Labwork 1

February 2021

The purpose of this labwork is to practice defining recursive functions for some simple examples.

Recursive function definitions

A function is *recursive* if it calls itself in the body. A typical example is the factorial function

$$fac : \mathbb{N} \to \mathbb{N}, \quad fac(n) := \begin{cases} 1 & \text{if } n = 0, \\ n \cdot fac(n-1) & \text{if } n > 0. \end{cases}$$

Typically, a recursive function f is defined by cases:

- One or more *base cases*: these are the "simple" cases when the value of the f can be computed directly, without any need to perform a recursive call.
- One or more *recursive cases*: these are the cases when the value of f is computed as follows:
    1. we compute the value of f for some simpler input arguments
    2. next, we combine the previously computed values to obtain the value we want.

For example, the definition of $fac(n)$ consists of

- one base case, for $n = 0$. In this case we simply return 1 as result.
- one recursive case: when $n > 0$. In this case we perform the recursive call $fac(n-1)$ to compute the value of fac for the smaller input $n-1$, and then combine the value of $fac(n-1)$ with the value of n to obtain the value of $fac(n)$.

In Racket, the recursive definition of `fac` is

```
(define fac (lambda (n) (if (= n 0) 1 (* n (fac (- n 1))))))
```

This definition can also be written in the simplified form

```
(define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
```

The following exercises are intended to train you in identifying suitable recursive definitions (base cases+recursive cases) for some simple functions.

Problem 1

Write down a recursive definition for the function `(l-ref l i)` which returns the $\mathtt{i}+1$-th element of a list `l`. If `l` has less than $\mathtt{i}+1$ elements, the function call should return `#f`.

Answer: We know that a list is either empty (that is, `null`) or a list of the form `(cons h t)` consisting of a first element `h` and a shorter list `t` (the *tail* of the list). Note that, if the list `(cons h t)` has n elements, then the list `t` has $n-1$ elements.

Thus, we distinguish two cases:

1. `l` is empty. In this case, `l` has no i+1-th element, therefore `(l-ref l i)` should return `#f`
2. `l` has a first element `h`, followed by the sublist of elements `t`. We distinguish two sub-cases:
   (a) $\mathtt{i}=0$. In this case, the $\mathtt{i}+1$-th element of `l` is the first element of `l`, which is `(car l)`.
   (b) $\mathtt{i}>0$. In this case, the $\mathtt{i}+1$-th element of `l` is the `i`-th element of `t`, which is `(cdr l)`.

The Racket encoding of this recursive definition is straightforward:

```
(define (l-ref l i)
  (if (null? l) #f
      (if (eqv? i 0)
          (car l)                    ; base case
          (l-ref (cdr l) (- i 1))    ; recursive case
      )))
```

Note: In Racket source code, the symbol '`;`' indicates the start of a comment, which continues till the end of the line.

Problem 2

Define a recurisve function `(len l)` which computes the length of a list `l`.

Answer: We distinguish two cases:

1. `l` is the empty list `null`. In this case, the length is 0.
2. `l` consists of a head element `h` followed by a sublist `t`. In this case, the length of `l` is longer than `t` by 1.

In Racket, this definition looks as follows:

```
(define (len l)
  (if (null? l) 0 (+ 1 (len (cdr l)))))
```

Problem 3

Write down a recursive definition for the function `(app l1 l2)` which computes the result of concatenating lists `l1` and `l2`. For example:

```
> (app '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)
> (app '() '(a b c))
'(a b c)
 > (app '(a b c) '())
'(a b c)
```

Answer: We can reason by induction on the structure of list `l1`:

1. If `l1` is `null`, then `(app l1 l2)` should return `l2`.
2. If `l1` is not `null` then we can reason as follows:
   - First, we append `(cdr l1)` with `l2`. Let's assume the result is `l3`.
   - `l3` coincides with the tail of the list `(app l1 l2)`. To obtain the list `(app l1 l2)`, we must add `(car l1)` in front of list `l3`. Thus, `(app l1 l2)` coincides with `(cons (car l1) (app (cdr l1) l2))`

In Racket, this recursive definition is encoded as follows:

```
(define (app l1 l2)
  (if (null? l1) l2 (cons (car l1) (app (cdr l1) l2))))
```

Problem 4

Write down a recursive definition of the function `(rev l)` which computes the list obtained by reversing list `l`. For example:

```
> (rev null)
'()
> (rev '(1 2 3 4))
'(4 3 2 1)
```

Suggestion: Note, that in order to compute `(rev (l)`, we can proceed as follows: We call the auxiliary function `(rev-aux l null)`, where `(rev-aux l1 l2)` behaves as follows:

1. 1. As long as `l1` is not `null`, remove `(car l1)` from `l1` and add it as first element of `l2`.

