un fichier

Au lieu de programmer à l'aide de l'interpréteur CAML, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) et de le(s) compiler pour obtenir un exécutable.

Pour cela, on écrit le programme dans un fichier d'extension `.ml`. On le compile par la commande

```
ocamlc -o Prog Prog.ml
```

Ceci invoque le **compilateur bytecode**.

Le **compilateur natif** produit des exécutables souvent plus performants. On y fait appel par la commande

```
ocamlopt -o Prog Prog.ml
```

Dans les deux cas, l'exécutable se lance par

```
./Prog
```

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

1. on construit un fichier objet (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

   ```
   ocamlc -c F.ml
   ```

   ou bien

   ```
   ocamlopt -c F.ml
   ```

Projets de plusieurs fichiers

Pour compiler un projet comportant **plusieurs fichiers**,

1. on construit un fichier objet (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

   ```
   ocamlc -c F.ml
   ```

   ou bien

   ```
   ocamlopt -c F.ml
   ```

2. on appelle l'éditeur de liens par la commande

   ```
   ocamlc -o Prog F1.cmo ... Fn.cmo
   ```

   ou bien

   ```
   ocamlopt -o Prog F1.cmx ... Fn.cmx
   ```

   où les `F1.cm*`, …, `Fn.cm*` sont les fichiers objet du projet.

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

```
open A;;
```

2. Utiliser `f` aux endroits désirés dans `B.ml`.

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

   ```
   open A;;
   ```

2. Utiliser `f` aux endroits désirés dans `B.ml`.

   Dans `B.ml`, le nom de `f` devient

   ```
   A.f
   ```

Projets de plusieurs fichiers et inclusions

Dans le cas où un fichier `B.ml` a besoin d'une fonction `f` définie dans un fichier `A.ml` du projet, il faut

1. inclure `A.ml` dans `B.ml` au moyen de l'instruction

   `open A;;`

2. Utiliser `f` aux endroits désirés dans `B.ml`.

   Dans `B.ml`, le nom de `f` devient

   `A.f`

Exemple :

```
(* A.ml *)

let double x =
2 * x
```

```
(* B.ml *)

open A

let quadruple x =
2 * (A.double x)
```

Dans ce projet, `B.ml` inclut `A.ml`. Ainsi, la fonction `double` de `A.ml` est visible dans `B.ml` par l'identificateur `A.double`.

Espaces de noms

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

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.

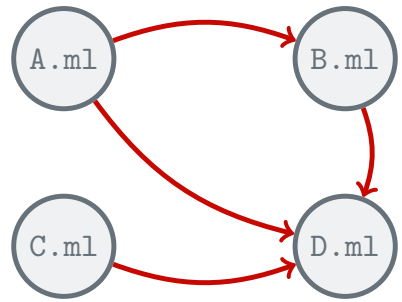
Ordre de compilation

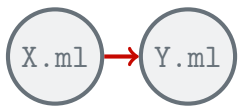
À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

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 :



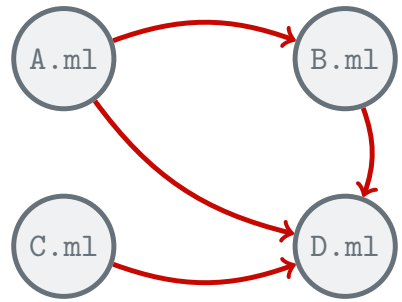
Toute flèche 

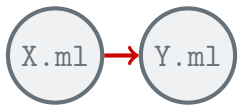
signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

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`.

