Lesson 0.4 Introduction to Racket

Preliminaries

The purpose of this lesson is to familiarize you with the basics of Racket (a dialect of Scheme). You will learn about

- Expressions
- Numbers, Booleans, and Arithmetic
- Functions
- Structs
- Images
- Lists

This is a long lesson: it covers the features of Racket that will be covered from now until Week 4.

In this lesson, we will often refer to "contracts" and "purpose statements". For the purpose of this lesson, you can treat these as informal notions. We will learn more about contracts and purpose statements in Lesson 1.4.

As you go through this lesson, you should follow along, trying all the examples in the DrRacket interaction window. Don't just try what's shown on the page-- try variations as well, and see what happens. (This is an example of what we mean when we say *active learning*.)

To do this, install DrRacket (if you have not done so already), and open it. The exact way in which you open DrRacket will depend on your OS. Then click "Choose Language..." either in the *Language* submenu in the menu bar or in the language selector in the bottom left corner of the DrRacket window and choose the "Beginning Student" language under "Teaching Languages / How to Design Programs".

1. Expressions

Maybe the most fundamental building block in Racket (and other languages) is an expression. An expression is a piece of syntax that *returns one result*. An expression can be as simple as a single number

```
5
```

or more complex as the calculation of the square root of a number

```
(sqrt 5)
```

(Don't be alarmed if you don't understand the above syntax just yet.)

You can try typing the above examples in the bottom half (a.k.a. Interaction Window) of DrRacket. There you will see a prompt > and next to it you can type an expression. Whenever you write an expression next to the prompt and press enter, DrRacket will "evaluate" the expression and print the "result" in the line underneath.

```
> 5
5

> (sqrt 5)
#i2.23606797749979
```

The `#i` prefix warns that the printed result is inexact. For example, it's a bad idea to compare inexact numbers for equality. If you really need to do this (say for an

automated test), use `check-=` (in rackunit), which takes a third argument representing the error tolerance.

In the next section we will see how we can use the interaction window of DrRacket as an advanced calculator.

2. Numbers, Booleans, and Arithmetic

As we mentioned above, some of the *primitive* expressions in Racket are numbers; positive and negative, integers and decimals

```
> 17
17

> -10
-10

> 3.1415
3.1415
```

Racket has lots of numbers besides these. For example, you can write any rational number as a primitive expression:

```
> 1/2
0.5

> 12/10
1.2

> 9/5
1.8
```

Racket also has a set of arithmetic *operators* that work with numbers.

```
; + : Number Number -> Number
; Adds two numbers and returns the result

; - : Number Number -> Number
; Subtracts two numbers and returns the result

; * : Number Number -> Number
; Multiplies two numbers and returns the result

; / : Number Number -> Number
; Divides two numbers and returns the result
```

`+`, `-`, `*`, `/` are the names of the operators.

`Number Number -> Number` is the *Contract* for each of them. It says that these operators take two numbers as arguments and return a number as a result. If you are going to use an operator correctly, you need to know its contract!

Now let's see how we can *apply* these operators to some *operands* (a.k.a. arguments) and start doing something interesting with our Racket calculator.

To apply an operator to some operands we need a pair of parentheses that enclose both the operator and the operands, in that order.

```
> (+ 3 5)
8
```

The operator is always the *first* thing in the parentheses, followed by the operands. Here `+` is the operator, `3` is the first operand, and `5` the second operand. The order of the operands is important:

```
> (- 13 7)
6

> (- 7 13)
-6
```

Whenever you see the "parentheses-notation" in Racket you should immediately recognize that it is the *application* of an operator to several operands, and you should be able to recognize that the first thing between the parentheses is the operator, the second thing is the first operand, the third is the second operand, etc.

So now we have a simple calculator where we can do one operation after the other. To calculate 3*2 + 5*3 we can type:

```
> (* 3 2)
6

> (* 5 3)
15

> (+ 6 15)
21
```

Remember that all of the above are expressions. Each one is being evaluated by DrRacket and a result is returned in its place. Having this in mind we can build more complex expressions, and make our calculator compute 3*2 + 5*3 writing just one big expression:

```
> (+ (* 3 2) (* 5 3))
21
```

In Racket (as in its predecessors, Scheme, and Lisp), there are no operator precedences to be memorized. Instead, the parentheses always tell you the order in which complex expressions are evaluated: parenthesized subexpressions are evaluated before the expressions that contain them. After all, you couldn't do it any other way.

