Practical Session 3: Scheme (List processing)

1 Getting started

Download and install Racket[1]. Racket is a general-purpose implementation of Scheme. It provides a command-line interpreter as well as a GUI called DrRacket.

finish instructions

2 The basics

Scheme uses prefix notation. This means that applying a function f to two arguments x and y is done with the following:

```
(f x y)
```

For instance, to compute $2 * (14 + 17)$, you will write:

```
(* 2 (+ 14 17))
```

Write and evaluate the following expression in Scheme:

$$\frac{(134 + 121) * \frac{74}{\sqrt{8}}}{97.1 - \frac{exp(7)}{78*71}}$$

3 Lists

Scheme is a dialect of LISP, which stands for *List Processor* and lists are at the core of LISP languages. They are the main data structure and the language is designed to manipulate them in many ways, and use them to define more complex data structures.

The basic building blocks of Scheme are *atoms*. They can be numbers, symbols (`'a`, `'mysymbol`), characters (`#\a`, `#\Z`), booleans (`#t` and `#f` for true and false),...

Atoms can be combined together with the `cons` function to form a *pair*. This function takes two argument and returns a pair, in the which the first item is the first argument and the second item is the second argument. The Scheme interpreter will write a pair as:

[1] `https://racket-lang.org/`

```
(a . b)
```

Given a pair, obtaining the first and second elements of the pair can be done with the infamous `car` and `cdr` functions.

```
(define mypair (cons 'a  'b))
(car mypair)        ; -> Returns the symbol 'a
(cdr mypair)        ; -> Returns the symbol 'b
```

A *list* is a pair, in which the second element (the `cdr`) is a list or the constant `null` (also written `'()`). For instance, this defines a list :

```
(define mylist (cons 'a  null))
(list? mylist)                  ; -> returns #t, true
```

You can see that a list has a recursive definition : a list is a pair where the second element is a list. The empty list `'()` gives us a base case from which to build lists. Racket provides facilities to build lists straightaway more easily than using multiple `cons`. For example, both of these lines define valid lists:

```
'(22 21 23 #f 45 0 'a  "lists can have heterogeneous elements")

(list 1 2 3 4 "A cat is fine too")
```

Try to guess the value of the following expressions. Then verify your answer in racket.

```
(cons 1 (cons 2 (cons 3 (cons 4 '() ))))
(car (cons 1 (cons 2 '())))
(car (cdr (list 1 2 3 4)))

(cadr '(1 2 3))
(caddr '(1 2 3))
(cdddr '(1 2 3))

(cons (cons 'a 'b) (cons 1 '() ))
(list (list 1 2 3) 4 5 6)
```

What is the difference between these two objects?

```
(cons 'a (cons 'b 'c))
(cons 'a (cons 'b (cons 'c '() )))
```