Conseils de programmation CAML

Ce document est une très légère adaptation des consignes que l'on trouve sur le site de CAML : `http://ocaml.org/learn/tutorials/guidelines.html`. Il s'agit d'un ensemble de conseils raisonnables de présentation des programmes Caml, ainsi que des conseils de programmation qui ont recueilli l'assentiment de programmeurs Caml chevronnés. Ces conseils sont valables pour vos programmes sur ordinateurs *ainsi que pour vos copies*.

1 Conseils généraux pour l'écriture

Soyez simples et lisibles Le temps passé à taper les programmes est négligeable par rapport au temps passé à les lire. C'est pourquoi on gagne un temps précieux à réaliser des programmes dont la lisibilité est optimale.

✔ Un programme est écrit une fois, modifié 10 fois, relu 100 fois.

2 Conseils de présentation des programmes

Pseudo loi des espaces N'hésitez pas à séparer les mots de vos programmes à l'aide de blancs ; la touche d'espacement est la plus facile à trouver, il n'y a pas de raison de s'en priver !

Délimiteurs Les symboles délimiteurs sont suivis d'une espace, les symboles d'opérations sont entourés d'espaces. Ce fut un grand progrès de la typographie de séparer les mots par des blancs pour rendre les textes écrits plus faciles à lire. Faites de même dans vos programmes si vous voulez des textes lisibles.

Ecriture des paires Un n-uplet est parenthésé et la ou les virgules (délimiteurs) sont suivies d'une espace. Exemples : `(1, 2)`, `let triplet = (x, y, z)`. Exceptions :

- Au lieu de `let (x, y) =`, on peut écrire `let x, y =`. On peut voir cela comme la définition simultanée de plusieurs valeurs.
- Dans le cas d'un filtrage simultané sur plusieurs valeurs, on peut écrire `match x, y with`. On peut voir cela comme un filtrage en parallèle sur plusieurs valeurs :

Écriture des listes On écrit `elt :: liste` avec des espaces autour du symbole `::` (car `::` est un opérateur infixe donc entouré d'espaces) et `[1; 2; 3]` (car `;` est un délimiteur donc suivi d'une espace).

Écriture des symboles d'opérations On aura soin de bien séparer les symboles d'opération par des espaces : non seulement les formules sont plus lisibles, mais les confusions d'opérateurs multi-symboles sont évitées. (Exceptions évidentes à cette règle, les symboles `!` et `.` ne sont pas séparés de leurs arguments.) Exemple : on écrit `x + 1` ou `x + !y`.

On pourrait vouloir écrire `x*y + 2*z` qui met bien en évidence la priorité de la multiplication sur l'addition. Cette idée est séduisante mais elle est illusoire, parce que rien dans le langage ne donne l'assurance que les espaces reflètent bien la signification de la formule. Par exemple `x * z-1` signifie bien `(x * z) - 1`, et non `x * (z - 1)` comme l'interprétation proposée des espaces semblerait le suggérer. Si vous voulez mettre en évidence les priorités, utilisez le moyen significatif que vous procure le langage : mettez des parenthèses.

3 Indentation des programmes

✔ Traitez l'indentation de vos programmes comme si elle déterminait la signification de vos programmes.

☞ **Remarque 1**

Dans certains cas, en PYTHON *par exemple, elle détermine vraiment leur signification !*

Valeur de l'indentation Les changements d'indentation des lignes de programme sont généralement de 2 ou 4 espaces. Adoptez une valeur d'indentation et respectez-la dans tout le programme.

Le *corps* d'une fonction est indenté.

Indentation des `let ... in` L'expression qui suit une définition introduite par `let` est indentée au même niveau que le mot clé `let`. Si l'expression `expr1` est courte, on peut tout écrire sur une même ligne :

OCaml
```
let ident = expr1 in
expr2
```

Dans le cas d'une série de `let`, la règle précédente implique que ces définitions soient placées au même niveau d'indentation (l'idée est qu'une série de `let ... in` est analogue à un ensemble d'hypothèses dans un texte mathématique, d'où l'indentation au même niveau de toutes les hypothèses) :

OCaml
```
let ident1 = expr1 in
let ident2 = expr2 in
...
```

Si l'expression est longue (e.g. plus d'une ligne) elle est placé dans le corps du `let ... in` et est indentée :

OCaml
```
let ident =
  expr1
in
expr2
```

Indentation des `if cond then expr1 else expr2` L'indentation des conditionnelles dépend de la taille des expressions qui les composent. Si `cond`, `expr1` et `expr2` sont petites on écrit simplement sur une ligne :

OCaml
```
if cond then expr1 else expr2
```

Sinon on privilégiera la forme :

OCaml
```
if cond1 then
  expr1
else if cond2 then
  expr2
else
  expr2
```

Le cas des expressions conditionnelles multiples se traite comme si `else if` était un mot-clé.
S'il y a des délimiteurs `begin ... end` :

