

Chapitre 6

BONNES PRATIQUES DE PROGRAMMATION

Notions	Commentaires
Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante.	Ces annotations se font à l'aide de commentaires.
Programmation défensive. Assertion. Sortie du programme ou exception levée en cas d'évaluation négative d'une assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées ou pour le contrôle de débordements. Plus généralement, les étudiants sont sensibilisés à réfléchir aux causes possibles (internes ou externes à leur programme) d'opérer sur des données invalides et à adopter un style de programmation défensif. [...]
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

Notions	Commentaires
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant est capable d'écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.
[...] Chemins faisables. Couverture des sommets, des arcs ou des chemins (avec ou sans cycle) du graphe de flot de contrôle.	Les étudiants sont capables d'écrire un jeu de tests satisfaisant un critère de couverture des instructions (sommets) ou des branches (arcs) sur les chemins faisables.
Test exhaustif de la condition d'une boucle ou d'une conditionnelle.	Il s'agit, lorsque la condition booléenne comporte des conjonctions ou disjonctions, de ne pas se contenter de la traiter comme étant globalement vraie ou fausse mais de formuler des tests qui réalisent toutes les possibilités de la satisfaire. On se limite à des exemples simples pour lesquels les cas possibles se décèlent dès la lecture du programme.

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

Ce cours sert à deux choses :

- Donner des compléments sur les langages de programmation au programme ; compléments qui peuvent aider à écrire du code plus simple à (re)lire.
- Présenter des bonnes pratiques de programmation. On reste sur des conseils très généraux ; beaucoup de règles spécifiques dépendront de l'équipe dans laquelle vous travaillerez.

SOMMAIRE

0. Compléments sur les types de base	108
0. Évaluation paresseuse des opérateurs booléens	108
1. Détails sur le passage / renvoi des arguments	109
2. En C : instruction pré-processeur #define	109
<i>Header guards (p. 110). (Mi-HP) Utilisation pour créer des tableaux statiques (p. 110).</i>	
1. Types avancés	111
0. En OCaml : paires, triplets, etc	111
1. Types enregistrements	111
<i>En C (p. 111). En OCaml (p. 112).</i>	
2. Types énumérations	112
<i>(HP) En C (p. 112). En OCaml (p. 112).</i>	
3. OCaml : types construits	113
<i>Notion de type construit (p. 113). let destructurant (p. 113). Le type 'a option (p. 114).</i>	
4. Types récursifs	114
2. Les exceptions	115
0. (HP) En C	115
1. En OCaml	115
<i>Rattraper une exception (p. 115). Lever une exception (p. 116). Définir une exception (p. 116).</i>	
3. Écrire mieux	117
0. Nommer	117
1. Structurer	118
2. Documenter	118
3. Anticiper	119
4. Simplifier	119
5. Progresser	119
4. Tester mieux	120
5. Déboguer mieux	120

0 Compléments sur les types de base

0.0 Évaluation paresseuse des opérateurs booléens

Définition 1 (Évaluation paresseuse des opérateurs logiques).

En C et en OCaml (et Python), les opérateurs `||` et `&&` évaluent d'abord leur opérande de gauche. Selon sa valeur, ils peuvent ne même pas évaluer celui de droite :

- dans `a || b`, le terme de gauche (`a`) est évalué en premier. S'il vaut `true`, on ne calcule pas celui de droite car `true` est absorbant pour `||`.
- dans `a && b`, le terme de gauche (`a`) est évalué en premier. S'il vaut `false`, on ne calcule pas celui de droite car `false` est absorbant pour `&&`.

Exemple.

```

3 let rec mem x lst =
4   match lst with
5   | t :: q -> t = x || (mem x q)
6   | [] -> false

```



Dans la fonction ci-contre, le `||` de la ligne 5 est paresseux. Donc si `t = x`, on évalue pas le terme de droite du `||`. Ainsi, cela permet que dès que l'on a trouvé `x` dans la liste, on arrête le parcours de la liste.

Remarque.


- L'évaluation paresseuse permet d'économiser l'écriture d'un saut conditionnel de la forme « si terme de gauche vaut X, ne pas évaluer le terme de droite ». On peut bien sûr écrire le saut conditionnel, mais ce qui peut alléger le code est bienvenu.

- L'évaluation paresseuse consiste à tester **d'abord le terme de gauche puis ensuite (et seulement si nécessaire) le terme de droite**. Si vous voulez que le terme de droite soit évalué en premier... il faut en faire le terme de gauche.
Le compilateur ne peut pas prédire quel terme sera le plus probablement absorbant, il se contente d'appliquer une règle simple.
- Il n'est pas garanti d'avoir une évaluation paresseuse sur les multiplications d'entiers.

