Info : Bases de la programmation

Table des matières

1 Intro

Principaux paradigmes :

- Impératif (C, Python) ;
- Déclaratif :
  - Fonctionnel (OCaml) ;
  - Logique ;
- Orienté objet.

2 Programmation fonctionnelle

2.1 Bases du langague OCaml

2.1.1 Déclarations

Syntaxe :

```
let <name> = <expression>
let a = 5
let n = 5 in n*n (*-> 25*)

let a = 3 and b = 2 in a + b

```

2.1.2 Expression

Types de bases :

- `unit` : associé à une unique valeur, `void`, notée `()` ;
- `int` : les entiers (bornés, signés). On peut utiliser `min_int` et `max_int`. Fonctions : `+, -, *, /, mod` (`/` donne le quotient) ;
- `float` : les flotants. Fonctions : `+., *, -., /., **, sqrt, log, exp, sin, asin`, ... On peut utiliser `1.789e3` ;
- `char` : les caractères, notés entre `'` ;
- `string` : chaines de caractères, notés entre `"`. Fonctions utiles : `"bon" ^ "jour"` (concaténation), `String.length` ;
- Les couples : `t1 * t2` est un couple avec `t1` et `t2` des types. Fonctions : `fst`, `snd` ;
- `bool` : `true, false`. Fonctions : `not`, `&&`, `||` (évaluation paresseuse). Comparaisons : `=, <>, <, >, <=, >=`

2.1.3 Filtrage par motif

Exemples :

```
match x with
    | y when y = < 0 -> -y
    | _ -> x

match (a, b) with
    | (x, 0) | (0, x) -> 0
    | (x, y) -> x + y

match (a, b) with
    | (x, 0) | (0, x) -> begin match x with
                                  | 0 -> -1
                                  | _ -> 0
                         end
    | (x, y) -> x + y

```

2.1.4 Les fonctions

Exemples :

```
let f x = x*x
let g (x:int) (y:int) : int = x + y (*version curryfiee*)
let g2 ((x, y):int*int) : int = x + y (*version decuryfiee*)
f 2
g 5 7

let h = g 1

f2 = function x -> x + y
f3 = fun x y -> x + y

let test0 : int -> bool = function n -> n = 0

```

2.2 Récursivité

2.2.1 Fonctions récursives en OCaml

Exemple :

```
let rec fact (n:int):int = n * fact(n - 1)
(*fact 10
will return :
Stack overflow during evaluation (looping recursion?).*)

let rec fact (n:int):int =
    match n with
        | 0 | 1 -> 1
        | _ -> n * fact(n - 1)

```

2.2.2 Les listes

Les listes en OCaml sont :

- homogènes ;
- On ne peut accéder qu'à la tête directement ;
- On ne peut pas modifier une liste, seulement en créer de nouvelles.

$\boxed{a_1} \longrightarrow \boxed{a_2} \longrightarrow \ldots \longrightarrow \boxed{a_n} \longrightarrow \texttt{nil}$

Exemples :

```
let l = x1::x2::xn::[]
(*l est de type 'a list*)

let l2 = [0 ; 1 ; 2 ; 3]

List.hd l2 (*return 0*)
List.tl l2 (*return [1 ; 2 ; 3]*)

let rec len (l:'a list):int =
    match l with
        | [] -> 0
        | _::q -> 1 + len q

let l3 = [1 ; 2] @ [3 ; 4] (*return l3 = [1 ; 2 ; 3 ; 4]*)

```

Gammes :

```
let rec mem (x:'a) (l:'a list):bool =
    (*Test d'appartenance*)
    match l with
        | [] -> false
        | t::q -> x = t || mem x q

let rec concat (l1:'a list) (l2:'a list) : 'a list =
    (*Concatenation*)
    match l1 with
        | [] -> l2
        | t::q -> t::(concat q l2)

let rec exist (f:'a -> bool) (l:'a list) : bool =
    (*Teste si il existe un el x tq f x = true*)
    match l with
        | [] -> false
        | t::q -> f t || exist f q
```

2.2.3 Cas particuliers de récursivité

Récursivité croisée :

```
let rec f_1 args_1 = code_1
and f_2 args_2 = code_2
...
and f_n args_n = code_n

```

Récursivité terminale

Une fonction est récursive terminale si son résultat est le résultat d'un appel récursif sans calcul supplémentaire. Cela permet des optimisations dans la gestion des appels de fonction.

