TP 3 : Evaluation and Scoping (Lisp)

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercise I : Macros & lazy evaluation (Lisp)

Lisp's macros allow code to be rewritten [1] without evaluating it, or partially evaluating it. This can be very handy, for example, if we want to create a function that will display something before evaluating its arguments, this is impossible because Lisp has a strict order of evaluation. However, with macros it is possible :

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

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

Macros are functions, the evaluation of which is made at compile-time ! Also, using the character *backtick* (`) specifies to the interpreter that we do not want to evaluate its content. This code is therefore equivalent to :

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

Note that it is necessary to escape the argument `arg` of the macro by writing `,arg` otherwise the code produced would be :

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

Which would obviously produce an error since the variable `arg` has never been declared.

Q1 – Define `orOp` and `andOp` macros for logical operations || and &&. We will be careful to return the value of one of the two operands and not `t`.

Q2 – Verify that you now have the same behavior in Haskell and in lisp on the calls :

— `True || (error "failure")`

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

— `(error "failure") || True`

Q3 – Verify that `(orOp (princ "a") (princ "b"))` displays neither "ab" nor "aa", but rather "a". Check also that `(orOp (funcall (lambda () nil)) (princ "b"))` displays "b"

Exercise II : Infinite lists

It is possible in Haskell to define infinite lists using the syntax `[1..]` which produces the list of integers strictly positive. It is also possible to specify a step iteration by doing `[1,3..]` which produces the list of odd positive integers.

Q1 – Build the infinite list of odd numbers greater or equal to 3. Display the first 10 numbers in this list by using the `take: Int → [a] → [a]` function.

Q2 – It is also possible to define infinite lists recursively, for example `ints=1:[x+1 | x <- ints]` is a another way to produce the list of strictly positive integers. Define the (recursive) list of powers of 2.

Q3 – Using the function `takeWhile: (a → Bool) → [a] → [a]`, write a `smallerRoot` function that takes a number n and an increasing list l (potentially infinite) and which retrieves integers from l smaller or equal to $\sqrt{n}$.

Q4 – Ŵrite a function `notDiv: Int → [Int] → Bool` which takes an integer n and a list of integers l and returns true if and only if n is not divisible by any number in l. You can use the function `all:(a → Bool) → [a] → Bool`

Q5 – Haskell allows you to define lists by comprehension by doing `[x | x <- [1,3..], x `mod` 3 /= 0]` which produces the list of odd integers not divisible by 3. Using this syntax, define the list of prime numbers. We can build it recursively by saying that the list of prime numbers begins with 2, followed by the odd numbers i which are not divisible by any prime number smaller or equal to $\sqrt{i}$

Exercise III : Switch

The *switch* statement performs a branching by comparing the value of an expression, called the target, with different cases treated as separated exclusive blocks. It works like a generalized conditional with any number of branches, not just two. In the event that no case corresponds to the target, an expression to return by default can be specified.

Order of evaluation

A switch first evaluates the target, and depending on the result obtained, executes the instructions of the corresponding case. A case is a couple of expression (e_1, e_2) where e_1 is an expression to be compared with the target, and e_2 the expression to return when evaluating the target is e_1. The different cases are evaluated from top to bottom, as long as no cases match the target. It's important to understand that the switch, like the conditional, must have a lazy behavior and only evaluate the

different cases and their associated instructions only when necessary.

```
switch (print "1:"; 42) {
  case (print "2"; 0): print "you␣can't␣see␣me"
  box (print "3"; 40+2): print "4";
  default: print "you␣can't␣see␣me"
}
```

For example, the code above produces the output "1234".

Q1 – Propose an implementation of a `switch` function by Haskell using lists to represent the different case. We recall that in Haskell, we can construct a couple expression with the syntax `(a,b)`

Q2 – Check on an example that you have the correct evaluation order.

In Haskell the non-strict evaluation order [2] allows a function to evaluate its body before all of its arguments are evaluated. This helps to avoid errors when the evaluation of some arguments should fail. For languages with strict evaluation order, a classic way to delay the evaluation of an expression is to encapsulate it in an anonymous function that takes no argument. For example the code `(lambda () (print ``foo''))` will not produce a display until it is applied, while `(funcall (lambda () (print ``foo'')))` will indeed output the string "foo". These encapsulated expressions are called *thunks* [3], or suspensions.

We will now write two versions of the switch in lisp. A way to do this is to delay the evaluation of the switch arguments by encapsulating the expressions of the different branches of the switch in *thunks*.

Q3 – Propose a function `fswitch` which takes as a parameter a target, a list of cases represented by *thunks* and a case by default also represented by a *thunk* which evaluates the underlying expressions following the specified order of evaluation previously.

Remarks :

— we will consider using `equalp` rather than `=` to have a polymorphic comparison.

— we can use lists of two elements to represent pairs.

Q4 – Verify that the following code displays : "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")))
```

To avoid the syntactic heaviness due to the use of lambdas, we can use macros together with thunks.

Q5 – Propose a second version of the switch in lisp which uses the `'` (quote) operator rather than *thunks* to delay the evaluation of the arguments, and which forces the evaluation using the `eval` function.

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

Q6 – Check that the following code displays : "default case"

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