0.1 Détails sur le passage / renvoi des arguments

Lorsqu'on réalise un appel de fonction, l'appel a lieu par valeur. C'est à dire que l'on calcule une valeur pour chacun des arguments, et c'est ensuite cette valeur qui est recopiée dans le bloc d'appel de la fonction.

Remarque.

-  La « valeur » d'un tableau C est son adresse. Autrement dit, si une fonction modifie le contenu du tableau, elle modifie le contenu de l'original.
- En C, pour passer un struct on recopie la valeur de ses arguments.
- En OCaml, tout ce qui n'est pas un booléen ou un entier est en fait un pointeur vers son contenu (que l'on n'a pas le droit de modifier).
- Tout cela est également valable pour le *renvoi* d'arguments, ou plus généralement pour tout moment où l'on a besoin de stocker/communiquer la valeur de quelque chose.

Pourquoi c'est important ? Parce que cela signifie que le calcul de la « valeur » des entrées est fait une seule fois, avant le début de l'appel.

Exemple. Si l'on exécute le code ci-dessous, l'entier 2 sera affiché une seule fois : en effet, la valeur de `foo 1` est calculée *avant* l'appel à `bar`.

```

9  (** Affiche 2*x et le renvoie *)
10 let foo x =
11   let _ = print_int (2*x) in
12   let _ = print_char '\n' in
13   2*x
14
15  (** Renvoie 3*x *)
16 let bar x =
17   (* on appelle x trois fois *)
18   x + x + x
19
20 let y = bar (foo 1)
```



Mais dans quel ordre sont évalués les arguments d'une fonction à plusieurs arguments ? Il n'y a pas de garantie. Le premier argument peut-être évalué avec le second, ou l'inverse.

0.2 En C : instruction pré-processeur #define

Cette sous-section concerne *uniquement* le langage C.

En C, `#define` permet de définir une macro. Une macro est quelque chose qui est cherché-remplacé par le précompilateur : lorsque l'on écrit `#define MACRO valeur`, le pré-compilateur va remplacer toutes les occurrences de `MACRO` par `valeur`.

Dans le cadre du programme de MP2I/MPI, vous ne devez pas savoir les utiliser ; mais vous devez savoir reconnaître et comprendre leurs utilisations.

Remarque. `#define` appartient à une grande famille nommée les **instructions pré-processeur**. De manière générale, en C, toute instruction (toute "ligne") qui commence par `#` est une instruction pré-processeur. Ainsi, `#include` en est une.

0.2.0 Header guards

Si l'on inclut plusieurs fois le même fichier de manière naïve, cela mène à des erreurs : si j'inclus deux fois un fichier, je définis ses fonctions deux fois... or il est interdit de redéfinir un identifiant global (comme un nom de fonction par exemple).

La solution consiste à utiliser des **header guards**, qui garantissent que l'inclusion a lieu une seule fois. Plus précisément, ils garantissent que les inclusions suivantes n'ajoutent rien à la première. On dit qu'on *rend l'inclusion idempotente*.

En pratique, cela demande d'écrire les interfaces ainsi :

```
1  /* Un exemple d'interface
2
3  #ifndef _H_EXEMPLE_H_
4  #define _H_EXEMPLE_H_ 0
5
6  Le contenu du header ici :
7  ...
8  ...
9  ...
10 ...
11 ...
12
13 #endif
```

Il faut comprendre ce header comme ceci : « si la macro `_EXEMPLE_H` n'est pas encore définie : la définir, faire le contenu du header, fin du si ». Ainsi, la première fois que le fichier est inclus durant la compilation, la macro n'est pas encore définie et on rentre dans le Si. Les fois suivantes, la macro est déjà définie, on n'entre pas dans le Si... et on saute donc l'inclusion de la librairie (puisque l'entiereté du header, qui décrit la librairie, est caché derrière de `#ifndef`).

0.2.1 (Mi-HP) Utilisation pour créer des tableaux statiques

Quand on crée un tableau en C avec `type T[...]`, la longueur `...` ne **doit pas** impliquer des identifiants. On ne peut donc pas écrire `int T[len]`, mais on peut écrire `int T[100 + 1]`.

Remarque. Pourquoi cette règle ? Parce que de tels tableaux sont alloués sur la pile, et qu'il est *très* utile de pouvoir prédire la taille de chacun des blocs d'activation de la pile. Il faut donc que la taille des blocs soit connue à la compilation - on dit qu'elle doit être **statique** - ce qui interdit les tableaux dont la longueur dépend d'une variable/fonction.