Ex : calc $S = \sum_{k=0}^{n} k$:

```
let sum (n:int):int =
    let rec aux (n:int) (acc:int) : int =
        (*Calc de acc + S*)
        match n with
            | 0 -> acc
            | _ -> aux (n - 1) (acc + n)
    in aux n 0

```

Ici, l'accumulation est calculée avant l'appel récursif (passage par valeur) : on économise l'espace mémoire.

3 Efficacité et sûreté des algo

3.1 Complexité

3.1.1 Intro

On distingue complexité temporelle (mesure du temps de calcul) et complexité spatiale (mesure espace mémoire utilisé). Notions floues. On s'intéresse en général à l'ordre de grandeur des quantités (spatiale) et au nombre d'opérations (complexité temporelle).

3.1.2 Notations de Bachmann-Landau

Soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ des suites d'entiers positifs.

u est dominée par v se note $u_n = \mathcal{O}(v_n)$.
$u_n = \mathcal{O}(v_n) \Leftrightarrow \exists (c, N) \in \mathbb{R}_+^* \times \mathbb{N} \mid \forall n \geq N,\ u_n \leq v_n$

u domine v se note $u_n = \Omega(v_n)$.
$u_n = \Omega(v_n) \Leftrightarrow \exists (c, N) \in \mathbb{R}_+^* \times \mathbb{N} \mid \forall n \geq N,\ u_n \geq v_n$

u est de l'ordre de v se note $u_n = \Theta(v_n)$.
$u_n = \Theta(v_n) \Leftrightarrow u_n = \mathcal{O}(v_n) \wedge u_n = \Omega(v_n)$

Propriétés :
$\forall \alpha, \beta > 0,\ (\log n)^\alpha = \mathcal{O}(n^\beta)$
$\forall \alpha \in \mathbb{R}, \forall \beta > 0,\ n^\alpha = \mathcal{O}(\beta^n)$
$$\forall \alpha \geq \beta > 0,\ \begin{cases} n^\beta = \mathcal{O}(n^\alpha) \\ (\log n)^\beta = \mathcal{O}((\log n)^\alpha) \\ \beta^n = \mathcal{O}(\alpha^n) \end{cases}$$

$u_n = \mathcal{O}(w_n) \ \wedge \ v_n = \mathcal{O}(w_n) \Rightarrow u_n + v_n = \mathcal{O}(w_n)$
$u_n = \mathcal{O}(v_n) \Rightarrow u_n \cdot w_n = \mathcal{O}(v_n \cdot w_n)$

Ex : $3u_n = \mathcal{O}(u_n)$
$n \cdot \mathcal{O}(1) = \mathcal{O}(n)$ (abus de notation).

3.1.3 Complexité dans le pire cas

Voc : Si la complexité est en :
$\mathcal{O}(1)$: temps constant ;
$\mathcal{O}(n)$: complexité linéaire (ex : factorielle, chercher un élément dans une liste) ;
$\mathcal{O}(n^2)$: complexité quadratique (ex : supression des doublons d'une liste) ;
$\mathcal{O}(\log n)$: complexité logarithmique (ex : recherche dichotomique) ;
$\mathcal{O}(n \log n)$: complexité quasi-linéaire ;
$\mathcal{O}(a^n)$: complexité exponentielle.
On utilise le même voc pour Θ.

