Practical Session 4: Scheme (Functional programming)

1 Functional Programming

Scheme follows the functional programming paradigm. To put it simply, functional programming is built around the concept of *functions*, viewed as mathematical objects: a function takes some inputs and returns an output, with no other side-effects. Functional programming avoids mutation of data: an argument passed to a function is never modified itself, it's a new object which is returned as the result of applying a function to an argument. In functional programming, a program is viewed as a series of functions applied to input data.

2 First-class functions

Like many other functional programming languages, Scheme has first-class functions. This means that functions can be treated like any other entities and the program, and more specifically, a function can be passed as argument to, can be returned from a function and can stored in a variable.

For instance, Scheme has a built-in `filter` function. This function takes two arguments: the first is a predicate p and the second is a list l. It returns the list l, in which all the elements for which p is false have been removed. Consider this example:

```
(define mylist '(1 2 -3 4 5 -6 7 8 -9))
(filter positive? mylist)

 -> '(1 2 4 5 7 8)
```

`positive?` is of course a built-in Scheme predicate which returns `#t` if its argument is a positive number.

In the previous session, you wrote a `sum-cubes` function which may have looked like the following:

```
(define (sum-cubes a b)
   (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

Rewrite a more generic `sum-func` which takes three arguments f, a and b and which returns the sum of f applied to all integers in $\{a, a+1, ...b\}$. Uses this function to perform a `sum-cubes`, for instance:

```
(sum-func cube 0 3)
-> 36
```

Finally, define `sum-cubes` in terms of this new `sum-func`.

2.1 Anonymous functions

Sometimes it can be useful to write a small function as argument to another directly in the function call, without `define`'ing it entirely. This is where `lambda`'s become handy. For instance, to use filter with a predicate that filter all items equal to 5:

```
(filter (lambda (x) (not (equal? x 5))) mylist)
-> '(1 2 -3 4 -6 7 8 -9)
```

The `lambda` expression creates an anonymous function which, to x associates the boolean $x! = 5$.

In the previous session, you defined a `how-many` function. A function providing a more generic version of this is actually built into Scheme: `count`. Try to implement an equivalent `how-many` function using `count` and an anonymous function.

Many functions built into Scheme take a functional parameter. We have already seen `filter`, and others are `apply`, `map`, `foldr`, `foldl`.

1. Try to guess / understand what each does based on the following examples:

```
(define mylist '(1 2 -3 4 5 -6 7 8 -9))

(map (lambda (x) (+ x 1)) mylist)
(apply + mylist)
(apply max mylist)
(foldl + 10 mylist)
```

2. Re-implement `sum-cubes` in terms of `map` and `apply`. You may also find the `range` builtin useful.

3. Write a function which takes a list of lists and returns a list containing the maximum of each list.

```
(define listoflists '((1 2 3 1) (45 1 3 4 5) (4 5 64)
        (4 6) (144) (0 4 4) (14 464 4 7 6)))

(max-list listoflists)
-> '(3 45 64 6 144 4 464)
```

2.2 Higher-order functions

As said above, it is possible for a function to return a function (think for example about what the `lambda` builtin returns).

Write a function `mean` which takes as argument a numeric function $f : \mathbb{R} \to \mathbb{R}$ and returns the function:

$$f' : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \qquad x, y \mapsto \frac{f(x) + f(y)}{2}$$

Use this function with the `cube` function to generate a `mean-cube` function. Then compute the mean-cube of 4 and 16.

```
(mean-cube 4 16)
-> 2080
```