	OCaml	C	Python
test d'égalité	=	==	==
test de différence	<>	!=	!=
division euclidienne	/	/	//
modulo	mod	%	%
et	&&	&&	and
ou	||	||	or

Opérateurs en OCaml, C et Python

	OCaml
Définition	`let r = ref ...`
Accéder à la valeur	`!r`
Modifier la valeur	`r := ...`

Références

- `()` est une valeur de type `unit`, qui signifie rien.
- Si `f : 'a -> 'b -> 'c` alors `f` prend deux arguments de type `'a` et `'b` et renvoie un résultat de type `'c`.
  De plus, `f x` (application partielle) est la fonction de type `'b -> 'c` qui à `y` associe `f x y`.
- On créé souvent une liste avec une fonction récursive :

```
let rec range n = (* renvoie [n - 1; ...; 1; 0] *)
    if n = 0 then []
    else (n-1)::range (n - 1)
```

On peut éventuellement utiliser une référence sur une liste :

```
let range n =
    let l = ref [] in (* moins idiomatique *)
    for i = n - 1 downto 0 do
        l := i::!l
    done; !l
```

- **Impossible de modifier une liste** `l`.
  `e::l` renvoie une **nouvelle liste** mais ne modifie pas `l`.
  `let l = [] in ...` ne sert à rien.
- **Impossible d'écrire** `l.(i)` pour une liste `l` : il faut utiliser une fonction récursive pour parcourir une liste.

```
let rec appartient e l = match l with
    | [] -> false
    | t::q -> e = t || appartient e q
```

- Chaque cas d'un `match` sert à **définir de nouvelles variables**, et ne permet pas de comparer des valeurs.

Exemple : Dans `appartient`, il ne faut pas écrire `| e::q -> ...` (ceci remplacerait le `e` en argument).
Pour tester l'égalité : `| t::q -> if t = e then ...` ou `| t::q when t = e -> ...`.

- Les indices d'un tableau à n éléments vont de 0 à $n-1$:

```
let appartient e t =
    let r = ref false in
    for i = 0 to  n - 1 do
        if t.(i) = e then r := true
    done; !r
```

- **Impossible de renvoyer une valeur à l'intérieur d'une boucle** (pas de `return` en OCaml) :

```
let appartient e t =
    for i = 0 to n - 1 do
        if t.(i) = e then true (* NE MARCHE PAS !!! *)
    done; false
```

- Types union et enregistrement :

```
(* type union : l'un OU l'autre *)
type 'a option = None | Some of 'a
(* on utilise match (et fonction récursive)
   sur les types union *)
let add x y = match x, y with
    | None, _ | _, None -> None
    | Some x1, Some y1 -> Some (x1 + y1)

(* type enregistrement : l'un ET l'autre *)
type fraction = { num : int; denum : int }
let f = { num = 1; denum = 4 }
f.denum (* donne 4 *)
```

Remarque : l'égalité (avec =) est automatiquement définie avec un nouveau type. Exemple : si `t1` et `t2` sont des arbres binaires, alors `t1 = t2` vaut `true` si `t1 = V` et `t2 = V` ou si `t1 = N(r1, g1, d1)`, `t2 = N(r2, g2, d2)` avec `r1 = r2`, `g1 = g2` et `d1 = d2`.

- Quelques fonctions utiles (avec les équivalents sur `Array`) :
  - `Array.make_matrix n p e` renvoie une matrice $n \times p$ dont chaque élément est `e`
  - `List.iter f l` appelle `f` sur chaque élément de `l`
  - `List.map f [a1; ...; an]` renvoie `[f a1; ...; f an]`
  - `List.filter f l` renvoie la liste des `e` tels que `f e`
  - `List.exists f l` et `List.for_all f l`.

Type	Exemple	Obtenir valeur	Modification	Taille	Création
`array`	`[|1; 2|] : int array`	`t.(i)`	`t.(i) <- ...`	`Array.length` (en O(1))	`Array.make n e` (en O(n))
`string`	`"abc" : string`	`s.[i]`		`String.length` (en O(1))	`String.make n e`
`list`	`[1; 2] : int list`	Fonction récursive		`List.length` (en O(n))	
`tuple`	`(1, "abc", [1; 2]) : int*string*int list`	`let a, b, c = t`			

Remarque : une « liste » Python est en réalité un tableau.

