Règles à utiliser pour programmer votre projet OCaml

1 Règles anti-bugs

Elles sont principalement au nombre de neuf. Leurs buts sont d'éliminer autant que possible des bugs apparaissant dans deux phases bien séparées de programmation : la phase d'écriture de la première version du programme, et la phase de maintenance ou de mise à jour (réalisée quelques mois ou années plus tard). **Ces règles sont à utiliser impérativement pour l'écriture de votre projet.** Notons qu'elles sont facilement transposables à tout autre langage de programmation.

règle	*utilité*
1/ Séparez vos fonctions en inserant des en-têtes spécifiques	*maintenance*
2/ Commentez au fur et à mesure	*maintenance*
3/ Évitez les fonctions de plus de 50 lignes	*écriture*
4/ Testez les prérequis de vos fonctions	*maintenance*
5/ Donnez des noms à vos fonctions, variables et paramètres qui décrivent précisément ce qu'elles font	*maintenance*
6/ Débuggez au fur et à mesure vos fonctions	*écriture*
7/ Évitez autant que possible les variables globales	*maintenance*
8/ Supprimez judicieusement les warnings	*écriture*
9/ Indentez correctement votre code et évitez les lignes longues	*maintenance*

1.1 L'utilisation des en-têtes dans les fonctions

1.1.1 Utilité des en-têtes

Lors de l'écriture d'un gros programme, on est souvent amené à rechercher dans quel ordre il faut passer les paramètres d'une fonction, ou bien encore ce qu'elle fait précisément et dans quels cas on peut l'utiliser. Il serait donc utile de pouvoir retrouver rapidement la fonction en question dans le listing et d'avoir tous les renseignements qui nous intéressent sans avoir à relire tout le code de la fonction.

Pour trouver facilement la fonction dans le listing, nous vous proposons d'adopter les deux règles suivantes :

1. sauter plusieurs lignes entre deux fonctions consécutives ;
2. placer un signe distinctif suffisamment voyant en début de fonction : une longue ligne de commentaire composée uniquement de « = ».

De quels renseignements avons nous besoin à propos d'une fonction lorsque nous développons ou débuggons ? Il faut connaître avec précision :

1. ce que fait ladite fonction ;
2. ce qu'elle retourne ;
3. à quoi correspondent ses paramètres ;

4. quelles exceptions risquent d'être levées et non rattrapées par la fonction ;
5. quelles sont les hypothèses sur les paramètres pour que la fonction s'exécute correctement ;
6. si le fonctionnement correct de la fonction dépend de la manière dont une fonction précédente a été écrite, il faut le savoir de manière à ce que, si l'on modifie cette dernière, on sache que votre fonction ne produira plus un résultat correct.

1.1.2 Implémentation avec OCamldoc

Tous ces renseignements, nous vous proposons de les mettre en commentaires juste en dessous de la ligne de « = » en utilisant la syntaxe OCAMLDOC : tout commentaire OCAMLDOC commence par `(**` et se termine normalement par `*)`. Les primitives OCAMLDOC que vous utiliserez seront les suivantes :

- `@return` *texte* : indique ce que retourne la fonction ;
- `@param` *nom texte* : indique à quoi correspond le paramètre *nom* ;
- `@raise` *nom texte* : indique que l'exception *nom* risque d'être levée ;
- `@warning` *texte* : permet de spécifier un warning ;
- `@pre` *texte* : indique une hypothèse qui doit être vraie pour que la fonction s'exécute correctement ;
- `@depend` *texte* : indique que la fonction actuelle est dépendante de la manière dont une autre fonction a été définie.