Toutefois, on a parfois envie d'avoir plusieurs tableaux qui ont la même longueur et de pouvoir changer dans le code source la longueur de tous ces tableaux d'un coup. De plus, donner un nom à la longueur est toujours utile : un nom de variable est plus explicite que 10, 35 200. On peut utiliser un `#define` pour cela :

```
1  #define NB_ELEM 35
2
3  type tab[NB_ELEM]; // tableau statique à NB_ELEM
```

Remarque.

- On ne peut pas utiliser de `const` à la place de la macro ici. On pourrait par contre utiliser `constexpr`, présent en C depuis la version C23.
- Cet usage des macros n'est pas mentionné dans les éléments à savoir reconnaître dans programme officiel. Pourtant, je l'ai déjà vu en concours : considérez donc qu'il faut savoir le reconnaître¹.

1. Heureusement, même sans comprendre ce qu'est une macro, on devine assez bien ce qu'il se passe. La traduction de « define

1 Types avancés

1.0 En OCaml : paires, triplets, etc

Cette sous-section concerne *uniquement* le langage OCaml.

On peut définir à l'aide de `x` (de type `t0`) et `y` (de type `t1`) deux expressions la paire composée de ces deux expressions : il s'agit simplement de `(x,y)` (elle est de type `t0*t1`).

On peut faire de même avec des triplets, quadruplets, etc.

1.1 Types enregistrements

Définition 2 (Enregistrement).

Un type enregistrement est un type dont les éléments sont des n-uplets. Le nombre et le type des coordonnées sont fixés, et chaque coordonnée a un identifiant. On appelle ces coordonnées des **champs**.

Par exemple, on pourrait vouloir définir le type `carte_bus` comme étant un triplet :

- `nom` : le nom d'un personne
- `nb_trajets` : le nombre de trajets restant sur la carte bus
- `age` : l'âge de la personne

Les types enregistrements sont très utiles pour définir des structures de données. En effet, une structure de données est souvent composée de plusieurs informations (par exemple, `arr`, `entree` et `sortie` pour un tableau circulaire). Utiliser un type enregistrement permet qu'au lieu de manipuler une variable par information, on manipule uniquement une « grosse » variable qui est un enregistrement.

L'autre intérêt est de pouvoir, via une interface, cacher le contenu de l'enregistrement au code client ; afin de simplifier : on préfère manipuler une file sans connaître son implémentation que d'avoir à réfléchir aux indices d'un tableau circulaire.

1.1.0 En C

Pour définir le type enregistrement donné en exemple précédemment en C :

```
1 struct carte_bus {
2     char* nom;
3     int nb_trajets;
4     unsigned age;
5 }; // Notez le ; final
```

Le type obtenu est nommé `struct carte_bus`, on peut lui donner un nouveau nom ainsi :

```
1 typedef struct carte_bus nouveau_nom;
```

Pour créer un élément de ce type, on peut le déclarer sans l'initialiser puis modifier le contenu de ses champs. On peut aussi initialiser tous ses champs lors de la création grâce à un initialiseur :

```
1 carte_bus moi = {.nom = "adomenech", .nb_trajets = 12, .age = 42};
```

On peut accéder (pour lire la valeur ou la modifier) à un champ `ch` d'une variable `var` via `var.ch`

» est assez transparente.

1.1.1 En OCaml

Pour définir le type enregistrement donné en exemple précédemment en OCaml :

```
1 type carte_bus = {
2   nom : string;
3   nb_trajets : int;
4   age : int
5 }
```



Pour créer un élément de ce type :

```
1 let moi = {nom = "adomenech"; nb_trajets = 12; age = 42}
```



On peut lire le champ `ch` d'une variable `var` via `var.ch`

Remarque. Lorsque nous introduirons l'impératif en OCaml, nous verrons une variante des types enregistrements qui permet de créer des champs mutables.

1.2 Types énumérations

Définition 3 (Énumération).

Un type énumération est un type défini en listant exhaustivement les valeurs possibles d'un élément de ce type.

Exemple.

- On peut considérer le type booléen comme un type énuméré à deux membres : `true` et `false` (il n'est pas codé comme cela, mais c'est une bonne introduction).

Dans les deux sous-sous-section suivantes, je présente des types énumérations qui contiennent trois membres, mais on pourra bien sûr faire de même à un, deux, quatre, cinq, six, etc.

1.2.0 (HP) En C

En C, voici comment définir un type énumération nommé `enum neveu` qui ne contient que trois valeurs `RIRI`, `FIFI` et `LOULOU` :

```
1 enum neveu = {RIRI, FIFI, LOULOU};
```



On peut bien sûr renommer en utilisant `typedef enum neveu new_name`

Remarque.