3.1.4 Relation de récurence

Relation de la forme $C(n) = C(n-1) + an$
On résout directement :

$$C(n) = C(n-2)+a(n-1)+an = ... = C(1)+a\sum_{k=2}^{n} k = C(1)+\frac{(n-1)(n+1)}{2} = \Theta(n^2)$$

Ex : tri par insertion :

```
let rec insert (x:'a) (l:'a list) : 'a list =
    (*Liste triee par ordre croissant*)
    match l with
    | [] -> [x]
    | t::q when t < x -> t::insert x q
    | _ -> x::l

let rec i_sort (l:'a list) : 'a list =
    match l with
    | [] -> []
    | t::q -> insert t (i_sort q)

```

Tri fusion :

```
let rec partition (l:'a list) : 'a list * 'a list =
    match l with
    | [] -> [], []
    | [a] -> [a], []
    | a::b::q -> let (l1, l2) = partition q in (a::l1, b::l2)


```

```
let rec fusion (l1:'a list) (l2:'a list) : 'a list =
    (*Fusionne 2 listes triees pour return une liste triee par
   ordre croissant.*)
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | t1::q1, t2::q2 when t1 <= t2 -> t1::fusion q1 l2
    | t1::q1, t2::q2 -> t2::fusion l1 q2

let rec tri_fusion (l:'a list) : 'a list =
    match l with
    | [] | [_] -> l
    | _ -> let (l1, l2) = partition l in fusion (tri fusion l1) (
   tri_fusion l2)

```

3.2 Spécifications et tests

3.2.1 Rappels

– Lors de la déclaration d'une fonction, il est possible de préciser la signature, *i.e* le type de ses param et de son résultat.

– Lorsque le code d'une fonction s'appuie sur des hypothèses concernant ses param (qui ne st pas garanties par la signature), il est recommandé de les préciser en comment.

Ex :

```
let rec sum (n:int):int =
    (*On suppose n positif*)
    match n with
    | 0 -> 0
    | _ -> n + sum(n - 1)

```

3.2.2 Spécification d'une fonction

Fournir la spécification d'une fonction, c'est préciser quelles st les données attendues en param (type et hypothèses), et expliquer ce qu'elle calcule (type et nature du résultat vis-à-vis des param d'entrée).

Rq : Il n'y a pas vraiment de standard pr l'écriture d'une spécification, le plus important est d'être précis.

Ex :

```
(*Fonction insert :
    - Parametres :
        + element x de type 'a
        + liste l triee de type 'a list

    - Resultat :
        Liste triee, contenant x et tous les elements de l
*)
let rec insert (x:'a) (l:'a list):'a list = ...

```

3.2.3 Motivation des tests

– Le typage en OCaml permet de vérifier statiquement la cohérence de l'écriture des expressions.

– Les tests sont une manière dynamique de vérifier que l'éxécution d'un programme passe bien.

– Faire des tests permet de détecter les erreurs du programmeur, mais aussi de l'utilisateur.

Exemple d'erreur de l'utilisateur : l'utilisateur ne fournit pas un param qui vérifie la spécification (contrainte de type ou de valeur).

Il y a des fonctions de lecture permettant d'interrompre le programme pr ask des valeurs à l'utilisateur :

– `read_int: unit -> int`

– `read_float: unit -> float`

– `read_line: unit -> string`

Usage : `read_int()`

Erreur : on a éxécuté `sum(read_int())` et l'utilisateur entre :

−2 : `Stackoverflow` ...

2.3 : `Failure of "int_to_string"`

Exemple d'erreur par le programmeur : faute de frappe (+1 au lieu de +2) ou alors une icompréhension sur le résultat d'une fonction.

Ex : Que calcule `x mod 3` ? Le reste de la division euclidienne de x par 3 si $x \geq 0$. Si $x < 0$, le résultat est négatif : le reste moins 3.

3.2.4 Détecter une erreur

Pour détecter une erreur, il faut pouvoir :

(1) : exprimer des contraintes : exprimer des booléens ;

(2) : vérifier que les contraintes sont satisfaites, i.e qu'elles s'évaluent en `true`.