- Si `x` est une variable, `&x` est son adresse (dans la mémoire RAM).

- Un pointeur `p` est une variable contenant une adresse. `*p` donne alors la valeur à l'adresse contenue dans `p`. Par exemple :

```
int x = 42;
int* p = &x; // p contient l'adresse de x
print("%d\n", *p); // affiche 42
*p = 0; // x vaut maintenant 0
```

`NULL` (qui vaut en fait 0) est une valeur spéciale pour un pointeur qui ne pointe sur rien.

- Par défaut, les variables sont stockées dans une zone mémoire appelée pile (stack) et sont automatiquement supprimées lorsqu'on sort de leur portée. Les arguments d'une fonction sont aussi stockés dans la pile.

- Il est également possible de stocker une variable dans le tas (heap). La taille de la zone mémoire à allouer n'a alors pas besoin d'être connue à la compilation.

```
int* p = (int*)malloc(sizeof(int));
// p est un pointeur vers un entier
*p = 42;  // p pointe vers un entier valant 42
print("%d\n", *p); // affiche 42
free(p) // libère la mémoire pointée par p
```

Pour renvoyer un pointeur créé dans une fonction, il faut utiliser `malloc` :

```
int* f() {
    int* p = (int*)malloc(sizeof(int));
    *p = 42;
    return p;
}
```

Sinon, le pointeur renvoyé pointera vers une zone mémoire qui n'existe plus :

```
int* f() {
    int x = 42;
    return &x; // x est détruit
}
int* y = f();
print("%d\n", *y); // erreur
```

- Une structure est un regroupement de variables (des attributs). Par exemple :

```
struct Complexe {
    double re;
    double im;
};
Complexe conjugue(Complexe c) {
    Complexe res;
    res.re = c.re;
    res.im = -c.im;
    return res;
}
Complexe z = {.re = 1, .im = 2};
Complexe w = conjugue(z);
```

- Passage d'argument par valeur (ou : copie) : l'argument est copié dans la fonction. La variable originale n'est pas modifiée.

```
void f(int x) {
    x = 42;
}

int y = 0;
f(y); // y n'est pas modifié
```

- Passage d'argument par adresse (ou référence) : l'adresse de l'argument est passée à la fonction. La variable originale peut être modifiée.

```
void f(int* x) {
*x = 42;
}

int y = 0;
f(&y); // y est modifié
```

En OCaml, un tableau est passé par adresse :

```
let f t =
    t.(0) <- 42

let y = [|0|] in
f y (* y est modifié *)
```

- Création d'un tableau statique en C :

```
int t[10]; //  tableau de 10 entiers
t[0] = 42;
print("%d", t[0]);
int t2[] = {1, 2, 3}; // autre façon
```

- Création d'un tableau dynamique en C (la taille peut alors dépendre d'une variable) :

```
int* t = (int*)malloc(10*sizeof(int));
t[0] = 42;
print("%d", t[0]);
```

- En C, les tableaux dynamiques ne connaissent pas leur taille. Il faut donc passer la taille en argument :

```c
int somme(int* t, int n) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        res += t[i];
    }
    return res;
}
```

- Une chaîne de caractères en C est un tableau de caractères terminé par le caractère nul `'\0'`. Par exemple :

```c
char s[] = {'h', 'e', 'l', 'l', 'o', '\0'};
print("%s", s); // affiche hello
char s1 = "hello"; // équivalent
```

- Comme une chaîne de caractères termine par `'\0'`, il est possible de connaître sa taille en parcourant les caractères jusqu'à rencontrer `'\0'` (`strlen` existe dans `string.h`) :

```c
int strlen(char* s) {
    int i = 0;
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

- Si on connaît ses dimensions à l'avance, on peut déclarer une matrice statique en C :

```c
int t2[2][3] = {
    {0, 1, 2},
    {3, 4, 5}
};
print("%d", t2[1][2]); // affiche 5
```

- Sinon, il faut la définir dynamiquement :

