Racket Lab (optional)

This lab is meant to quickly introduce students who haven’t seen Racket before. It goes through the basic types and operators in Racket as well as a few exercises that you can try out if you wish. The lab is completely optional so feel free to utilize it as much or little as you like. While we will not be hosting a separate lab time for it, you are more than welcome to ask questions at TA hours about the lab or about general Racket syntax.

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)
2.23606797749979
```

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

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 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 guarantees that these operators take two numbers as arguments and return a number as a result. Knowing the contract of an operator is imperative in order to use it correctly!

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

To apply an operator on some operands we need a pair of parentheses that enclose both the operator and the operands, in 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:
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 $3 \times 2 + 5 \times 3$ we can type:

```racket
> (* 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 $3 \times 2 + 5 \times 3$ writing just one big expression:

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

Q: What is the order of evaluation in complex expressions?

When things get too complicated use indentation to make the expression more readable:

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

; False
> false
false
```

It also provides operators that can be applied on booleans

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

; or : Boolean Boolean -> Boolean
; Logic disjunction (Inclusive)

; not : Boolean -> Boolean
; Logic negation

> (not false)
ture

> (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 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
```
So for example:

```scheme
> (< 300.0001 300)
false
> (= (+ (* 3 50)
    (* 3 25))
    (* 3
    (+ 50 25)))
true
```

Q: What will this return?

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

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

## Conditional

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.

```scheme
(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 action. The first test that succeeds triggers its associated action. The final else clause is chosen if no other test succeeded.

For example:

```scheme
> (cond
  ...)
```
Defining Functions

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 (op-name arg-1 arg-2 ...) expr)
```

With this syntax we can define a new operator called `op-name` that takes a number of arguments `arg-1, arg-2, etc., and when applied evaluates the expression `expr` and returns the result. `expr` can refer to the arguments, to produce its resulting value. For example let's define the Boolean operation 'nand', using 'and' and 'not':

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

We can use our addition just like any other operator:

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

Let's also define a function that computes 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))
```

Let's also define `abs` using `cond`:

```
```
(define (abs x)
  (cond
   [(< x 0) (- 0 x)]
   [else x]))

Ex 1: Compute the number of seconds in a leap year (a leap year has 366 days).

Ex 2: Write an expression that tests if the result of 100/3 is greater than the result of (100 + 3) / (3 + 3).

Ex 3: Write the definition of a function that converts a temperature from degrees Fahrenheit to degrees Celsius. The formula for the conversion is \( C = \frac{F - 32}{9} \). The contract, purpose statement and examples for this function are:

; f->c : Number -> Number
; GIVEN: a temperature in degrees Fahrenheit as an argument
; RETURNS: the equivalent temperature in degrees Celsius.
; Examples:
; (f->c 32)  => 0
; (f->c 100) => 37.77777777777778

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

Ex 4: 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 compute the amount of the tip in dollars. The contract, purpose statement, and examples of `tip` are the following:

; tip : NonNegNumber Number[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 5: Define a function called `sq` that computes the square of a number. Write the contract, purpose statement, examples and definition of this function.
Ex 6: One of the solutions of the quadratic equation is given by the formula:

\[ x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

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.

Ex 7: Define a function called circumference that computes the circumference of a circle. The contract, purpose statement, and usage of circumference are:

```
; circumference : Number -> Number
; GIVEN: the radius r of a circle
; RETURNS: its circumference, using the formula 2 * pi * r.
; Examples:
; (circumference 1)  =>  6.283185307179586
; (circumference 0)  =>  0
```

(pi is a predefined constant in Racket) Test your function with, at least, the given examples.

Ex 8: The area included in a circle of radius r is given by the formula $3.1415 \times r^2$. Using the interaction window of DrRacket as a calculator, compute the area included in circles of radius 1, 5, and 7.

Write the contract, purpose statement, examples, and the definition of a function called circ-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 9: 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. (what is its difference with modulo?)

Define a predicate even? that takes a number as an argument and returns true if this number is divisible by 2, and false otherwise. (You will probably need to use remainder, or something similar, in the implementation of even?.)
Ex 10: 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.

Complex Data Types: Structs

Number and Boolean are primitive data types in Racket. It is often useful to have more complex data types. Racket provides the possibility to just create those as we need them. The general scheme is:

```
(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. 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 the types of the Fields can be different for every data type, 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 can even use a data type that you defined yourself!

- **data-type-name? : Any → Boolean**
  
  This predicate takes anything as 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. The Racket student languages include a pre-defined data type called **Posn**. It represents a position in a two-dimensional plane, and is defined as follows:
(define-struct posn (x y))

(Note that in this Lab write the type name with an upper-case first letter while we write names in definitions always in lower-case).

The above code lets Racket create the following functions:

- make-posn
- posn?
- posn-x
- posn-y

If you are using a teaching language, Posn may already defined and the above functions are available to you.

All of this lab has been reused from cs5010 at Northeastern. You can access their entire lab handout at http://www.ccs.neu.edu/home/wand/cs5010/lab01/racket-intro.htm.