Exceptions

1 Principe des exceptions

Comme beaucoup de langages, OCaml dispose d'un mécanisme pour gérer les erreurs (dans un sens très large) appelé *exceptions*. Le principe de base des exceptions est le suivant :

— une exception est générée lors de l'évaluation d'une certaine expression ;
— cette exception peut être rattrapée (c'est-à-dire gérée) par la fonction dans laquelle elle s'est produite (bloc `try ... with`) ;
— si ce n'est pas le cas, l'exécution de la fonction se termine et l'exception est transmise à la fonction appelante, qui peut à son tour la rattraper (toujours par un bloc `try ... with`) ;
— on continue ainsi à remonter la pile d'appels ; si l'on arrive au bout de la pile sans que personne n'ait géré l'exception, l'exécution du programme se termine et l'exception est signalée à l'utilisateur.

Remarques

⇨ « Beaucoup de langages », mais pas le C. Il existe un mécanisme en C qui permet de construire quelque chose d'essentiellement équivalent (sans syntaxe particulière en revanche) : `setjmp`/`longjmp`. C'est complètement (mais alors vraiment complètement) hors-programme.

⇨ Quand on dit que « l'exécution de la fonction se termine », cela signifie qu'on sort complètement du flot de contrôle normal : tous les blocs d'activation qui sont traversés sans que l'exception ne soit gérée sont dépilés.

Supposons par exemple que l'on souhaite écrire une fonction `somme_inverses : int array -> int` renvoyant la somme des inverses des éléments d'un tableau. Logiquement, on procéderait ainsi :

```
let somme_inverses t =
  let s = ref 0 in
  for i = 0 to Array.length t - 1 do
    s := !s + 1 / t.(i)
  done;
  !s
```

Évidemment, si l'un des éléments est nul, une exception sera levée :

```
utop[54]> somme_inverses [| 12; 0; 1; 3 |];;
Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Si l'on décide arbitrairement de renvoyer `max_int` dans le cas où l'un au moins des éléments est nul, on peut utiliser un bloc `try ... with` :

```
let somme_inverses t =
  let s = ref 0 in
  try
    for i = 0 to Array.length t - 1 do
      s := !s + 1 / t.(i)
    done;
    !s
  with
  | Division_by_zero -> max_int
```

Dans ce cas, dès que l'un des éléments vaut `0`, l'exception `Division_by_zero` est levée ligne 5 et l'on saute à la ligne 9 : comme l'exception rattrapée est bien celle qui a été levée, on renvoie le `max_int` situé à droite de la flèche.

```
utop[57]> somme_inverses [| 12; 0; 1; 3 |];;
- : int = 4611686018427387903
```

La partie `with` du bloc `try ... with` fait un filtrage par motif : dans l'exemple ci-dessus, le seul motif accepté est `Division_by_zero`. Si l'exception que l'on essaie de rattraper ne correspond pas au motif (c'est-à-dire, ici, si ce n'est pas *exactement* `Division_by_zero`), elle n'est pas rattrapée et elle est donc transmise à la fonction appelante (comme s'il n'y avait pas eu de `try...with`). Par exemple :

```
let somme_inverses t =
  let s = ref 0 in
  try
    for i = 0 to Array.length t do  (* Ligne modifiée *)
      s := !s + 1 / t.(i)
    done;
    !s
  with
  | Division_by_zero -> max_int
```

```
utop[45]> somme_inverses [| 2; 7; 1 |];;
Exception: Invalid_argument "index out of bounds".
```

1. Que va-t-il se passer si l'on appelle la fonction `somme_inverses` (modifiée) sur le tableau `[| 2; 0; 1 |]` ?
2. Vérifier en exécutant le code.

Il est tout à fait possible d'avoir plusieurs motifs dans le `with`. Par exemple, dans le code ci-dessous, la fonction renvoie :
- la valeur finale de `!s` si aucune exception n'est levée (ce qui n'arrivera jamais vu l'erreur dans les bornes de la boucle) ;
- `max_int` si la première exception levée dans le bloc `try` est `Division_by_zero` ;
- deux fois la valeur de `!s!` (au moment où l'exception est levée) si la première exception levée est `Invalid_argument x` pour un certain `x`.

Si jamais la première exception levée n'est pas l'une des deux qui sont rattrapées dans le `with`, elle est transmise à la fonction appelante.