If your expression is complicated, break it across multiple lines. When you do this, ALWAYS use indentation to make the expression more readable:

```
> (+ (+ (- 20 5)
        (+ 10 4))
     (* (- 100 93)
        (* 3.5
           (- 5 3))))
78
```

In practice, expressions this long should be split using functions. We'll see how to do that a little later.

We now know how to use Racket as a decent calculator to do arithmetic. But let's not stop there. Let's see how we can also do *logical calculations*.

Racket has some more primitive expressions, the set of *booleans*:

```
> true
true

> false
false
```

It also provides operators that can be applied on booleans

```
; and : Boolean Boolean -> Boolean
; Logical conjunction
```

```
; or : Boolean Boolean -> Boolean
; Logical disjunction (Inclusive)

; not : Boolean -> Boolean
; Logical negation


> (not false)
true

> (and true false)
false

> (or (and true
           (or true false))
      (or (not true)
          (not (and (not false)
                    true))))
true
```

There are also operators that connect the world of numbers and the world of booleans. These operators perform *tests* on numeric (or other) data and return a boolean value. These are called *predicates*.

```
; = : Number Number -> Boolean
; Tests two numbers for equality

; < : Number Number -> Boolean
; Tests if the first operand is less than the second

; > : Number Number -> Boolean
; Tests if the first operand is greater than the second

; <= : Number Number -> Boolean
; Tests if the first operand is less or equal than the second

; >= : Number Number -> Boolean
; Tests if the first operand is greater or equal than the second
```

So for example:

```
> (< 300.0001 300)
false

> (= (+ (* 3 50)
         (* 3 25))
      (* 3
         (+ 50 25)))
true
```

Q: What happens when you try this?

```
> (< 3 (< 2 1))
```

(Remember, that > is the prompt, not the "greater-than" predicate)

Be careful of the contracts of operators to avoid these *type errors*.

3. Conditionals

There are times that we need the value of an expression to change depending on some condition. Racket provides a construct to implement this kind of branching.

```
(cond
  [test-1 expr-1]
  [test-2 expr-2]
  [test-3 expr-3]
  ...
  [else expr-n])
```

The cond is a multi-branch conditional. Each clause has a test and an associated expression. The first test that succeeds triggers its associated expression, and the value of the associated expression is returned. The final else clause is chosen if no other test succeeded.

Here's an example:

```
(cond
  [(< x 0) -1]
  [(= x 0) 1]
  [(> x 0) 2])
```

This returns -1, 1, or 2, depending on the whether the value of the variable `x` is negative, zero, or positive. We haven't seen variables yet, but we will soon.

Here's another example:

```
> (cond
    [(< (sqrt 5) (/ 5 2)) true]
    [else false])
```

This example is UGLY CODE, for two reasons:

First, it's a two-way `cond`. Racket provides an `if` construct that should be used instead. So a better version might be:

```
(if
  (< (sqrt 5) (/ 5 2))
  true
  false)
```

Second, the value of this expression is always the same as the value of

```
(< (sqrt 5) (/ 5 2))
```

If you write code like this on your assignments, you WILL be penalized.

If you are ever tempted to write

```
(if condition true false)
```

just write

```
condition
```

instead. Of course, if the expression had been

```
(if (< (sqrt 5) (/ 5 2))
    27
    42)
```

then this issue would not have come up.

Ex 3.1: Write an expression whose value is the number of seconds in a leap year (a leap year has 366 days). Next, write 2 more expressions that have the same value.

Ex 3.2: Write an expression that returns `true` if the result of 100/3 is greater than the result of (100 + 3) / (3 + 3) and `false` otherwise.