OCaml
```
if cond then begin
  expr1
end else begin
  expr2
end
```

Indentation des filtrages Les clauses de filtrages sont toutes introduites par une barre verticale y compris la première et sont toutes alignées au niveau de la barre verticale qui commence chaque clause. Si l'expression correspondant à une clause ne tient pas sur la ligne, on ira à la ligne immédiatement après la flèche de la clause (qui terminera donc la ligne), puis l'on indentera normalement, à partir du début du filtre correspondant. Les flèches des clauses n'ont pas à être alignées. Il y a des espaces après `|` et autour de `->`. Exemple :

OCaml
```
match expr with
| motif1 -> expr1_courte
| motif2 ->
    expr2_longue
| motif3 when cond2 ->
    expr3_longue
```

Indentation des appels de fonctions Dans toute la mesure du possible on évitera les arguments constitués d'une expression complexe : dans ce cas on définira les « gros » arguments par une construction let. Par exemple au lieu de

OCaml
```
let temp =
  f x y z
  ``large
  expression''
  ``other large''
  expression'' in
  ...
```

on écrira

OCaml
```
let t =
  ``large
  expression''
in
let u =
  ``other large''
  expression''
in
let temp =
  f x y z t u in
  ...
```

Indentation des opérations Lorsqu'une opération comporte des arguments complexes, ou en présence de multiples appels à la même opération, l'on terminera la ligne à l'opérateur, et l'on n'indentera pas le reste des opérations. Par exemple :

OCaml
```
x + y + z +
t + u
```

La liaison des expressions trop grosses pour être écrites dans une opération s'impose franchement en cas de combinaison d'opérateurs. Au lieu de l'expression peu lisible :

OCaml
```
(x + y + z * t) /
  (``large
    expression'')
```

on écrira

OCaml
```
let u =
  ``large
  expression''
in
(x + y + z * t) / u
```

Ces conseils s'étendent à tous les opérateurs (par exemple `::`).

4 Conseils de programmation

4.1 Méthode de programmation

Programmez simple et clair Quand c'est fini, relisez, simplifiez et clarifiez. À toutes les étapes de la création, utilisez votre tête !

Découpez vos programmes en petites fonctions Les petites fonctions sont plus faciles à maîtriser.

Mettez en facteur les morceaux de code répétés en les définissant dans des fonctions séparées Le partage du code ainsi obtenu facilite la maintenance puisque toute correction ou amélioration est automatiquement répercutée. En outre, le simple fait d'isoler et de nommer un morceau de code permet quelquefois d'identifier une fonctionnalité insoupçonnée.

N'utilisez pas le copier-coller en programmant Coller du code signifie à coup sûr l'introduction d'un défaut de partage du code du programme et une négligence à identifier une fonction auxiliaire utile ; donc cela signifie qu'on perd du partage dans le code du programme. La perte de partage de code conduit à des difficultés dans la maintenance : une erreur dans le code copié doit être corrigé à chaque occurrence du

bug dans chacune des copies ! De surcroît, il est difficile de reconnaître les mêmes 10 lignes de code repétées 20 fois dans un programme. En revanche, si ces 10 lignes sont définies par une fonction auxiliaire, il n'y a aucun problème à savoir où ces lignes sont utilisées : ce sont tous les sites d'appel de la fonction auxiliaire. Couper-coller du code rend donc les programmes plus difficiles à comprendre.

✔

En conclusion, copier-coller du code produit des programmes plus difficiles à lire, à comprendre et à maintenir : il faut absolument l'éviter.

4.2 Commentaires dans les programmes

N'hésitez jamais à commenter lorsqu'il y a une difficulté. S'il n'y a aucune difficulté, il n'y a pas lieu de commenter. Préférez un commentaire au début de la fonction qui explique le fonctionnement de l'algorithme utilisé et gardez les commentaires dans le code de la fonction pour les détails.

4.3 Choix des identificateurs

Il est difficile de choisir des identificateurs dont le nom évoque la signification du morceau de programme correspondant. C'est pourquoi on y attachera un soin particulier, en privilégiant la clarté et la régularité du nommage.

Toujours nommer du même nom les arguments de fonctions qui ont la même signification. S'il y a plusieurs arguments de même signification on les suffixera par des numéros.

Ne pas utiliser d'abréviations pour les noms globaux Les identificateurs globaux (y compris et surtout ceux des fonctions) peuvent être longs, car il importe de comprendre à quoi ils servent loin de leur définition.

Séparer les mots par des soulignés C'est la convention en CAML (et en PYTHON) et il faut la respecter. Exemple : `int_of_string` et non pas `intOfString`.