```c
int** id(int n) { // matrice identité
    int** t = (int**)malloc(n*sizeof(int*));
    for(int i = 0; i < n; i++) {
        t[i] = (int*)malloc(n*sizeof(int));
        for(int j = 0; j < n; j++) {
            if(i == j)
                t[i][j] = 1;
            else
                t[i][j] = 0;
        }
    }
    return t;
}
```

Voir mp2i-info.github.io pour plus de détails.

Extrait du programme officiel

A Langage C

La présente annexe liste limitativement les éléments du langage C (norme C99 ou plus récente) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants à la fin de la première année. Ces éléments s'inscrivent dans la perspective de lire et d'écrire des programmes en C ; aucun concept sous-jacent n'est exigible au titre de la présente annexe.
À l'écrit, on travaille toujours sous l'hypothèse que les entêtes suivants ont tous été inclus : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`. Mais ces fichiers ne font pas en soi l'objet d'une étude et aucune connaissance particulière des fonctionnalités qu'ils apportent n'est exigible.

A.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage C doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

— Typage statique. Types indiqués par le programme lors de la déclaration ou définition.
— Passage par valeur.
— Délimitation des portées par les accolades. Les retours à la ligne et l'indentation ne sont pas signifiants mais sont nécessaires pour la lisibilité du code.
— Déclaration et définition de fonctions, uniquement dans le cas d'un nombre fixé de paramètres.
— Gestion de la mémoire : pile et tas, allocation statique et dynamique, durée de vie des objets.

Définitions et types de base

— Types entiers signés `int8_t`, `int32_t` et `int64_t`, types entiers non signés `uint8_t`, `uint32_t` et `uint64_t`. Lorsque la spécification d'une taille précise pour le type n'apporte rien à l'exercice, on utilise les types signé **`int`** et non signé **`unsigned int`**. Opérations arithmétiques `+`, `-`, `/`, `*`. Opération `%` entre opérandes positifs. Ces opérations sont sujettes à dépassement de capacité. À l'écrit, on élude les difficultés liées à la sémantique des constantes syntaxiques. On ne présente pas les opérateurs d'incrémentation.
— Le type **`char`** sert exclusivement à représenter des caractères codés sur un octet. Notation `'\0'` pour le caractère nul.
— Type **`double`** (on considère qu'il est sur 64 bits). Opérations `+`, `-`, `*`, `/`.
— Type `bool` et les constantes `true` et `false`. Opérateurs `!`, `&&`, `||` (y compris évaluation paresseuse). Les entiers ne doivent pas être utilisés comme booléens, ni l'inverse.
— Opérateurs de comparaison `==`, `!=`, `<`, `>`, `<=`, `>=`.
— Les constantes du programme sont définies par **`const`** *type c* **`=`** *v*. On n'utilise pas la directive du préprocesseur **`#define`** à cette fin.

Types structurés

— Tableaux statiques : déclaration par *type* `T[`*s*`]` où *s* est une constante littérale entière. Lecture et écriture d'un terme de tableau par son indice `T[`*i*`]` ; le langage ne vérifie pas la licéité des accès. Tableaux statiques multidimensionnels.
— Définition d'un type structuré par **`struct`** *nom_s* **`{`***type*$_1$ *champ*$_1$**`;`** … *type*$_n$ *champ*$_n$**`;}`** et ensuite **`typedef struct`** *nom_s nom* (la syntaxe doit cependant être rappelée si les étudiants sont amenés à écrire de telles définitions). Lecture et écriture d'un champ d'une valeur de type structure par *v* . *champ* ainsi que *v*`->`*champ*. L'organisation en mémoire des structures n'est pas à connaître.
— Chaînes de caractères vues comme des tableaux de caractères avec sentinelle nulle. Fonctions `strlen`, `strcpy`, `strcat`.

Structures de contrôle

— Conditionnelle **`if (`***c***`)`** s_T, **`if (`***c***`)`** s_T **`else`** s_F.
— Boucle **`while (`***c***`)`** *s* ; boucle **`for (`***init***`;`** *fin*`;` *incr***`)`** *s*, possibilité de définir une variable dans *init* ; **`break`**.

