

**Assignment: 1**

**Due:** Tuesday, January 12, 2016 9:00 pm

**Language level:** Beginning Student

**Files to submit:** `functions.rkt`, `conversion.rkt`, `grades.rkt`,  
`participation.rkt`

**Warmup exercises:** HtDP 2.4.1, 2.4.2, 2.4.3, and 2.4.4

**Practise exercises:** HtDP 3.3.2, 3.3.3, and 3.3.4

**Assignment Policies**

- No assignments will be accepted beyond the submission date. No exceptions.
- For this and all subsequent assignments, the work you submit must be entirely your own.
- Do not look up either full or partial solutions on the Internet or in printed sources.
- Make sure to follow the style and submission guide available on the course Web page when preparing your submissions. Please read the course Web page for more information on assignment policies and how to submit your work.
- **For this assignment only**, you do not need to include the design recipe in your solutions. A well-written function definition is sufficient.

**Grading**

- **This assignment (or any later one) will not be graded and no marks will be recorded until you have first received full marks in Assignment 0.**
- Completing A0 after the A1 deadline will result in a mark of 0 on Assignment 1.
- Your solutions for assignments will be graded on both correctness and on readability.

**Correctness**

- **Be sure to check your basic test results after each submission!** They will be emailed to your student account within a few minutes of your assignment submission.
- If you do not get full marks on the basic test, then your submission will not be markable by our automated tools, and you will receive a low mark (probably 0) for the correctness portion of the corresponding assignment question.
- On the other hand, getting full marks on the basic test does **not** guarantee full correctness marks. It only means that you spelled the name of the function correctly and passed some extremely trivial tests.
- **Thoroughly testing your programs is part of what we expect of you on each assignment.**

## Readability

- You should use constants where appropriate.
- All identifiers should have meaningful names, unless specifically stated otherwise such as in question 1.

Here are the assignment questions you need to submit.

1. **Translate** the following function definitions into Racket, using the names given. Place your solutions in the file `functions.rkt`.

Note that when you are asked to **translate** a function, it should be a direct translation. When asked to translate  $(a + b)$  the translation is  $(+ a b)$ , not  $(+ b a)$ . When translating  $x^2$ , either  $(sq x)$  or  $(* x x)$  is acceptable.

- (a) An example from mathematics (Stirling's upper bound):

$$\textit{stirling}(n) = n^{(n+\frac{1}{2})} \cdot e^{1-n}$$

(Hint: review the difference between the built-in Racket functions *exp* and *expt*.)

- (b) An example from algebra (*harmonic mean*):

$$\textit{hm}(x, y, z) = \frac{3}{\frac{1}{x} + \frac{1}{y} + \frac{1}{z}}$$

- (c) An example from physics (*ballistic motion*):

$$\textit{height}(v, t) = v \cdot t - \frac{1}{2} \cdot g \cdot t^2$$

where  $g$  is the constant 9.8 (acceleration due to gravity)

2. The constant 9.8 (used in the previous question) represents the acceleration due to gravity in units of metres per second squared ( $m/s^2$ ). This is a metric unit; in the United States, so-called “imperial” units are usually used instead of metric. There, the constant  $g$  would likely have the value of 32, in units of feet per second squared ( $ft/s^2$ ). As you can see, it is very important to know what units you’re working with when writing programs that deal with real-world measurements. In fact, NASA’s Mars Climate Orbiter crashed into Mars in 1999 because some of the programmers were assuming metric units while others were assuming imperial units!<sup>1</sup>

In this question, you will write functions to convert between units. Place your solutions in the file `conversion.rkt`. You should use meaningful constant names. Do not perform any “rounding”. You do not have to worry about “divide by zero” errors.

- (a) The unit of speed most often used in physics is metres per second ( $m/s$ ). A common imperial unit of speed is miles per hour ( $mph$ ). Write a function  $m/s \rightarrow mph$  which consumes a speed in the units of  $m/s$  and produces the same speed in units of  $mph$ . It may be helpful to know that one mile is exactly 1609.344  $m$ . (Remember that in your function name,  $\rightarrow$  is typed as `->`.)
- (b) A more unusual unit of speed is *fathoms per fortnight* ( $fpf$ ). One fathom is exactly 1.8288  $m$  and a fortnight is 14 days. Write a function  $fpf \rightarrow mph$  which consumes a speed in units of  $fpf$  and produces the same speed in units of  $mph$ .
- (c) An even more unusual unit of speed is *Smoots per nanocentury* ( $S/nc$ ). One Smoot is exactly 1.7018  $m$  and one nanocentury is exactly 3.15576  $s$ . Write a function  $mph \rightarrow S/nc$  which consumes a speed in units of  $mph$  and produces the same speed in units of  $S/nc$ .

---

<sup>1</sup>[ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO\\_report.pdf](ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf)

3. The end of the Winter term has come, and you decide to use Racket to calculate your grade going in to the CS 135 final. For this question, you do not need to worry about the course requirements of passing the exam and assignment components of the course separately.

- (a) Write a function *cs135-grade-sofar* that consumes three numbers (in the following order):
- the midterm grade,
  - the participation grade, and
  - the overall assignments grade.

