TRAVAUX PRATIQUES XXIII

Compléments sur le langage OCaml

Le but de ce TP est de découvrir de nouveaux concepts fondamentaux des langages de programmation modernes.

.1 Un peu de déboguage

Dans le fichier `dichotomie.ml` se trouve une recherche par dichotomie supposée renvoyer l'indice d'un élément dans un tableau (et -1 si il est absent).

1. Déboguez-la. Pour cela, vous pouvez commencer par faire plus de tests, en réfléchissant aux différents cas pertinents à prendre en compte.

A Le type option

Ceci est une parenthèse rapide sur un type que l'on a déjà vu : le type `'a option` .

La fonction de recherche dichotomique précédente est, dans l'idée, ce que les mathématiques appellent une fonction partielle : si `x` est présent dans `t` on renvoie son indice et sinon... il n'y a pas de valeur à renvoyer sur cette entrée. L'astuce utilisée jusqu'à présent est de renvoyer `-1`. Mais c'est une mauvaise pratique parce que :

— On ne comprend pas. Si l'on a pas en tête la spécification précise de la fonction, ce `-1` ne veut rien dire.

— Si jamais on essaye de faire des calculs avec l'indice renvoyé, faire un calcul avec ce `-1` ne fera pas d'erreur ! Principe général : *fail fast*. Un code simple à débuguer est un code qui renvoie une erreur le plus tôt possible : si votre fonction `f` est appelée par `g` qui est appelé par `h` et que ce n'est que dans `h` que l'on se rend compte qu'il y a un bug, le déboguage sera *infernal*. [1]

— (Lié aux deux précédents) Un jour, votre code sera utilisé par quelqu'un d'autres que vous. Cette personne n'aura pas en tête tout ce que vous avez fait. Elle modifiera peut-être votre code sans le comprendre, et sans lire vos commentaires. Il faudra que votre code soit clair et résiliant aux bêtises. Renvoyer un nombre magique comme `-1` n'est ni clair ni résiliant aux bêtises.

On va donc virer cet affreux `-1`. À la place, on veut utiliser le type option qui permet de dire soit « j'ai trouvé une réponse : celle-ci » soit « je n'ai pas trouvé de réponse ».

Voici sa définition (vous n'avez pas à le recoder, il est pré-codé) !

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

Autrement dit, si votre algorithme a trouvé une réponse `v` , renvoyez `Some v` . S'il n'a pas trouvé de réponse (par exemple parce qu'elle n'existe pas), renvoyez `None` .

2. Modifiez la dichotomie pour qu'elle renvoie une option.

B Les exceptions

On va maintenant voir un concept plus général : les exceptions. Une exception, c'est une « erreur » : lorsqu'une exception est trouvée, le code arrête de s'évaluer. Si rien n'est fait, on obtient un message d'erreur avec l'erreur en question.

3. Dans `utop` , tentez la déclaration suivante : `let x = 3/0` . Que signifie le message obtenu ? Faites la même chose mais en compilé.

Les exceptions servent donc à signaler les erreurs, mais pas que [2] : il s'agit en fait de rediriger exceptionnellement le

1. Croyez-mon expérience : vous ne voulez pas avoir à déboguer de tels monstres.
2. Sinon elles s'appelleraient des erreurs, pas des exceptions...

flot du code (pour rappel, le flot du code est son ordre d'exécution, cf graphe de flot de contrôle).

Il est possible de **rattraper une exception**, c'est à dire de préciser quoi faire en cas d'exception (on peut même préciser plusieurs exceptions différentes à rattraper). On utilise pour cela la construction `try ... with ...` :

OCaml
```
try
  expr0
with
  | Exception1 -> val1
  | Exception2 -> val2
  | etc etc etc
```

Exemple :

OCaml
```
let division (x : float) (y : float) =
  try
    x /. y (* rappel : /. est la division entre flottants *)
  with
    | Division_by_zero -> Float.infinity
```

4. Testez le code précédent.

On peut définir ses propres exceptions. Pour cela, on utilise la phrase OCaml ci-dessous. Un npm d'exception doit toujours commencer par une majuscule

OCaml
```
exception Un_super_nom
```

Une exception peut prendre un argument :

OCaml
```
exception Nom of int
```

On peut lever (déclencher) volontairement une exception avec l'expression `raise Nom_de_l_exception` .
Remarque : Il est important de comprendre que dès qu'une exception est rencontrée, l'évaluation en cours s'interrompt *immédiatement*. Lever une exception est donc une façon d'interrompre une suite d'appels récursifs ou de tours de boucles. Par exemple :

OCaml
```
exception Break

let demo (t : 'a array) (x : 'a) =
  try
    for i = 0 to Array.length t -1 do
      if t.(i) = x then raise Break
    done;
    false
  with
    | Break -> true
```

5. Que fait la fonction ci-dessus ? Combien de tours de boucle ont lieu ?

6. Modifier votre dichotomie pour qu'elle renvoie l'indice si elle en trouve un et lève l'exception `Not_found` si elle n'en trouve pas.

