Practical Session 5: Scheme

1 Parameter lists

You may have noticed that some Scheme functions are able to take an arbitrary number of arguments. For instance:

```
(* 1 2 3 4 5 6 7 8 9 10 1.132)
(+ 9 8 7 4)
(map * '(1 2 3) '(4 5 6) '(7 8 9) '(10 11 12))
```

This is a very useful syntax which can be found in modern languages as well: the star operator in Ruby or Python for instance. The way in which a function taking an arbitrary number of arguments can be defined in Scheme is through the dot operator: the parameter name following the dot operator at the end of a function parameter list will be bound to a list of all the arguments passed to the function when it is called. The `apply` function which we have already encountered before is the counterpart of the dot operator: it takes a list, unpacks it, and applies the function to all arguments of the list.

Try to implement a `my-plus` function which sums up all its arguments. Use the dot operator:

```
(define (my-plus . mylist)
        ... )
```

You may want to re-use the `accumulate` function you wrote in the previous session.

2 Scoping

So far, all the functions that we have written were relatively small. This is common in functional programming. However, as our programs become larger we may sometimes have the need to introduce local variables to store the result of temporary calculations. We thus need a way of declaring variables temporarily in some locally defined scope. This is the purpose of the `let` function:

```
(let ((x 10) (y 5)) (* x y))
```

The expression above creates a temporary scope containing a binding to the variables `x` and `y`, and then evaluates `(* x y)` with these bindings.

Try to evaluate the following expressions:

```
(let ((x 10) (y x)) (* x y))
(let* ((x 10) (y x)) (* x y))
```

Can you understand the difference between these two forms? Can you write an expression equivalent to the second expression, using only `(let ((x 10)) ...)` and `(let ((y x)) ...)`?

3 Lexical scoping

When trying to resolve a reference, an interpreter has two different possibilities: use lexical scoping or dynamic scoping. In lexical scoping, the resolution of an identifier depends on where it was defined in the source code, whereas in dynamic scoping its depends on where it is used at runtime.

Consider the following piece of code :

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y))
(f 5)
```

Can you try to guess the two possible values of `(f 5)` in the case of a statistically and dynamically scoped interpreter? If you run this in the racket R5RS interpreter, can you tell what kind of scoping Scheme uses?

When running `(f 5)` above, what happens is this: the interpreter must resolve `f`. It is defined as `(g 2)`. It must now resolve `(g 2)`, which is `(+ x y)` and `y` is obviously 2 but the value of `x` is unknown in the direct scope of `g`. With dynamic scoping, the interpreter will look for a binding of `x` in the function calling `g`. In this case, it is `f` where `x` is equal to 5. But in static scoping, it will not look in the calling function but in the direct outer block of `g`, which here is the global scope, where `x` has been defined as 1.

It is very important to understand the difference and to know which approach the language uses, especially when doing functional programming: if a function is passed as argument, you should know how the variables are going to be bound if they are not completely local to the function if you want to be able to predict the behaviour of the function.

What about

```
(let ((a 5))
        (let ((fun (lambda (x) (max a x))))
                (let ((a 10) (x 20))
                        (fun 1))))
```

What is the effective binding of `x` and `a` in `fun`? (In case of doubt, add `x` statements in the body of the `lambda` expression to print the values of x and a.)

4 Mutability

Modern functional languages emphasize the immutability of data: once a data structure is created, it can no longer be modified but rather new structures must be created, usually by applying a transformation on your existing structure.

Think about `map` for instance. In Scheme, what happens to `mylist` after evaluating these two expressions?

```
(define mylist (list 1 2 4 2 57 9 .1 3 12 -75))
(map (lambda (x) (/ (sqrt x) x)) mylist)
```

But Scheme's list structures are actually mutable. You may have noticed that when you tell racket to use the R5RS standard, it will display lists as a series of `mcons` constructs. This is because the internal implementation of the Scheme standard in the Racket language prevents mutable lists by default, and lists built with `cons` in the Racket language cannot be modified in memory later on. However, it provides special functions to use, when the user specifically wants to have mutability: `mcons, mcar, mcdr`. And because accordin to the R5RS standard pure Scheme (unlike Racket) has mutable lists by default, the Scheme `cons` actually corresponds to racket's `mcons`, which is why there is some confusion.

We will now play around with mutable lists in Scheme (don't forget to use `#lang r5rs`).

```
(define mutable (list 1 2 4 2 57 9 .1 3 12 -75))

(set-car! mutable 20)
(set-car! (cdr mutable) 10)

(set-cdr! (cdr mutable) '(just changing my cdr))
```

What is the value of `mutable` after each of the above expressions?
Now evaluate:

```
(set-cdr! mutable mutable)
```

What would the value of this last expression? What would happen if you evaluated `mutable`? What is the `car`, `cadr`, `caddr` and `cadddr` of `mutable`?
Now evaluate:

```
(define immutable '(this list wont change))
(set-cdr! mutable immutable)
(set-car! (cdddr mutable) 'can)
```

What is the value of `immutable` after each of the above expressions? What happened?

Write an `append! x y` function which appends list `y` to `x`, changing the actual structure of `x` in memory (it is a good Scheme practice to suffix functions which mutate some or all of their inputs with a `!`).

Consider the following `mystery` function. Can you guess what it does without evaluating it?

```
(define (mystery x)
   (define (loop x y)
      (if (null? x)
         y
         (let ((temp (cdr x)))
            (set-cdr! x y)
            (loop temp x))))
   (loop x '()))
```

Assume we have two lists:

```
(define v '(a b c d))
(define w (mystery v))
```

What does `w` look like? What has `v` become? Why?