CvP - Programming Assignment 1

- Deadline: October 26, at the start of the werkcollege.
- Submit your solution electronically via liacscvp2018@gmail.com
- Clearly state your name and student number your solution file.
- You can only use pure functional constructs. You are allowed to use only the built-in Scheme functions used in Chapter 15 of the book.
- We prefer you use Racket as a Scheme interpreter:
  - the Racket Lang environment http://racket-lang.org/ (which is available in the lab machines).

  If this doesn't run on your machine, you can use a Scheme interpreter of your choice. For example:
  - the MIT one https://www.gnu.org/software/mit-scheme/

  Refer to the respective documentations for instructions on how to use them.
- If you are using racket, don't forget to change the line on top of your code from `#lang racket` to `#lang scheme`.
- If you are using racket, please hand in the .rkt file, if you are not using racket, please put your functions inside a .txt file and hand in that file.

Question 1 Define and test a Scheme function `drop` that drops all occurrences of an element from a list. The expected behaviour is shown below.

```
> (drop 'a '(a b b c))
(a c)
```

Question 2 Define and test a Scheme function `intersection` that takes two lists and returns their intersection. If an element appears more than once in one list or the other, it should appear in the output list only once. The expected behaviour is shown below.

```
> (intersection '(1) '(1))
(1)
> (intersection '(1 2) '(1))
(1)
> (intersection '(2 1 2) '((1) 2))
(2)
```

Question 4 Define and test a Scheme functions `even` and `odd` that takes a list and returns the list of elements at the respective even and odd positions in the list. The expected behaviour is shown below.

```
> (even '(1 2 3 4 5 6))
(2 4 6)
> (odd '(1 2 3 4 5 6))
(1 3 5)
```

Question 5 Define and test a Scheme function `zip` that takes two lists and pairs up corresponding elements of the two lists. If one of the lists is longer than the other, the extra elements are ignored. The expected behaviour is shown below.

```
> (zip (odd '(1 2 3 4 5 6)) (even '(1 2 3 4 5 6)))
(1 2 3 4 5 6)
> (zip '(1 2 3) '(a b c))
(1 a 2 b 3 c)
> (zip '(1 2 3) '(a))
(1 a 2)
```

Question 6 Define and test a Scheme function `flatten` that takes a list and returns a list of all its primitive elements (numbers, atoms, etc.) and the ones of its sub lists. The expected behaviour is shown below.

```
> (flatten '())
()
> (flatten '(1))
(1)
> (flatten '((1 2) 3))
(1 2 3)
```

Question 7 Define and test a Scheme function `rac` that returns last element of a list. The expected behaviour is shown below.

```
> (rac '(1 2 3))
3
```

Question 8 Define and test a Scheme function `rdc` that drops the last element of a list. The expected behaviour is shown below.

```
> (rdc '(1 2 3))
(1 2)
```

Question 9 Consider we want to implement a compiler for a simple language. A large part of this is to implement its denotational semantics. This question is about implementing the look-up table, which is part of most imperative programming languages. It is a list that contains pairs of a variable name, and its value, e.g.,

```
((A 1) (B 12) (C 3))
```

NB: function arguments of letters need a `'` to become recognized, so enter `'A` to enter `A` as argument.

(a) Create a Scheme function `get` that returns the value of a given variable in a given table.

(b) Create a Scheme function `update_table` that takes the arguments: a look-up table `T`, a variable `v` to be edited, and a value `x` to be assigned. If variable `v` is in look-up table `T`, then `update_table` returns the original look-up table `T` with the value of `v` set to `x`. If variable `v` is not in `T`, then `update_table` returns the the original look-up table `T` with the pair `(v, x)` appended to the end.

Question 10 Write a Scheme function `langrec` that recognizes sentences that can be created using the following grammar with start symbol $\langle A \rangle$:

$$\langle A \rangle \rightarrow a\langle B \rangle c \mid ac$$
$$\langle B \rangle \rightarrow b\langle B \rangle \mid b \mid \langle D \rangle$$
$$\langle D \rangle \rightarrow d\langle D \rangle \mid d \mid \langle A \rangle$$

The sentences should be given to the function in the form of a list of terminals. If a sentence belongs to the language generated by this grammar the function should return true, if not it should return false. The expected behaviour is shown below.

```
> (langrec '(a c))
#t
> (langrec '(b))
#f
> (langrec '(a c a))
#f
> (langrec '(a b c))
#t
```

```
> (langrec '(a d c))
#t
> (langrec '(a a c c))
#t
> (langrec '(a a a c c c))
#t
> (langrec '(a a c c c))
#f
> (langrec '(a a a c c))
#f
> (langrec '(a b c d))
#f
> (langrec '(a b d c))
#t
> (langrec '(a d b c))
#f
> (langrec '(a b d d d d d d d d d d c))
#t
> (langrec '(a b d d d d d b d d d d c))
#f
```

Hint: Define three functions `(A list)`, `(B list)`, and `(D list)` that decide whether a list of symbols can be parsed from the respective start symbols $\langle A \rangle$, $\langle B \rangle$, and $\langle D \rangle$.