Cours d'informatique

MP2I

Lou Chalmain

2022-2023

Table des matières

Chapitre 1

Initiation à Ocaml

Ocaml est un langage de programmation relevant notamment du paradigme de la programmation fonctionnelle, c'est à dire qu'il repose sur l'évaluation de fonctions, mais permettant également d'adopter un style de programmation impérative.

Un programme Ocaml est une suite d'expressions et de déclarations séparées par un double point-virgule `;;` dans lesquelles l'indentation et les retours à la ligne n'a pas d'importance mais facilitent grandement la lisibilité et la compréhension du code.

Une expression peut elle-même être une séquence de plusieurs expressions séparées d'un point-virgule `;`. Dans ce cas c'est l'évaluation de la dernière expression qui est renvoyée.

1.1 Variables

1.1.1 Déclaration et types

La déclaration d'une variable s'effectue en Ocaml grâce au mot-clé `let` selon la syntaxe suivante :

```
let ma_variable = ma_valeur;;
```

Exemple 1.1.

```
# let a = 8;;
val a : int = 8
```

La sortie indique qu'on vient de créer une ***variable*** dont :

- le ***nom*** est `a`
- le ***type*** est `int`
- la ***valeur*** est `8`

Pour faciliter la compréhension du code on choisit des noms de variables explicites lorsque cela fait sens (comme `prix` ou `distance` plutôt que `a` ou `x`).
Dans tous les cas on choisira des mots ***sans espaces*** et ***non accentués*** (on pourra utiliser le symbole `_`).

Exemple 1.2.

```
# let duree_en_h = 3.5;;
val duree_en_h : float = 3.5
```

Une variable définie comme ci-dessus est ***immuable*** (sa valeur n'est pas modifiable) et possède obligatoirement un ***type***, on parle de ***typage statique***.
Le type est ***inféré*** par le compilateur ou l'interpréteur : il est déduit des différentes fonctions et opérateurs impliqués dans l'expression évaluée lors de la déclaration.

Le tableau suivant récapitule les types de base et les opérateurs associés :

<table>
<tr><th></th><th>Type</th><th>Opérateur</th><th></th></tr>
<tr><td rowspan="5">Entiers relatifs</td><td rowspan="5">int</td><td>+</td><td>addition</td></tr>
<tr><td>-</td><td>soustraction</td></tr>
<tr><td>*</td><td>multiplication</td></tr>
<tr><td>/</td><td>division euclidienne</td></tr>
<tr><td>mod</td><td>reste de la diviion euclidienne (si positifs)</td></tr>
<tr><td rowspan="5">Nombres à virgule flottante</td><td rowspan="5">float</td><td>+.</td><td>addition</td></tr>
<tr><td>-.</td><td>soustraction</td></tr>
<tr><td>*.</td><td>multiplication</td></tr>
<tr><td>/.</td><td>division exacte</td></tr>
<tr><td>**</td><td>puissance</td></tr>
<tr><td rowspan="3">Booléen
(true ou false)</td><td rowspan="3">bool</td><td>not</td><td>négation</td></tr>
<tr><td>&&</td><td>ET logique</td></tr>
<tr><td>||</td><td>OU logique</td></tr>
<tr><td>Caractère</td><td>char</td><td></td><td>ex : 'c'</td></tr>
<tr><td>Chaîne de caractères</td><td>string</td><td>^</td><td>concaténation : "une "^"chaine" = "une chaine"</td></tr>
<tr><td>Rien</td><td>unit</td><td></td><td>() est de type unit</td></tr>
</table>

Remarque 1.1. Les nombres à virgule flottante (ou ***flottants***) s'écrivent avec un point et non une virgule ! Sinon, il s'agira d'un couple d'entiers (type `int * int`).

Exemple 1.3.

```
# 2.718;;
- : float = 2.718
# 2,718;;
- : int * int = (2, 718)
```

Remarque 1.2. Pour les caractères on utilise des guillemets simples (`'`) tandis que pour les chaînes de caractères on utilise des guillemets doubles (`"`).

Remarque 1.3. La longueur d'une chaîne de caractères s'obtient avec `String.length`.
L'accès aux différents caractères de la chaîne se fait selon la syntaxe `s.[i]` où i est l'indice souhaité (à partir de l'indice 0), mais on ne peut pas les modifier : on dit que les chaînes de caractères sont ***immuables***.

Si besoin, on pourra convertir une expression en son équivalent d'un autre type. On dispose notamment des fonctions de conversion suivantes :

Type de départ	Type d'arrivée	Fonction	Exemple	Correspondance
`int`	`float`	`float_of_int`	`4`	`4.`
`float`	`int`	`int_of_float` (troncature à l'unité)	`3.14`	`3`
`int`	`string`	`string_of_int`	`4`	`"4"`
`float`	`string`	`string_of_float`	`3.14`	`"3.14"`
`int`	`char`	`char_of_int` (correspondance ASCII)	`64`	`'@'`
`char`	`int`	`int_of_char` (correspondance ASCII)	`'a'`	`97`

1.1.2 Overflow

Les entier et les flottants sont sujets au dépassement de capacité. En effet, les variables sont stockées dans la mémoire RAM de l'ordinateur, or celle-ci n'est pas infinie.

• Le plus grand entier représentable est stocké dans la variable `max_int`, si on augmente ce nombre de 1, on tombe sur le plus petit entier représentable.
On parle de dépassement d'entier ou encore d'***integer overflow***.

• De même, `max_float` est le plus grand flottant représentable.

Exemple 1.4.

```
# max_int;;
- : int = 4611686018427387903
# max_int + 1;;
- : int = -4611686018427387904
# max_float;;
- : float = 1.79769313486231571e+308
```

Remarque 1.4. Le dépassement d'entier est notamment responsable du crash de la fusée Ariane 5...

1.1.3 Comparaisons et type bool

Pour effectuer une comparaison entre deux objets de même type on utilise les opérateurs `=`, `<`, `>`, `<=`, `>=` ou `<>` (pour tester la différence).
Le résultat est de type `bool` : `true` (vrai) ou `false` (faux).

Exemple 1.5.

```
# 4.2 < 2.4;;
- : bool = false
# 7 >= (-2);;
- : bool = true
# 0.1 +. 0.2 <> 0.3;;
- : bool = true          (* ce resultat est plus etonnant, mais deja vu en TP *)
# 64 mod 5 = 0;;
- : bool = false
```

Les opérateurs `&&` et `||` effectuent une ***évaluation paresseuse*** des expressions situées de part et d'autre, cela signifie que :

• pour évaluer l'expression `a && b` (ET logique), c'est l'expression `a` qui est évaluée la première.

⟶ si le résultat est `true`, l'évaluation de `b` est retournée
⟶ sinon c'est directement `false` qui est retourné

• pour évaluer l'expression `a || b` (OU logique), c'est l'expression `a` qui est évaluée la première.

⟶ si le résultat est `false`, l'évaluation de `b` est retournée
⟶ sinon c'est directement `true` qui est retourné

Exemple 1.6.

```
# 4 < 6 && 1024 <= 1024;;
- : bool = true
# 5.2 > 8. || 13 mod 2 = 0;;
- : bool = false
# not (4 = 7);;
- : bool = true
```

1.1.4 Variables locales

La construction vue ci-dessus permet de définir une ***variable globale***.
Une variable est dite ***locale*** lorsqu'elle n'est définie que dans le cadre d'une expression.
On utilise pour cela la construction `let ... in`

Exemple 1.7. On souhaite convertir 47 heures en minutes :

```
# let min_par_h = 60 in
       47 * min_par_h;;
- : int = 2820
# min_par_h;;
Error: Unbound value min_par_h     (* min_par_h n'est plus definie *)
```

1.1.5 Références

On a vu que les variables définies comme précédemment étaient ***immuables*** (leur valeur n'est pas modifiable).

Lorsqu'on souhaite utiliser des variables ***mutables*** on utilise des ***références***, que l'on déclare à l'aide du mot-clé `ref` en adoptant la syntaxe suivante :

```
let ma_ref = ref ma_valeur;;
```

- L'accès à la valeur de la réfécence se fait en ajoutant un ! devant son nom : `!ma_ref`
- L'assignation d'une nouvelle valeur s'effectue selon la syntaxe `ma_ref := autre_valeur;;`

Exemple 1.8.

```
# let x = ref 3;;
val x : int ref = {contents = 3}
# x := 4 * !x;;
- : unit = ()
# x;;
- : int ref = {contents = 12}
# !x;;
- : int = 12
```

Remarque 1.5. Une référence dont la valeur est de type `int` est de type `int ref`.