Note : les tags `@pre` et `@depend` sont utilisables sur les machines de l'UFR mais ils ne font pas partie des distributions standards d'OCAMLDOC. Pour pouvoir les utiliser sur une machine hors UFR, reportez-vous à la sous-section 1.1.3. Voici un exemple de commentaire de fonction :

```
(* ================================================================== *)
(** cette fonction renvoie le resultat de la division entiere du premier
    parametre par le second.
    @param x numerateur
    @param y denominateur
    @return division euclidienne de x par y
    @pre y doit etre different de 0 *)
(* ================================================================== *)
let divise x y = x/y;;


(* ================================================================== *)
(** cette fonction prend en argument deux couples contenant les coordonnees
    de 2 points et renvoie un couple contenant les coordonnees du milieu
    de ces points.
    @param x couple contenant les coordonnees du premier point
    @param y couple contenant les coordonnees du deuxieme point
    @return un couple contenant les coordonnees du milieu de x et de y
    @raise Out_of_bounds une exception Out_of_bounds est levee lorsque l'un
           des points n'est pas dans le quadrant positif. *)
(* ================================================================== *)
let milieu x y =
  let (x1,y2) = x and (y1,y2) = y in
    if (x1 < 0) or (y1 < 0) or (x2 < 0) or (y2 < 0) then
      raise Out_of_bounds
    else
      ((x1 + y1)/2, (x2 + y2)/2);;
```

OCAMLDOC fonctionnant avec le système de modules d'OCaml, il serait judicieux de placer avant le commentaire de votre première fonction un premier commentaire OCAMLDOC indiquant une description succincte de ce que contient votre fichier (ou module). Par exemple :

```
(* ================================================================== *)
(** Ce module implemente.....
    @author Christophe Gonzales
    @version 1.0 alpha *)
(* ================================================================== *)
```

Un dernier mot pour vous expliquer rapidement pourquoi utiliser OCAMLDOC. C'est un outil de documentation qui vous permet de générer automatiquement un manuel de référence de votre programme dans de multiples formats. En ce qui nous concerne, nous n'utiliserons qu'une documentation HTML, mais il est aussi possible de générer du LaTeX (qui permettra de générer du postscript ou du pdf), des pages texinfo (lisibles en hypertexte sous emacs par exemple), des pages de manuel UNIX, etc. Votre documentation OCAMLDOC vous permettra donc non seulement de rendre votre listing plus lisible (les commentaires OCAMLDOC sont directement lisibles dans votre code source), mais en outre vous pourrez utiliser votre documention online avec n'importe quel browser. Notez que vous devrez inclure votre documentation OCAMLDOC dans votre rapport final.

Enfin, rien ne vous dispense de compulser la documentation d'OCAMLDOC incluse dans le manuel de référence d'OCaml, qui contient tout un tas de directives de formatage pour personaliser vos documentations.

1.1.3 Lancement d'OCamldoc

Une fois vos commentaires mis en place, il suffit de lancer :

```
ocamldoc -d docs -g /home/Info-mio/OCAML/ocamldoc_projet.cmo -I +threads monprogramme.ml
```

le `-d docs` permet de spécifier que vous voulez placer tous les fichiers html dans un répertoire `docs`[1] et le `-g /home/Info-mio/OCAML/ocamldoc_projet.cmo` que vous allez utiliser une variante du générateur d'html spécifique à votre projet de MIAS.

Il peut sembler un peu rébarbatif de se rappeler les options à passer à OCAMLDOC, aussi serait-il judicieux de rajouter à la fin du fichier `.bashrc` se trouvant dans votre home directory (s'il n'existe pas, créez le) :

```
alias ocamldoc='ocamldoc -d docs -g /home/Info-mio/OCAML/ocamldoc_projet.cmo -I +threads'
```

Pour que cette ligne soit prise en compte, fermez les consoles ou terminaux ouverts et ouvrez-en de nouveaux. Vous n'avez plus alors qu'à taper `ocamldoc monprogramme.ml` pour générer la documentation.

Hébergement d'OCamldoc chez vous :

Un dernier mot si vous voulez installer `ocamldoc_projet.cmo` chez vous : soit vous le copiez des machines de l'UFR sur votre machine, soit vous récuperez le source `ocamldoc_projet.ml` et vous le compilez à l'aide de la commande suivante :

```
ocamlc -I +ocamldoc -c ocamldoc_projet.ml
```

Vous obtiendrez alors sur votre machine un fichier `ocamldoc_projet.cmo`.
À l'UFR, les fichiers `ocamldoc_projet.ml` et `ocamldoc_projet.cmo` se trouvent :

- dans le répertoire `/home/Info-mio/OCAML`
- sur le web : `http://www.infop6.jussieu.fr/deug/2003/mias/mias-o/public/ocaml/`

[1] Attention, le répertoire `docs` doit avoir été créé avant de lancer ocamldoc.