- On met des majuscules, car c'est la convention usuelle pour les macros. Le pré-compilateur va chercher-remplacer les occurrences de ces trois variables par respectivement 0, 1 et 2.
- Ils ne sont pas au programme.

1.2.1 En OCaml

En OCaml, voici comment définir un type énumération nommé `neveu` qui ne contient que trois valeurs `Riri`, `Fifi` et `Loulou` :

```
1 type neveu = Riri | Fifi | Loulou
```



On appelle Riri, Fifi et Loulou des **constructeurs** du type `neveu` (des constructeurs à 0 arguments pour être précis²).

On peut ensuite bien sûr donner ces valeurs des variables :

```
1 let a = Fifi
2 let b = Loulou
```



Remarque. ⚠ En OCaml, les majuscules sur les constructeurs sont obligatoires.

1.3 OCaml : types construits

1.3.0 Notion de type construit

En section 1, on a vu comment utiliser des constructeurs à 0 arguments pour définir une énumération en OCaml. On peut également faire des constructeurs de type prenant un argument.

Exemple. Voici par exemple un type `decision` ayant deux constructeurs. Le premier, `Indecis`, ne prend pas d'arguments ; tandis que le second, `Choix`, prend en argument un booléen.

```
1 type decision = Indecis | Choix of bool
```



Les constructeurs ne peuvent prendre que 0 ou 1 argument. Si on veut leur en donner plus, il faut faire des constructeurs prenant en argument des paires, ou des triplets, etc.

Un type ainsi défini en donnant ses constructeurs s'appelle un **type construit**.

1.3.1 let destructurant

Si `p` désigne une paire, on peut utiliser un `let` pour accéder aux deux expressions de la paire. Voici par exemple des fonctions qui renvoient respectivement la première et la seconde coordonnée d'une paire :

```
1 let fst = fun paire ->
2   let (x,_) = paire in x
```



```
1 let snd = fun paire ->
2   let (_,y) = paire in y
```



On dit que de tels `let` sont des **let destructurant** : ils « ouvrent » la structure de ce qu'il y a à droite du signe `=` pour accéder à son contenu.

Pour destructurer un type construit, il faut faire une disjonction de cas :

```
1 type decision = Indecis | Choix of bool
2 let f = fun (d : decision) ->
3   match d with
4   | Indecis -> ...
5   | Choix b -> ...
```



Remarque. Les listes sont des types construits³, et c'est pour cela qu'on y applique des `match`.

2. Je vous renvoie au cours sur les structures de données pour comprendre cette appellation.

3. Le type `'a list` est certes un type construit récursif, mais un type construit quand même.

1.3.2 Le type 'a option

En informatique, on veut souvent faire des fonctions qui renvoient une solution si elle existe, et ne renvoient rien sinon.

Exemple.

- Recherche de l'indice d'un élément dans un tableau. Si l'élément est absent du tableau, il n'y a pas d'indice à renvoyer.
- Recherche d'une star dans un groupe (cf DS2). S'il n'y en a pas, il n'y a rien à renvoyer.
- Renvoyer la tête d'une liste. Si la liste est vide, il n'y a rien à renvoyer.

Dans de nombreux langages modernes, il existe un type pensé pour désigner « quelque chose, ou rien ». En OCaml, il s'agit du type `'a option`. Il est déjà codé, mais à titre informatif voici sa définition :

```
1 type 'a option = None | Some of 'a
```



C'est le type que l'on utilisera dans ces situations en OCaml. Si l'on trouve une solution, on renverra `Some la_solution`, et sinon `None`.

Remarque.

- Si on récupère un élément de type `'a option` renvoyé par une fonction, il faut bien sûr le filtrer pour faire une disjonction de cas sur s'il vaut `None` ou `Some truc`.
- À la différence du `None` de Python, ce `None`-ci a bien un type : `'a option`. Ainsi, si une fonction renvoie une `'a option` et que l'on oublie de la filtrer, le compilateur détectera une erreur *à la compilation*. Cela évite des bugs classiques, par exemple celui qui consiste à obtenir un indice via une recherche par dichotomie, puis accéder à cet indice sans avoir vérifié qu'il n'est pas `None`.⁴

1.4 Types récursifs

Que ce soit en C ou en OCaml, un type peut-être récursif : un de ses champs ou de ses constructeurs pour s'appeler lui-même :

Exemple. Voici deux exemples, un en C et un en OCaml :

```
11 struct cellule_s {
12     int elem;
13     struct cellule_s* next;
14 };
```

list.h

```
16 type 'a list =
17     [] | (::) of 'a * 'a list
```

list.ml

(Extrait du code source d'OCaml!)