4. Function Definitions

At this point our DrRacket calculator can do a great deal of things. It's almost like a scientific calculator, but we are not there just yet. It would be nice if we were able to define our own operators on numbers and booleans and extend the functionality of our calculator. Racket has a special syntax for doing just that:

```
(define (fcn-name arg-1 arg-2 ...) expr)
```

With this syntax we can define a new function called *fcn-name*. This new function takes a number of arguments *arg-1*, *arg-2*, etc., and when applied evaluates the expression *expr* and returns the result. We call *expr* the "body" of the function.

Note: in this lesson, we use *slanted font* to indicate a pattern to be filled in.

For example let's define the Boolean operation 'nand', using 'and' and 'not':

```
; nand : Boolean Boolean -> Boolean
; RETURNS the negation of the conjunction of the two given booleans.
(define (nand x y)
  (not (and x y)))
```

We can use our new function just like any other operator:

```
> (nand true true)
false
```

In this example, we see that *expr* can refer to the arguments to produce its resulting value.

This example also shows that we should always equip our functions with a contract, like those we wrote for the basic operators, and with a *purpose statement* that describes what value the function returns. Contracts and purpose statements are a key part of our design method, and we will see much more about them in subsequent lessons.

Next, let's define a function that returns the average of two numbers:

```
; average : Number Number -> Number
; RETURNS: the average of its arguments
; usage:
; (average 3 5)    =>  4
; (average -7 7)   =>  0

(define (average x y)
  (/ (+ x y) 2))
```

We can use conditionals in a function definition. For example, we can define the absolute-value function `abs`:

```
; abs : Real -> Real
; RETURNS: the absolute value of the given real number.
(define (abs x)
  (if (< x 0)
      (- 0 x)
      x))
```

Here we wrote the contract as `Real -> Real` rather than `Number -> Number` because Racket has complex numbers, and this function definition doesn't give the right answer if `x` is complex.

Ex 4.1: Write the definition of a function that converts a temperature from degrees Fahrenheit to degrees Celsius. The formula for the conversion is C = (F-32) * (5/9) .

The contract, purpose statement and examples for this function are:

```
; f->c : Real -> Real
; GIVEN: a temperature in degrees Fahrenheit as an argument
; RETURNS: the equivalent temperature in degrees Celsius.
; Examples:
; (f->c 32)  => 0
; (f->c 100) => 37.77777777777777....
```

Test your function with *at least* the given examples. How is the value of `(f->c 100)` displayed?

Ex 4.2: Define a function called `tip` that takes two arguments, a number representing the amount of a bill in dollars, and a decimal number between 0.0 and 1.0, representing the percentage of tip one wants to give (e.g. 0.15 = 15%). `tip` should return the amount of the tip in dollars. The contract, purpose statement, and examples of `tip` are the following:

```
; tip : Real Real[0.0,1.0] -> Number
; GIVEN: the amount of the bill in dollars and the
; percentage of tip
; RETURNS: the amount of the tip in dollars.
; Examples:
; (tip 10 0.15)  => 1.5
; (tip 20 0.17)  => 3.4
```

Test your function with *at least* the given examples.

Ex 4.3: Define a function called `sq` that computes the square of a number. Write the contract, purpose statement, examples and definition of this function. Follow the examples of contracts and purpose statements above.

Ex 4.4: One of the solutions of the quadratic equation a*x^2 + b*x + c = 0 is given by the formula:

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Write the contract, purpose statement, examples, and definition of a function `quadratic-root` that takes as arguments `a`, `b`, and `c`, and computes the root of the corresponding quadratic equation. Test your code on simple examples, like x^2-4 = 0 or x^2-2x+1 = 0. What does your function give as a solution for x^2+4 = 0 ?

Ex 4.5: Define a function called `circumference` that computes the circumference of a circle. The contract, purpose statement, and examples for `circumference` are:

```
; circumference : Real -> Real
; GIVEN: the radius r of a circle
; RETURNS: its circumference, using the formula 2 * pi * r.
; Examples:
; (circumference 1)  =>  2*pi
; (circumference 0)  =>  0
```

(`pi` is a predefined constant in Racket) Test your function with *at least* the given examples. How is the value of twice pi displayed?