— Définition et déclaration de fonction, passage des paramètres par valeur, y compris des pointeurs. Cas particuliers : passage de paramètre de type tableau, simulation de valeurs de retour multiples.

Pointeurs et gestion de la mémoire

— Pointeur vers un objet alloué, notation *type*`*` p `= &`v. On considère que les pointeurs sont sur 64 bits.
— Déréférencement d'un pointeur valide, notation `*`p. On ne fait pas d'arithmétique des pointeurs.
— Pointeurs comme moyen de réaliser une structure récursive. Pointeur `NULL`.
— Création d'un objet sur le tas avec `malloc` et `sizeof` (on peut présenter `size_t` pour cet usage mais sa connaissance n'est pas exigible). Libération avec `free`.
— Transtypage de données depuis et vers le type `void*` dans l'optique stricte de l'utilisation de fonctions comme `malloc`.
— En particulier : gestion de tableaux de taille non statiquement connue ; linéarisation de tels tableaux quand ils sont multidimensionnels.

Divers

— Utilisation de `assert` lors d'opérations sur les pointeurs, les tableaux, les chaînes.
— Flux standard.
— Utilisation élémentaire de `printf` et de `scanf`. La syntaxe des chaînes de format n'est pas exigible.
— Notion de fichier d'en-tête. Directive `#include "`*fichier*`.h"`.
— Commentaires `/* ... */` et commentaires ligne `//`

A.2 Éléments techniques devant être reconnus et utilisables après rappel

Les éléments suivants du langage C doivent pouvoir être utilisés par les étudiants pour écrire des programmes dès lors qu'ils ont fait l'objet d'un rappel et que la documentation correspondante est fournie.

Traits généraux et divers

— Utilisation de `#define`, `#ifndef` et `#endif` lors de l'écriture d'un fichier d'en-tête pour rendre son inclusion idempotente.
— Rôle des arguments de la fonction `int main(int argc, char* argv[])` ; utilisation des arguments à partir de la ligne de commande.
— Fonctions de conversion de chaînes de caractères vers un type de base comme `atoi`.
— Définition d'un tableau par un initialisateur `{`t_0`,` t_1`, ...,` t_{N-1}`}`.
— Définition d'une valeur de type structure par un initialisateur `{.`c_1 `=` v_1`, .`c_2 `=` v_2`, ...}`.
— Compilation séparée.

Gestions des ressources de la machine

— Gestion de fichiers : `fopen` (dans les modes `r` ou `w`), `fclose`, `fscanf`, `fprintf` avec rappel de la syntaxe de formatage.
— Fils d'exécution : inclusion de l'entête `pthread.h`, type `pthread_t`, commandes `pthread_create` avec attributs par défaut, `pthread_join` sans récupération des valeurs de retour.
— Mutex : inclusion de l'entête `pthread.h`, type `pthread_mutex_t`, commandes `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`.
— Sémaphore : inclusion de l'entête `semaphore.h`, type `sem_t`, commandes `sem_init`, `sem_destroy`, `sem_wait`, `sem_post`.

B Langage OCaml

La présente annexe liste limitativement les éléments du langage OCaml (version 4 ou supérieure) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

B.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage OCaml doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

— Typage statique, inférence des types par le compilateur. Idée naïve du polymorphisme.
— Passage par valeur.
— Portée lexicale : lorsqu'une définition utilise une variable globale, c'est la valeur de cette variable au moment de la définition qui est prise en compte.
— Curryfication des fonctions. Fonction d'ordre supérieur.
— Gestion automatique de la mémoire.
— Les retours à la ligne et l'indentation ne sont pas signifiants mais sont nécessaires pour la lisibilité du code.

Définitions et types de base

— `let`, `let rec` (pour des fonctions), `let rec` … `and` …, `fun` x y `->` e.
— `let` v `=` e `in` e', `let rec` f x `=` e `in` e'.
— Expression conditionnelle `if` e `then` e_V `else` e_F.
— Types de base : `int` et les opérateurs `+`, `-`, `*`, `/`, l'opérateur `mod` quand toutes les grandeurs sont positives ; exception `Division_by_zero` ; `float` et les opérateurs `+.`, `-.`, `*.`, `/.` ; `bool`, les constantes `true` et `false` et les opérateurs `not`, `&&`, `||` (y compris évaluation paresseuse). Entiers et flottants sont sujets aux dépassements de capacité.