Possibilité : expression conditionnelles – Syntaxe : `if <exp> then <exp> else <exp>`

La première exp doit être de type bool. L'expression conditionnelle s'évalue en la valeur de l'expression après `then` si ce booléen s'évalue en `true`, et en celle de la dernière expression sinon -> les 2 expression (branches `then` et `else` doivent être de mm type)

Si la branche `then` est de type unit, la branche `else` est facultative.

`if b then e` ⇔ `if b then e else ()`

3.2.5 Signaler une erreur

Por signaler une erreur, on peut interroger l'éxécution du programme qui ?détecte ? l'erreur ou bien la poursuivre en renvoyant une valeur par défaut rpz l'absurde résultat.

Ex : `List.hd [] -> Exception Failure “hd”`

On peut réécrire hd :

```
let hd (l:'a) (default:'a) =
    match l with
    | [] -> default
    | t::_ -> t
```

Pb : change le type de la fonction. Il n'y a pas tj une valeur par défaut à fournir : ex inverse nb et div par 0.

Type option : on peut signaler l'absence de résultat grâce au type `'a option` (polymorphe) qui a 2 contructeurs :

- `None`, qui rpz l'absence de valeur
- `Some : 'a -> 'a option` (Some x rpz la valeur x)

Ex :

```
let rec quotient (a:int) (b:int) : int option =
    (*On suppose a>=0 et b >= 0*)

    if b = 0 then None
    else if a < b then Some 0
    else match quotient (a - b) b with
        | None -> None
        | Some q -> Some (1 + q);;
```

Exception : on a vu le msg `Exception Failure ...`

C'est le résultat de l'application de la fonction `failwith`, qui prend une chaîne de caractères en entrée et qui interrompt le programme.

Ex :

```
let rec quotient (a:int) (b:int) : int =
    if b = 0 then failwith "Division par 0"
    else if a < b then 0
    else 1 + quotient (a - b) b
```

Rq : le résultat de la fonction `failwith` est de type `int`.

En fait, `failwith` est de type `string -> 'a`

`Failure` est cas particulier d'exception.

• On déclare une exception grâce à la syntaxe `exception <Nom>` (le nom commence par une maj)

• Une exception est de type `exn`

• On soulève une exception grâce à `raise <Nom>`, ce qui interrompt le programme et affiche `Exception: <Nom>`.

`raise` est de type `exn -> 'a`

`failwith s` est équivlent à `raise(Failure s)`

Rq : les expressions conditionnelles, les types option et les exceptions ne sont pas spécifiques à la gestion d'erreur.

En particulier, on peut rattraper une exception pour continuer l'éxécution autrement.

Syntaxe :

```
try <exp> with
| <nom> -> <exp>
| <nom> -> <exp>
...
```

Les exceptions non stoppées remontent jusqu'à l'utilisateur.

3.2.6 Un cas particulier : les assertions

Une assertion s'écrit `assert <exp>` où l'expression est de type `bool`.

Si l'expression s'évalue à `false`, l'exception `Assert_failure` est levée. Sinon, le résultat est `()`, donc de type `unit`.

Usage : on évalue l'assertion *puis* on éxécute le reste du code, ie. on éxécute des instructions l'une après l'autre.

On utilise une nouvelle construction syntaxique. La séquence

Syntaxe : `<exp_1> ; <exp_2> ; ... ; <exp_n>`

`<exp_1> ; <exp_2> ; ... ; <exp_(n-1)>` sont de type `unit`.

Le type de la séquence est le type de l'expression `<exp_n>`

Ex :

```
let rec sum (n:int):int =
    assert (n >= 0);
    if n = 0 then 0 else n + sum (n - 1);;
```

3.2.7 Corriger des erreurs

Éxécuter des tests permet de détecter une erreur, mais aussi la partie du code qui est fautive.