Ex 4.6: The area included in a circle of radius `r` is given by the formula `pi * r^2`.

Using the interaction window of DrRacket as a calculator, compute the area included in circles of radius 1, 5, and 7.

Then write the contract, purpose statement, examples, and the definition of a function called `circle-area` that computes the area included in a circle of radius `r`, using the above formula. Use the three calculations you did above as your examples.

Ex 4.7: Find out what the operator `remainder` does by typing it in the definitions window, highlighting it, and pressing F1.

Try applying `remainder` on some examples to make sure you understand what it does. (how is it different from `modulo`? Find some arguments for which `remainder` and `modulo` give different results.)

Define a predicate `is-even?` that takes a number as an argument and returns true if this number is divisible by 2, and false otherwise. Do not use the function `even?` that is predefined in BSL Use `remainder` or something similar.

Ex 4.8: Define a function that takes three numbers as arguments and returns the sum of the two larger numbers. As always, write down contract, purpose statement, and examples.

5. Structs

Like other programming languages, Racket has many kinds of data structures. In this course, we will primarily use two built-in data structures: *structs* and *lists*. Let's first talk about structs.

In Racket, we define a struct type by writing:

```
(define-struct data-type-name (field-1 field-2 ...))
```

The above code tells racket to introduce a type with a certain name that has certain named fields. (As before, we write *data-type-name* to indicate a place where you should write the name of your desired data type, etc.) Racket then creates some functions for us:

- `make-`*`data-type-name`* `: T1 T2 ... -> `*`Data-Type-Name`*

  This function takes as many arguments as there are fields in our type definition. Note that the contract here says `T1` and `T2` instead of `Number` or `Boolean`. That is because different data types will generally have fields of different types, i.e. you might want to create one where the first field is of type `Number` and another one where the first field is of type `Boolean`. You might even have a field that should contain a value of a data type that you defined yourself!

- *`data-type-name`*`? : Any -> Boolean`

  This predicate takes any value as an argument and returns `true` if that argument has the type that we just defined (i.e. if it was created by `make-`*`data-type-name`* (and of course `false` in any other case).

- *`data-type-name-field-1`* `: `*`Data-Type-Name`* `-> T1`
  `data-type-name-field-2` `: `*`Data-Type-Name`* `-> T2`
  ...

  These functions can be used to extract the values that were given as arguments to

`make-`*`data-type-name`*.

Let's see how that works in an example. Let's define a data type called `Point`. It represents a position in a two-dimensional plane, and is defined as follows:

```
(define-struct point (x y))
```

(Note that we always write the type name with an upper-case first letter while we write names of functions, variables, and fields in lower-case (see the Style Guide).

The definition above causes Racket to define the following functions:

- `make-point`
- `point?`
- `point-x`
- `point-y`

Ex 5.1: What are the values of the following expressions? Be sure to predict the answers in your head before trying the expressions.

- `(make-point 5 3)`
- `(point? 5)`
- `(point? true)`
- `(point? (make-point 2 1))`
- `(point-x (make-point 8 5))`
- `(point-y (make-point 42 15))`

Ex 5.2: What do you think the contracts for the `point`-functions should be?

Ex 5.3: What will happen if you type `(make-point true false)`? what is the result of `(point-x (make-point true false))`? Now what do you think the contracts for the `point`-functions are?

Compare the contracts you just wrote down with the contracts for the `posn` struct in the Intermediate Student Language (ISL). You can look up the contracts for `posn` in the Racket Help Desk. There are many ways to reach the Help Desk. For example, enter "make-posn" into the definition window, select it, and right-click or press F1.

Ex 5.4: Which functions will Racket create when we execute this: `(define-struct student (id name major))`?

6. More about Struct definitions

The contracts for `make-point` and `point-x` that you wrote down in Exercise 5.2 probably looked like

- `make-point : Number Number -> Point`
- `point-x : Point -> Number`

But in Exercise 5.3 you learned that `(point-x (make-point true false))` works and returns `true`. which is not a Number! That is because Racket does not care

that much about types and contracts - it will only stop if you ask it to do something it just can't, like asking for the x-component of a Boolean, but otherwise it will shove values around without complaining.