Remarque.

- Nous verrons de nombreux types d'arbres au second semestre, qui sont des types récursifs.
- Ces types sont récursifs, on travaillera donc avec par récurrence. Nous verrons dans le chapitre sur l'induction comment étendre la récurrence dans le cas où chaque élément du type est défini à l'aide de *plusieurs* autres éléments. Par exemple :

```
1 type 'a bst = Nil | Node of ('a bst) * 'a * ('a bst)
```



- Par rapport aux types basés sur des tableaux, les types récursifs ont l'avantage d'être très flexibles dans la façon dont on « lie » des éléments entre eux et d'être très facilement redimensionnables. Ils ont l'inconvénient d'être plus difficiles à optimiser par la mémoire cache.

4. De manière générale, le compilateur d'un langage typé évite bien des erreurs.

2 Les exceptions

Une **exception** est un signal envoyé par le code qui interrompt l'exécution et attend que l'on signale que faire.

Par exemple, la fonction `int` de Python permet de convertir une donnée en entier. Si la donnée n'est pas interprétable comme un entier, elle déclenche l'exception `ValueError` ; on dit qu'on **lève** l'exception `ValueError`. Pour demander une valeur⁵ entière à l'utilisateur, on peut faire ainsi en Python :

```
1 x = 0
2 #Tant que la valeur de l'utilisateur n'est pas un entier, réessayer
3 while True:
4     try:
5         x = int(input("Entrez un entier :"))
6         break
7     except ValueError:
8         print("Vous n'avez pas entré un entier, réessayez.")
```



Dans l'exemple ci-dessus, on utilise un `try - except` : on essaye de faire un morceau de code. S'il ne lève aucune exception, tout va bien. Si une exception est déclenchée, on passe immédiatement au `except` : on *redirige exceptionnellement* l'exécution du code.

Point important : lorsqu'une exception est levée, l'exécution de la ligne de code en cours s'interrompt immédiatement, et on « remonte » dans le chemin suivi par le code jusqu'à trouvé un `except` qui gère cette exception. Ainsi, une exception peut faire quitter plusieurs appels de fonction d'un coup⁶.

Les exceptions ont deux rôles principaux :

- Signaler des erreurs, et y réagir. C'est le cas de `ValueError` présentée ci-dessus, ou encore de `DivisionByZero`.
- Sauter rapidement d'un point du code à un autre. Cela permet par exemple de quitter une imbrication d'appels récursifs très rapidement, et sans alourdir le code.

2.0 (HP) En C

Les exceptions telles que présentées ci-dessus n'existent pas en C. Il y a cependant une gestion d'erreur (rudimentaire) en C : si une fonction échoue, elle modifie une variable globale nommée `errno`. Le reste du code peut donc vérifier la valeur de `errno` pour savoir si la fonction s'est bien déroulée ou non.

2.1 En OCaml

2.1.0 Rattraper une exception

En OCaml, l'équivalent de `try - except` est `try - with` :

```
1 try
2     expr0
3 with
4 | NomDUneException -> expr1
5 | AutreExceptionEnvisage -> expr2
6 | ... etc
```



Le tout est une expression, dont le type est la valeur de `expr0` et `expr1` et `expr2` (et ainsi de suite) ; celles-ci doivent donc avoir le même type (comme dans un `then - else`).

5. Via `input` qui est l'équivalent de `scanf`.

6. De même qu'un `return` peut faire quitter plusieurs boucles imbriquées d'un coup, une exception peut faire quitter plusieurs blocs syntaxiques (boucles, fonctions, `if-else`, etc) d'un coup.

2.1.1 Lever une exception

Réciproquement, pour lever une exception, on utilise `raise NomDeLException` .

Exemple. Voici (à peu près) le code source de `List.hd` :

```
29 let hd lst =
30   match lst with
31   | [] -> raise (Failure "hd")
32   | a::_ -> a
```

 list.ml

Ici, si la liste est vide, on renvoie l'exception `Failure "hd"` (il s'agit d'une exception prenant un argument).

