TP 3 : Évaluation et Scoping

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercice I : Macros & évaluation paresseuse (Lisp)

Les macros de Lisp permettent de ré-écrire [1] du code sans l'évaluer, ou en l'évaluant partiellement. Cela peut s'avérer très pratique, par exemple, si on veut créer une fonction qui affichera quelque chose avant l'évaluation de ses arguments, cela est impossible car Lisp a un ordre d'évaluation strict. Cependant, avec les macros, c'est possible :

```
(defmacro surround(arg)
  `(progn
     (format t "before~%")
     ,arg
     (format t "after~%")))

(surround (format t "hello␣world~%"))
```

Les macros sont des fonctions, dont l'évaluation s'effectue à la compilation ! De plus, l'utilisation du caractère *backtick* (`) spécifie à l'interprète que l'on ne veut pas évaluer son contenu. Ce code est donc équivalent à :

```
(progn (format t "before~%") (format t "hello␣world~%") (format t "after~%"))
```

Notons qu'il faut échapper l'argument `arg` de la macro en écrivant `,arg` sans quoi le code produit serait :

```
(progn (format t "before~%") arg (format t "after~%"))
```

Ce qui produirait évidemment une erreur étant donné que la variable `arg` n'a jamais été déclarée.

Q1 – Définissez des macros `orOp` et `andOp` pour les opérations logiques || et &&. On fera attention à retourner la valeur d'une des deux opérandes et non pas `t`.

Q2 – Vérifiez que vous avez à présent le même comportement en Haskell et en lisp sur les appels :

— `orOp True (error "failure")`

1. https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_macro_expansion

— `orOp (error "failure") True`

Q3 – Vérifiez de plus que `(orOp (princ "a") (princ "b"))` n'affiche ni "ab", ni "aa", mais bel et bien "a". Verifiez aussi que `(orOp (funcall (lambda () nil)) (princ "b"))` affiche bien "b"

Exercice II : Listes infinies

Il est possible en Haskell de définir des listes infinies à l'aide de la syntaxe `[1..]` qui produit la liste des entiers strictement positifs. Il est aussi possible de spécifier un pas d'itération en faisant `[1,3..]` qui produit la listes des entiers positifs impairs.

Q1 – Construisez la liste infinie des nombres impairs plus grands ou égaux à 3. Affichez les 10 premiers nombres de cette liste en utilisant la fonction `take: Int → [a] → [a]`.

Q2 – Il est aussi possible de construire des listes infinies récursives, par exemple `ints=1:[x+1 | x <- ints]` est une autre façon de produire la liste des entiers strictement positifs. Définissez la liste (récursive) des puissances de 2.

Q3 – En utilisant la fonction `takeWhile: (a → Bool) → [a] → [a]`, écrivez une fonction `smallerRoot` qui prend un nombre n et une liste croissante l (potentiellement infinie) et qui récupère les entiers de l plus petits ou égaux à $\sqrt{n}$.

Q4 – Écrivez une fonction `notDiv: Int → [Int] → Bool` qui prend un entier n et une liste d'entiers l et qui renvoie vrai si et seulement si n n'est divisible par aucun nombre dans l. Vous pourrez utiliser la fonction `all:(a → Bool) → [a] → Bool`

Q5 – Haskell permet définir des listes par compréhension en faisant `[x | x <- [1,3..], x `mod` 3 /= 0]` qui produit la liste des entiers impairs non divisibles par 3. En utilisant cette syntaxe, définissez la liste des nombres premiers. On peut la construire récursivement en disant qu'elle commence par 2, et est suivie des nombres impairs i qui ne sont divisibles par aucun nombre premier plus petit ou égal à $\sqrt{i}$

Exercice III : Switch

Le *switch* est une instruction qui permet d'effectuer un branchement en comparant la valeur d'une expression, qu'on appellera la cible, avec différents cas, traités comme des blocs exclusifs séparés. Cela fonctionne comme une conditionnelle généralisée avec n'importe quel nombre de branches, pas seulement deux. Dans le cas ou aucun cas ne correspond à la cible, une expression à retourner par défaut peut être spécifiée.

Ordre d'évaluation

Un switch évalue d'abord la cible, et, selon le résultat obtenu, exécute les instructions du cas correspondant. Un cas est un couple d'expression (e_1, e_2) où e_1 est une expression à comparer avec la cible, et e_2 l'expression à retourner lorsque l'évaluation de la cible correspond à e_1. Les différents cas sont évalués de haut en bas, tant qu'aucun cas ne correspond à la cible. Il est important de

comprendre que le switch, comme la conditionnelle, doit avoir un comportement paresseux et ne doit évaluer les différents cas et leurs instructions associées que lorsque c'est nécessaire.

```
switch (print "1:"; 42) {
  case (print "2"; 0): print "you␣cant␣see␣me"
  case (print "3"; 40+2): print "4";
  default: print "you␣cant␣see␣me"
}
```

Par exemple, le code ci-dessus produit l'affichage "1234".

Q1 – Proposez une implémentation d'une fonction `switch` en Haskell en utilisant des listes pour représenter les différents cas. On rappelle qu'en Haskell, on peut construire un couple d'expression avec la syntaxe `(a,b)`

Q2 – Vérifiez sur un exemple que vous avez le bon ordre d'évaluation.

En Haskell l'ordre d'évaluation non strict [2] permet à une fonction de s'évaluer avant que tout ses arguments soient évalués. Cela permet d'éviter des erreurs lorsque l'évaluation de certains arguments en produit. Pour les langages avec ordre d'évaluation strict, un moyen classique pour délayer l'évaluation d'une expression est de l'encapsuler dans une fonction anonyme ne prenant pas d'argument. Par exemple le code `(lambda () (print ''foo''))` ne produira pas d'affichage tant qu'il n'est pas appliqué, tandis que `(funcall (lambda () (print ''foo'')))` affichera bel et bien la chaine "foo". Ces expressions encapsulées s'appellent des *thunk*[3], ou suspensions.

Nous allons à présent écrire deux versions du switch en lisp. Une façon de faire est de délayer l'évaluation des arguments du switch en encapsulant les expressions des différentes branches du switch dans des *thunks*.

Q3 – Proposez une fonction `fswitch` qui prend en paramètre une cible, une liste de cas représentés par des *thunks* et un cas par défaut aussi représenté par un *thunk* qui évalue les expressions sous-jacentes en suivant l'ordre d'évaluation spécifié précédemment.

Remarques :

— on pensera à utiliser `equalp` plutôt que `=` pour avoir une comparaison polymorphe.

— on pourra utiliser des listes de deux éléments pour représenter des couples.

Q4 – Vérifiez que le code suivant affiche bien : "case 3"

```
(fswitch
  3
  (list
    (list (lambda() 1) (lambda () (princ "case␣1")))
    (list (lambda() 2) (lambda () (princ "case␣2")))
    (list (lambda() 3) (lambda () (princ "case␣3"))))
  (lambda () (princ "default␣case")))
```

2. https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_need
3. https://en.wikipedia.org/wiki/Thunk

Q5 – Proposez une seconde version du switch en lisp qui utilise l'opérateur ’ (quote) plutôt que des *thunks* pour délayer l'évaluation des arguments, et qui force l'évaluation à l'aide de la fonction `eval`.

Q6 – Vérifiez que le code suivant affiche bien : "default case"

```
(switch
  4
  '((1 (princ "case␣1"))
    (2 (princ "case␣2"))
    (3 (princ "case␣3")))
  '(princ "default␣case"))
 (terpri))
```