L'outil le plus simple pour déterminer d'où vient la faute est de savoir comment la corriger est l'usage d'affichages localisés en combinaison d'un jeu de test (concernant l'ensemble des possibilités pour l'éxécution d'un programme).

Fonction d'affichage :

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

→ ce sont des instructions à utiliser au sein de séquences

→ On aborde la programmation impérative.

Rq : les instructions en OCaml sont des expressions de type `unit` et affectent l'état de la machine. On dit qu'elles produisent des effets de bord et elles st par conséquent appelées expressions impures (par oppo à purement fonctionnelles)

4 Programmation impérative

4.1 Bases du language C

4.1.1 Expressions

Comme en OCaml, les expression st de type construit par combinaison de variables, de constantes, de fonctions et de différentes constructions syntaxiques pouvant renvoyer une valeur. Les expression sont également associées à des types.

Les types de base sont :

- `void` : un type vide qui joue un rôle similaire à `unit` en OCaml ;
- `int` : les entiers, constructibles via des constantes (0, 1, ...), des fonctions (+, ∗, +, / (quotient div eucl), % (mod)) ;
- `double` : les nombres flottants (même si `float` existe également). On peut utiliser des constantes (3.14159, 1.789e3), des fonctions (+, −, ∗, /) ;
- `char` : les caractères entre '' (ex : `'test'`) ;
- `bool` : les valeurs de vérité. Les constantes booléennes sont `true` et `false`. Les fonctions booléennes sont :
  - ∗ la négation : `! <exp>` ;
  - ∗ la disjonction : `<exp> || <exp>` ;
  - ∗ la conjonction : `<exp> && <exp>`
  - ∗ les comparaisons : `==, !=, <, >, <=, >=`

4.1.2 Instructions

Elles désignent des ordres à éxécuter, qui ont un effet sur l'état de la machine (la mémoire). On dit qu'elles produisent des effet de bord.

Exemples d'instructions :

- La déclaration : on déclare une variable en précisant son type , son nom et facultativement sa valeur :

  `<type> <nom> [=<exp>];`

  Ex :

```
1 int x = 42;
2 double z;
3
```

- Déclaration simultanée ; on peut déclarer plusieurs variables de *même type* en les séparant par des virgules. Ex :

```
1 int x, y;
2 int i = 0, j;
3
```

- Constantes : par défaut, on peut modifier la valeur des variables. Pour interdire la modification d'une variable, on utilise le mot-clé `const`. La valeur est obligatoire.

Ex : `const double x = 3.4;`

– Affectation : pour modifier la valeur d'une variable, on utilise l'instruction affectation.
Syntaxe : `<nom> = <exp>;`

– L'incrémentation : (hors programme) on peut ajouter 1 à la valeur d'une variable entière grâce à l'opérateur d'incrémentation : `<nom>++;`. Les propriétés sont H.P. Rq : décrémentation `<nom>--;` Avec nos restrctions, c'est équivalent d'écrire `i++` ou `i = i + 1` ou `i += 1;`

– L'assertion : on écrit `assert(<exp>);` ;

– Entrées / sorties : On peut afficher une chaîne de char grâce à la fonction `printf`. Ex :

```
printf("Bonjour\n");
```

On peut afficher la valeur d'une expression grâce à une chaîne de format contenant les codes :

- `%d` pour les entiers ;
- `%f` pour les flottants ;
- `%lf` pour les `double` (long float) ;
- `%c` pour les caractères ;
- `%s` pour les chaînes.

Ex :

```
printf("%d et %f", x + 1, 4.5);
```

Pour demander une valeur à l'utilisateur, on utilise la fonction `scanf` avec la chaîne de format adaptée. Ex :

```
scanf("%d", &x);
//%d : We want an int
//& : Operator to get the variable memory acces
//x : Variable in which we want to store the value (declared
    before)

```

4.1.3 Structures de contrôle

– Instruction conditionnelle :
`if (<exp>) <instr> [else <instr>]`

Ex :

```
if (b) x = x + 2;

if (a && !b)
    x = x + 2;
else
    y = 2 * x;

```

Pour éxécuter plusieurs instructions dans une branche d'une instruction conditionnelle, on délimite un bloc par des accolades. Ex :

```
if (b) {
    x = x + 2;
    y = 2 * y;
}
```

– Itérations : la programmation itérative est un cas particulier de la programmation impératve.