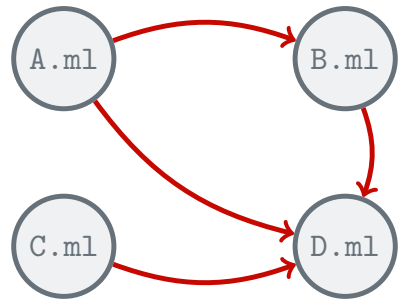
Dans cet exemple, on a les trois ordres suivants possibles :

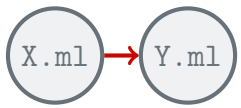
- `D.ml`, `C.ml`, `B.ml`, `A.ml`;
- `D.ml`, `B.ml`, `C.ml`, `A.ml`;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

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`.

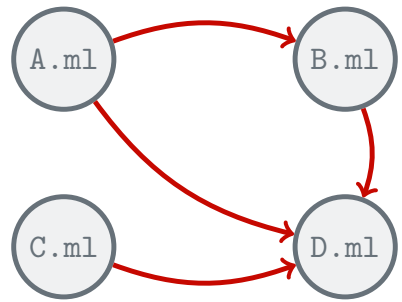
Calcul des dépendances automatisé

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`.

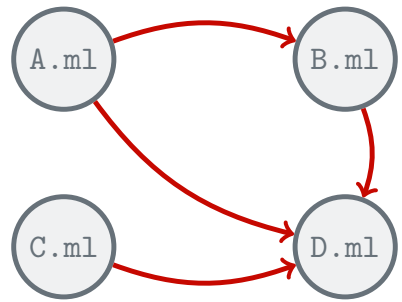
Reprenons le graphe d'inclusions



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

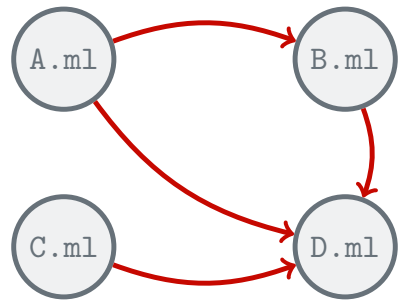
```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
```

```
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
`*`	2	Produit cartésien binaire
`*`	$\geqslant 2$	Produit cartésien multiple
`{ }`	$\geqslant 1$	Produit nommé
`|`	$\geqslant 1$	Somme

Plan

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

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

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

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

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

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

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

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

```
type point = int * int
```

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

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

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

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

```
type point = int * int
```

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

Un couple s'écrit

```
(e1, e2)
```

Les parenthèses sont facultatives.

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

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

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

```
type point = int * int
```

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

Un couple s'écrit

```
(e1, e2)
```

Les parenthèses sont facultatives.

```
# (3.5, 21);;
- : float * int = (3.5, 21)
```

Produit cartésien binaire

Étant donnés deux types `T1` et `T2`,

```
T1 * T2
```

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

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

```
type point = int * int
```

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

Un couple s'écrit

```
(e1, e2)
```

Les parenthèses sont facultatives.

```
# (3.5, 21);;
- : float * int = (3.5, 21)
```

```
# ((), ((), ()));;
- : 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 multiple** de `T1`, …, `Tn`.

n-uplets

Étant donnés des types `T1`, …, `Tn`,

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

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

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

n-uplets

Étant donnés des types `T1`, …, `Tn`,

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

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

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

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

n-uplets

Étant donnés des types `T1`, …, `Tn`,

`T1 * ... * Tn`

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

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

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

Un n-uplet s'écrit

`(e1, ..., en)`

Les parenthèses sont facultatives.

n-uplets

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

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

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

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

```
# 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}
```

Produit nommé

Étant donnés des types `T1`, …, `Tn` et des identificateurs `ID1`, …, `IDn`,

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

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

Il contient pour valeurs les enregistrements dont les **champs** sont `ID1`, …, `IDn` de types respectifs `T1`, …, `Tn`.

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

Un enregistrement s'écrit

```
{ID1 = V1 ; ... ; IDn = Vn}
```

où `V1`, …, `Vn` sont des valeurs de types respectifs `T1`, …, `Tn`.