2. As soon as l1 is null, return l2.

For example, to reverse '(1 2 3 4), we call:

```
   (rev-aux '(1 2 3 4) '())
→ (rev-aux '(2 3 4) '(1))
→ (rev-aux '(3 4) '(2 1))
→ (rev-aux '(4) '(3 2 1))
→ (rev-aux '() '(4 3 2 1))
→ '(4 3 2 1)
```

The final result is the reversed version of '(1 2 3 4).
In Racket, the recursive definition of rev-aux is encoded as follows:

```
(define (rev-aux l1 l2)
  (if (null? l1)
      l2
      (rev-aux (cdr l1) (cons (car l1) l2))))
```

and rev is defined by

```
(define (rev l) (rev-aux l null))
```

Homeworks

HW1 A list is *good* if it is either empty, or it is of the form

(list s_1 n_1 ... s_m n_m)

where $s_1, \ldots, s_m$ are symbols, and $n_1, \ldots, n_m$ are numbers. Define recursively a predicate (good-list? l) which returns #t if l is a good list, and #f otherwise. For example:

```
> (good-list? null)      > (good-list? '(a 1 b 2 c 3/4))
#t                       #t
> (good-list? "abc")     > (good-list? '(1 a b))
#f                       #f
```

Remember that list? recognises lists, null? recognises the empty list, symbol? recognises symbols, and number? recognises numbers.

HW2 Define recursively a function (symb-value l s) which takes as input a *good* list l and a symbol s which occurs in l, and returns the number that appears immediately after s in l.

For example

```
> (symb-value '(x 2 y 3 z 4 t 5) 'z)
4
> (symb-value '(a 3.14) 'a)
3.14
```

Other recommended exercises:

HW3 Define recursively the predicate (`mem l v`) which returns `#t` if `v` is an element of the list `l`, and `#f` otherwise. Use the predicate `equal?` to check if two elements are equal.

For example:

```
> (mem? '(1 a b a c) 'a)          > (mem? '(1 a b a c) 'd)
#t                                 #f
> (mem? '(1 (2 3) 4) '(2 3))      > (mem? '() '())
#t                                 #f
```

HW4 Define recursively a function (`add l`) which takes as input a list of numbers, and computes the sum of its elements. If `l` is `null`, the function should return `0`.

HW5 Define recursively a function (`mult l`) which takes as input a list of numbers, and computes the sum of its elements. If `l` is `null`, the function should return `1`.

HW6 A nested list of numbers is either the empty list, or a list whose elements are either numbers, or nested lists of numbers. Define recursively a predicate (`nlist? l`) which returns `#t` if `l` is a nested list of numbers, and `#f` otherwise. For example:

```
> (nlist? null)     > (nlist? '(((1) 2) 3.2 ((4))))
#t                  #t
> (nlist? 1)        > (nlist? '(4 ((-5) a)))
#f                  #f
```

HW7 Define recursively the following functions that take as input a nested list of numbers `l`:

(a) (`add-all l`) which computes the sum of all elements in list `l`. If `l` contains no number, this function should return 0.

(b) (`max-elem l`) which returns the maximum number that occurs in list `l`. If If `l` contains no number, this function should return `0`.

(c) (`max-depth l`) which returns the maximum number of nested parentheses in list `l`. For example:

```
> (max-depth '())                 > (max-depth '(1 2 3))
1                                 1
> (max-depth '((1 2) (3 ((0)))))  > (max-depth '(1 (())))
3                                 3
```

You can make use of the predefined function (`max` m n) which returns the maximum of numbers m and n.