Racket also does not care what the values represent. If the x-component of your Point, is a number, you can use it to represent a percentage of something or a temperature in degress celsius, or any other quantity that that is represented as a number.

Racket does not generate the correct contracts of the point-functions, nor do they appear out of thin air. We have to write down some information for other programs to know what contracts to follow. That is, somewhere we have to specify the following information:

- What actual types do the fields of the struct have (instead of T1, T2, etc.)?
- What do the fields represent (temperature (in degrees celsius or fahrenheit?)? distances (centimeters? meters? feet?)? ...)

The way we do this is a comment next to the struct-definition:

```
(define-struct point (x y))
;; A Point is a (make-point Number Number).
;; It represents a position on the screen.
;; Interpretation:
;;   x = the x-coordinate on the screen (in pixels from the left).
;;   y = the y-coordinate on the screen (in pixels from the top).
```

Here, the first line tells us what the types of the fields are (i.e. both x and y are Numbers) and the following tells us what the type in general and its fields in particular represent. Now it is easy for everyone to see that `(make-point true false)` is not allowed and may lead to errors somewhere, and that x is neither degrees celsius nor meters nor feet. We'll see much more about this in Lesson 1.3.

Ex 6.1: Write down reasonable comments for the definition of the type Student from Ex 5.4 that define the types of the fields and their interpretation.

7. Images

In this class, we will work with images quite often. To work with images, include the following code at the top of your file:

```
(require 2htdp/image)
```

This loads the image library. Here are the basic functions for calculating with images:

```
;; bitmap : Path -> Image
;; GIVEN: a path to a file containing an image as a .jpg or .png
;; The path is given as a string, e.g. "myfile.jpg"
;; RETURNS: the image found in that file.

;; above : Image ... -> Image
;; GIVEN an arbitrary number of images
;; RETURNS: an image in which the given images appear one above
;; the other from top to bottom.

;; beside : Image ... -> Image
;; GIVEN an arbitrary number of images
;; RETURNS: an image in which the given images appear next to each
;; from left to right.

;; An OutlineMode represents a style for drawing an shape.
;;  - "outline" --only the shape's outline is drawn
```

```
;;  - "solid"   --the whole inside of the shape is filled

;; rectangle : Number Number OutlineMode Color -> Image
;; RETURNS: an image of a rectangle with given width, height,
;; drawing mode and color

;; circle : Number OutlineMode Color -> Image
;; RETURNS: an image of a circle with given radius, drawing mode
;; and color.

;; text : String Number Color -> Image
;; RETURNS: An image consisting of the given string, rendered
;; in the given color and with the given number as text size.

;; empty-scene : Number Number -> Image
;; RETURNS: The image of an empty rectangle with given width and
;; height.  An empty rectangle is a white rectangle with a black
;; outline.

;; place-image : Image Real Real Image -> Image
;; GIVEN: An Image img1, 2 Reals x and y, and an Image img2
;; RETURNS: An image like img2, but with img1 placed with its
;; center placed at the given coordinates.  The coordinates are
;; given relative to the top-left corner of img2.  The resulting
;; image is cropped to the dimensions of img2.
```

Observe that the contract for rectangle says that it expects one of exactly two strings as its third argument: either the string "`outline`" or the string "`solid`". If you give it some other string, you've violated `rectangle`'s contract, and its behavior is unpredictable! Maybe it will just produce an error, or maybe it will erase your disk! Who knows? The contract doesn't tell us.

Similarly, we've written `Color` for the fourth argument to `rectangle`. Look in the Help Desk to find out the permissible values for a Color.

Ex 7.1: Create a folder on git and save your Racket file there. Then also copy some image to that folder (either take it from your computer or download one from the internet). Then put the following in your racket file:

```
(define my-image (bitmap "the file name of your image"))
```

Play around with some of the image functions, also try something like

```
(above my-image my-image my-image)
```

Ex 7.2: Create some solid blue rectangles with the following dimensions:

- 2x4
- 4x8
- 8x16
- 16x32

Ex 7.3: Give the dimensions of the next 2 rectangles in the sequence. Write down a formula that describes the n-th rectangle in this sequence. Write down a contract, purpose statement, examples, and definition for a function `rectangle-in-sequence` that takes an argument `n`, where `n` is a number that tells the function to return the `n`th rectangle in this sequence. Be sure to test your function on at least the examples above.