Pour bien comprendre la différence entre une variable standard et une référence, voici un exemple d'évaluation d'expressions a priori équivalentes :

Sans référence :

```
# let x = 4;;
val x : int = 4
# let add y = x + y;;
val add : int -> int = <fun>
# add 8;;
- : int = 12
# let x = 32;;
val x : int = 32
# add 8;;
- : int = 12
```

Avec référence :

```
# let x = ref 4;;
val x : int ref = {contents = 4}
# let add y = !x + y;;
val add : int -> int = <fun>
# add 8;;
- : int = 12
# x := 32;;
- : unit = ()
# add 8;;
- : int = 40
```

Remarque 1.6. On n'utilisera des références que lorsque c'est vraiment nécessaire.

1.2 Fonctions

1.2.1 Déclaration et signature

On rappelle que Ocaml est avant tout un langage de ***programmation fonctionnelle***. Les fonctions y occupent donc naturellement une place importante.

Une fonction Ocaml ne devrait pas générer d'***effet de bord***. On parle d'effet de bord lorsque l'évaluation d'une expression (dite ***impure***) affecte l'environnement extérieur à cette expression, par exemple en modifiant la valeur d'une variable globale, en faisant de l'affichage sur écran ou bien en modifiant le contenu d'un fichier.

Pour déclarer une fonction on utilise la syntaxe suivante : `let fonction argument = expression;;`

Exemple 1.9.

```
# let carre x = x * x;;
val carre : int -> int = <fun>
# carre 12;;        (* inutile de mettre des parentheses *)
- : int = 144
# carre (-9);;        (* sauf cas particuliers *)
- : int = 81
```

Une fonction possède une ***signature***, inférée par le compilateur ou l'interpréteur.
En notant `type_entree` le type de `argument` et `type_sortie` le type de `expression`,
la signature de la fonction est : `type_entree -> type_sortie`

Exemple 1.10. La fonction `carre` définie plus haut à pour signature `int -> int`.

Remarque 1.7. Lorsqu'on évalue l'expression `carre (3 * 4)` l'expression `3 * 4` est d'abord évaluée puis le résultat est passé comme argument à la fonction. C'est donc `carre 12` qui est évalué.
On dit que Ocaml est un ***langage strict*** : l'évaluation des arguments se font avant l'appel de la fonction, contrairement aux langages dits paresseux pour lesquels l'évaluation d'un argument n'est effectuée qu'au moment de sa première utilisation. (Voir le fonctionnement des opérateurs `&&` et `||` page 7)

Une fonction peut tout à fait ne pas avoir d'argument.

Exemple 1.11.
La fonction `reset_x` déclarée ci-dessous, de type `unit`, permet de réinitialiser la référence `x` à la valeur 0.

```
# let x = ref 12;;
val x : int ref = {contents = 12}
# let reset_x () = x := 0 ;;
val reset_x : unit -> unit = <fun>
# reset_x ();;
- : unit = ()
# x;;
- : int ref = {contents = 0}
```

Exemple 1.12. La fonction `swap` déclarée ci-dessous, de type `unit`, permet d'échanger les valeurs de deux références.

```
# let swap x y = let z = !x in     (* la variable locale z stocke la valeur de x *)
      x := !y;
      y := z
  ;;
val swap : 'a ref -> 'a ref -> unit = <fun>
# let x, y = ref 16, ref 64;;
val x : int ref = {contents = 16}
val y : int ref = {contents = 64}
# swap x y;;
- : unit = ()
# x, y;;
- : int ref * int ref = ({contents = 64}, {contents = 16})
```

On notera que l'expression `x, y;;` est de type `int ref * int ref` : il s'agit d'un couple de références sur entiers.

Pour déclarer une fonction à plusieurs variables on utilise la syntaxe :

```
let fonction arg1 arg2 arg3 ... = expression;;
```

La signature d'une telle fonction est alors de la forme : `type1 -> type2 -> type -> ... -> type_sortie`

En réalité il s'agit d'une suite de fonctions à un seul argument imbriquées entre elles. C'est ce qu'on appelle la ***curryfication***. Un avantage notable est qu'on peut appliquer partiellement une fonction.

Exemple 1.13.

```
# let prod x y = x * y;;
val prod : int -> int -> int = <fun>
# prod 2;;
- : int -> int = <fun>
# let double = prod 2;;
val double : int -> int = <fun>
# double 3;;
- : int = 6
```

On trouve fréquemment des variables locales dans des fonctions.

Exemple 1.14.

```
# let conversion_h_min heure = let min_par_h = 60 in
             heure * min_par_h;;
val conversion_h_min : int -> int = <fun>
# conversion_h_min 3;;
- : int = 180
```

1.2.2 Polymorphisme

Il peut arriver que les types ne puissent par être inférés car il n'y a pas de contrainte particulière induite par l'expression évaluée (contrairement à l'utilisation de l'opérateur + par exemple).
Dans ce cas on note les différents types possibles `'a`, `'b`, etc. On conserve la même notation lorsque deux type doivent être identiques dans une signature.

Exemple 1.15.

```
# let first x y = x;;
val first : 'a -> 'b -> 'a = <fun>
# let second x y = y;;
val second : 'a -> 'b -> 'b = <fun>
# first 4 8;;
- : int = 4
# second true false;;
- : bool = false
```

1.2.3 Type unit

Certaines expressions ne renvoient aucune valeur après évaluation, ce qui se note `()`. Ces expressions sont de type `unit`. Celles-ci peuvent en revanche générer des effets de bord. C'est le cas par exemple lorsqu'une expression utilise l'une des fonctions `print_int`, `print_float`, `print_string` ou `print_newline` qui permettent de faire de l'affichage sur écran.

Exemple 1.16.

```
# let a = 4;;
val a : int = 4
# print_string "la racine de ";
  print_int (a*a);
  print_string " vaut ";
  print_int a;
  print_newline ();;
la racine de 16 vaut 4
- : unit = ()
```

Une fonction dont le résultat est de type `unit` est appellée une ***procédure***. Les fonctions `reset_x` et `swap` des exemples 1.11 et 1.12 (page 9) sont également des procédures. L'appel de ces fonctions n'engendre pas d'affichage mais modifie l'environnement extérieur en modifiant les valeurs de références.

Les fonctions `read_int`, `read_float` et `read_line`, permettent quant à elles de lire respectivement un entier, un flottant et une chaîne de caractères saisie par l'utilisateur.

Exemple 1.17.

```
# let nb = read_int () in
       print_string "Le double de ";
       print_int nb;
       print_string " vaut ";
       print_int (2 * nb);
       print_newline ();;
16             (* cette valeur est saisie par l'utilisateur *)
Le double de 16 vaut 32
- : unit = ()
```

1.2.4 Fonctions anonymes - ordre supérieur

Il n'est pas toujours utile de donner un nom à une fonction. C'est le cas par exemple si l'on souhaite n'appeler cette fonction qu'une seule fois. On utilise alors la syntaxe suivante : `fun x -> ...`

Exemple 1.18.

```
# (fun x -> x * x - x) 8;;
- : int = 56
```

Ainsi, il est tout à fait équivalent d'écrire : `let f x = expr` et `let f = fun x -> expr`

En fait, une fonction Ocaml est une valeur comme les autres. La deuxième syntaxe ci-dessus est d'ailleurs tout à fait identique à celle utilisée pour déclarer des variables de l'un des types vus précédemment. On peut donc tout à fait passer une fonction comme argument d'une autre fonction.

On dit qu'une fonction Ocaml est d'***ordre supérieur*** lorsqu'elle prend en argument une ou plusieurs fonctions.

Exemple 1.19.

```
# let eval f g = (f 0) * (g 0);;
val eval : (int -> int) -> (int -> int) -> int = <fun>
# eval (fun x -> x + 2) (fun x -> 2 * x - 4);;
- : int = -8
```

1.2.5 Fonctions récursives

Une fonction est dite ***récursive*** lorsqu'elle est déclarée en utilisant une expression faisant appel à elle-même.

On déclare une fonction récursive Ocaml à l'aide du mot-clé `rec` :
`let rec ma_fonction arg = ...`

Exemple 1.20. La fonction ci-dessous permet de calculer la factorielle d'un nombre :

$$!n = \begin{cases} 1 & \text{si } n = 0 \\ n \times !(n-1) & \text{sinon} \end{cases}$$

```
# let rec factorielle n =
       if n = 0 then 1
       else n * factorielle (n-1);;
val factorielle : int -> int = <fun>
# factorielle 5;;
- : int = 120
```

Remarque 1.8. Attention, lorsqu'on écrit une fonction récursive il faut s'assurer d'avoir un « cas de base » afin d'être certain que l'appel de cette fonction ne générera pas une infinité d'appels à cette même fonciton.
Dans le cas de la fonction `factorielle` définie ci-dessus, il s'agit du cas $n = 0$.