```
let somme_inverses t =
  let s = ref 0 in
  try
    for i = 0 to Array.length t do  (* Ligne modifiée *)
      s := !s + 1 / t.(i)
    done;
    !s
  with
  | Division_by_zero -> max_int
  | Invalid_argument message -> 2 * !s
```

Pour déclencher (on dit *lever*) volontairement une exception, on utilise la fonction `raise`, appliquée à l'exception souhaitée :

```
let mini t =
  let n = Array.length t in
  if n = 0 then raise (Invalid_argument "tableau vide");
  let m = ref t.(0) in
  for i = 1 to n - 1 do
    m := min !m t.(i)
  done;
  !m
```

```
utop[52]> mini [| |];;
Exception: Invalid_argument "tableau vide".
```

Remarque

⇨ Nous avons déjà levé volontairement des exceptions, par deux moyens :
- `failwith message`, qui équivaut à `raise (Failure message)` ;

— `assert(condition)`, qui lève l'exception `Assert_failure (f, l, c)` si `condition` s'évalue à `false` (`(f, l, c)` est un triplet qui indique le fichier, la ligne et le caractère où l'assertion a échoué).

2 Exemple d'utilisation : lecture d'un fichier

La manière la plus simple (Ce qui ne veut certainement pas dire la meilleure) de lire un fichier texte en OCaml est d'utiliser les fonctions suivantes :

— `open_in : string -> in_channel` qui prend en argument un nom de fichier (c'est-à-dire un chemin vers un fichier) et renvoie une valeur de type `in_channel` ;
— `input_line : in_channel -> string` qui lit des caractères dans le `in_channel` jusqu'à tomber sur un retour à la ligne, et renvoie la chaîne de caractères lue (sans le `"\n"`) ;
— `close_in : in_channel -> unit` pour fermer le fichier une fois qu'on a fini de l'utiliser.

Il y a deux remarques importantes :

— une valeur de type `in_channel` se comporte comme un « flux », c'est-à-dire que les lignes sont « consommées » au fur et à mesure qu'elles sont lues (si l'on appelle deux fois de suite `input_line` sur le même `in_channel`, on obtiendra une ligne puis la suivante) ;
— si on appelle `input_line` sur un `in_channel` « épuisé » (dans lequel il n'y a plus rien à lire), l'exception `End_of_file` est levée. C'est en fait le seul moyen de savoir (en utilisant ces fonctions) qu'on a terminé la lecture du fichier.

Exemple

⇨ Le programme suivant accepte un nombre quelconque d'arguments en ligne de commande, qui doivent tous être des noms de fichier (qui doivent exister). Il les lit ligne par ligne, les uns après les autres, et affiche leur contenu sur la sortie standard.

```
let print_file filename =
  let f = open_in filename in
  let rec loop () =
    try
      let s = input_line f in
      print_endline s;
      loop ()
    with
    | End_of_file -> () in
  loop ();
  close_in f
```

```
let () =
  let n = Array.length Sys.argv in
  (* On commence à 1 puisque
   * Sys.argv.(0) n'est pas un
   * "vrai" argument. *)
  for i = 1 to n - 1 do
    print_file Sys.argv.(i)
  done
```

On pourrait penser qu'un tel programme ne sert à rien, mais en réalité c'est que fait la commande `cat` (Elle est un peu plus versatile, mais c'est la manière la plus courante de l'utiliser), qui est d'usage très courant. Typiquement, on redirigerait la sortie : `cat part1 part2 part3 > file`.

1. Écrire une fonction qui accepte un nom de fichier en argument et renvoie le nombre de lignes de ce fichier.
2. Écrire un programme qui accepte des noms de fichier (en nombre quelconque) comme arguments et affiche :
   — une ligne pour chaque fichier, avec le nombre de lignes du fichier puis le nom du fichier ;
   — une dernière ligne pour le total.

