4. Pratique

4.1 Entrées et sorties

4.2 Compilation

Programmation dans un fichier

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

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
```

Structure des fichiers

Un fichier `.ml` d'un projet contient

1. soit des déclarations de fonctions ;
2. soit des déclarations de fonctions, suivies du séparateur `;;`, suivi d'une expression principale.

La **valeur** de cette expression principale est le **résultat** produit par l'exécution du programme.

Voici le schéma d'écriture d'un tel fichier :

```
(* Partie de declarations de fonctions. *)

;;

(* Expression principale. *)
```

– Exemple –

```
(* Partie de déclarations de fonctions. *)
let f x =
    x + 1

let g x y =
    y - x
```

```
(* Separateur. *)
;;

(* Expression principale. *)
print_int ((f 5) + (g 10 (f 8)))
```

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

1. inclure `A.ml` dans `B.ml` au moyen de

   ```
   open A
   ```

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

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

   ```
   A.f
   ```

– Exemple –

```
(* A.ml *)

let double x =
    2 * x
```

```
(* B.ml *)

open A

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

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

Espaces de noms

Un espace de nom est une application (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Ils permettent de **restreindre la visibilité** de certaines définitions.

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.

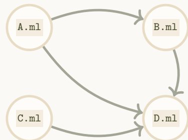
Ordre de compilation

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`.

Il faut donc compiler le projet dans l'ordre dicté par un **tri topologique** du graphe dual du graphe d'inclusions du projet.

– Exemple –

Considérons le graphe d'inclusions suivant :



Toute flèche X.ml → Y.ml signifie que le fichier `X.ml` inclut le fichier `Y.ml`.

On a les trois ordres suivants possibles :

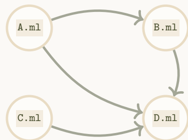
- `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

Calcul des dépendances automatisé

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

– Exemple –

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`.

5. Types

Plan

5. Types

5.1 L’algèbre des types
5.2 Types produit
5.3 Types somme
5.4 Types paramétrés

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 (par exemple, 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.

On rappelle que les **types scalaires** dont nous disposons sont

`int`, `float`, `char`, `string`, `bool`, `unit`.

Voici les opérateurs de types que nous allons considérer :

Opérateur	Arité	Nom
`->`	2	Flèche
`*`	2	Produit cartésien binaire
`*`	$\geqslant 2$	Produit cartésien multiple
`{ }`	$\geqslant 1$	Produit nommé
`|`	$\geqslant 1$	Somme

Plan

5. Types

5.1 L'algèbre des types

5.2 Types produit

5.3 Types somme

5.4 Types paramétrés

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`).

– Exemple –

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

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

Un couple s'écrit par `(e1, e2)`. Les parenthèses sont facultatives.

– Exemples –

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

```
# (1, (2, 3));;
- : int * (int * int) = (1, (2, 3))
# ((1, 2), 3);;
- : (int * int) * int = ((1, 2), 3)
```

Accès aux coordonnées d’un couple

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

– Exemples –

```
# 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 ...
```

– Exemple –

```
# let add c =
      let (c1, c2) = c in
      c1 + c2;;
```

```
val add : int * int -> int = <fun>
```

Non associativité du produit cartésien

L'opérateur de types `*` est **non associatif** : si `T1`, `T2` et `T3` sont des types, `(T1 * T2) * T3` est un type différent de `T1 * (T2 * T3)`.

– Exemple –

```
# let conc_21 x =
      let (x1, x2) = x in
      let (x11, x12) = x1 in
      x11 ^ x2 ^ x12;;
val conc_21 : (string * string) * string -> string
    = <fun>
```

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

```
# let conc_12 x =
      let (x1, x2) = x in
      let (x21, x22) = x2 in
      x21 ^ x1 ^ x22;;
val conc_12 : string * (string * string) -> string
    = <fun>
```

```
# conc_12 ("a", ("b", "c"));;
- : string = "bac"
```

Les types `(string * string) * string` et `string * (string * string)` sont bien différents.

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 ⩽ i ⩽ n`, `ei` est de type `Ti`.

– Exemple –

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

Un n-uplet s'écrit

```
(e1, ..., en)
```

Les parenthèses sont facultatives.

– Exemple –

```
# (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.

Si `c` est un *n*-uplet, on accède à ses coordonnées au moyen de

```
let (e1, ..., en) = c in ...
```

– Exemple –

```
# let p = (2.5, 3.4, -1.2) in
      let (x, y, z) = p in
      x +. z;;
- : float = 1.3
```

Il est possible de ne recueillir que la `k`e de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = c in ...
```

– Exemple –

```
# 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`.

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

– Exemple –

```
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`.

– Exemple –

```
# {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`

– Exemple –

```
# let p = {nom = "Alan Turing" ; age = 41} in p.nom;;
- : string = "Alan Turing"
```

```
# let nom_du_plus_age p1 p2 =
      if p1.age > p2.age then
          p1.nom
      else if p1.age < p2.age then
          p2.nom
      else
          "";;
val nom_du_plus_age : personne -> personne -> string = <fun>
```

« Modification » des champs d’un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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`.

– Exemple –

```
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

Il est incorrect de définir des types enregistrements qui ont un même identificateur de champ.