Remarque 1.9. Une même fonction mathématique peut être implémentée de différentes façons, de manière récursive ou non. Nous verrons dans un prochain chapitre qu'il faudra rester vigilant notamment quant à la complexité en temps. En effet, une fonction récursive peut générer un arbre d'appels important rendant l'exécution impossible en un temps acceptable.

1.3 Structures de contrôle

1.3.1 Structure conditionnelle if ... then ... else ...

L'expression `if condition then expr1 else expr2` évalue d'abord l'expression `condition`, qui doit être un booléen. Si le résultat est `true`, c'est `expr1` qui est évaluée puis retournée. Sinon, c'est `expr2`.

Exemple 1.21.

```
# let abs x = if x > 0.
                  then x
            else -. x
;;
val abs : float -> float = <fun>
```

Remarque 1.10. Le `else` n'est pas toujours nécessaire. On n'écrit pas de `else` s'il n'y a rien à évaluer dans le cas où la condition ne serait pas satisfaite.

Remarque 1.11. Si on souhaite dstinguer plus de deux cas on peut imbriquer d'autres `if` :

```
if condition1 then expr1 else
if condition2 then expr2 else
...
if conditionp then exprp else expression;;
```

1.3.2 begin ... end

Lorsqu'une expression, suivant par exemple un `then`, est une ***séquence*** de plusieurs expressions, il est nécessaire d'utiliser des parenthèses ou bien la structure `begin ... end` afin de grouper ces expressions.

Ainsi, on écrira au choix :

```
if condition then
       (expr1;
       expr2;
       expr3)
```

ou bien

```
if condition then
   begin
       expr1;
       expr2;
       expr3
   end
```

À défaut, malgré une bonne indentation (qui on le rappelle n'a aucune importance), cela sera interprété comme :

```
(if condition then
       expr1);
       expr2;
       expr3
```

1.3.3 Structure itérative : boucle for

Une boucle `for` a la syntaxe suivante :

```
for var = indice_debut to indice_fin do
   expression
done
```

Exemple 1.22.

```
# for i = 1 to 9 do
       print_string "Le carre de ";
       print_int i;
       print_string " vaut ";
       print_int (i*i);
       print_newline ()
done;;
Le carre de 1 vaut 1
Le carre de 2 vaut 4
Le carre de 3 vaut 9
Le carre de 4 vaut 16
Le carre de 5 vaut 25
Le carre de 6 vaut 36
Le carre de 7 vaut 49
Le carre de 8 vaut 64
Le carre de 9 vaut 81
- : unit = ()
```

1.3.4 Structure itérative : boucle while

Une boucle `while` a la syntaxe suivante :

```
while condition do
    expression
done
```

Exemple 1.23.

```
# let pgcd n m = let q, r = ref n, ref m in
       while (!r <> 0) do
              let reste = !q mod !r in
              q := !r;
              r := reste
       done;
  !q;;
val euclide : int -> int -> int = <fun>
# pgcd 256 48;;
- : int = 16
# pgcd 246 13;;
- : int = 1
```

On peut utiliser la fonction `pgcd` pour déclarer une fonction `premiers_entre_eux` qui détermine si deux entiers naturels sont premiers entre eux ou non :

```
# let premiers_entre_eux n m = (pgcd n m = 1);;
val premiers_entre_eux : int -> int -> bool = <fun>
# premiers_entre_eux 256 48;;
- : bool = false
# premiers_entre_eux 246 13;;
- : bool = true
```

1.4 Types structurés prédéfinis

1.4.1 n-uplets

Un n-uplet Ocaml, est formé de n valeurs, de type éventuellement différents, séparées par des virgules. On déclare un n-uplet selon la syntaxe suivante :

```
let mon_n_uplet = val1, ..., valn;;
```

On obtient alors une variable de type `type1 * ... * typen` où `type1`, ..., `typen` sont les types des valeurs formant le n-uplet.

Exemple 1.24.

```
# let tuple = (5, true, "hello", '@');;
val tuple : int * bool * string * char = (5, true, "hello", '@')
```

Pour accéder aux différents éléments d'un n-uplet, il faut procéder à une déclaration ***déstructurante***.

Exemple 1.25.

```
# let (val1, val2, val3, val4) = tuple;;
val val1 : int = 5
val val2 : bool = true
val val3 : string = "hello"
val val4 : char = '@'
```

Remarque 1.12. Les éléments d'un n-uplet ne sont pas modifiables en place, ils sont ***immuables***.

1.4.2 Listes

Une liste Ocaml est construite à partir de la liste vide `[]`, à laquelle on a éventuellement ajouté des éléments en tête de liste, tous de même type, avec l'opérateur `::` (prononcer « cons »).

On obtient alors une liste de type `'a list` où `'a` correspond au type de tous les éléments de la liste.

Une liste de `n` éléments peut ainsi être déclarée selon la syntaxe suivante :

```
let ma_liste = e1 :: e2 :: ... :: en :: [];;
```

Une autre syntaxe possible est la suivante : `let ma_liste = [e1; e2; ...; en];;`

La fonction `List.length` permet d'accéder à la longueur d'une liste.

Exemple 1.26.

```
# let premiers = [2; 3; 5; 7; 11; 13; 17; 19];;
val premiers : int list = [2; 3; 5; 7; 11; 13; 17; 19]
# let voyelles = 'a' :: 'e' :: 'i' :: 'o' :: 'u' :: 'y' :: [];;
val voyelles : char list = ['a'; 'e'; 'i'; 'o'; 'u'; 'y']
# List.length voyelles;;
- : int = 6
```

Les liste Ocaml sont ***immuables***. L'opérateur `::` (« cons ») est utilisé pour construire une nouvelle liste comportant un élément supplémentaire en tête de liste, mais ne permet pas de modifier une liste.

Exemple 1.27.

```
# let liste = ["farine"; "oeufs"; "lait"; "huile"];;
val liste : string list = ["farine"; "oeufs"; "lait"; "huile"]
# "chocolat"::liste;;
- : string list = ["chocolat"; "farine"; "oeufs"; "lait"; "huile"]
# liste;;
- : string list = ["farine"; "oeufs"; "lait"; "huile"]
```

Contrairement aux n-uplets ou aux chaînes de caractères, on ne peut pas accéder à un élément quelconque d'une liste sans passer par une fonction récursive (voir section 1.5.2 page 18).

Enfin, deux listes peuvent être concaténées en une nouvelle liste grâce à l'opérateur `@` mais attention, la complexité est en $\mathcal{O}(n)$ où n est la longueur de la seconde liste (voir prochain chapitre).

Exemple 1.28.

```
# [1.; 0.1; 0.01; 0.001] @ [0.0001; 0.00001];;
- : float list = [1.; 0.1; 0.01; 0.001; 0.0001; 1e-05]
```

1.4.3 Tableaux

Un tableau Ocaml est constitué d'un nombre fixé d'éléments, tous de même type.

Un tableau est de type `'a array` où `'a` correspond au type de tous les éléments du tableau.

Un tableau de `n` éléments peut être déclaré selon la syntaxe suivante :

```
let mon_tableau = [| e1; e2; ...; en |];;
```

Remarque 1.13. Le tableau vide se note `[| |]`.

La fonction `Array.length` permet d'accéder à la longueur d'un tableau.

Exemple 1.29.

```
# let tab = [| 8; 15; 6; 14; 18|];;
val tab : int array = [|8; 15; 6; 14; 18|]
# Array.length tab;;
- : int = 5
```

Contrairement aux listes, un tableau est ***mutable*** : on peut modifier en place ses éléments.

- `tab.(i)` permet d'accéder à l'élément `i` du tableau `tab` (à partir de l'indice 0).
- `tab.(i) <- new` permet de remplacer l'élément d'indice `i` par la valeur `new`.

On peut également créer un tableau en fixant son nombre d'éléments et en les initialisant tous à la même valeur.

Exemple 1.30.

```
# let etat = Array.make 8 true;;
val etat : bool array = [|true; true; true; true; true; true; true; true|]
# etat.(1) <- false; etat.(4) <- false;;
- : unit = ()
# etat;;
- : bool array = [|true; false; true; true; false; true; true; true|]
# etat.(8) <- false;;
Exception: Invalid_argument "index out of bounds".
```

Remarque 1.14. Dans l'exemple ci-dessus on observe une erreur « index out of bounds » lorsqu'on cherche à accéder à l'élément d'indice 8 car les indices vont de 0 à 7 pour un tableau à 8 éléments.
Nous parlerons des exceptions dans une prochaine section.