Ex 7.4: Write the following function:

```
;; rectangle-from-proportions: PosReal PosReal -> Rectangle
;; RETURNS: a solid blue rectangle, whose width in pixels is given
;; by the first argument, and whose proportion (ratio of height to
;; width, i.e. height = width * proportion) is given by the second
;; argument.
```

Test your function on some examples. How do you know that the image your function created is the right one? (Hint: use `image-width` and `image-height` from the image library).

Ex 7.5: Try to assemble a human shape from circles and rectangles using the image functions above. It does not need to look fancy, just imagine a head, a chest and arms and feet. Then use the stepper to see how DrRacket assembles your image.

Ex 7.6: Here is a struct definition:

```
(define-struct person (first-name last-name age height weight))
```

Write down a reasonable comment part of that data definition that specifies types and interpretations of the fields. Then write the function `person-image` that takes a person and returns an image like the ones in Ex 7.5, but in a way that the height and width of this image is related to the height of the person (i.e. if one person is twice the size of another person, the image for the first person should be twice as high and wide as the image of the second person).

Ex 7.7: Write a function `person+name-image`, which returns an image like the one returned by `person-image` except that the full (first + last) name of the person is drawn below the image of the person. You may want to use the built-in function `string-append`. Look in the help desk to see what it returns.

8. Lists

Structs can already consist of more than one value, but very often we need to store an arbitrary number of values. A fundamental part of programming in Racket lets us do just that: lists. In contrast to the structs we have seen, there are two basic ways to create a list value. The first is rather self-explanatory:

```
;; empty : List
;; The empty list
```

But somehow, we also have to get values into lists, so we have a second constructor:

```
;; cons: Any List -> List
;; Given some value and a list, returns a new list in which the given value is t
;; first element and the given list is the rest of the new list.
```

Can you guess how to create a list with one element? The answer is to use `cons` with that element and the empty list, that is:

```
> (cons "Something" empty)
(cons "Something" empty)
```

And a list of two elements? Well, this time we'll have to use `cons` twice:

```
> (cons "Something" (cons "Some other thing" empty))
(cons "Something" (cons "Some other thing" empty))
```

Note that `cons` *prefixes* an element to the list-- that is, it returns a new list that is like

the old list, except that the new element is added at the front.

Ex 8.1: Write down an expression whose value is the list of numbers from 1 to 5.

Ex 8.2: Write down an expression whose value is a list of 5 booleans, alternating between true and false, starting with true

Writing down all these constructors is a bit cumbersome, so here's a shorthand for creating lists:

```
;; list : Any ... -> List
;; Takes any number of values and returns a list of those values.
;; Example: (list 1 2 3 4 5) => (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty)))
```

That is a lot shorter! Now that we have seen how to construct lists, what can we do with them? We observe their contents by taking them apart again, piece by piece:

```
;; list-fn : List -> ??
; (define (list-fn lst)
;   (cond
;     [(empty? lst) ...]
;     [else (... (first lst) (list-fn (rest lst)))]))
```

If you forgot what `cond` is about, look a few headlines back. Apart from that, you'll see that there is a predicate `empty?` that lets you check if a list is empty. If it is not, then the list must have been constructed by using `cons`, and that means that there must be a first element of the list and some rest - and as you may have guessed, the functions `first` and `rest` return just those.

A interesting thing is the call to `list-fn` in the last line of `list-fn`. This is what we call *recursion*. There is a lot to be said about recursion later in the term, but for now we'll just use this technique to process lists.