1.2 Les commentaires (hormis ceux d'ocamldoc)

Les commentaires doivent être écrits **en même temps** que vous programmez car c'est à ce moment là que vous avez tout en tête. Prenez un soin particulier à commenter toutes les portions de programme que vous avez eu du mal à réaliser : si cela a été le cas, c'est que le programme ne vous semble pas ultra naturel au premier abord, donc n'espérez pas le comprendre sans problème six mois après. Au moment où l'on vient d'écrire le programme, on se dit que, maintenant, on sait comment le faire, mais six mois après, par expérience, vous pouvez être assurés que vous aurez oublié et que le programme vous semblera totalement hermétique.

1.3 Taille des fonctions

Il faut impérativement éviter les fonctions de plus de 50 lignes (2 écrans en mode texte). En effet, plus une fonction est longue, plus elle a de chances de contenir des bugs. La limite des 50 lignes est préconisée pour l'écriture du noyau Linux, cf. le chapitre 4 sur la page ouèbe :
`http://pantransit.reptiles.org/prog/CodingStyle.html`
Donc si des pros de la programmation utilisent cette restriction, vous le pouvez aussi.

1.4 Tests des prérequis

Il n'est pas rare de voir des fonctions qui ne s'exécutent correctement que dans certains cas et qui plantent lamentablement dans d'autres. Au moment où le programmeur développe son logiciel, il a en mémoire les cas favorables et il s'arrange pour que la fonction en question ne soit appelée que dans ces cas. Las, quelques mois plus tard, il doit effectuer quelques modifications dans son programme et oublie qu'il se trouve maintenant dans un cas défavorable. Le programme ne fonctionne plus correctement et un débuggage s'impose. Problème : où se trouve le bug ? Si on le sait, il n'est pas difficile de le corriger. En revanche, si on ne le sait pas, le débuggage peut s'avérer complexe car le bug ne se trouve pas forcément sur la ligne où le programme a planté. Si vous testez au début de chaque fonction si l'on est dans les conditions d'application correcte de celle-ci (les prérequis), il n'y a pas de lézard : le programme plantera précisément à l'endroit du bug.

Comment s'y prendre ? C'est très simple : commencez par écrire votre fonction. Relisez votre code *attentivement* et, pour chaque instruction, essayez de voir si vous pourriez la faire planter. Si vous trouvez un tel cas, c'est un prérequis. Exemple simpliste :

```
let fonction_buggee x y x0 y0 =
  let alpha = (y -. y0) /. (x -. x0) in
  let delta = sqrt (100. /. (1. +. alpha *. alpha))
  in
    (x +. delta, y +. alpha *. delta);;
```

Vous aurez inévitablement reconnu que cette fonction essaye de calculer la position de l'un des points d'intersection entre la droite passant par les points (x, y) et $(x0, y0)$, et le cercle de centre $(x0, y0)$ et de rayon 10. Le problème ? C'est la ligne du « `let alpha` » : si $x = x0$, une erreur survient. Donc, **même si vous savez qu'actuellement votre programme n'appellera jamais la fonction avec** $x = x0$, vous devez tester si vous êtes dans ce cas et prendre les mesures adéquates pour arrêter votre programme :

```
let fonction_pas_buggee x y x0 y0 =
  (* test des prerequis *)
  if x = x0 then failwith "fonction_pas_buggee : erreur x = x0"
  else

    (* execution de la fonction *)
    let alpha = (y -. y0) /. (x -. x0) in
```

```
    let delta = sqrt (100. /. (1. +. alpha *. alpha))
    in
      (x +. delta, y +. alpha *. delta);;
```

Notons qu'ici on aurait pu calculer l'intersection plutôt que de faire échouer la fonction. Mais dans le cas général, il faut faire échouer tout le programme, de telle sorte qu'en regardant l'empilement des appels de fonction dans un debugger, on puisse savoir exactement où se situe le bug. Rappelons enfin qu'il **faut absolument** documenter les prérequis : tests et documentation sont complémentaires, pas exclusifs.