```
$ ./lc.out cat.ml lc.ml lecture1.ml
18 cat.ml
22 lc.ml
29 lecture1.ml
69 total
```

Remarques

⇨ C'est ce que fait la commande `wc` (*word count*) appelée avec l'option `--lines`.

⇨ Si l'on utilise une *wildcard* comme `*`, le *shell* fait l'expansion avant de passer les arguments au programme. Ça a l'air compliqué, mais ça veut dire que si l'on appelle `./lc.out *.ml`, c'est comme si on avait passé *tous les fichiers du répertoire dont le nom se termine par* `.ml` *comme argument.*

```
$ wc --lines ../*.tex
 178 ../exceptions.tex
  18 ../td-exceptions.tex
 196 total
$ ./lc.out ../*.tex
178 ../exceptions.tex
18 ../td-exceptions.tex
196 total
```

⇨ Les plus attentifs auront remarqué que la sortie de `wc --lines` est plus joliment formatée dans l'exemple ci-dessus. S'ils devaient dans les jours prochains se trouver confrontés à une violente crise d'ennui, ils pourraient éventuellement chercher à régler ce problème, en commençant sans doute par chercher comment fixer la largeur d'un champ dans `Printf.printf`.

3 Utilisation dans le flot de contrôle

Comme nous l'avons vu à la partie précédente, les exceptions ne servent pas *que* à gérer les « erreurs » : quand on appelle `input_line`, on s'attend bien à déclencher l'exception `End_of_file` à un moment ou un autre. On peut en fait aller plus loin et utiliser les exceptions pour manipuler le *flot de contrôle* (le chemin suivi par l'exécution).

Remarque

⇨ OCaml a été conçu pour que les exceptions soient rapides (à lever et à récupérer), ce qui rend raisonnables (et même relativement idiomatiques) les techniques présentées dans cette partie. Dans la plupart des langages, ce serait catastrophique (mais ces langages proposent généralement des mots-clés tels que `break` et `return`, et les techniques présentées y sont donc bien moins utiles).

Supposons qu'on souhaite tester la présence d'un élément `x` dans un tableau `t` (non trié). Pour l'instant, nous avons vu trois manières de procéder :

— faire une boucle `for` qui parcourt systématiquement le tableau jusqu'au bout :

```ocaml
let appartient_for x t =
  let trouve = ref false in
  for i = 0 to Array.length t - 1 do
    if t.(i) = x then trouve := true
  done;
  !trouve
```

— faire une boucle `while` pour s'arrêter dès qu'on a trouvé l'élément (deux variantes) :

```ocaml
let appartient_while_1 x t =
  let n = Array.length t in
  let trouve = ref false in
  let i = ref 0 in
  while !i < n && not !trouve do
    if t.(!i) = x then trouve := true;
    i := !i + 1;
  done;
  !trouve
```

```ocaml
let appartient_while_2 x t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) <> x do
    i := !i + 1
  done;
  !i < n
```

— utiliser une fonction auxiliaire récursive pour s'arrêter dès qu'on a trouvé l'élément :

```
let appartient_rec x t =
  let rec loop i =
    if i = Array.length t then false
    else t.(i) = x || loop (i + 1) in
  loop 0
```

En utilisant une exception, il y a une possibilité supplémentaire : emballer une boucle `for` dans un bloc `try...with` et lever une exception dès qu'on trouve l'élément. Dans ce cas, il est **fortement recommandé** de définir une nouvelle exception :

```
exception Trouve

let appartient_exn x t =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x then raise Trouve
    done;
    false
  with
  | Trouve -> true
```

Remarque

⇨ Ici, je ne recommande pas du tout la version qui utilise une exception, inutilement lourde. On préférera :
- la version avec boucle `for` si parcourir la liste jusqu'au bout ne pose pas de problème ;
- une version avec `while` ou la version récursive sinon.

Une exception peut être paramétrée par un objet d'un certain type : c'est par exemple le cas de `Failure` et de `Invalid_argument`, qui sont toutes deux paramétrées par une chaîne de caractères. Pour définir une nouvelle exception paramétrée par un entier, par exemple, on écrirait :