Voici, à titre indicatif, des exceptions courantes en OCaml :

- `Invalid_argument` : sert à signaler que l'argument donné à une fonction ne respecte pas les pré-conditions.
- `Not_found` : lorsqu'une fonction qui *doit* renvoyer quelque chose ne trouve pas la valeur à renvoyer. En général, une fonction qui essaye de renvoyer quelque chose mais peut échouer va soit renvoyer quelque chose ou lever une exception; soit être codée comme renvoyant un type option.
- `End_of_file` : levée quand on essaye de lire un fichier alors qu'on a entièrement lu.
- `Failure` : levée par `failwith`. En fait⁷, la fonction `failwith` est définie ainsi :

```
1 let failwith msg =
2   raise (Failure msg)
```



2.1.2 Définir une exception

Pour définir (de manière globale) une exception :

```
1 exception NomDeLException
```



Il s'agit d'une déclaration globale (et non d'une expression).

Remarque.

- Une exception peut être déclarée comme prenant un argument (éventuellement un n-uplet), la syntaxe est la même que pour les constructeurs de type.
- Le nom d'une exception doit débiter par une majuscule.
- Les déclarations locales d'exceptions existent, mais ne sont pas au programme.

7. Et c'est explicitement au programme!

3 Écrire mieux

Un code est écrit une fois, modifié dix fois, et relu cent fois. Il faut donc simplifier son écriture, anticiper les modifications futures, et ne jamais négliger la lisibilité.

Proverbe de programmeur·se (issu de <https://ocaml.org/docs/guidelines>)

Toujours penser à la bonne poire qui devra déboguer et maintenir votre code : c'est le vous du futur. Prenez soin du vous du futur.

Proverbe personnel

Un·e programmeur·se passe (beaucoup) plus de temps à relire et éditer du code qu'à en écrire ; aussi créer du code de qualité n'est jamais du temps perdu⁸. Je présente ici quelques recommandations fondamentales. Vous pouvez exceptionnellement en dévier, uniquement exceptionnellement et pour de bonnes raisons.

3.0 Nommer

- **Utiliser des identifiants explicites.** Par exemple, si une variable `p` contient le nombre d'éléments pairs d'un tableau, ne l'appellez pas `p` ; appelez-le `nb_pairs`.
 - **Pour les variables locales, vous pouvez utiliser des abréviations pour raccourcir le nom ; c'est même recommandé :** une variable locale est souvent utilisée, les raccourcir permet d'accélérer la relecture.⁹ Et si jamais la·e relecteur·ice a un doute sur le sens de l'abréviation, la déclaration est sous ses yeux (une fonction est courte) pour lever le doute !
 - **Cependant, pour les identifiants de fonctions et de variables globales, il est déconseillé d'utiliser des abréviations :** en effet, elles sont déclarées loin avant dans le fichier (voire dans un autre fichier), et c'est fatigant de retourner chercher leurs définitions. Préférez un nom le plus explicite possible quitte à ce qu'il soit long.
- **Évitez les variables globales mutables.** Devoir suivre les changements de valeur d'une
- **N'utilisez pas des noms trop similaires dans une même portée.** Lorsque deux noms de variables ne diffèrent que par un ou deux caractères, on fait des erreurs d'écriture/relecture. Une variable globale, c'est *fatigant*.
- **Si dans différentes fonctions on retrouve des variables locales qui jouent le même rôle, donnez-leur le même nom.**¹⁰
- **Faites attention avec `i`, `j` ou `k`.** Ce sont de bons noms pour des variables qui sont juste des indices ; mais si c'est un indice qui a aussi un sens supplémentaire alors ce sont des noms catastrophiques. On en revient à la première règle : utiliser des noms explicites !
- **Choisissez une convention de casse et tenez-vous y.** Il existe deux grandes conventions pour écrire des noms de variables : séparer les sous-mots par des underscores (`par_exemple_cela` , on parle de **snake_case**), ou par des majuscules (`parExempleCela` , on parle de **camelCase**).

8. Même en TP à l'oral de concours : quand on découvre un bug dans une fonction écrite il y a 40min, il vaut mieux que le code soit bien écrit pour retrouver et corriger rapidement le bug.