1.5 Les noms des fonctions, des variables et des paramètres

En soi, on peut penser que le nom que vous donnez à vos fonctions n'est pas très important. Grave erreur : au moment où vous en programmez une, vous savez précisément ce qu'elle fait. Mais au bout de quelque temps, vous allez oublier et c'est sans doute à ce moment là que vous allez vouloir modifier votre code. Si vous savez que vous avez une fonction `somme : int->int->int` dans votre code et que vous voulez réaliser la somme de 3 et de 4 (autrement dit, ça fait 7), pourquoi ne pas utiliser cette fonction ? Eh bien peut-etre parce que son code est le suivant :

```
let somme x y = (3 * x + 7 * y ) / 10 ;;
```

Eh oui, c'était une somme pondérée et non une somme au sens classique. Résultat : on obtient 3 au lieu de 7, et tout le reste du programme produit un résultat erroné. Si l'on avait appelé la fonction `somme_ponderee`, on aurait su tout de suite qu'elle ne correspondait pas à l'opération que l'on voulait effectuer. Bien évidemment, les mêmes règles s'appliquent aussi aux noms des variables et des paramètres que vous passez à vos fonctions.

1.6 Le débuggage incrémental

On a déjà vu des étudiants programmer entièrement un projet sans jamais le compiler — ou si peu — et, une fois ce dernier achevé, l'exécuter pour la première fois et lancer une phase intensive de débuggage. Cette approche nous semble très sous-optimale. Comme nous l'avons déjà dit, le plus dur dans le débuggage, c'est de savoir sur quelle ligne se trouve le bug. Si, chaque fois que vous écrivez une nouvelle fonction, vous la compilez et vous la débuggez. Si bug il y a, c'est obligatoirement sur la dernière fonction (puisque les autres sont déjà débuggées). Il s'ensuit que la recherche du bug est très simplifiée puisqu'il se trouve sur un nombre de lignes très limité (moins de 50 lignes). Imaginons par exemple que nous ayons programmé (très salement) un code permettant de calculer la $n^{\text{ème}}$ valeur de la suite $u_{n+1} = u_n^3 \times 2$, u_0 étant égal à 1, ainsi qu'une fonction calculant la valeur de u_{n+2}/u_n pour un n passé en paramètre :

```
let prochain_terme p =
  p * p + p * 2;;

let rec u n =
  if n = 0 then 1 else prochain_terme (u (n - 1));;

let v n = (u (n + 2)) / (u n);;

v 0;;
v 1;;
v (-2);;
```

Tout d'abord, remarquons que ce code n'a aucun commentaire ni aucun prérequis, ce qui indique déjà un piètre niveau de programmation. Supposons que nous soyons en train de débugger la fonction `v`.

Cette fonction ne renvoie pas de résultat correct. Où se trouve le bug ? Après inspection de la fonction `v`, on s'aperçoit que le code de celle-ci est correct. Si l'on inspecte le code de `u`, on peut s'apercevoir que les prérequis ne sont pas testés puisque la fonction devrait tourner indéfiniment pour $n < 0$, ce qui explique pourquoi `v (-2)` plante lamentablement. Mais cela n'explique pas pourquoi `v 0` et `v 1` produisent des résultats incorrects. Le problème principal provient en fait de la fonction `prochain_terme` dans laquelle une faute de frappe a substitué un `*` par un `+`. On voit donc que, parce que l'on n'a pas débuggé les fonctions les unes après les autres, on est obligé de rechercher le bug dans tout le listing. Tout aurait été beaucoup plus simple si l'on avait d'abord débuggé `prochain_terme` :

```
let prochain_terme p = p * p * p * 2;;
```

On aurait pu alors écrire correctement la fonction `u` :

```
(* ================================================================== *)
(** cette fonction calcule le nieme terme de la suite u_n = u_n^3 * 2, pour
    n>=0. Lorsque n=0, elle renvoie 1 et elle leve une exception lorsque n<0.
    @param n valeur de l'indice pour lequel on calcule la valeur de la suite
    @return la valeur du nieme terme de la suite
    @raise Failure cette exception est levee si l'on passe une valeur negative de n *)
(* ================================================================== *)
let rec u n =
  if n = 0 then 1
  else if n < 0 the failwith "u : valeur negative de n"
  else prochain_terme (u (n - 1));;
```