Let us rather talk about what we do in this function: we get some list as an argument, and we know that there are two cases how that list could have been constructed: either using `empty` or using `cons`. Well, there has to be a result in any case, but if the list is empty, we cannot extract a value from it, so we have to replace the dots in the first case with a meaningful value in case the list is empty. If, on the other hand, there is a value and a rest, both of them may influence the final result of our computation. Therefore, we calculate the result of the computation of the rest of the list (which may be empty or have some more elements, but less than the list we were given) and then *combine* it with the first element. Let's look at an example:

```
;; sum : List -> Number
;; RETURNS: the sum of the numbers in the given list
;; EXAMPLES:
;; (sum empty) = 0
;; (sum (list 1)) = 1
;; (sum (list 1 2 3) = 6

(define (sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (sum (rest lst)))]))
```

Look what we inserted here: the sum of all the numbers in an empty list is clearly 0. And if we take the number that is first in a list and add it to the sum of all the numbers in the rest of the list, we clearly get the sum of all numbers in the list. We do not necessarily have to use any of the values in the list in every case:

```
;; list-length : List -> Number
;; RETURNS the length of the given list
```

```
;; EXAMPLES:
;; (list-length empty) = 0
;; (list-length (list 1)) = 1
;; (list-length (list 1 2 3) = 3

(define (list-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (list-length (rest lst)))]))
```

Here, the length of an empty list is again clearly 0, and the length of a list where there is a first element and a rest must be 1 + the length of that rest.

Ex 8.3: Write a function that returns the product of all the numbers in a list (Hint: be careful with the empty list)

Look at the following pieces of code. Is there anything wrong with either one or both of them?

```
(list-length (list 1 5 "a" true 3))

(sum (list 1 5 "a" true 3))
```

`list-length` works just fine, but `sum` does not. Racket complains because this code violates the contract of +: + expects a number as its first argument, and here it is given `true`. `(+ true 3)` is not a valid operation. And indeed, altough we have not mentioned contracts for a while, we have severely violated our contracts. Could you guess what the contract for `first` would be? As we said that the contract of `cons` is

```
;; cons : Any List -> List
```

Therefore we cannot know anything for sure about the first element in a list. Thus, the right contract would be

```
;; first : List -> Any
;; WHERE: the list is non-empty
```

But + requires Numbers as arguments, so it is not safe to supply it with arguments of type Any. To fix this, we introduce further conventions that Racket itself does not care about as long as everything works (like with list-length): we introduce specialized lists for certain types, for example the type ListOfNumber, (in short: LoN). You can still use `empty`, `cons`, `list`, `first` and `rest`, but you can change the contracts to have (Some-Type) instead of Any and ListOf(Some-Type) instead of List. So for ListOfNumber we'll have

```
;; cons  : Number ListOfNumber -> List-Of-Number
;; first :ListOfNumber -> Number
```

but for ListofBoolean we'll have

```
;; cons  : Boolean ListOfBoolean -> ListOfBoolean
;; first : ListOfBoolean -> Boolean
```

Ex 8.4: Design a function that, given a list of booleans, returns true if all booleans in the list are true. Write down contract, purpose statement and examples, and test your function.

Ex 8.5: Design a function that takes a list of Points and draws a solid blue circle with radius 10 at every Point in that list into a 300x300 scene.

Ex 8.6: Design a function that takes a list of strings and draws the combined text of those strings, separated by spaces.

Ex 8.7: There are two ways to do Ex 8.6 with the functions available to you. Try the way that you did not use to solve Ex 8.6.

Ex 8.8: Design a function that takes a list of lists of strings as an argument that treats each of the lists of strings as a line (assembled like in Ex 8.6) in a text and renders the whole text as an image.

Ex 8.9: Look up the `beside/align` function on the Racket Help Desk. Use it to design a function that takes a list of people (as defined in Ex 7.6) and uses the function from Ex21 to draw these people, placing them beside each other to form some kind of a group photo.

Ex 8.10: Design a function that, given a list of booleans, returns a list with each boolean negated (e.g. `(neg-list (list true false true)) => (cons false (cons true (cons false empty))`).

Ex 8.11: Design a function that, given a list of Numbers, returns a list of Images, where each image is a circle that has a radius based on a number of the input list.

Ex 8.12: Design a function that takes a list of Points and returns the sum of the distances to each of those Points from (0,0). You should write a helper function to calculate the distance.