Principe : on répète le même bloc d'instruction un certain nombre de fois.

∗ Boucles conditionnelles : on répète les instructions tant qu'une condition est satisfaite. Syntaxe :

```
while (x % 7 != 0)
    x--;
```

∗ Boucles inconditionnelles / boucles for : on répète les instructions un nombre déterminé de fois. En C, le nombre d'itérations est conditionné par trois expression. Syntaxe :

```
for (<exp1> ; <exp2> ; <exp3>) <instr/bloc>
```

`<exp1>` est l'initialisation, éxécutée avant la boucle ;

`<exp2>` est une condition. Les itérations s'arrêtent lorsqu'elle n'est plus satisfaite ;

`<exp3>` est une instruction éxécutée entre chaque itération (souvent l'incrémentation d'un compteur).

Ex :

```
for (i=0 ; i < 10 ; i++) { //Rq : i doit etre declare avant, et
    vaudra 10 a la fin de l execution
    x = x * 2;
    y = y / 2 * i
}
```

On peut également déclarer un compteur de boucle qui sera supprimé à la fin de l'éxécution :

```
for (int i = 0 ; i < 10 ; i++) {
    ...
}
```

∗ Interruption des itérateurs : on peut interrompre une boucle avec l'instruction `break`

4.1.4 Fonctions

– Déclaration : Syntaxe : `<type_ret> <func_name> (<type_arg1> <arg1>, ..., <type_argn> <argn>) <bloc>`

Exemple :

```
double moyenne (double x, double y, double z) {
    return (x + y + z) / 3;
}
```

L'instruction `return [<exp>];` permet de sortir de la fonction en renvoyant la valeur de l'expression. i le type de la fonction est `void`, on utilise pas d'expression et le `return` est facultatif.

– Utilisation : pour applique une fonction, à des arguments, on utilise la syntaxe `<nom>(<exp1>, ..., <expn>)`.

Les arguments sont passés par valeur.

Si le type de retour est `void`, on peut voir l'appel de la fonction comme une instruction.

Ex : `f(x ,y);`

– Fonction `main` : c'est la fonction principale d'un programme C. Elle est obligatoire si on veut éxécuter du code.

Elle est sans paramètres (pour l'instant), et son type de retour est `int`. Par défaut, la valeur de retour est 0. On écrit donc :

```
int main() {
    <block>
    return 0;
}

```

– Variables locales : les variables déclarées ds un bloc sont limitées à ce bloc. Pour déclarer une var globale, on la déclare en dehors de toute fonction.

4.1.5 Tableaux

On se limite pour l'instant aux tableaux statiques i.e. dont la taille est fixée d'avance par une expression littérale.

– Déclaration : `<type> <name>[<size>];`

– Initialisation : On peut initialiser le contenu du tableau à l'aide d'une boucle, ou bien au moment de la déclaration à l'aide de la syntaxe `<type> <name>[<size>] = <exp1>, ..., <expn>`

Ex :

```
int tab[2];
int t[3] = {0, 1, 2};

```

– Accès et écriture : on accède à une case d'un tableau par un indice.

Syntaxe : `<name>[<index>];`

Pour écrire dans une case, on utilise l'instruction d'affectation. Ex : `t[0] = 2;`

```
for (int i=0 ; i < n ; i++){
    t[i] = 2*i
}

```

– Tableaux multidimentionnels : ce sont des tableaux de tableaux.