On peut alors envisager de programmer `v` :

```
(* ================================================================== *)
(** cette fonction calcule la valeur de u_{n+2} / u_n pour n>=0 et leve une
    exception si n est strictement negatif.
    @param n valeur de l'indice pour lequel on calcule u_{n+2} / u_n
    @return la valeur de u_{n+2} / u_n
    @raise Failure cette exception est levee si l'on passe une valeur negative de n *)
(* ================================================================== *)
let v n =
  if n < 0 then failwith "v : valeur negative de n"
  else (u (n + 2)) / (u n);;

v 0;;
v 1;;
v (-2);;
```

Le débuggage incrémental suppose bien évidemment que vous avez « correctement » débuggé les fonctions précédentes et que virtuellement aucun bug ne vous a échappé. Vous devez donc prêter une attention toute particulière au code de la fonction que vous êtes en train de débugger (y passer un peu de temps) et essayer par tous les moyens de la faire planter. Il ne faut surtout pas se dire qu'a priori votre fonction est correcte et qu'un examen rapide devrait vous indiquer si c'est le cas : c'est le meilleur moyen de ne pas voir le bug.

1.7 Les variables globales

Les variables globales sont à proscrire si elles ne sont pas nécessaires, ou bien si elles ne sont pas utilisées par beaucoup de fonctions. En ocaml, les variables connues par une fonction ainsi que leurs valeurs sont celles au *moment de la création* de la fonction. Cela peut induire quelques effets indésirables si l'on n'y prend pas garde :

```
let alpha = 3;;
let f x = alpha * x + 2;;
let alpha = 5;;
f 5;;
```

`f 5` vaut bien évidemment 17 car c'est le premier `alpha` qui est pris en compte par la fonction. Sur un long programme et si l'on lit un peu rapidement le listing, on risque de se méprendre. Donc, si c'est possible, préférez le passage d'arguments :

```
let f alpha x = alpha * x + 2;;
```

1.8 Les warning sont indésirables

On peut toujours s'arranger pour avoir un programme sans warning à la compilation, et c'est précisément ce que vous ferez. Il existe des warnings bénins et des warnings indiquant réellement des bugs. Il n'est pas toujours évident de faire la distinction entre les deux. En outre, le warning bénin d'un jour peut se métastaser en warning à bug le lendemain. Donc éliminez tous les warnings surgissant à la compilation. Bobby Lapointe disait qu'il y a deux façons de jouer du violon : soit vous jouez bien, soit vous jouez tzigane. Pour l'élimination des warnings, c'est pareil : soit vous prenez le temps de bien comprendre le message, vous essayez de comprendre pourquoi ce message apparaît et vous en déduisez les modifications à effectuer dans votre programme, soit vous essayez au pif des modifications jusqu'à ce que le warning disparaisse et vous obtiendrez un programme bancal. Inutile de préciser l'approche qui semble préférable.

1.9 Le tuareg sera votre berger

Comme tout langage, il existe des « styles » de programmation : une philosophie, une indentation, une mise en page particulière. Lorsque vous travaillez sous emacs, le mode `tuareg` vous offre la pagination « classique ». Ce mode est déjà installé sur les machines de l'UFR et il vous suffit de taper « M-x tuareg-mode » sous emacs pour l'activer (M-x correspond à Meta-x, ou encore à taper simultanément sur les touches Alt et x). Si vous voulez réindenter votre code, vous pouvez en sélectionner une partie en cliquant sur la bouton gauche de votre souris, en la bougeant tout en gardant le bouton appuyé puis, après l'avoir relaché, en tapant « M-x indent-region ».

Hébergement du tuareg chez vous :

Tout d'abord, il faut récupérer l'archive contenant ce mode. La dernière en date est disponible sur la page web :

`http://www.infop6.jussieu.fr/deug/2003/mias/mias-o/public/ocaml/tuareg-mode.tar.gz`

Copiez ce fichier sur votre machine et installez-le à l'aide de la commande suivante :

```
tar xvz -C /usr/share/emacs/site-lisp -f tuareg-mode.tar.gz
```

1.10 De la non utilisation du copier/coller

Autant que possible, vous factoriserez dans des boucles les répétitions de séquences d'instructions de tailles significatives. Les copier/coller sont en effet assez dangereux car plutôt générateurs de bugs.