— Comparaisons sur les types de base : `=`, `<>`, `<`, `>`, `<=`, `>=`.
— Types `char` et `string` ; `'`x`'` quand x est un caractère imprimable, `"`x`"` quand x est constituée de caractères imprimables, `String.length`, s`.[`i`]`, opérateur `^`. Existence d'une relation d'ordre total sur `char`. Immuabilité des chaînes.

Types structurés

— n-uplets ; non-nécessité d'un `match` pour récupérer les valeurs d'un n-uplet.
— Listes : type `'a list`, constructeurs `[]` et `::`, notation `[`x`;` y`;` z`]` ; opérateur `@` (y compris sa complexité) ; `List.length`. Motifs de filtrage associés.
— Tableaux : type `'a array`, notations `[|`…`|]`, t`.(`i`)`, t`.(`i`)` `<-` v ; fonctions `length`, `make`, et `copy` (y compris le caractère superficiel de cette copie) du module `Array`.
— Type `'a option`.
— Déclaration de type, y compris polymorphe.
— Types énumérés (ou sommes, ou unions), récursifs ou non ; les constructeurs commencent par une majuscule, contrairement aux identifiants. Motifs de filtrage associés.
— Filtrage : `match` e `with` p_0 `->` v_0 `|` p_1 `->` v_1 … ; les motifs ne doivent pas comporter de variable utilisée antérieurement ni deux fois la même variable ; motifs plus ou moins généraux, notation `_`, importance de l'ordre des motifs quand ils ont des instances communes.

Programmation impérative

— Absence d'instruction ; la programmation impérative est mise en œuvre par des expressions impures ; `unit`, `()`.
— Références : type `'a ref`, notations `ref`, `!`, `:=`. Les références doivent être utilisées à bon escient.
— Séquence `;`. La séquence intervient entre deux expressions.
— Boucle `while` c `do` b `done` ; boucle `for` v `=` d `to` f `do` b `done`.

Divers

- Usage de `begin` … `end`.
- `print_int`, `print_float`, `print_string`, `read_int`, `read_float`, `read_line`.
- Exceptions : levée et filtrage d'exceptions existantes avec `raise`, `try` … `with` … ; dans les cas irrattrapables, on peut utiliser `failwith`.
- Utilisation d'un module : notation $M.f$. Les noms des modules commencent par une majuscule.
- Syntaxe des commentaires, à l'exclusion de la nécessité d'équilibrer les délimiteurs dans un commentaire.

B.2 Éléments techniques devant être reconnus et utilisables après rappel

Les éléments suivants du langage OCaml doivent pouvoir être utilisés par les étudiants pour écrire des programmes dès lors qu'ils ont fait l'objet d'un rappel et que la documentation correspondante est fournie.

Traits divers

- Types de base : opérateur `mod` avec opérandes de signes quelconques, opérateur `**`.
- Types enregistrements mutables ou non, notation {c_0 : t_0; c_1 : t_1; …}, {c_0 : t_0; `mutable` c_1 : t_1; …}; leurs valeurs, notations {c_0 = v_0; c_1 = v_1; …}, $e.c$, $e.c$ <- v.
- Fonctions de conversion entre types de base.
- Listes : fonctions `mem`, `exists`, `for_all`, `filter`, `map`, `iter` du module `List`.
- Tableaux : fonctions `make_matrix`, `init`, `mem`, `exists`, `for_all`, `map` et `iter` du module `Array`.
- Types mutuellement récursifs.
- Filtrage : plusieurs motifs peuvent être rassemblés s'ils comportent exactement les mêmes variables. Notation `function` p_0 -> v_0 | p_1 -> v_1 ….
- Boucle `for` v = f `downto` d `do` b `done`.
- Piles et files mutables : fonctions `create`, `is_empty`, `push` et `pop` des modules `Queue` et `Stack` ainsi que l'exception `Empty`.
- Dictionnaires mutables réalisés par tables de hachage sans liaison multiple ni randomisation par le module `Hashtbl` : fonctions `create`, `add`, `remove`, `mem`, `find` (y compris levée de `Not_found`), `find_opt`, `iter`.
- `Sys.argv`.
- Utilisation de `ocamlc` ou `ocamlopt` pour compiler un fichier dépendant uniquement de la bibliothèque standard.

Gestions des ressources de la machine

- Gestion de fichiers : fonctions `open_in`, `open_out`, `close_in`, `close_out`, `input_line`, `output_string`.
- Fils d'exécution : recours au module `Thread`, fonctions `Thread.create`, `Thread.join`.
- Mutex : recours au module `Mutex`, fonctions `Mutex.create`, `Mutex.lock`, `Mutex.unlock`.