4.4 Utilisation des parenthèses dans les expressions

Les parenthèses sont significatives : elles indiquent la nécessité d'utiliser une priorité inhabituelle. Elles doivent donc être utilisées à bon escient et non pas saupoudrées au hasard dans les programmes. Pour cela, vous devez connaître les priorités habituelles, c'est-à-dire les combinaisons d'opérations qui ne nécessitent pas de parenthèses. Fort heureusement ce n'est pas compliqué pour qui connaît un peu de mathématiques ou s'efforce de suivre les règles suivantes :

Opérateurs arithmétiques Mêmes règles qu'en mathématique. Par exemple : `1 + 2 * x` est identique à `1 + (2 * x)`.

Applications de fonctions Mêmes règles qu'en mathématique dans l'usage des fonctions *trigonométriques*. En mathématique, on écrit `sin x` pour signifier `sin (x)`. De même `sin x + cos x` signifie `(sin x) + (cos x)` pas `sin (x + (cos x))`. On utilise les même conventions en CAML : on écrit `f x + g x` pour signifier `(f x) + (g x)`.

Cette convention est généralisée à tous les opérateurs (infixes) : `f x :: g x` signifie `(f x) :: (g x)`, `f s ^ g s'` signifie `(f s) ^ (g s')`, `failwith s ^ s'` signifie `(failwith s) ^ s'` et pas `failwith (s ^ s')`.

Comparaisons et opérateurs booléens Les comparaisons sont des opérateurs infixes, donc les règles précédentes s'appliquent. C'est pourquoi `f x < g x` signifie `(f x) < (g x)`. Pour des raisons de typage (pas d'autre interprétation sensée), l'expression `f x < x + 2` signifie `(f x) < (x + 2)`. De même `f x < x + 2 && x > 3` signifie `((f x) < (x + 2)) && (x > 3)`.

Les priorités relatives des opérateurs booléens sont celles des mathématiques.

De même que `1 + 2 * x` signifie `1 + (2 * x)`, `true || false && x` signifie `true || (false && x)`.

✔

Dans le doute ou dès que vous pensez que la lisibilité en est améliorée, utiliser des parenthèses.

Délimitation des constructions dans les programmes Lorsqu'il est nécessaire de délimiter des constructions syntaxiques dans les programmes, on utilise comme délimiteurs les mots-clés begin et end plutôt que des parenthèses. Exemple :

OCaml
```
match x with
| 1 ->
  begin match y with
  | ...
  end
| 2 ->
...
```

Il faut impérativement délimiter le filtrage imbriqué, sinon les clauses du filtrage englobant sont automatiquement associées au filtrage englobé.

Filtrages On n'aura jamais peur d'abuser du filtrage ! En revanche, on aura soin d'éviter les filtrages non exhaustifs. Ils seront complétés avec soin, sans utiliser une clause « attrappe-tout » comme `| _ -> ...` quand il est possible de s'en passer.

Les filtrages non-exhaustifs induits par des clauses avec gardes doivent aussi être corrigés. Un cas typique consiste à supprimer une garde redondante.

`let` déstructurant On n'utilisera le `let` destructurant que dans le cas où le filtrage est exhaustif. Dans ce cas, on dit que le filtre est *irréfutable*. Typiquement, on se limitera donc à des définitions de type produits (n-uplets ou enregistrements) ou à des définitions dont le filtre appartient à un type somme à un seul cas. Dans tous les autres cas, on préfèrera une construction `match ... with` explicite.

`let _ = ...` La liaison `let _ = ...` ne définit rien du tout, se contentant en fait d'évaluer l'expression à droite du signe `=`. Elle ne doit être utilisée que ponctuellement, et pour expliciter un résultat ignoré.

Par exemple, on utilise une liaison vide, pour supprimer l'avertissement de :

OCaml
```
List.map print_int liste;
print_newline ()
```

on écrit

OCaml
```
let _ = List.map f l in
print_newline ()
```

On peut aussi utiliser la fonction prédéfinie `ignore : 'a -> unit` qui ignore son argument pour rendre `unit`. Mais le meilleur moyen est encore de réfléchir sur son code et de se demander pourquoi l'on obtient cet avertissement. Le compilateur émet une alerte parce que le programme calcule un résultat inutile puisqu'il est jeté. Donc, s'il n'est pas complètement inutile, ce calcul ne sert que par ses effets ; il devrait donc le signaler en retournant la valeur rien. La plupart du temps l'alerte indique qu'on n'a pas appelé la bonne fonction, ou bien que la fonction nécessaire n'existe pas et doit être écrite. Ici, on est dans le premier cas, et il fallait évidemment écrire

OCaml
```
List.iter f l;
print_newline ()
```

Avertissements du compilateur Les avertissements du compilateurs sont destinés à prévenir des erreurs potentielles ; c'est pourquoi il faut impérativement en tenir compte et corriger vos programmes si leur compilation produit de tels avertissements. De plus, les programmes dont la compilation produit des avertissements ont un parfum d'amateurisme qui ne sied certainement pas à vos propres oeuvres !