Introduction to Racket

Racket[1] is a programming language that has evolved from Scheme and Lisp. It has both a statically-typed and a dynamically-typed version, but we will focus on the dynamically-typed version (i.e. there is no type-checking that occurs at "compile time"). There is extensive online documentation[2] that you should familiarize yourself with.

Atoms and S-expressions

At its core Racket is a very simple language. There are essentially two ways you can form expressions in the language: atoms and s-expressions.

An **atom** is a single quantity. They are also called simple values[3]. Typical examples would be:

- Numbers[4]. Racket has various kinds of numbers, both "exact" (fractions, integers, complex numbers) and "inexact" (floating point).
- Booleans[5] are denoted by `#t` and `#f`. Also in general every value but `#f` is treated as "falsy".
- Strings[6] are as usual surrounded by double quotes.
- Symbols[7] are a not typically seen kind of value. They start with a single quote, followed by alphanumerics. Symbols are not like strings. Think of them as fields/keys. Testing for symbol equality is quick, but there are not many other operations you can perform on symbols.
- There are other atoms that we will not look at.

Almost everything else in Racket is an **s-expression**, though that is not how they are referred to in the language. Such an expression consists of a list of other atoms or expressions, surrounded in parentheses. It is invariably the case that the expression is interpreted as a the first argument, a function or operator, applied to the remaining arguments. For instance addition of numbers would be denoted as `(+ 2 3 4)`.

Some people find an overabundance of parentheses disruptive; In fact every parenthesis is needed, and there are no ambiguities about the meaning of an expression. For instance we don't need anyone to disambiguate for us the meaning of `2 + 3 * 4` because

[1]https://racket-lang.org/
[2]https://docs.racket-lang.org/
[3]https://docs.racket-lang.org/guide/Simple_Values.html
[4]https://docs.racket-lang.org/guide/numbers.html
[5]https://docs.racket-lang.org/guide/booleans.html
[6]https://docs.racket-lang.org/guide/strings.html
[7]https://docs.racket-lang.org/guide/symbols.html

it would be written in Racket as (+ 2 (* 3 4)), as opposed to (* (+ 2 3) 4) if multiplication was meant to occur first. *Parsing a Racket program is trivial.*

It is imperative that you use vertical alignment as much as possible to indicate grouping. For instance the above expression would often be written as:

```
(+ 2
   (* 3 4))
```

where we have vertically aligned the 2 with the other terms to be added. Here's how a conditional might work:

```
(if (> 3 2)
    "The 'then' branch"
    "The 'else' branch")
```

Or here is a list of pairs, where the first element in each pair is a symbol. Pairs are formed using cons.

```
(list (cons 'a 2) (cons 'b 3) (cons 'c 1))
```

The working environment

We will be working in DrRacket, which offers a console next to a document editor. The document editor is typically used to write definitions and "load" those definitions in. The console is used for more interaction.

You will notice the first line in the document is #lang racket. This specifies that we should use the non-statically-typed Racket. There are many other "languages" that can be used in this environment. For example there is a language plai focusing on programming language interpretation. The first line in the file suggests what language is to be used to interpret the rest of the file.

While we are at it, comments in Racket start with a semicolon, and go to the end of the line. One often uses two semicolons for longer and more permanent comments. There are no block comments available.

As you type in the document editor you will see that sections are highlighted when you close a parenthesis, to help you know what you have closed.

Definitions

There are a number of different ways to create definitions. The most standard is the define macro. We will see more later.

The simplest form of define would be:

```
(define a 5)
```

From now on a stands for 5. We can also use this to define functions, using the anonymous function construct in Racket, called lambda instead of fun:

```
(define add3 (lambda (x) (+ x 3)))
(add3 5)
```

We would most often space the definition vertically, like so:

```
(define add3
  (lambda (x)
    (+ x 3)))
```

So let us examine the form of a lambda expression: The second item is a parenthesized list of the variable names, in this case only one such name x, and the third item is the body of the function. The first item is of course the keyword lambda.

Definitions done via define are visible throughout the file. In particular, they are automatically set for recursion. We can therefore do the following, for example, to add all numbers up to n:

```
(define addUp
  (lambda (n)
    (if (<= n 0)
        0
        (+ n (addUp (- n 1))))))
```

There is a "syntactic sugar" for the function definition, that would make the above look as follows:

```
(define (addUp n)
  (if (<= n 0)
      0
      (+ n (addUp (- n 1)))))
```

Lists

Lists are a crucial structure in Racket. They are formed essentially as a sequence of "pairs" terminating with null. There is no separate notion of pairs as in tuples. You can simply "pair" two values together via cons, and if you do this systematically where the second value in the pair is always a list you get lists. For example the following are equivalent ways of specifying a list:

```
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3)
```

There are three operations that we can perform on lists:

- null? tells us if the list is empty. For instance (null? (list 1 2 3)) will return false, while (null? null) will return true.
- car and cdr are used on any kind of pair to give us the first and second entry in the pair. For instance we have the following: (define l (list 1 2 3)) (car l) ;; returns 1 (cdr l) ;; returns

For example here is a function that adds up all elements in a list:

```
(define sum-list
  (lambda (xs)
    (if (null? xs)
        0
        (+ (car xs)
           (sum-list (cdr xs))))))
(sum-list l)         ;; should return 6
```

Practice problems

1. Write a function multUp that given a positive integer n multiplies all the numbers from 1 to n.
2. Write a function fib that computes the n-th Fibonacci number. The Fibonacci numbers start with the 0th and 1st number equaling 1, and every subsequent number is the sum of the two previous numbers. Do not worry about efficiency.
3. Write a function lookup that takes as input a list of symbol-number pairs and also a symbol and looks that symbol up in the list. If it finds it then it returns the corresponding number, otherwise it returns #f. You can compare two symbols using eq?. An example use would be that (lookup (list (cons 'a 2) (cons 'b 3)) 'b) should return 3.
4. Write a version of lookup that allows for the possibility that an entry in the list is not a pair at all. It should simply skip those entries. You can use pair? to test if something is a pair, and not to negate. An example use would be: (lookup (list (cons 'a 2) 45 (cons 'b 3)) 'b)
5. Read up on cond on the documentation[8] and use it to simplify the previous function.
6. Write a function max that given a list of numbers returns the largest number. If the list is empty then it should return (error "empty list").

[8]https://docs.racket-lang.org/guide/conditionals.html#%28part._cond%29