9. Je précise : « court » ne signifie pas « une seule lettre ». Cela signifie « moins de dix lettres ».

10. Comme en maths où *f* désigne toujours les fonctions et *x* les arguments.

3.1 Structurer

- **Parenthésiez pour lever les ambiguïtés**, ou pour accélérer la relecture du code. Ne parenthésiez pas lorsque cela n'aide pas.
- **Identifiez votre code**. Python impose l'indentation (et c'est très bien) ; en C et OCaml il faut y faire attention. Voici des règles générales :
 - **À chaque entrée dans un bloc syntaxique, ajoutez un niveau d'indentation. À chaque sortie, enlevez-en un.**¹¹
 - **Une indentation mesure toujours le même nombre d'espaces (2 ou 4).**
 - **Utilisez des espaces ou des tabulations pour indenter, mais pas un mélange des deux.** Dans le cas des tabulations, les éditeurs de texte de développement proposent une option pour régler leur largeur ou pour les transformer automatiquement en espace.
 - **Une accolade fermante a le même niveau d'indentation que la ligne de l'accolade ouvrante associée.**
- **Utilisez des lignes vides pour séparer les blocs logiques de votre code.** Séparez toujours les fonctions du même nombre de lignes vides (typiquement, 2 ou 3). Dans vos fonctions, faites une ligne vide uniquement pour aider à la relecture en divisant la fonction en « paragraphes » (en groupes de lignes qui ont chacun un rôle précis). Inutile toutefois de découper une fonction si elle est déjà simple : laissez les choses simples être simples.
- **Gardez vos fonctions courtes : évitez de dépasser 20 lignes.**
- **Gardez vos lignes courtes : évitez de dépasser 80 caractères.**
- **Une ligne doit correspondre à une seule action.** N'écrivez pas $h(g(f(x)))$, c'est illisible.¹²
- **Si créer une variable rend le code plus lisible, créez une variable.** Une variable ne coûte (vraiment) pas cher¹³ ; alors qu'ajouter un nom à une quantité intermédiaire peut aider à relire le code. En particulier, évitez les « nombres magiques » : si un nombre traine dans votre code sans que l'on comprenne son sens, c'est un mauvais code.
- **Déclarez les variables locales au plus proche de leur utilisation.** Lire une déclaration d'une variable locale qui ne servira que dans 20 lignes, c'est *fatigant*.
- **Si créer une fonction rend le code plus lisible, créez une fonction.** Cela permet de donner un nom explicite au rôle d'un ou plusieurs paragraphes du code.
- **Si plusieurs morceaux de code réalisent la même tâche, créez et utilisez une fonction réalisant cette tâche.** Cela évite la redondance de code qui est source d'erreur : si plusieurs morceaux de code font la même tâche, il est vite arrivé de mettre à jour un des morceaux de code mais pas l'autre et de créer ainsi un bug.
- **Découpez votre code en plusieurs fichiers.** Chaque fichier doit correspondre à une famille de tâche (définir une structure de données, définir des tests, etc). Cela simplifiera *vraiment* la relecture et le débogage.

3.2 Documenter

- **Spécifiez vos fonctions.** Cette spécification se fait typiquement dans l'interface.
- **Utilisez les commentaires pour expliquer les morceaux de code compliqué.** Un commentaire d'une ligne pour un paragraphe de code suffit en général : il faut simplement dire *à quoi servent* ces lignes de code. Ne commentez pas ce qui est déjà simple à relire, et ne paraphrasez pas le code.
- **Annotez les boucles compliquées d'un commentaire en donnant un invariant utile.** On le fait généralement en langage naturel et non en langage mathématique.
- **Si une fonction correspond à un algorithme compliqué, vous pouvez à la débiter par un gros commentaire explicatif.**

11. Notez que c'est ainsi que Python définit ses blocs syntaxiques.

12. En programmation orientée objet, c'est d'ailleurs une des sources les plus courantes de code inutilement compliqué à relire.

13. Et le compilateur peut les enlever lors de ses optimisations du code.

3.3 Anticiper

- **Sachez ce que vous allez coder avant de le coder : de quoi aurez-vous besoin, où et comment allez-vous stocker les données, comment les manipuler, etc ; en bref comment votre code sera organisé.** Si vous n'avez aucune idée du code à écrire, sortez du brouillon et réfléchissez au brouillon.
- **Un bon code ne génère aucun Warning à la compilation.** En C, vous devez utiliser `-Wall -Wextra` pour avoir tous les warnings possibles.
- **Faites de la programmation défensive.** Utilisez des `assert` pour garantir que les pré-conditions sont vérifiées. Il faut que le non-respect des pré-conditions soit élémentaire à déboguer !
- **Garantisiez l'exhaustivité des disjonctions de cas.** Toute disjonction de cas (resp. filtrage par motif) qui n'est pas trivialement exhaustive doit se terminer par un `else` (resp. par un `| _ ->`), quitte à ce que cette branche ne contienne qu'une levée d'exception.¹⁴
- **Faites des messages d'erreurs explicites.** Les `assert` c'est très bien, mais ils donnent un message d'erreur peu lisible¹⁵. Utiliser `if` qui affiche un message d'erreur avant d'interrompre le programme ou de lever une exception, c'est encore mieux.
- **TESTEZ VOS FONCTIONS!!** Plus d'information à ce sujet dans la prochaine section.