1.11 Quelques recommandations diverses

Voici quelques règles simples à appliquer pour ne pas « galérer » lorsque vous écrirez de grosses (et même de petites) applications :

1/ Le source d'un programme est destiné à être lu : i) sur votre console d'ordinateur ; ii) sur un listing. Il faut donc qu'il puisse apparaître de manière lisible sur ces deux média. Aussi, vous n'écrirez pas de ligne plus longue que 75 caractères.

2/ Vous devez être particulièrement vigilants à obtenir un code **très lisible**. Cela suggère que vous réfléchissiez un peu à ce que vous comptez écrire avant de programmer. Toute bonne programmation commence avec une feuille et un crayon. Lorsque vous pensez avoir une bonne vision de ce que vous devez écrire, alors seulement vous passez sur machine. Vous devez toujours obtenir des programmes simplement compréhensibles par un autre programmeur.

3/ A l'exception des index de boucles, vous devez avoir des noms de variables significatifs et pas trop longs. Des variables telles que `toto`, `xxx`, `X23`, `lavariabledelamortquituequiesttroplongue` sont à proscrire car on ne voit pas du tout ce qu'elles représentent.

4/ Vous n'emploierez jamais de variable nommée `'O'` ou `'l'` (o majuscule ou l minuscule) car on peut trop facilement les confondre avec les constantes entières 0 et 1.

5/ Voyons maintenant une règle permettant une bonne évolutivité de vos programmes : on n'écrit jamais de constantes directement dans le code, mais on utilise des définitions en début de programme : `let rayon = 1.32;;`. Ainsi, si vous voulez modifier la valeur de la constante, il n'y a qu'une ligne à modifier dans tout le programme.

Maintenant, vous voilà prêt à programmer. Ces règles ont l'air un peu embêtantes à première vue, mais elles sont très simples à respecter et elles vous éviteront bien de déboires. En particulier, elles devraient vous permettre de limiter au maximum les phases de débugage.

2 Votre rapport final

Lorsque l'on évalue un projet, la partie « programmation » est bien évidemment importante. En particulier, il est préférable d'avoir un code propre et évolutif plutôt qu'un code faisant plus de choses mais écrit de manière porcine. Cela dit, la programmation doit s'accompagner d'un rapport précisant ce qui a été réalisé, comment cela l'a été et pourquoi cela a été réalisé de cette manière. L'idée en est fort simple : imaginez que vous soyez dans une entreprise et que vous ayez développé un gros projet. Vous décidez de changer d'entreprise. Votre successeur sur ce projet doit-il être obligé de lire l'ensemble de vos 50000 lignes de code pour savoir ce qui doit être modifié pour changer la couleur de fond de la page d'accueil de votre logiciel ? Assurément non, vous devez donc aider vos successeurs (voire vous-même si vous reprenez la programmation du logiciel après plusieurs mois passés à faire autre chose) à rentrer facilement dans votre code. Pour cela, chaque binôme devra rédiger un dossier de projet comprenant :

- *Une description générale du problème.* C'est fort utile de savoir, au bout de quelques mois ce que fait exactement votre programme.
- *Une justification précise du découpage fonctionnel.* Là encore, l'architecture du programme paraît évidente lorsqu'on est en train d'écrire ce dernier, mais semble de plus en plus obscure au fur et à mesure que le temps passe. Une description de l'architecture du programme peut éviter d'avoir à relire une partie du listing lorsqu'on fait évoluer le logiciel.
- *Une description approfondie des fonctions ayant un contenu algorithmique important.* Ce type de fonctions étant, d'une manière générale assez complexe, mieux vaut les expliquer au moment où on les programme, car après, on risque des *nervous breakdowns comme on dit de nos jours* à essayer de comprendre ce que l'on a écrit. Notons qu'une explication précise et brève de ce que font chacune des fonctions de votre programme est aussi nécessaire. Cela dit ce type d'explication relèvera de votre documentation OCAMLDOC.
- *Un listing commenté du programme et la documentation générée par* OCAMLDOC.
- *Si des modifications ou des ajouts ont été apportés au sujet initial, une description précise de ces modifications ou ajouts.*