Ex :

```
int const a = 20, b = 10;
int t[a][b];

for (int i=0 ; i < a ; i++) {
    for (int j=0 ; j < b ; j++) {
        t[i][j] = i + j;
    }
}

```

– Chaînes de caractères : ce sont des tableaux de `char` qui contiennent obligatoirement le caractère spécial `'\0'`

Entrées : on écrit `scanf("%f", name);`

4.1.6 Transtypage

Les conversions de type sont implicites.

Ex : `1 + 2.5` est correct.

On peut écrire une conversion explicite grâce à la syntaxe `(<type>)<exp>`

Cas particulier pr les chaînes : fonctions `atoi` pr les entiers et `atof` pr les flottants.

4.1.7 Commentaires

```
//This is a comment.

/*This is a comment
on
multiple
lines*/

```

4.2 Programmation impérative en OCaml

4.2.1 Rappel : les instructions en OCaml

Les instruction en OCaml sont des expressions de type `unit`, dites *impures* car elles produisent des effet de bord.

Tous les objets vus jusqu'ici sont immuables, i.e. on ne peut pas changer leur valeur.

Il existe toutefois des objets mutables en OCaml.

4.2.2 Références

Les instructions affectent la mémoire de la machine, mais on ne souhaite pas manipuler directement les adresses mémoire. On préfère pour cela associer un identifiant à un

emplacement mémoire, d'où la déclaration d'une référence.

Syntaxe : `ref <exp>` est une référence à un emplacement mémoire qui contient la valeur de l'expression. Pour la manipuler, on lui donne un nom à l'aide d'une déclaration globale.

Ex : `let x = ref [1; 2; 3; 4]`

Le type des références est `'a ref` où `'a` désigne le type de la valeur stockée.

Il faut distinguer le contenant (la ref) du contenu (la valeur), qui l'on obtient à l'aide de l'opérateur de déréférencement : si `<nom>` est une ref, `!<nom>` est la valeur contenue dans la ref.

On peut modifier le contenu d'une ref grâce à une affectation

Syntaxe : `<nom> := <exp>`

Ne pas comfondre

	Affectation	Test d'égalité	Négation	Déréférencement
OCaml	`:=`	`=` (ou `==`)	`not`	`!`
C	`=`	`==`	`!`	plus tard

```
let x = ref 0 in
    x := 2;
    x := !x + 3
    !x

```

Comme en C, on dispose d'opérateurs d'incrémentation et de décrémentation (pour les `int ref`)

```
let x = ref 0 in
    incr x;
    !x

(*- int x = 1*)

let x = ref 2 in
    decr x;
    !x

(*- int x = 1*)

```

4.2.3 Boucles

Ce sont des instructions, donc des expressions de type `unit`.

Ex : `incr` et `decr` sont de type `int ref -> unit`.

– Boucles conditionnelles :

```
while <exp1> do <exp2> done
```

où `<exp1>` est de type `bool` (la condition) et `<exp2>` est de type `unit` (le corps de la boucle).

– Boucles inconditionnelles :

Beaucoup plus contraintes qu'en C, une boucle `for` en OCaml déclare un indice de boucle (de type `int`) qui varie par pas de 1 entre une valeur initiale et une valeur finale (incluses).

Syntaxe :

```
for <nom> = <exp1> to <exp2> do <exp3> done
```

où `<exp1>` et `<exp2>` sont de type `int` et `<exp3>` est de type `unit`.

`<nom>` est un identifiant *uniquement* dans `<exp3>`

"équivalent en C" :

```
for (int <nom>=<exp1> ; <nom> <= <exp2> ; <nom>++) {
    <exp3>
}
```

On peut écrire des boucles descendantes :

```
for <name>=<exp1> downto <exp2> do <exp3> done
```

4.2.4 Tableaux

Le type des tableaux dont les éléments sont de type `'a` est `'a array`.

Tableaux homogènes.