Enfin, on peut copier un tableau grâce à la fonction `Array.copy` . Attention en revanche, il s'agit d'une copie superficielle : si on modifie les éléments du tableau initial, cela n'affecte pas la copie. En effet, les deux tableaux pointent vers les mêmes valeurs, mais occupent des espaces mémoire différents. On pourra s'en convaincre avec l'exemple ci-dessous :

Exemple 1.31.

```
# let init = [|1; 2; 3; 4|];;
val init : int array = [|1; 2; 3; 4|]
# let copie = Array.copy init;; (* copie superficielle *)
val copie : int array = [|1; 2; 3; 4|]
# copie = init;;
- : bool = true
# copie == init;; (* l'operateur == compare les adresses memoire *)
- : bool = false
# let tab = init;;
val tab : int array = [|1; 2; 3; 4|]
# tab.(0) <- 64;;
- : unit = ()
# init.(0);;   (* toute modification de tab modifie egalement init *)
- : int = 64
```

1.4.4 Types option

Lorsqu'on souhaite indiquer la présence ou non d'une valeur on utilise le type `'a option` . Les variables de ce type peuvent alors valoir :

- `None` : ce qui correspond à l'absence de valeur
- `Some val` : ce qui correspond à la présence de la valeur `val`

Exemple 1.32. La fonction suivante détermine si un caractère est présent dans une chaîne de caractères ou non et donne sa dernière position le cas échéant :

```
# let position chaine c =
       let pos = ref None in
       let long = String.length chaine in
       for i = 0 to long - 1 do
              if chaine.[i] = c then pos := Some i
       done;
!pos;;
val position : string -> char -> int option = <fun>
# position "une chaine" 'h';;
- : int option = Some 5
# position "une chaine" 'z';;
- : int option = None
```

1.5 Filtrage de motifs

1.5.1 Syntaxe

Lorsqu'on souhaite renvoyer un résultat différent selon la valeur obtenue après l'évaluation d'une expression, on effectue un ***filtrage de motifs*** selon la syntaxe suivante :

```
match expression with
       | motif1 -> resultat1
       | motif2 -> resultat2
       ...
       | _ -> autre_resultat
```

Remarque 1.15. Le ***motif universel*** _ signifie « tout autre motif ».

Exemple 1.33. La fonction `factorielle` de l'exemple 1.20 page 12 aurait pu être déclarée de la façon suivante :

```
# let rec factorielle n = match n with
       | 0 -> 1
       | _ -> n * (factorielle (n-1) )
;;
val factorielle : int -> int = <fun>
# factorielle 5;;
- : int = 120
```

Le mot-clé `function` permet même d'obtenir une syntaxe encore plus concise en effectuant un filtrage sur le dernier argument de la fonction.

Exemple 1.34. Toujours pour la fonction `factorielle`, cela donne :

```
# let rec factorielle = function
       | 0 -> 1
       | n -> n * (factorielle (n-1) )
;;
val factorielle : int -> int = <fun>
# factorielle 5;;
- : int = 120
```

Remarque 1.16. La correspondance de motif est recherchée dans l'ordre des motifs proposés, il est donc important de les ordonner correctement. Dans l'exemple précédent, il est impératif d'indiquer le motif « 0 »avant le cas général.

Exemple 1.35.

```
# let fruit = function
        | "poire" -> true
        | "fraise" -> true
        | "pomme"-> true
        | "courgette" -> false
        | "potimarron" -> false
  ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
""
val fruit : string -> bool = <fun>
```

Attention à bien s'assurer que le filtrage est ***exhaustif***. Ce n'est pas le cas dans cet exemple (d'où le message d'alerte). La fonction est bien déclarée mais on obtiendra par exemple :

```
# fruit "poire";;
- : bool = true
# fruit "courgette";;
- : bool = false
# fruit "cerise";;
Exception: Match_failure ("//toplevel//", 76, -26).
```

1.5.2 Le cas des listes

Le filtrage de motifs permet de parcourir des listes en utilisant des fonction récursives.

Exemple 1.36. La fonction suivante permet de compter le nombre d'occurences d'une valeur dans une liste :

```
# let rec compte liste elem = match liste with
        | [] -> 0
        | e::q -> compte q elem + (if e = elem then 1 else 0)
;;
val compte : 'a list -> 'a -> int = <fun>
# compte [true; false; true; true; false] true;;
- : int = 3
# compte [true; false; true; true; false] false;;
- : int = 2
```

Remarque 1.17. On notera dans l'exemple précédent l'utilisation d'une expression avec `if-then-else`.

Remarque 1.18. On aurait pu vouloir indiquer directement un motif `elem::q` , cela ne fonctionne pas : les motifs ne doivent pas comporter de variables utilisées antérieurement ni deux fois la même variable.

Le filtrage peut être plus précis encore en indiquant des motifs plus spécifiques.

Exemple 1.37. La fonction suivante compte le nombre de fois qu'un élément recherché est présent deux fois consécutives :

```
# let rec couple liste elem = match liste with
        | [] -> 0
        | e1::e2::q when e1 = e2 && e1 = elem -> 1 + couple (e2::q) elem
        | e::q -> couple q elem
;;
val couple : 'a list -> 'a -> int = <fun>
# couple [1; 2; 2; 5; 2; 2; 2; -4; 0; 2] 2;;
- : int = 3
```

Remarque 1.19. On notera l'utilisation de `when` dès la description du motif à la place de l'utilisation d'un `if` ultérieurement.

1.6 Exceptions

Lorsqu'un comportement exceptionnel intervient (une erreur le plus souvent) il est signalé par une exception. On dit qu'une exception est ***levée***.

Exemple 1.38.

```
# 1/0;;
Exception: Division_by_zero.
```

Il est possible de ***rattraper*** une exception avec la construction `try ... with .`
À l'évaluation de l'expression `try expr with except -> val` on commence par évaluer l'expression `expr`. Si tout se passe bien, c'est cette valeur qui est renvoyée. Si l'exception `except` est levée, c'est la valeur `val` qui est renvoyée.

Exemple 1.39.

```
# try 1/0 with Division_by_zero -> 0;;
  - : int = 0
```

Remarque 1.20. On peut utiliser la même syntaxe que pour le filtrage de motifs dans le cas où plusieurs exceptions sont susceptibles d'être levées :

```
try expr with
    | except1 -> val1
    | except2 -> val2 ...
```

Il peut parfois être utile de lever soi-même une exception. on utilise pour cela le mot-clé `raise`.
Par ailleurs on peut déclarer de nouvelles exceptions selon la syntaxe `exception Mon_exception` voire `exception Mon_exception of type` s'il s'agit d'une exception avec argument.

Exemple 1.40.

```
# exception Droite_verticale;;
exception Droite_verticale
# let coef_dir a b = let (xa, ya) = a and (xb, yb) = b in
       if xa = xb then raise Droite_verticale
       else (yb -. ya) /. (xb -. xa) ;;
val coef_dir : float * float -> float * float -> float = <fun>
# coef_dir (0., 0.) (1., 4.);;
- : float = 4.
# coef_dir (1., 3.) (1., -2.);;
Exception: Droite_verticale.
```

Remarque 1.21. Le nom de l'exception déclarée doit impérativement commencer par une majuscule.

Remarque 1.22. Une expression commençant par le mot-clé `raise` peut avoir un type différent selon le contexte, en cohérence avec le type de l'expression complète.

Dans le cas où l'exception levée n'est pas ratrapable et on souhaite indiquer un message d'erreur plus précis, on a recours à l'exception prédéfinie `Failure of string`.
Celle-ci peut s'utiliser avec la syntaxe `failwith "message d'erreur"`.
Ce qui correspond en fait à `raise (Failure "message d'erreur")`.

1.7 Types construits

Un nouveau type de données peut être déclaré à l'aide du mot-clé `type`.

1.7.1 Types énumérés (ou somme ou union)

Un type ***énuméré*** (ou encore type ***somme*** ou type ***union***) est déclaré en indiquant l'ensemble (fini) des valeurs possibles de ce type, appelées ***constructeurs***, séparées par un | .

Par exemple la déclaration `type saison = Printemps | Ete | Automne | Hiver`
énumère les 4 valeurs possibles correspondant au type `saison`.

Remarque 1.23. Les constructeurs commencent obligatoirement par une majuscule, contrairement aux noms de variables qui commencent eux obligatoirement par une minuscule.

Remarque 1.24. L'ordre de déclaration des différents constructeurs induit une relation d'ordre sur le type énuméré. Ainsi on aura `Printemps < Ete < Automne < Hiver`.

Un constructeur peut éventuellement avoir des arguments, qu'on indique à l'aide du mot-clé `of`. Un type peut par ailleurs être construit récursivement. On peut avantageusement utiliser le filtrage de motif sur les types énumérés.