This function should produce the weighted grade in the course (as a percentage, but not necessarily an integer) going in to the final exam. For example, (*cs135-grade-sofar* 100 100 100) would produce 100. You may need to review the mark allocation in the course. You can assume that all grades are percentages and are given as numbers between 0 and 100, inclusive.

- (b) The end of exams has now come, and you have received your final grade in the course. Write a function *cs135-final-exam* that consumes two numbers. The first number is the weighted grade going in to the final exam (as calculated above). The second number is the final grade received in the course. This function should calculate the grade received on the final exam.

Place your solutions in the file `grades.rkt`.

4. **2% Bonus:** In CS 135, your class participation grade (from clickers) is calculated in the following manner:

- Each question is worth two marks.
- You receive two marks for a correct answer, and one mark for an incorrect answer.
- You receive zero marks if you do not answer.
- To account for imperfect attendance, forgotten clickers, dead batteries etc., only your best 75% of the questions are used to calculate your grade.

Write a function *cs135-participation* that consumes three parameters (in order):

- the total number of clicker questions asked in the year,
- the number of questions you answered correctly, and
- the number of questions you answered incorrectly.

Your function must produce your class participation grade as a percentage. For convenience, you may assume the total number of questions is a positive Integer divisible by four.

**Note:** you may only use the features of Racket given up to the end of Module 1. You may use **define** and **mathematical** functions, but not **cond**, **if**, lists, recursion, Booleans, or other things we'll get to later in the course. Specifically, you may use any of the functions in section 1.5 of this page:

<http://docs.racket-lang.org/htdp-langs/beginner.html>

Place your solution in the file `participation.rkt`. Note that bonus questions are typically “all or nothing”. Incorrect or very poorly designed solutions may not be awarded any marks.

This concludes the list of questions for you to submit solutions. Don't forget to always check your email for the basic test results after making a submission.

## Challenges and Enhancements

The teaching languages provide a restricted set of functions and special forms. There are times in these challenges when it would be nice to use built-in functions not provided by the teaching languages. We may be able to provide a teachpack with such functions. Or you can set the language level to “Pretty Big”, which provides all of standard Racket, plus the special teaching language definitions, plus a large number of extensions designed for very advanced work. What you lose in doing this are the features of the teaching languages that support beginners, namely easier-to-understand error messages and features such as the Stepper.

This **enhancement** will discuss exact and inexact numbers.

DrRacket will try its best to work exclusively with *exact* numbers. These are *rational* numbers; i.e. those that can be written as a fraction  $a/b$  with  $a$  and  $b$  integers. If a DrRacket function produces an exact number as an answer, then you know the answer is exactly right. (Hence the name.)

DrRacket has a number of different ways to express exact numbers. 152 is an exact number, of course, because it is an integer. Terminating decimals like 1.60934 from question 2 above are exact numbers. (How could you determine a rational form  $a/b$  of this number?) You can also type a fraction directly into DrRacket; 152/17 is an exact number. Scientific notation is another way to enter exact numbers; 2.43e7 means  $2.43 \times 10^7 = 24300000$  and is also an exact number.

It is important to note that adding, subtracting, multiplying, or dividing two exact numbers always gives an exact number as an answer. (Unless you’re dividing by 0, of course; what happens then?) Many students, when doing problems like question 2, think that once they divide by a number like 1.60934, they no longer have an exact answer, perhaps because their calculators don’t treat it as exact.

But try it in DrRacket: (`/ 2 1.60934`). DrRacket seems to output a number with lots of decimal places, and then a “...” to indicate that it goes on. But right-click on the number, and a menu will allow you to change how this (exact) number is displayed. Try out the different options, and you’ll see that the answer is actually the exact number 100000/80467.

You should use exact numbers whenever possible. However, sometimes an answer cannot be expressed as an exact number, and then *inexact numbers* must be used. This often happens when a computation involves square roots, trigonometry, or logarithms. The results of those functions are often not rational numbers at all, and so exact numbers cannot be used to represent them. An inexact number is indicated by a `#i` before the number. So `#i10.0` is an inexact number that says that the correct answer is probably somewhere around 10.0.

Try (`sqr (sqrt 15)`). You would expect the answer to just be 15, but it’s not. Why? (`sqrt 15`) isn’t rational, so it has to be represented as an inexact number, and the answer is only approximately correct. When you square that approximate answer, you get a value that’s only approximately 15, but not exactly.

You might say, “but it’s close enough, right?” Not always. Try this:

```
(define (addsub x)
  (- (+ 1 x) x))
```

This function computes  $(1 + x) - x$ , so you would expect it to always produce 1, right? Try it on some exact numbers:

```
(addsub 1)
(addsub 12/7)
(addsub 253.7e50)
```

With exact numbers, you always get 1, as expected. What about with inexact numbers?

`(addsub (sqrt 15))` => #i1.0, which is fine. `(addsub (sqrt 2))` => #i0.9999999999999998, which is close to 1; that's more or less what we expect from inexact numbers. But `(addsub (exp 40))` => #i0.0. That answer is very different from 1! Can you find inputs which give different answers from these?

If you go on to take further CS courses like CS 251 or CS 371, you'll learn all about why inexact numbers can be tricky to use correctly. That's why in this course, we'll stick with exact numbers wherever possible.