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.

Write a (filter p l) 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 keeps all items smaller than 5:

```
(filter (lambda (x) (< x 5)) mylist)
-> (1 2 -3 4 -6 -9)
```

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

2.2 Map and apply

Some functions defined in the Scheme standard take a functional parameter: for instance **apply** or **map**. They are important functional programming tools which are also found in many modern languages (sometimes under a different name)

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)
```

2. Write a (range a b) utility function which produces a list of all integers from a to b, included. Then use this function to re-implement sum-cubes in terms of **map** and **apply**.

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)
```

Another common operation we might want to do on list is to *accumulate* the result of a function over all the arguments of this list. In many languages, this function is called fold. However, the Scheme standard does not define this function.

Write an accumulate function. This function will have the following interface :

```
(accumulate init-value function list)
```

And its value is the application of the two-parameter function successively on all elements of **list**, with the first parameter being the accumulated value so far, starting with init−value. In other words, (accumulate x0 f '(l1 l2 ... ln)) should return

$$f(l_n, f(l_{n-1}, \ldots \quad f(l_2, f(l_1, x_0)))$$

For instance :

```
(accumulate 1 * '(2 3 4))
> 24

(accumulate 14 + '(2 3 4))
> 23
```

Take care of the parameter order ($f(a, b)$ may not be equal to $f(b, a)$! Respect the order given above).
Can you use this function to reverse a list?
Why can you not easily use this function to implement your collatz function of the previous weeks?

2.3 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
```

3 Special forms and parameter evaluation

From the syntax, it looks like if in Scheme is just another function of three arguments: (if condition true−expr false−expr). However, we will see that there is a slight caveat to this approach and that, under the hood, things are a bit different.

Implement a (my−if condition true−expr false−expr) function in terms of **cond**. Try it with the following expressions:

```
(define x 4)
(my-if (positive? x) x (- x))

(my-if (positive? x)
       (display "positive")
       (display "negative"))
```

What goes on in your my−if function is that the interpreter first evaluates *all* the parameters passed to the the function. In the first example, this does not matter, because x and (− x) simply evaluate to 4 and -4, without any side-effects. But in the second example, both arguments have the side-effect of displaying some text on the output, and their evaluation produces #<void> (this is an implementation choice in racket. The standard simply specifies that display, returns "an unspecified value"). In other words, both the true and false blocks are evaluated upon parameter passing, and the function call actually becomes (my−if #t #<void> #<void>) and the display have already printed out their arguments.

3.1 Hygienic macros

Because of this, some constructs in Scheme are actually *special forms* and not functions. In a special form, the arguments are only evaluated when called within the special form, and not when passed as arguments. It is possible to define your own special forms using define−syntax and syntax−rules. You can refer to the R5RS[1] for a reference, or some tutorials available online[2] for help on how to use these constrcuts.

Rewrite my−if using a hygienic macro. Do you observe the problem discovered above?

[1] `http://www.schemers.org/Documents/Standards/R5RS/`
[2] See for instance `http://www.willdonnelly.net/blog/scheme-syntax-rules/`