Exemple 1.41. On peut par définir un type `figure` comme ceci :

```
# type figure = None | Elem_Liste of (int * int) * figure;;
type figure = None | Elem_Liste of (int * int) * figure
# let rec nb_points fig = match fig with
       | None -> 0
       | Elem_Liste (point, sous_fig) -> 1 + nb_points sous_fig
;;
val nb_points : figure -> int = <fun>
# let triangle =
       Elem_Liste ( (3, 2), Elem_Liste ( (-1, 5), Elem_Liste ((0, 0), None) ));;
val triangle : figure =
       Elem_Liste ((3, 2), Elem_Liste ((-1, 5), Elem_Liste ((0, 0), None)))
# nb_points triangle;;
- : int = 3
```

1.7.2 Types enregistrements

Lorsque l'on souhaite définir des objets ayants plusieurs caractéristiques on peut déclarer leur type en renseignant entre accolades les différents ***champs*** possibles, en précisant à chaque fois le type associé.
La syntaxe est la suivante :

```
type mon_type_enr = { champs_1 : type_1 ; champs_2 : type_2 ; ... ; champs_n : type_n }
```

Pour déclarer une variable de type enregistrement on indiquera alors :

```
let var = { champs_1 = val_1 ; champs_2 = val_2 ; ... ; champs_n = val_n }
```

L'accès aux différentes valeurs de champs d'une variable s'effectuent ensuite en indiquant `var.champs` où `champs` est l'un des différents champs possibles.

Exemple 1.42.

```
# type rvb = { r : int ; v : int ; b : int};;
type rvb = { r : int; v : int; b : int; }
# let jaune = { r = 255; v = 255; b = 0};;
val jaune : rvb = {r = 255; v = 255; b = 0}
# jaune.r;;
- : int = 255
```

Par défaut, les variables de type enregistrement sont immuables.
Si l'on souhaite donner un caractère mutable à certains champs, on l'indique au moment de la déclaration du type enregistrement à l'aide du mot-clé `mutable`. La valeur d'un champs peut alors être modifiée en indiquant `var.champs <- nouvelle_valeur`.

Exemple 1.43.

```
# type eleve = {nom : string; prenom : string; mutable age : int};;
type eleve = { nom : string; prenom : string; mutable age : int; }
# let alice = {nom = "Wonder"; prenom = "Alice"; age = 18};;
val alice : eleve = {nom = "Wonder"; prenom = "Alice"; age = 18}
# alice.age <- alice.age + 1;;
- : unit = ()
# alice;;
- : eleve = {nom = "Wonder"; prenom = "Alice"; age = 19}
# alice.nom <- "Dupond";;
Error: The record field nom is not mutable
```

Remarque 1.25. Les références vues précédemment sont en fait des enregistrements dont le champ `contents` est mutable.

Les types enregistrement peuvent aussi être définis récursivement.

Exemple 1.44.

```
# type personne = {prenom : string; enfants : personne list};;
type personne = { prenom : string; enfants : personne list; }
# let moi = {prenom = "Jade"; enfants = []};;
val moi : personne = {prenom = "Jade"; enfants = []}
# let mon_frere = {prenom = "Antoine"; enfants = []};;
val mon_frere : personne = {prenom = "Antoine"; enfants = []}
# let maman = {prenom = "Josephine"; enfants = [moi; mon_frere]};;
val maman : personne =
 {prenom = "Josephine";
      enfants =
              [{prenom = "Jade"; enfants = []}; {prenom = "Antoine"; enfants = []}]}
```

1.7.3 Types mutuellement récursifs

Pour déclarer des types ***mutuellement récursifs***, c'est-à-dire qui sont construits l'un à partir de l'autre, on le fait conjointement avec le mot-clé and.

Exemple 1.45.

```
# type teinte = {nom : string; composition : couleur list} and couleur = Rouge | Vert
    | Bleu | Melange of teinte;;
type teinte = { nom : string; composition : couleur list; }
and couleur = Rouge | Vert | Bleu | Melange of teinte
# let purple = {nom = "violet"; composition = [Rouge; Bleu]};;
val purple : teinte = {nom = "violet"; composition = [Rouge; Bleu]}
# let peinture = Melange purple;;
val peinture : couleur =
  Melange {nom = "violet"; composition = [Rouge; Bleu]}
```

1.8 Ocaml sur sa propre machine

1.8.1 Installation de Ocaml

Sous Linux/macOS

On peut installer Ocaml via `opam` (gestionnaire de paquet spécialisé pour Ocaml) ou bien avec son gestionnaire de paquet habituel

Exemple 1.46. Sous Ubuntu :

```
sudo apt-get install opam
```

Exemple 1.47. Sous macOS avec Homebrew :

```
brew install opam
```

Opam doit être initialisé avec les commandes suivantes :

```
opam init -y
eval $(opam env)
```

Sous windows

Suivre les instructions de la page `http://fdopen.github.io/opam-repository-mingw/installation/` (choisir l'installateur graphique version 32-bit ou 64-bit selon votre machine).

Cela va créer un environnement adapté Cygwin (suite de logiciels libres permettant de fournir un environnement type Unix) et y installer opam et Ocaml.

On initialise `opam` depuis le terminal Cygwin avec les commandes suivantes :

```
opam init -y
eval $(opam env)
```

1.8.2 Boucle d'intéraction (toplevel) et compilateur

Pour faciliter l'apprentissage de ce nouveau langage, nous avons jusqu'à présent utilisé un ***interpréteur*** Ocaml en mode interactif.
Pour utiliser une boucle d'intéraction (ou ***toplevel***) Ocaml depuis un terminal (éventuellement Cygwin) on saisit simplement la commande suivante :

```
$ ocaml
```

Ou bien, pour avoir la possibilité d'utiliser les flèches afin de récupérer la saisie précédente :

```
$ ledit ocaml
```

On obtient alors une information sur la version d'Ocaml utilisée ainsi qu'une invite # à la suite de laquelle il convient de saisir les expressions à évaluer :

```
$ ledit ocaml
        OCaml version 4.02.3
#
```

On quitte ce mode comme ceci :

```
# #quit;;
$
```

Cependant, cette utilisation ne permet pas de sauvegarder ses lignes de code.

Pour exécuter un programme Ocaml enregistré sur sa machine on commence par l'enregistrer dans un fichier ayant une extension `.ml`.

Par exemple on peut enregistrer le programme ci-dessous dans un fichier nommé `hello.ml` :

```
let () = print_string "Hello world !\n"
```

Ce programme peut alors être exécuté dans un terminal avec la commande suivante :

    $ ocaml hello.ml

Cette commande demande à l'interpréteur Ocaml d'exécuter le programme `hello.ml` ce qui a pour effet d'afficher la chaîne de caractères "Hello World !" comme prévu.

Remarque 1.26. Jusqu'à présent on utilisait systématiquement le double point-virgule ; ; pour conclure la saisie des déclarations. Ce n'est plus utile si on ne passe pas par le mode interactif. Pour s'en passer il faut néanmoins s'assurer que notre programme correspond bien à une suite de déclarations.
Le programme suivant génère par exemple une erreur :

```
let a = 16
print_int a ; print_newline ()
```

En effet rien n'indique que la déclaration `let` est terminée à la fin de la première ligne, les expressions correspondant à l'affichage sont donc également incluses dans la déclaration.
On corrige cela aisément comme ceci :

```
let a = 16
let () = print_int a ; print_newline ()
```

Enfin, pour produire un fichier exécutable on utilise le ***compilateur*** Ocaml.
La commande qui convient est la suivante :

    $ ocamlopt hello.ml -o hello

Cette commande permet de compiler le code du fichier `hello.ml` et de produire du code exécutable par un ordinateur (en langage machine). On ajoute l'option `-o hello` pour préciser le nom que l'on souhaite donner au fichier exécutable généré. Ce dernier s'exécute ensuite avec la commande suivante :

    $ ./hello

1.8.3 Utilisation de VSCodium

VSCodium est un éditeur de code multi-plateforme, open source, supportant une dizaine de langages de programmation. Il s'agit d'une distribution sous licence libre du logiciel VS Code (Visual Studio Code) de l'éditeur Microsoft.

Installer VSCodium depuis `https://vscodium.com/`

Sous Windows, cliquer sur « Download latest release », selectionner la dernière version puis exécuter par exemple `VSCodiumUserSetup-x64-1.71.2.22258.exe`

Il faut ensuite installer deux paquets permettant d'utiliser Ocaml depuis VSCodium. On saisit pour cela dans un terminal (Cygwin pour Windows) la commande suivante :

```
opam install ocaml-lsp-server ocamlformat
```

Sous Windows, VSCodium doit être lancé depuis l'environnement cygwin avec une commande du type (adapter selon l'emplacement de votre exécutable) :

```
"/cygdrive/c/Users/LCHALMAIN/AppData/Local/Programs/VSCodium/VSCodium.exe"
```

Le répertoire Unix `/cygdrive/c/Users/LCHALMAIN` correspond au chemin Windows `C:\Users\LCHALMAIN`.
On peut également créer un raccourci sur le bureau puis dans les propriétés du raccourci éditer la cible en

ajoutant avant le chemin vers l'utilitaire `ocaml-env-win` :
`C:\Ocaml64\usr\local\bin\ocaml-env-win.exe "C:\Users\LCHALMAIN\chemin\vers\VSCodium.exe"`

Sur VSCodium, ajouter les extensions Ocaml Platform (et éventuellement French Language Pack) en cliquant sur . On choisira également dans les paramètres (> Éditeur de texte > Suggestions) de ne pas accepter les suggestions après appui sur « Entrée ».