– Exemple –

```
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.

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`.

– 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

5. Types

5.1 L’algèbre des types

5.2 Types produit

5.3 Types somme

5.4 Types paramétrés

Somme

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

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

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

Il contient exactement `n` **valeurs** : `Id1`, …, `Idn`. Ces valeurs sont appelées constructeurs.

– Exemple –

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

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

– Exemples –

```
# 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**.

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`.

– Exemple –

```
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).

– Exemples –

```
# 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.

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

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

– Exemple –

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

```
# let e1 = Cellule (1, Vide);;
val e1 : liste_int = Cellule (1, Vide)
# let e2 = Cellule (2, e1);;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))
# let e3 = Cellule (3, e2);;
val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom `e3` est lié à la liste de valeur

3	2	1

Exemple 2 — arbres binaires d'entiers

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

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

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

– Exemple –

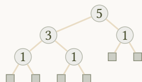
On construit des arbres binaires d'entiers de la manière suivante :

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

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

```
# let a3 = (Noeud (a2, 5, a1));;
val a3 : arbre_b_int = Noeud (Noeud (Noeud (Vide,1,Vide), 3, Noeud (Vide,1,Vide)), 5, Noeud (Vide,1,Vide))
```

Le nom `a3` est lié l'arbre binaire de valeur



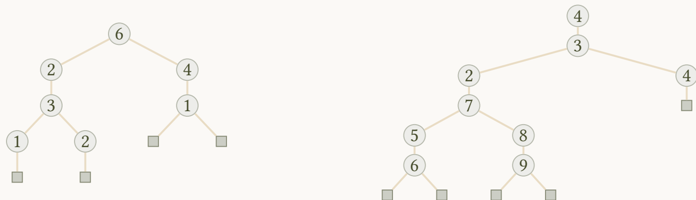
Exemple 3 — arbres unaires binaires d'entiers

Un arbre unaire binaire d'entiers est

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

– Exemples –

Voici deux arbres unaires binaires, respectivement dont la racine est d'arité 2 et dont la racine est d'arité 1 :



Exemple 3 — arbres unaires binaires d'entiers

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

On commence par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

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

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

On se base sur la définition des arbres unaires binaires pour construire finalement le type recherché :

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

Exemple 3 — arbres unaires binaires d'entiers

– Exemple –

Écrivons une expression de type `arbre_12` qui représente l'arbre

```
# let a2 = Noeud2 (Vide1, 2, Vide1) in
 let a1 = Noeud1 (1, a2) in
 let a22 = Noeud2 (a1, 2, a1) in
 let a11 = Noeud1 (1, a22) in
 Arbre2 (Noeud2 (a11, 2, a11));;
```

```
- : arbre_12 =
Arbre2
 (Noeud2
   (Noeud1 (1,
     Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
      Noeud1 (1, Noeud2 (Vide1, 2, Vide1)))),
   2,
   Noeud1 (1,
    Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
     Noeud1 (1, Noeud2 (Vide1, 2, Vide1))))))
```

5. Types

5.1 L'algèbre des types
5.2 Types produit
5.3 Types somme
5.4 Types paramétrés

Paramètres dans les types

Tout comme les **fonctions** qui admettent des **paramètres** (voués à être substitués par des **valeurs**), il est possible de définir des **types** avec des **paramètres** (voués à être substitués par des **types**).

On parle alors de types paramétrés.

La définition d'un nouveau type paramétré `ID` se fait par

```
type ('P1, ..., 'Pn) ID = OP
```

où `P1`, ..., `Pn` sont des identificateurs et `OP` fait intervenir des types, des opérateurs de types et `'P1`, ..., `'Pn`.

Les `'P1`, ..., `'Pn` sont des paramètres de types.

Lorsque `n = 1`, la définition se fait simplement (sans parenthèses) par

```
type 'P1 ID = OP
```

Rôle des paramètres de type

Supposons que `T` soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

On dit que `T` est paramétré par `'P1`, …, `'Pn`.

Dans la définition de `T`, les occurrences de `'Pi`, `1 ⩽ i ⩽ n`, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un **ensemble de types**.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme des constantes) ;
2. des paramètres de types (vus comme des variables) ;
3. des opérateurs de types.

Exemple 1 — listes génériques

Le type à un paramètre ci-contre permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. On obtient ainsi des **listes génériques**.

```
type 'e liste =
    |Vide
    |Cellule of 'e * 'e liste
```

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

– Exemples –

- Une liste de caractères :

```
# Cellule ('a', (Cellule ('b', Vide)));;
- : char liste = Cellule ('a', Cellule ('b', Vide))
```

- Une liste de listes d'entiers :

```
# Cellule ((Cellule (1, Vide)), Vide);;
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

- Une liste dont le type des éléments n'est pas déterminé :

```
# Vide;;
- : 'a liste = Vide
```