7. Écrire une fonction `inverse_contenu (m : float array array) -> unit` qui prend en argument une matrice de flottants (représentée par le tableau de ses lignes, donc un tableau de tableaux de flottants) et la modifie de sortie à remplacer tous les $m_{i,j}$ par $\frac{1}{m_{i,j}}$.

Elle lèvera l'exception `Invalid_argument` si une des valeurs de la matrice est nulle. [3]

8. Écrire une fonction `matrix_search_position : 'a array array -> 'a -> int*int` qui prend en argument une matrice, un élément, et renvoie un indice (i, j) où se trouve cet élément dans la matrice. S'il est absent, vous renverrez `Not_found`. [4]

Vous n'utiliserez que des boucles `for`, des `try with`, des exceptions, et des références (et des `if then else`, bien sûr). Pas de récursion ou de fonction intelligente du module `Array` (`Array.length` est autorisé, évidemment).

3. C'est l'exception qu'on lève généralement en OCaml lorsque l'on se rend compte que l'argument d'une fonction ne respecte pas les préconditions.
4. Là encore, `Not_found` est une exception usuelle en OCaml.

Dans ce dernier exemple, la redirection exceptionnelle de flot aide à créer un code concis et lisible. Nous verrons d'autres exemples similaires avec le retour sur trace.

Notez qu'il ne faut pas abuser de cette utilisation des exceptions : pour que le code soit clair, son flot doit être le plus clair possible. Les redirections exceptionnelles doivent donc rester exceptionnelles. [5]

Terminons en mentionnant que la fonction `failwith` que l'on a déjà utilisée revient en fait à lever l'exception `Failure of string`. Autrement dit, `failwith "blabla"` est équivalent à `raise (Failure "blabla")` . [6]

C Lecture et écriture dans un fichier

C.1 Permissions

Les dossiers et fichiers linux peuvent être modifiés [7], mais pas par n'importe qui. On parle des **permissions**. Dans un terminal, vous pouvez afficher les permissions des fichiers avec la commande `ls -l` . Les permissions sont indiquées sous la forme d'une suite de lettres. Par exemple :

```
-rwxr--r--
```

Déchiffrons cela ensemble. Il y a 4 informations ici : la première lettre, puis 3 blocs de trois lettres :

```
-   rwx   r--   r--
```

La première lettre indique s'il s'agit d'un fichier (`-`) ou d'un dossier (`d`).

Ensuite, chaque groupe de trois lettres indique dans l'ordre : est-ce qu'on a le droit de lire (`r`) ou non (`-`), le droit de modifier (`w`) ou non (`-`), et le droit d'exécuter (`x`) ou non (`-`).

Le premier bloc de 3 lettres indique les permissions pour l'utilisateur propriétaire du fichier. Le second pour les autres utilisateurs qui font partie du même groupe que le proprio : on peut par exemple définir un groupe d'utilisateurs « élèves », et tous les fichiers marqués comme exécutables par les élèves seraient exécutable par n'importe quel membre du groupe élève. Enfin, le dernier indique les permissions pour les autres utilisateurs. [8]

À retenir : les permissions sont une suite de lettres. `-` signifie une absence, une lettre une autorisation. La première lettre indique si c'est un dossier ou non, et ensuite on a les permissions de l'utilisateur propriétaire, puis de son groupe, puis des autres.

C.2 Lire et écrire un fichier depuis un programme OCaml

Un des objectifs du programme de MP2I/MPI est de savoir lire et écrire dans un fichier, en C et en OCaml (après rappel des fonctions à utiliser). Voyons aujourd'hui comment faire en OCaml.

Définition 1

Pour lire/écrire dans un fichier, comme pour lire/écrire dans un terminal, on manipule des canaux (*channel* en Anglais) sur lesquels on "envoie" ou "récupère" notre texte.
Le canal de sortie qui permet d'écrire sur le terminal est prédéfini et se nomme `stdout` (ainsi que `stderr` pour les erreurs). Les fonctions d'affichage usuelles (comme `printf` en C ou `print_string` en OCaml) utilisent le canal `stdout` par défaut, sans qu'on ait besoin de le préciser.
De même, la lecture passe par des canaux d'entrée d'où on "extrait" du texte. Pour le terminal, le canal prédéfini se nomme `stdin` .

Pour lire et écrire dans un fichier, on commence donc par accéder aux canaux de lecture et d'écriture dans ce fichier. On demande à notre programme d'ouvrir ces canaux. Pour cela, on utilise les fonctions `open_in` et `open_out` respectivement.

