Practical Session 5: Scheme

1 Parameter lists

You may have noticed that some Scheme functions are able to take an arbitrary number of arguments. See 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 syntactic sugar which can be found in modern languages as well: the star operator in Ruby or Python for instance. The way this is done in Scheme is with the dot operator: this operator builds a list from all the arguments that follow it and passes this list to the function. The `apply` function that we have already seen is its counterpart: 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 wrote were relatively small. This is common in functional programming. However, some functions may be larger, and require temporary variables for instance. We need to have a way to declare variables locally, inside functions. This is the purpose of the `let` function:

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

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 its location in the source code, whereas in dynamic scoping its depends on its use at runtime. Perhaps more simply, this means that to resolve a statically scoped binding, the interpreter will look in the current block, and failing to that in the outer block, and then the outer block, etc. In dynamic scoping, the interpreter will look in the scope of the function, then the calling function etc.

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 interpreters? If you run this in the racket R5RS interpreter, can you tell chat 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` in `fun`?

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)
```

The main advantage (among others) of immutable data structures is that they are inherently thread-safe. They avoid many problems and make it easier to understand the code, keep it consistent even when exceptions occur, and make it easy to parallelize.

But Scheme actually provides mutable data structures. 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 of the internal implementation of the Scheme standard in racket: 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`. But because in Scheme lists are mutable by default, the Scheme `cons` actually corresponds to racket's `mcons`, which is why there is some confusion.

We will play around with mutable lists in Scheme.

```
        (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))

        (set-cdr! mutable mutable)
```

What is the value of this last expression? What would happen if you called (don't) `(apply + mutable)` for instance?

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

What is the value of `immutable` here? What happened?