1.9 Modules

Ocaml dispose d'une bibliothèque fournissant un certain nombre de types et de fonctions. Cette bibliothèque est composée de ***modules*** correspondant à certains usages (manipuler des listes, des tableaux, etc.).

Un module est constitué de deux fichiers de même nom :

- Un fichier d'***implémentation*** dont l'extension est `.ml`, contenant des déclarations de types et de valeurs.
- Un fichier d'***interface*** dont l'extension est `.mli`, contenant les déclarations de types et les signatures des valeurs déclarées dans le fichier d'implémentation.

Les types et valeurs d'un module sont accessibles en utilisant le nom du module commençant par **une majuscule** et une notation pointée.

Exemple 1.48. La fonction `List.length` correspond à la fonction `length` déclarée dans le module `List` (dans le fichier `list.ml`).

Pour utiliser directement les éléments d'un module, sans cette notation, on peut éventuellement ouvrir ce module en indiquant `open module`. Attention dans ce cas à ce qu'il n'y ait pas de déclarations différentes pour un même identificateur dans des modules différents, il vaut donc mieux éviter cette pratique pour éviter toute ambiguïté.

Remarque 1.27. Sous Linux, on trouve la bibliothèque Ocaml à l'emplacement `/usr/lib/ocaml`.

Remarque 1.28. Il n'est pas attendu d'un élève de CPGE de connaître la bibliothèque Ocaml. Les fonctions disponibles dans les modules présentés ci-dessous peuvent être utilisées ***à condition qu'il y en ait la mention explicite***.

1.9.1 Le module List

Les fonctions suivantes peuvent être utilisées après rappel :

Fonction	Signature	Description
`List.length`	`'a list -> int`	Renvoie le nombre d'éléments d'une liste
`List.mem`	`'a -> 'a list -> bool`	Teste si un élément est présent dans une liste ou non
`List.exists`	`( 'a -> bool ) -> 'a list -> bool`	Teste si un prédicat est vérifié pour au moins l'un des éléments d'une liste
`List.for_all`	`( 'a -> bool ) -> 'a list -> bool`	Teste si un prédicat est vérifié pour tous les éléments d'une liste
`List.filter`	`( 'a -> bool ) -> 'a list -> 'a list`	Renvoie la liste des éléments d'une liste qui vérifient un certain prédicat
`List.map`	`( 'a -> 'b ) -> 'a list -> 'b list`	Renvoie une liste correspondant aux éléments d'une liste auxquels une fonction a été appliquée
`List.iter`	`( 'a -> unit ) -> 'a list -> unit`	Applique une procédure à chacun des éléments d'une liste

Remarque 1.29. La fonction `List.length` est explicitement au programme et donc toujours utilisable.

1.9.2 Le module Array

Les fonctions suivantes peuvent être utilisées après rappel :

Fonction	Signature	Description
`Array.length`	`'a array -> int`	Renvoie le nombre d'éléments d'un tableau
`Array.make`	`int -> 'a -> 'a array`	Initialise un tableau contenant à chaque fois la même valeur
`Array.copy`	`'a array -> 'a array`	Renvoie une copie superficielle d'un tableau
`Array.make_matrix`	`int -> int -> 'a -> 'a array array`	Initialise un tableau à 2 dimensions contenant à chaque fois la même valeur
`Array.init`	`int -> ( int -> 'a ) -> 'a array`	`Array.init n f` renvoie le tableau `[|f 0; f 1; ...; f (n-1)|]`
`Array.mem`	`'a -> 'a array -> bool`	Teste si un élément est présent dans un tableau ou non
`Array.exists`	`( 'a -> bool ) -> 'a array -> bool`	Teste si un prédicat est vérifié pour au moins l'un des éléments d'un tableau
`Array.forall`	`( 'a -> bool ) -> 'a array -> bool`	Teste si un prédicat est vérifié pour tous les éléments d'un tableau
`Array.map`	`( 'a -> 'b ) -> 'a array -> 'b array`	Renvoie un tableau correspondant aux éléments d'un tableau auxquels une fonction a été appliquée
`Array.iter`	`( 'a -> unit ) -> 'a array -> unit`	Applique une procédure à chacun des éléments d'un tableau

Remarque 1.30. Les fonctions `Array.length`, `Array.make` et `Array.copy` sont explicitement au programme et sont donc toujours utilisables.

Chapitre 2

Analyse des programmes

2.1 Bonnes pratiques

Peu importe le langage utilisé, il est important d'avoir de bonnes habitudes de programmation afin de faciliter la compréhension du code, son analyse, ou encore faciliter le débuggage.

- Même si la syntaxe du langage ne l'impose pas, aérer et indenter son code notamment pour voir apparaître clairement les structures imbriquées.
- Donner des noms les plus explicites possible aux variables afin de faciliter la compréhension du code.
- Insérer des commentaires, pas pour paraphraser le code, mais pour donner des informations supplémentaires facilitant la compréhension ou bien indiquant des éléments importants pour l'analyse des programmes (par exemple, une précondition, un invariant).
- Apprendre à comprendre les erreurs de compilation pour identifier plus efficament les sources d'erreurs à l'avenir. Bien prendre en compte les messages d'erreurs même s'il ne s'agit que d'alertes.
- Compiler très régulièrement : il est beaucoup plus simple de trouver la provenance d'une erreur quand on sait où chercher, il ne faut donc pas avoir fait trop de modifications depuis la dernière compilation.
- Réaliser des jeux de tests pertinents pour s'assurer du bon fonctionnement d'un programme. Notamment sur des valeurs particulières, pour s'assurer qu'il n'y a pas de problème lorsqu'on est dans les conditions limites d'utilisation (pas de division par 0 par exemple), mais aussi pour tester les performances du programme (sur de grands entiers par exemple).
- Pour se donner l'intuition d'une solution à un problème donné, il peut être très utile d'effectuer à la main (papier/crayon) la résolution sur un cas particulier. On identifiera ainsi plus clairement les différentes étapes de l'algorithme à mettre en œuvre et les éventuels cas de base.
- Dans la phase d'écriture d'un programme on peut utiliser des assertions (fonction `assert`) afin de vérifier que les préconditions sont toujours vérifées et obtenir des indications supplémentaires sur les parties du code générant d'éventuelles erreurs. Exemple d'erreur : l'exception `Divizion_by_zero` est levée, une assertion permet d'identifier qu'à la ligne 12 la variable `i` prend la valeur 0, on pourra donc corriger plus facilement la source de l'erreur.

2.2 Correction

Lorsqu'on souhaite démontrer que le résultat renvoyé par une fonction est correct, la réalisation d'un jeu de tests est bien entendu insuffisant.

Pour prouver la correction d'un algorithme on doit tout d'abord énoncer clairement sa ***spécification*** en précisant :

- les différentes entrées admissibles, pour lesquelles on peut imposer des ***préconditions***
- le résultat attendu après l'exécution de l'algorithme

Exemple 2.1. Pour la fonction `factorielle` définie dans l'exemple 1.20 page 12 la spécification est la suivante :

- `factorielle` prend en argument un entier n. La précondition est que n doit être positif ou nul.
- Le résultat attendu est $n!$.

La preuve de la correction s'effectue alors généralement en utilisant un raisonnement par récurrence, notamment lorsqu'on étudie des fonctions récursives, ou bien lorsque la présence d'une boucle permet de raisonner sur le nombre d'itérations.

Théorème 2.1 (**Principe de récurrence simple**).

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si les deux conditions suivantes sont vérifiées :

- $P(0)$ est vérifiée ***(Initialisation)***
- lorsqu'on suppose, pour un certain entier n, que $P(n)$ est vérifiée (c'est l'***hypothèse de récurrence***), alors $P(n+1)$ est encore vérifiée ***(Hérédité)***

alors la propriété $P(n)$ est vérifiée pour tout entier $n \in \mathbb{N}$.

Exemple 2.2. Correction de la fonction `factorielle`
Montrons par récurrence sur n que pour tout $n \in \mathbb{N}$, la propriété $P(n)$: « `factorielle n` $=$ `n!` » est vérifée.