- Création de tableau :
  - Directement : `[|<exp1>; <exp2>; ...; <expn>|]`

    Tableau vide : `[||]`
  - À l'aide de la fonction `Array.make : int -> 'a -> 'a array`

    Ex : `Array.make 5 0.` est `[|0.; 0.; 0.; 0.; 0.|]`

    Pour les matrices, (les tableaux de tableaux), on utilise `Array.make_matrix : int -> int -> 'a -> 'a array array`

    Ex : `Array.make_matrix 2 3 0` vaut `[| [|0; 0; 0|]; [|0; 0; 0|] |]`
  - À l'aide de la fonction `Array.init : int -> (int -> 'a) -> 'a array`

    Ex : `Array.init n f` crée le tableau `[|f 0; f 1; ...; f (n - 1)|]`

- Lectures / écritures

  On accède aux éléments par indice (entre 0 et length -1)

  Syntaxe : `<nom>.(<exp>)`

  Où `<nom>` est le nom du tableau et `<exp>` représente l'indice (type `int`).

  Pour modifier une case d'un tableau, on écrit `<nom>.(<exp1>) <- <exp2>`

  Où `<exp1>` est l'indice de la case de `<exp2>` la valeur à stocker.

  Accès dans une matrice par accès succesifs : `<nom>.(<exp1>).(<exp2>)`.

- Longueur

On utiliser la fonction `Array.length : 'a array -> int`

Pr les matrices,

`Array.length m` est le nombre de lignes ;

`Array.length m.(0)` est le nombre de colones.

- Copie

  On copie un tableau existant grâce à `Array.copy : 'a array -> 'a array`

  Ex : `Array.copy [|0; 0; 0|]` renvoie `[|0; 0; 0|]` (nouveau tableau).

  Attention, la copie est superficielle.

  Ex :

```
let t = Array.make_matrix 2 3 0;;
let t' = Array.copy t;;
t'.(0).(1) <- 2;;
t;;
```

  Revoit `[| [|0; 2; 0|]; [|0; 0; 0|] |]`

  Pour créer une copie indépendante (deep copy), il faut appliquer `Array.copy` à chaque élément.

  Ex :

```
let t' = Array.copy t;;
for i=0 to Array.length t - 1 do
    t'.(i) <- Array.copy t.(i)
done;;
```

  Ou bien

```
let t' = Array.init (Array.length t) (function i -> Array.copy
   t.(i));
```

- Fonctions à savoir coder

  — `Array.mem : 'a -> 'a array -> bool` (test d'appartenance) ;

  — `Array.exists : ('a -> bool) -> 'a array -> bool` (existance d'un élément satisfaisant une propriété) ;

  — `Array.for_all : ('a -> bool) -> 'a array -> bool` (universalité d'une propriété sur les éléments du tableau) ;

  — `Array.map : ('a -> 'b) -> 'a array -> 'b array` (image point à point par une fonction)

  — `Array.iter : ('a -> unit) -> 'a array -> unit` (pour appliquer successivement une instruction à chaque élément du tableau (dans l'ordre))

  Ex : `Array.iter (function x -> print_int x ; print_newline()) t` affiche les éléments du tableau d'entiers `t` (un par ligne)

  Rq : la fonction `List.iter` existe.

4.2.5 Chaînes de caractères

Comme en C, on peut accéder aux caractères d'une chaîne par indice.

Syntaxe : `<nom>.[<exp>]`

Rq : il existe une fonction `String.make : int -> char -> string`

Attention, les chaînes de caractères ne sont pas des tableaux de caractères, i.e. les types `string` et `char array` sont distints.

En particulier, depuis OCaml 4.06, les chaînes de caractères sont immuables.

Autre différence essentielle entre OCaml et C : le typage.

On dit que le typage d'OCaml est *fort*, et que celui de C est *faible*.

Il n'y a pas de définition universellement reconnue des typages fort/faible.

Intuition : un typage fort apporte des garanties de sécurité vis à vis du programme en imposant des conditions plus strictes.

Ex : conversions de types implicites (impossible en OCaml, possible en C)

Autre ex : accès aux cases d'un tableau non vérifiées en C.