Ensuite, on lit ou écrit notre texte dans ces canaux via les commandes suivantes :

- `input_line : in_channel -> string` lit une ligne provenant du canal d'entrée donné en argument. Elle lit jusqu'à trouver un caractère de retour à la ligne et renvoi ce qu'elle a lu, à l'exception du retour à la ligne. Lève l'exception `End_of_file` si la fin du fichier est atteinte dès le début de la ligne (sans avoir rien lu).

5. C'est l'argument principal contre les `goto` : ils rendent le code dur à lire car on saute tout le temps d'un endroit à un autre.
6. Ca ressemble à un détail, mais c'est un détail au programme et qui est tombé à Mines 2 MPI cette année.
7. Quelle surprise.
8. Notez qu'il existe un utilisateur particulier, `root` (l'administrateur) qui n'entre dans aucune de ces catégories : c'est *sa* machine, il y a tous les droits partout : il ne sert donc à rien d'afficher ses droits.

- `output_string : out_channel -> string -> unit` écrit la chaîne de caractère donnée dans le canal de sortie donné en argument.

Attention, lorsqu'on écrit des chaînes de caractères dans un canal, les caractères s'accumulent dans le canal (dans ce qu'on appelle un *buffer*) sans être immédiatement écrits sur la destination. Lorsque le canal est suffisamment plein, il est vidé d'un seul coup dans le fichier/terminal destination. Ainsi, en cas d'erreur ou d'interruption prématurée, rien n'apparaitra sur le terminal/fichier même si votre commande `output_string` ou `print_string` a bien été exécutée. De même, l'affichage peut arriver plus tard que ce à quoi vous vous attendez. Pour provoquer l'écriture/affichage immédiatement, il faut *flush* le canal, avec la fonction `flush : out_channel -> unit` .

⚠**ATTENTION**, il ne faut jamais oublier de fermer un canal que vous avez ouvert ! En particulier parce que votre OS ne peut ouvrir qu'un nombre limite de fichiers simultanément, mais aussi parce que la fermeture du fichier va automatiquement *flush* le canal. Les fonctions permettant de fermer un canal sont `close_in` et `close_out` respectivement.

Exemples basiques :

- Voici un exemple de fonction qui écrit un message dans un fichier :

OCaml : comp.ml
```
let ecrit fichier message =
  let oc = open_out fichier in (* oc pour out_channel *)
  output_string oc message;
  close_out oc
```

Exemple d'utilisation : `let () = ecrit "texte.txt" "hello world !"` .

Remarque : Si le fichier donné en argument n'existe pas, il est créé lors du processus d'écriture.

⚠**Remarque :** on écrit SUR le fichier donné. S'il contenant déjà du texte, celui-ci sera écrasé par votre écriture (on peut par contre écrire plusieurs lignes dans le fichier à la suite, l'emplacement d'écriture se déplace au fur et à mesure qu'on écrit). On peut déplacer nous même la tête d'écriture pour éviter d'effacer un texte préalable, mais ceci est hors programme.

- Voici une fonction qui lit la première ligne d'un fichier donné en argument ligne par ligne (et affiche ces lignes dans le terminal).

OCaml : comp.ml
```
let lit_premiere_ligne fichier =
  let ic = open_in fichier in (* ic pour in_channel*)
  let line = input_line ic in
  print_string line; print_newline ();
  flush stdout; (*optionnel*)
  close_in ic
```

Exemple d'utilisation : `let () = lit_premiere_ligne "texte.txt"` .

- Voici une fonction qui lit intégralement un fichier donné en argument ligne par ligne (et affiche ces lignes dans le terminal). On lit le fichier jusqu'à atteindre l'exception `End_of_file` .

OCaml : comp.ml
```
let lit_fichier file =
  let ic = open_in file in
  try
    while true do
      let line = input_line ic in
      print_string line; print_newline ()
    done
  with End_of_file ->
    close_in ic
```

Exemple d'utilisation : `let () = lit_fichier "texte.txt"` .

En réalité, pour manipuler nos fichier proprement, il faudrait s'assurer de fermer les fichiers même si une erreur est rencontrée, mais ceci s'étend au delà du programme de MP2I/MPI. Ce n'est pourtant pas bien compliqué. Si vous voulez en savoir plus, je vous encourage à consulter a minima la documentation d'OCaml : https://ocaml.org/docs/file-manipulation , ainsi que la documentation du module `Stdlib` qui contient toutes les fonctions OCaml usuelles disponibles (vous pourriez y faire des trouvailles utiles).

9. Vérifiez que vous avez bien tout compris en lisant via OCaml le contenu du fichier ”texte.txt” et en affichant son contenu dans le terminal.
10. Via OCaml, créez un fichier ”cible.txt” qui contient le texte de votre choix (si possible sur plusieurs lignes).

D Pour occuper les plus rapides

11. En OCaml, jour 9 de l’advent of code 2023.