```
exception Mon_exception of int
```

On souhaite écrire une fonction `premiere_occ : 'a -> 'a array -> int option` telle que l'appel `premiere_occ x t` renvoie :
- `Some i` si `i` est la première occurrence de `x` dans `t` ;
- `None` si `x` n'apparaît pas dans `t`.

Écrire trois versions de cette fonction :

1. l'une utilisant une boucle `while` ;
2. l'une utilisant une fonction auxiliaire récursive ;
3. l'une utilisant une boucle `for` et un bloc `try...with`.

À nouveau, on préférera l'une des deux premières solutions ici.

Un exemple où ça a un intérêt, pour changer... On considère une matrice M (représentée sous la forme d'un `m : int array array`) et un entier x. On souhaite trouver la première occurrence de x dans M, où « première » est à comprendre dans l'ordre lexicographique sur les couples d'indices :

$$(i, j) \preceq (i', j') \iff \begin{cases} i < i' \\ \quad \text{ou} \\ i = i' \quad \text{et} \quad j \leqslant j' \end{cases}$$

Autrement dit, il s'agit de trouver l'occurrence le plus en « haut » de la matrice, et en cas d'égalité la plus à « gauche ». Pour pouvoir traiter le cas où x n'apparaît pas dans la matrice, on veut écrire une fonction ayant le type suivant :

```
cherche_matrice : int -> int array array -> (int * int) option
```

Sachant que l'on souhaite arrêter la recherche dès que possible, écrire deux versions de cette fonction :

1. l'une à base de boucles `while` ;
2. l'une utilisant des boucles `for` et une exception.

4 Rien à voir

Des affiches rectangulaires sont collées les unes après les autres sur un mur. Toutes les affiches ont la même largeur. On vous donne la hauteur Hi de chacune de ces affiches, dans l'ordre dans lequel elles sont collées. Le coin supérieur gauche de chaque affiche est toujours placé exactement sur le coin supérieur gauche du mur, et ce dernier est toujours plus grand que les affiches. Régulièrement, entre deux collages d'affiches, on vous demande d'indiquer combien d'affiches, parmi celles déjà collées, sont au moins en partie visibles, c'est à dire qu'une surface non nulle n'a été recouverte par aucune affiche collée depuis.

Limites de temps et de mémoire

Temps 2s sur une machine à 1 GHz.

Mémoire 32 Mo.

Garanties

— $1 \leqslant$ `nbRequetes` $\leqslant 100000$;

— $1 \leqslant H_i \leqslant 1000000$ où les H_i sont les hauteurs des affiches.

Entrée

— La première ligne de l'entrée est constituée d'un unique entier : `nbRequetes`.

— Chacune es `nbRequetes` lignes suivantes commence par un caractère pouvant être `'Q'` ou `'C'`. Un caractère `'Q'` correspond à la question : « Combien d'affiches sont actuellement visibles ? ». Un caractère `'C'` est suivi d'un entier `hauteur` sur la même ligne, séparé par un espace, et correspond au collage d'une affiche de hauteur `hauteur`.

Sortie La sortie de votre programme doit correspondre aux réponses aux questions posées en entrée, à raison d'une réponse par ligne, dans l'ordre d'apparition des questions.

Entrée

```
12
C 2
Q
C 4
C 2
Q
C 9
Q
C 9
C 2
Q
C 8
Q
```

Sortie

```
1
2
1
2
2
```

Entrée

```
10
Q
C 8
C 7
C 11
Q
C 2
C 4
C 3
Q
C 3
```

Sortie

```
0
1
3
```

1. En considérant les limites de ressources et les garanties sur les entrées données, en en notant $n =$ `nbRequetes` :
   (a) une complexité en $\Theta(n^2)$ est-elle satisfaisante ?
   (b) même question pour $\Theta(n \log n)$ et $\Theta(n)$;
   (c) peut-on stocker les hauteurs de toutes les affiches ?
2. Essayer de réfléchir à un algorithme efficace pour résoudre ce problème. Ce n'est pas évident [1], donc il ne faut pas hésiter à me demander une indication.
3. Implémenter votre solution en C, ou en OCaml, ou, mieux, dans les deux langages. Vous aurez sans doute besoin d'aide pour les entrées-sorties en OCaml, n'hésitez pas à me demander.

1. En plus d'être un problème de niveau 3 de France-IOI, c'est le début d'un oral d'informatique de Ulm. . .