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                                  # (concat "cd");;
      (aux u) ^ (aux ("b" ^ u));;         - : string = "cdacdbcdabcd"
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                                  # (concat "cd");;
      (aux u) ^ (aux ("b" ^ u));;         - : string = "cdacdbcdabcd"
val concat : string -> string = <fun>
```

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.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'imbriquer les expressions conditionnelles.

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'imbriquer les expressions conditionnelles.

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'imbriquer les expressions conditionnelles.

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur Caml la comprend sans ambiguïté.

Imbrications d'expressions conditionnelles

Une expression conditionnelle étant elle-même une expression, il est possible d'imbriquer les expressions conditionnelles.

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur Caml la comprend sans ambiguïté.

Il n'a pas besoin de marqueur de fin (comme le } de certains langages).

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
    if 3 >= 1 then 21 else 24
→   if true then 21 else 24
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```
$\rightarrow$ `if true then 21 else 24` $\rightarrow$ `21`,

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
  → if if true then 1 = 0 else true then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
  → if if true then 1 = 0 else true then 28 else 21
    → if 1 = 0 then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
  → if if true then 1 = 0 else true then 28 else 21
        → if 1 = 0 then 28 else 21
      → if false then 28 else 21
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
  → if true then 21 else 24 → 21,

if if "ab" = "ab" then 1 = 0 else true then 28 else 21
  → if if true then 1 = 0 else true then 28 else 21
    → if 1 = 0 then 28 else 21
      → if false then 28 else 21 → 21.
```

Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la valeur qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
  → if if true then 1 = 0 else true then 28 else 21
        → if 1 = 0 then 28 else 21
        → if false then 28 else 21 → 21.
```

Grâce au **principe de transparence référentielle**, ces deux expressions peuvent être remplacées par leurs valeurs, `21`, dans tout programme les employant sans modifier la valeur qu'il renvoie.

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`$^{\text{e}}$ 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`$^{\text{e}}$ 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`$^{\text{e}}$ 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 $k \leqslant n$,

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

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 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 clairement 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 clairement 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
```

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.

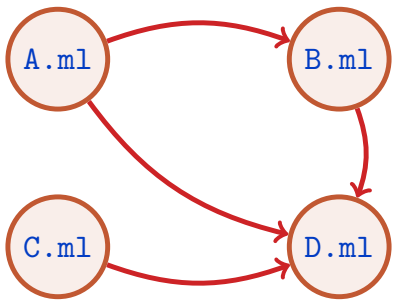
Ordre de compilation

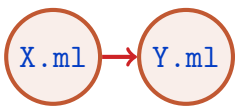
À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui inclut `Y.ml` avant d'avoir produit celui de `Y.ml`.

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 :



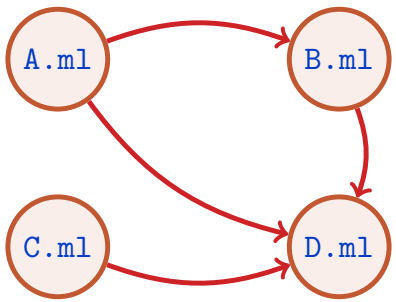
Toute flèche 

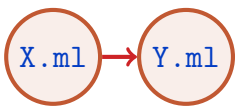
signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

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`.

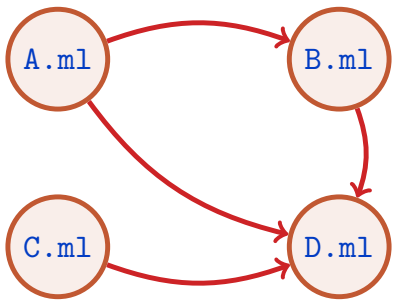
Dans cet exemple, on a les trois ordres suivants possibles :

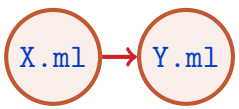
- `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

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`.

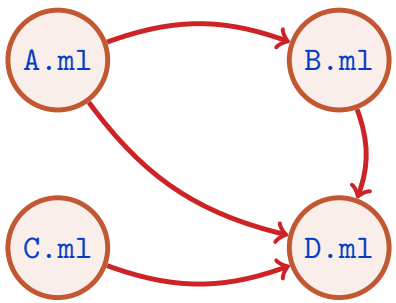
Calcul des dépendances automatisé

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`.

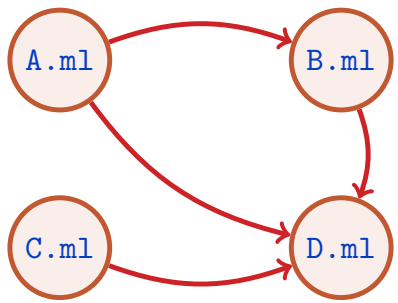
Reprenons le graphe d'inclusions



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

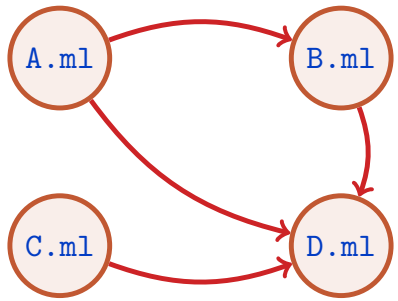
```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
```

```
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
`*`	$\geqslant 2$	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.