- Si $n = 0$: `factorielle n` $= 1 = 0!$. $P(0)$ est donc bien vérifée.
- Hérédité : On suppose que, pour un certain $n \in \mathbb{N}$, $P(n)$ est vérifée, montrons qu'alors $P(n+1)$ est encore vérifée.

$$
\begin{aligned}
\texttt{factorielle (n+1)} &= (n+1) \times (\texttt{factorielle n}) \\
&= (n+1) \times n! && \text{par hypothèse de récurrence} \\
&= (n+1)!
\end{aligned}
$$

Ainsi, par récurrence sur n, $P(n)$ est vérifée pour tout $n \in \mathbb{N}$.
La fonction `factorielle` est donc correcte puisqu'elle satisfait bien sa spécification.
Parfois cependant, on a besoin de davantage de souplesse quant au rang auquel l'hypothèse de récurrence doit être vérifée. Dans ce cas, on utilise plutôt une récurrence forte :

Théorème 2.2 (**Principe de récurrence forte**).

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si les deux conditions suivantes sont vérifiées :

- $P(0)$ est vérifiée ***(Initialisation)***
- lorsqu'on suppose, pour un certain entier n, que $P(k)$ est vérifiée pour tous les entier k tels que $k < n$ alors $P(n)$ est encore vérifiée ***(Hérédité)***

alors la propriété $P(n)$ est vérifiée pour tout entier $n \in \mathbb{N}$.

Exemple 2.3. On définit la fonction Ocaml récursive `puiss_2 : int -> int * int` dont la spécification est la suivante :

- `puiss_2` prend en argument un entier naturel n non nul (précondition)
- Le résultat renvoyé est l'unique couple $(a, p) \in \mathbb{N} \times \mathbb{N}^\star$ tel que $n = a \times 2^p$ et a n'est pas divisible par 2

```
# let rec puiss_2 n =          (* n > 0 *)
      if n mod 2 <> 0 then
            n, 0
      else let a, p = puiss_2 (n/2) in
            a, (p+1)
;;
val puiss_2 : int -> int * int = <fun>
```

```
# puiss_2 44;;
- : int * int = (11, 2)
# puiss_2 56;;
- : int * int = (7, 3)
```

On note $P(n)$: « `puiss_2 n` est l'unique couple $(a, p) \in \mathbb{N}^\star \times \mathbb{N}$ tel que $n = a \times 2^p$ et 2 ne divise pas a ».
Montrons par récurrence forte sur n que $P(n)$ est vraie pour tout $n \in \mathbb{N}^\star$.

- Si $n = 1$: comme 1 n'est pas divisible par 2, `puiss_2 1 = (1, 0)`.
  Or on a bien $1 = 1 \times 2^0$ et 2 qui ne divise pas 1. La propriété $P(1)$ est donc bien vérifiée.
- Hérédité : On suppose que, pour un certain $n \in [\![2; +\infty[\![$, $P(k)$ est vérifée pour tout $k \in \mathbb{N}^\star$ tel que $k < n$. Montrons qu'alors $P(n)$ est encore vérifée.
  - Si n n'est pas divisible par 2, alors `puiss_2 n = (n, 0)`. Or on a bien $n = n \times 2^0$.
  - Sinon, en notant `(a, p) = puiss_2 (n/2)` on a `puiss_2 n = (a, p+1)`.

Or,
$$\begin{aligned} n &= \frac{n}{2} \times 2 \\ &= a \times 2^p \times 2 \qquad \text{où 2 ne divise pas } a \text{ par hypothèse de récurrence} \\ n &= a \times 2^{p+1} \end{aligned}$$

Ainsi, par récurrence forte sur n, $P(n)$ est vérifiée pour tout $n \in \mathbb{N}^\star$.
On en déduit la correction de la fonction `puiss_2 n`.

Dans le cas d'une boucle (`for` ou `while`) on utilise un invariant de boucle pour prouver la correction :

Définition 2.1. Un ***invariant de boucle*** est une propriété qui doit :

- être vérifiée avant l'exécution de la boucle
- rester vérifiée après chaque itération

En particulier, cette propriété sera encore vérifiée à la fin de l'exécution de la boucle et peut donc, si elle est bien choisie, permettre de prouver la correction de l'algorithme.

Exemple 2.4. On définit la fonction Ocaml `puissance_2 : int -> int * int` dont la spécification est la même que précédemment mais utilise cette fois une boucle `while` :

- `puissance_2` prend en argument un entier naturel n non nul (précondition)
- Le résultat renvoyé est l'unique couple $(a, p) \in \mathbb{N} \times \mathbb{N}^\star$ tel que $n = a \times 2^p$ et a n'est pas divisible par 2

```
# let puissance_2 n = let a, p = ref n, ref 0 in          (* n > 0 *)
      while !a mod 2 = 0 do          (* invariant : !a * 2^!p = n *)
            a := !a / 2;
            p := !p + 1
      done;
(!a, !p);;
val puissance_2 : int -> int * int = <fun>
# puissance_2 8;;
- : int * int = (1, 3)
# puissance_2 31;;
- : int * int = (31, 0)
```

Montrons que la propriété $a \times 2^p = n$ est invariante.

- À l'initialisation des variables, $a = n$ et $p = 0$. On a donc bien $a \times 2^p = n \times 2^0 = n$.
- Supposons la propriété vraie au début d'un tour de boucle : $a \times 2^p = n$.
  On note respectivement a' et p' les nouvelles valeurs des variables a et p à l'issue du tour de boucle.
  Montrons que la proriété $a' \times 2^{p'} = n$ est encore vérifiée.
  On a $a' = \dfrac{a}{2}$ et $p' = p + 1$ alors $a' \times 2^{p'} = \dfrac{a}{\cancel{2}} \times 2^{p+\cancel{1}} = a \times 2^p = n$ (par hypothèse).

Ainsi, *si le programme termine* (cela reste à démontrer !), l'invariant donné ci-dessus est toujours vérifié à la fin de l'exécution du programme. Ce qui prouve la correction du programme puisque le couple (a, p) renvoyé vérifie bien l'égalité $n = a \times 2^p$ et 2 ne divise pas a (c'est cette dernière propriété qui aura interrompu l'exécution de la boucle `while`).

2.3 Terminaison

Lorsqu'un algorithme est récursif ou fait appel à une boucle `while` il est nécessaire de prouver sa terminaison. En effet, dans le cas d'un algorithme récursif, il faut s'assurer qu'on finit bien par appeler un cas de base qui interrompra la succession d'appels récursifs. Dans le cas d'une boucle `while`, il faut s'assurer que la condition d'entrée en boucle finit par ne plus être vérifiée. Dans les deux cas, on utilise pour cela un variant :

Définition 2.2. Un ***variant*** est un paramètre défini en fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives
- décroît strictement à chaque appel récursif ou itération d'une boucle (selon le cas)

Théorème 2.3. Tout algorithme possédant un variant termine.

Exemple 2.5. Revenons sur l'exemple 2.4 page 29. La variable a est un variant de boucle. En effet :

- à l'initialisation, $a = n \in \mathbb{N}^\star$
- à chaque itération, a prend la valeur $a' = \dfrac{a}{2}$ (condition d'entrée en boucle : 2 divise a), on a donc bien $a' \in \mathbb{N}^\star$ et $a' < a$.

On en déduit la terminaison de la fonction `puissance_2`.

Exemple 2.6. La fonction Ocaml récursive `mem` définie ci-dessous teste si un élément est présent dans une liste ou non.

```
# let rec mem elem = function
        | [] -> false
        | e::q when e = elem -> true
        | e::q -> mem elem q
;;
val mem : 'a -> 'a list -> bool = <fun>
# mem 4 [5; -2; 12; 6];;
- : bool = false
# mem "violon" ["piano"; "batterie"; "violon"; "trombone"];;
- : bool = true
```

La taille de la liste passée en argument est un variant puisque chaque nouvel appel récursif se fait sur la queue de la liste, qui a une donc une taille diminuée de 1. Cela justifie la terminaison de la fonction `mem`.

Remarque 2.1. Il n'est pas toujours évident de prouver la terminaison d'un algorithme. Par exemple, selon la conjecture de Syracuse, la fonction Ocaml définie ci-dessous termine pour toute entrée :

```
# let tps_de_vol n = let m = ref n and tps = ref 0 in
        while !m <> 1 do
                if !m mod 2 = 0 then
                        m := !m / 2           (* si !m est pair *)
                else
                        m := 3 * !m + 1;      (* si !m est impair *)
                tps := !tps + 1
        done;
!tps;;
val tps_de_vol : int -> int = <fun>
```

Mais cette conjecture n'a pour l'instant jamais été prouvée !

Remarque 2.2. On dit que la correction est ***partielle*** lorsque le résultat est correct, à la condition que l'algorithme s'arrête. On dit qu'elle est ***totale*** si de plus l'algorithme termine.