Produit nommé

Étant donnés des types `T1`, …, `Tn` et des identificateurs `ID1`, …, `IDn`,

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

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

Il contient pour valeurs les enregistrements dont les **champs** sont `ID1`, …, `IDn` de types respectifs `T1`, …, `Tn`.

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

Un enregistrement s'écrit

`{ID1 = V1 ; ... ; IDn = Vn}`

où `V1`, …, `Vn` sont des valeurs de types respectifs `T1`, …, `Tn`.

```
# {nom = "Haskell Curry" ; age = 81};;
- : personne = {nom = "Haskell Curry"; age = 81}
```

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

```
e.c
```

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

`e.c`

```
# 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}
```

« 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}
# 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}
# 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}
# 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 par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Par exemple :

```
(* A.ml *)

type a = {a : int ; b : int}
```

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Par exemple :

```
(* A.ml *)

type a = {a : int ; b : int}
```

```
(* B.ml *)

type b = {a : int ; c : char}
```

Contrainte sur les noms des champs

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** (car ils ont des espaces de noms différents).

On accède alors aux noms des champs d'un enregistrement en les préfixant par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Par exemple :

```
(* A.ml *)

type a = {a : int ; b : int}
```

```
(* B.ml *)

type b = {a : int ; c : char}
```

```
(* C.ml *)

open A
open B

let c1 = {B.a = 2 ; B.c = 'y'}
and c2 = {A.a = 2 ; A.b = 3}
```

Plan

Somme

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

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

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

Somme

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

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

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

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

Somme

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

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

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

Il contient `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
```

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
```

Une **valeur** d'un type somme s'écrit via son constructeur.

Somme

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

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

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

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

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

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

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
- : numero = Deux
```

Somme

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

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

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

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

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

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

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
- : numero = Deux

# let plusieurs n =
    n = Deux || n = Trois;;
val plusieurs : numero -> bool = <fun>
```

Somme

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

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

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

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

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

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

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
- : numero = Deux
```

```
# let plusieurs n =
    n = Deux || n = Trois;;
val plusieurs : numero -> bool = <fun>
```

```
# (plusieurs Un);;
- : bool = false
```

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
```

Une **valeur** d'un type somme s'écrit via son constructeur.

```
# Deux;;
- : numero = Deux

# let plusieurs n =
    n = Deux || n = Trois;;
val plusieurs : numero -> bool = <fun>
```

```
# (plusieurs Un);;
- : bool = false

# (plusieurs Trois);;
- : bool = true
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, …, `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
# type nombre = Entier of int | Rationnel of int * int | Infini
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, …, `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
# type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
# type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# (Entier 13);;
- : nombre = Entier 13
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
# type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# (Entier 13);;
- : nombre = Entier 13
```

```
# (Rationnel (2, 3));;
- : nombre
    = Rationnel (2, 3)
```

Constructeurs avec arguments

Certains constructeurs d'un type somme peuvent avoir un argument. Un argument est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, …, `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

```
# type nombre = Entier of int | Rationnel of int * int | Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

```
# (Entier 13);;
- : nombre = Entier 13
```

```
# (Rationnel (2, 3));;
- : nombre
    = Rationnel (2, 3)
```

```
# Infini;;
- : nombre = Infini
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

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

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

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

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

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

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

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

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

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))

# let e3 = (Cellule (3, e2));;
val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Exemple 1 — listes d'entiers

Une **liste d'entiers** est la liste vide ou bien une cellule contenant un entier qui est attachée à une liste d'entiers.

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

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

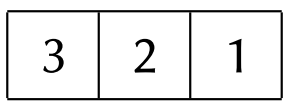
On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))

# let e3 = (Cellule (3, e2));;
val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom e3 est lié à la liste de valeur



Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

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

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

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

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

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

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

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
# let a2 = (Noeud (a1, 3, a1));;
val a2 : arbre_b_int =
  Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
```

Exemple 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

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

```
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 2 — arbres binaires d'entiers

Un **arbre binaire d'entiers** est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

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

```
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