Practical Session 3: Scheme (List processing)

1 Getting started

Many LISP dialects have been implemented throughout history. For the purpose of this course, we will use one of the best-known of these dialects: Scheme. Scheme is actually a language by itself, which is described in a standard called the *Revised ^{n} Report on the Algorithmic Language Scheme (RnRS)*, where n is the version number of this document. Although the document is still being revised periodically to this day (with R7RS having been published in 2013), the most commonly implemented version of this standard is still R5RS[1].

One general-purpose implementation of (this standard of) Scheme is available in Racket[2]. Racket is actually the name of a LISP dialect itself, but the Racket programming environment supports the use of both the Racket, R5RS and R6RS Scheme dialects, as well as some other variants. The Racket environment provides both a command-line interpreter as well as a GUI called DrRacket. In the GUI, you can write Scheme scripts or programs in the top panel. The bottom panel is an interpreter in which you can write expressions directly, or see the result of running the script or program typed in the top panel.

Every source file in DrRacket should start with an expression of the form #lang language where language tells the interpreter which dialect of Scheme you want to use. As mentioned above, for our practical sessions we will mostly use #lang r5rs. If you prefer to use your own text editor rather than the one provided by DrRacket, you can run your Scheme programs by executing the command `racket -r myscript.scm`.

2 The basics

Scheme uses prefix notation. This means that applying a function f to two arguments x and y is written as follows:

```
( f  x  y )
```

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

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

This may require a bit of getting used to, but you'll soon get the hang of this, don't worry. After all, it's only syntax.

[1] `http://www.schemers.org/Documents/Standards/R5RS/`
[2] `https://racket-lang.org/`

Write and evaluate the following expression in Scheme. (Hint: the Scheme function to calculate the square root is **sqrt**.)

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

Even control-flow statements use this prefix notation. For instance, an `if` condition would be written as:

```
(if test-expr true-expr false-expr)

(if (positive? a) "a is positive" "a is negative")
```

Try to re-write the following Pascal piece of code in Scheme:

```
a := -8;

if a > 0 then b := 2 * sqrt(a)
else b := 7 * sqrt(-a) / -a;

writeln(b)
```

To get you started, you can start from the following code skeleton:

```
(define a -8)
(define b ... )
(display b)
```

3 Lists

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

The basic building blocks of Scheme are *atoms*. They can be numbers (1, 1.3, 1/3, 1e3, ...), 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 constructor function takes two argument and returns a pair, of which the first item is the first argument and the second item is the second argument. After evaluation, the Scheme interpreter will write a pair (**cons** 'a 'b) as:

```
(a . b)
```

Once a pair has been constructed, retrieving its first and second elements again can be done through the infamous car and cdr functions. (The names of

car and cdr are historical since Lisp was originally implemented on the IBM 704 computer in the late 1950s, where car was used to retrieve the contents address of a register and cdr was used to retrieve the contents decrement register. As a mnemonic device to remember which one retrieves the first value and which one retrieves the second, just remember that the letter "a" precedes the letter "d" in the alphabet.)

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

A *list* is a special kind of pair (created with **cons**), in which the second element (the cdr) is either another list or the empty list '(). For instance, the code below defines a list with one element:

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

The empty list **null** or '() is also considered as a list:

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

In the definition above you can already see that a list has a recursive definition: a list is a pair where the second element is again a list. The empty list '() gives us a base case from which to build lists. Scheme provides facilities to build lists straightaway more easily than using multiple **cons**. For example, each of the lines below define valid lists:

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

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

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

Warning: Because the Scheme language uses mutable lists but Racket uses immutable lists (we will talk about list mutability in a following session), when using the R5RS standard, Racket will often show lists like this:

```
(list 1 2 3)
-> (mcons 1 (mcons 2 (mcons 3 '())))
```

This is because racket uses mcons instead of **cons** to construct mutable lists. To avoid this problem, always use the display function to print out lists.

Try to guess the value of the following expressions. Then verify your answer in Racket, using display to print your results.

```
(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)

(caddr '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
(caaddr '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
(cadar '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
```

What is the difference between these two objects?

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

4 Functions

Scheme adheres to the functional programming paradigm. In the next session, we will see what this means exactly and how functions in Scheme can even be passed around and manipulated as first-class items.

For now, we will just explore the basic syntax to define functions in Scheme and how many problems in Scheme can be defined elegantly in a recursive way. First look at the following trivial example of a function definition, then try to implement some of the more complex functions listed below.

```
(define (cube x) (* x x x))
```

4.1 Recursion on numbers

Write a function sum−cubes of two integer arguments a and b ($a < b$) which computes the sum of cubes of the integers in the interval $\{a, a+1, ..., b\}$. For instance, (sum−cubes 0 3) should return 36 ($0^3 + 1^3 + 2^3 + 3^3 = 36$).
Hint: $sumCubes(a, b) = a^3 + sumCubes(a+1, b)$

```
(define (sum-cubes a b)
  ...
)
```

Write a recursive function collatz? which checks the Collatz conjecture for a given integer n. Print all the steps of the check along the way (use writeln for instance). Remember from the previous practical sessions that the Collatz conjecture states that the following process always converges to 1:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Hint: You may find the built-in Scheme functions **cond**, even? and odd? useful. As a side-exercise, could you define versions of the even? and odd? predicates yourself?

4.2 Tail recursion

The Scheme language implements tail-recursion. To put it simply, tail-recursion occurs whenever a recursive function call is located at the end of the function: in this situation, the control does not need to go back from the called function to the caller, because there is nothing left to execute in the caller. Because of this, such recursive calls can actually be compiled as a loop under the hood, avoiding some of the performance issues that recursive functions typically suffer from (filling up the call stack). See the Racket doc[3] and the R5RS section 3.5 for more details on tail recursion.

Is the sum−cubes function you implemented above tail-recursive? Why (not)? If not, can you make it tail-recursive?

Hint: Accumulate the result along the way in an additional variable.

Compare the performance difference between a non tail-recursive sum−cubes and the tail-recursive implementation sum−cubes−tail with:

```
(time (sum-cubes 0 999999))
(time (sum-cubes-tail 0 999999))
```

Remark: The **time** function is not a Scheme primitive, but a Racket extension, so you will need to change your dialect to #lang racket for this to run. Notice how Scheme handles big numbers naturally through symbolic representation.

4.3 Recursion on lists

Using the car and cdr primitives, you can access the head and tail of a list. With this mechanism available, it is easy to recursively scan a list. As always with recursion, you will first need to define a base case: with lists, this is often

[3] https://docs.racket-lang.org/guide/Lists__Iteration__and_Recursion.html#%28part._tail-recursion%29

the case of the empty list '(). Then, figure out how what should happen when recursing from this base case to the case of a list with one element.

Write a function how−many which takes a first argument x and a second list argument l, and returns the occurrence count of x in l. For example:

```
(how−many 4 '(1 4 5 2 3 4 a b 4 4 c 4))
```

should return 5, because there are 5 occurrences of 4 in this list.

Is the how−many function you implemented above tail-recursive? If not, can you make it tail-recursive?

Write a function duplicate which takes a list and returns the same list where every element has been duplicated. For example:

```
(duplicate '(Do duck quacks echo?))
```

should return '(Do Do duck duck quacks quacks echo? echo?)

Re-write your collatz? function to generate a list of the successive values computed by the process.

```
(collatz−list 13)
−> '(13 40 20 10 5 16 8 4 2 1)
```