3.4 Simplifier

- **K.I.S.S. Keep It Stupid Simple :** *gardez votre code stupidement simple*. Ne vous surestimez pas, et utilisez tous les conseils précédents pour rendre le code stupidement simple. Évitez les optimisations qui rapportent peu mais nuisent à la lisibilité. Évitez les usines à gaz, concevez plutôt des fonctions et bibliothèques élémentaires qui font une seule chose mais la font très bien.
- **Premature optimisation is the root of all evil :** *l'optimisation prématurée est la racine de tous les maux* (citation de D. Knuth). Faites d'abord un programme qui marche, et ensuite seulement un programme optimisé (et uniquement si nécessaire). N'ajoutez qu'une optimisation à la fois, de la plus simple à la plus compliquée - et testez à chaque fois ! Pour réaliser ces mises à jour successives sans pour autant perdre la version de base qui fonctionne, n'hésitez pas à faire correspondre cela à plusieurs implémentations d'une même interface.

3.5 Progresser

Vos autres conseils à vous-même ici :

14. Pensez par exemple à `List.h` : si on lui donne une liste vide (ce que l'on est pas sensés faire), elle lève une exception.

15. Mieux vaut des `assert` que rien !!

4 Tester mieux

Les erreurs sont humaines, et arrivent : c'est normal. C'est pour s'en prévenir que l'on :

- Fait relire son code par d'autres.
- Prouve les morceaux compliqués du code.
- Fait des tests !

Un seul test ne suffit pas : il faut tout un **jeu de test**, c'est à dire un ensemble de tests qui recouvrent tous les points du code. Plus précisément, voici les pré-requis d'un bon jeu de test :

- **Chaque test est unitaire, c'est à dire qu'il teste une seule chose.** Il faut que lorsqu'un test échoue on puisse immédiatement pointer d'où provient le bug.
- **Testez une fonction avant de travailler sur la suivante.** C'est un corollaire du point précédent.
- **Il y a des tests pour les cas limites.** Si une fonction traite un tableau ou une liste, il faut vérifier que les cas particuliers du 1er élément et du dernier élément du tableau fonctionnent bien. Si une fonction est récursive, il faut vérifier que les cas de base fonctionnent bien.
- **Il y a aussi des tests pour les cas généraux.**
- **Si votre fonction doit avoir une bonne complexité, il y a des tests sur de grandes entrées.**
- **Si vous avez un vérificateur, générez des tests aléatoires et vérifiez le résultat de votre code sur ces tests.**
- **Le jeu teste toutes les façons de satisfaire ou non les conditions logiques.** Quand une condition contient des ET/OU, il faut tester toutes les façons de satisfaire ou non les clauses de la condition.
- **Le jeu couvre tous les sommets du Graphe de Flot de Contrôle, c'est à dire qu'il vérifie que toutes les lignes du code sont atteignables.** Cela permet de vérifier qu'on ne s'est pas trompés en écrivant des conditions impossibles dans des sauts conditionnels (en particulier quand on enchaîne les `else if`) et les boucles `while`.
- **Le jeu couvre tous les arcs du GFC, c'est à dire qu'il vérifie que tous les embranchements du code sont utilisables.**

Les deux derniers points sont, et de loin, les plus longs. Il est raisonnable de supposer que vous n'avez pas le temps de le faire dans des TP pensés pour 2/3/4h de programmation.

5 Déboguer mieux

Voici quelques conseils très généraux sur comment déboguer :

- **Avoir un code bien écrit, et doté de tests.** C'est le point le plus important.
- **Utilisez des `assert` pour vérifier que des conditions sont vraies.** C'est en fait le rôle principal de `assert` : déboguer. N'hésitez pas à mettre des `assert` au milieu du code pour vérifier ce que vous avez besoin de vérifier (qu'un indice est compris entre certaines bornes, qu'une certaine quantité est minorée/majorée, qu'un pointeur est non-NULL, etc).
- **Ajoutez des `print` pour voir l'état de certaines variables.** Ce n'est vraiment pas la meilleure méthode car on se retrouve vite submergé.e. On devrait préférer le prochain point, mais il n'est pas au programme en MP2I/MPL.
- **(HP) Utilisez des débogueurs comme `gdb` ou `ocamldebug`.** Je ne vais pas m'étendre dessus, mais voici deux liens si vous voulez creuser :
 - https://perso.ens-lyon.fr/daniel.hirschhoff/C_Caml/docs/doc_gdb.pdf (il y manque la fonctionnalité `frame X`, qui permet de se déplacer à l'appel `X` de la pile d'exécution)
 - <https://ocaml.org/docs/debugging#the-ocaml-debugger> (en anglais, et utilise quelques éléments avancés de OCaml)