2.4 Tests

Nous avons vu que pour prouver la correction d'un algorithme on avait recours à un raisonnement par récurrence (éventuellement forte) ou à un invariant. Cependant il n'est pas toujours possible d'effectuer ce type de raisonnement et la recherche d'un invariant peu parfois s'avérer fastidieuse. Aussi on commencera généralement par effectuer un jeu de tests afin de vérifier la correction d'un programme. Il n'est pas attendu en MP2I de savoir générer automatiquement des jeux de tests, mais de savoir écrire un jeu de test pertinent :

- Lorsque le domaine d'entrée peut être partitionné, tester des entrées pour chacun des ensembles
- Tester aux valeurs limites

Exemple 2.7. Si un programme prend en entrée une valeur de type `'a list`, on pourra le tester sur des entrées de type `int list`, `string list`, `float list` ou encore `bool list`.
On pensera également à tester la liste vide `[]`.

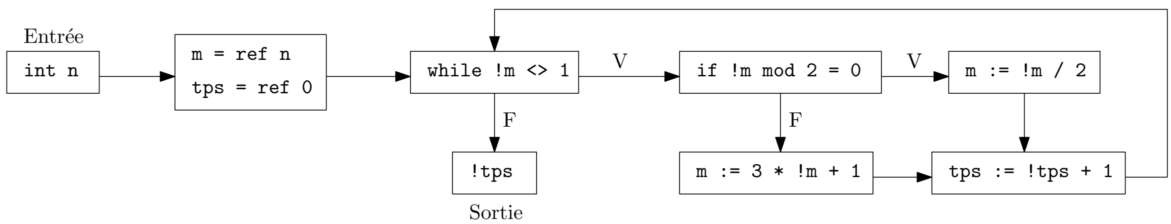
En complément, lorsqu'un programme contient des instructions conditionnelles on réalisera un test structurel en utilisant un graphe de flot de contrôle :

Définition 2.3. Le ***graphe de flot de contrôle*** d'un programme est un graphe orienté comportant :

- un sommet d'entrée - un sommet de sortie
- un sommet pour chaque bloc d'instructions élémentaires
- un sommet pour chaque condition et pour chaque boucle
- des arcs reliant les différents sommets - les arcs sortants d'une condition sont étiquetés par la valeur de la condition (V/F)

Une exécution, sur une entrée donnée, correspond alors à un ***chemin*** dans le graphe de flot de contrôle.

Exemple 2.8. Voici le graphe de flot de contrôle de la fonction `tps_de_vol` de la remarque 2.1 page 30.



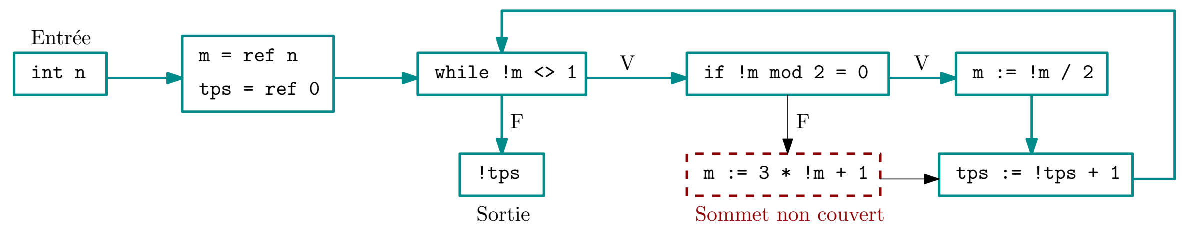
Définition 2.4. Dans un graphe de flot de contrôle, on dit qu'un chemin est ***faisable*** s'il existe une entrée pour laquelle l'exécution du programme correspond à ce chemin.

Remarque 2.3. L'existence de chemins infaisables est un problème ***indécidable*** (voir cours de spé).
Une fois déterminé le graphe de flot de controle, on peut réaliser un jeu de tests selon l'un ou l'autre des critères suivants (du plus faible au plus fort) :

- Couverture de toutes les instructions (i.e. de tous les sommets)
- Couverture de toutes les branches sur les chemins faisables (tous les arcs)
- Couverture de tous les chemins faisables

Remarque 2.4. Il peut y avoir une infinité de chemins faisables lorsque le graphe de flot de contrôle comporte des cycles. On pourra alors se contenter d'un jeu de tests satisfaisant un critère de couverture des chemins d'une longueur fixée ou bien choisir l'un des deux autres critères de converture (plus faibles).

Exemple 2.9. Dans le graphe de flot de contrôle de l'exemple précédent il y a une infinité de chemins faisables (par exemple, les puissances de 2 passées en entrées correspondent à des chemins faisables distincts). De plus, le seul test sur l'entrée 8 est insuffisant car ce test ne satisfait pas le critère de couverture des instructions puisque le chemin correspondant est le suivant :

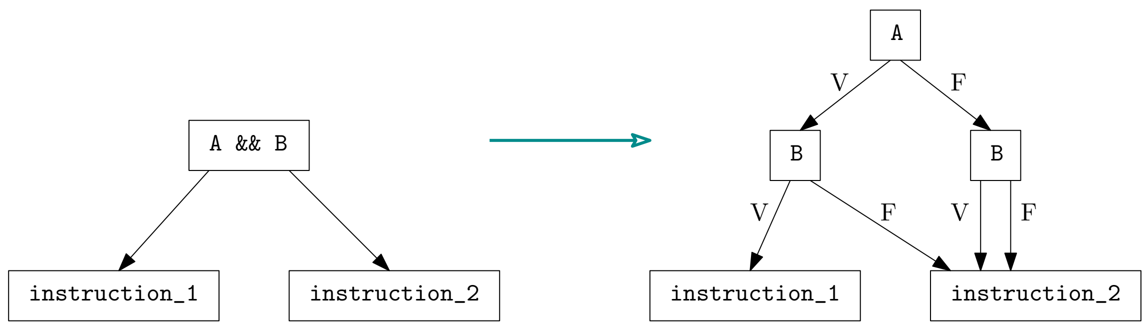


Enfin, lorsqu'une condition comporte des opérateurs booléens, on ne se contente pas de traiter l'expression comment globalement vraie ou fausse, on teste chaque cas possible.

Exemple 2.10. On donne la table de vérité de l'opérateur && :

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Dans le graphe de flot de contrôle on décomposera donc comme ceci :



2.5 Complexité

Maintenant qu'on sait justifier qu'un algorithme répond bien à un problème donné, il reste à évaluer ses performances. En effet, plusieurs algorithmes peuvent répondre à un même problème mais nécessiter un temps de calcul et un espace mémoire bien différents.

Définition 2.5. Pour un algorithme donnée :
- sa ***complexité temporelle*** indique le temps de calcul nécessaire à son exécution
- sa ***complexité spatiale*** indique l'espace mémoire utilisé lors de son exécution

2.5.1 Complexité temporelle

Définition 2.6. La ***complexité temporelle*** d'un algortihme correspond au nombre d'opérations élémentaires réalisées à son exécution sur une entrée, exprimée en fonction de la taille de l'entrée.

Exemple 2.11. Pour un tableau ou une liste en entrée, on exprimera la complexité temporelle en fonction de la longueur du tableau ou de la liste.

Remarque 2.5. Parmi les différentes opérations élémentaires possibles on trouve par exemple :

- les additions, multiplications, etc.
- les comparaisons sur des types simples
- les affectations

On suppose que toutes ces opérations se font en temps constant.

Exemple 2.12. La recherche d'un maximum dans une liste non triée nécessite d'effectuer $n-1$ comparaisons pour une liste de taille n

Définition 2.7. La complexité algorithmique peut être évaluée de plusieurs façons :

- dans le pire cas : c'est la complexité maximale qu'on peut atteindre
- dans le meilleur cas : c'est la complexité minimale attendue
- en moyenne : il s'agit de la moyenne des complexités obtenues pour toutes les entrées possibles d'une taille donnée

Remarque 2.6. Dans la majorité des cas, c'est la complexité dans le pire des cas qu'on étudie. Si rien n'est précisé, c'est qu'on demande une complexité dans le pire des cas.

2.5.2 Classes de complexité

Le tableau suivant regroupe les différentes complexité rencontrées et leur dénomination, classée dans l'ordre croissant de complexité :

Complexité	Dite	Exemple
1	constante	opérations élémentaires
$\log n$	logarithmique	dichotomie
n	linéaire	recherche séquentielle, boucle
$n \log n$	linéarithmique	tri fusion, diviser pour régner
n^2	quadratique	tri insertion, boucles imbriquées
$a^n \quad (a > 1)$	exponentielle	appels récursifs multiples

En pratique, on cherche simplement à connaître un ordre de grandeur de la ***complexité asymptotiqe***

2.5.3 Complexité spatiale

Index

~ ***Fin*** ~