CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment: 1

Due:	Tuesday, September 19, 2017 9:00 pm
Language level:	Beginning Student
Files to submit:	`functions.rkt`, `conversion.rkt`, `grades.rkt`, `bonus.rkt`
Warmup exercises:	HtDP 2.4.1, 2.4.2, 2.4.3, and 2.4.4
Practice exercises:	HtDP 3.3.2, 3.3.3, and 3.3.4

Assignment policies for this and all subsequent assignments

- No assignments will be accepted beyond the submission date and time. No exceptions.
- 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 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.
- Unless otherwise explicitly noted, you must include the design recipe in your solutions.

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!** MarkUs will display your basic test results shortly after you make a submission.
- If you do not get full marks on the basic tests, 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 tests 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 define constants where appropriate.
- All identifiers should have meaningful names, unless specifically stated otherwise such as in Question 1.

Other notes

- Check your clicker marks by following the link for "View Marks" on the course Web page to make sure your clicker is registered properly.
- Each assignment will start with a list of warmup exercises. You don't need to submit these, but we strongly advise you to do them to practice concepts discussed in lectures before doing the assignment. This week's warmup exercises are HtDP exercises 2.4.1, 2.4.2, 2.4.3, and 2.4.4.

Here are the assignment questions you need to submit.

1. Computers are helpful tools in many fields of human endeavour. Below are functions that are useful to compute, taken from a variety of areas.

   Translate the 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, use the Racket function (*sqr x*).

   For example, if we asked you to translate the function:

   $$mean(x1,x2) = \frac{x1+x2}{2}$$

   you would submit:

   ```
   (define (mean x1 x2)
     (/ (+ x1 x2) 2))
   ```

   Note: the design recipe is <u>not</u> required for Question 1 of this assignment.

   (a) An example from geometry (*Manhattan distance*):

   $$distance(x1,y1,x2,y2) = |x1-x2| + |y1-y2|$$

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

   $$Stirling(n) = n^{\left(n+\frac{1}{2}\right)} \cdot e^{1-n}$$

   Remember that `1/2` is the way to write the number $\frac{1}{2}$ in Racket, and note that *Stirling* begins with a capital letter, as it is a proper name.

(c) An example from statistics (*logit*):

$$logit(p) = \ln \frac{p}{1-p}$$

(d) An example from music (*even temperment*):

$$freq(base\text{-}frequency, interval) = base\text{-}frequency \cdot 2^{interval/12}$$

(Note the hyphenated name *base-frequency*. Names in Racket can have hyphens in them for readability, unlike in some other computer languages, where *base-frequency* would be interpreted as *base* minus *frequency*. Also note that while in practice, it would be helpful to use defined constants like *intervals-per-octave* instead of numbers like 12, in this question, you are asked to *translate* the given function, not improve it.)

(e) An example from finance (*Black-Scholes formula*):

$$d1(maturity, rate, volatility, spot\text{-}price, strike\text{-}price) =$$

$$\frac{1}{volatility \cdot \sqrt{maturity}} \cdot \left[\ln \left(\frac{spot\text{-}price}{strike\text{-}price} \right) + \left(rate + \frac{volatility^2}{2} \right) \cdot maturity \right]$$

(f) An example from physics (*ballistic motion*):

$$height(initial\text{-}velocity, time) = initial\text{-}velocity \cdot time - \frac{1}{2} \cdot g \cdot time^2$$

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

2. The above constant 9.8 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![1]

   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→mph* that consumes a speed in the units of m/s and produces the same speed in units of mph. You must use the fact that one mile is exactly $1609.344\,m$. (Remember that in your function name, → is typed as `->`.) Notice that writing fractions as `1609344/3600000` in Racket can help you write your examples and tests.

[1] ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, p.16

(b) A more unusual unit of speed is *Smoots per millifortnight* (S/mfn). You must use the facts that one Smoot is exactly $1.7018\,m$ and one millifortnight is exactly $1209.6\,s$. Write a function *mph→S/mfn* that consumes a speed in units of mph and produces the same speed in units of S/mfn. Note that the *S* in the function name is capitalized.

(c) In the United States, it is common to measure the fuel efficiency of a vehicle in miles per gallon (mpg). In Canada, it is more common to measure fuel efficiency in litres per 100 km (L/100km). Write a function *mpg→L/100km* that consumes a fuel efficiency in mpg and produces the same efficiency in units of L/100km.

As above, you must use the facts that one mile is exactly $1609.344\,m$, that there are $100000\,m$ in $100\,km$, and that one gallon in the United States is exactly 3.785411784 litres. (Remember, these are *exact*, not approximate, numbers, even though they may have many decimal places.)

3. (a) The end of the Fall term has come, and you decide to use Racket to calculate your final grade in CS 135. You've been participating actively throughout the term, so you'll be receiving full participation marks. However, your final grade still depends on your performance in the assignments and exams. For this question, you do not need to worry about the course requirements of passing the exam and assignment components of the course separately.

Write a function *final-cs135-grade* that consumes four numbers (in the following order):

1. the first midterm grade,
2. the second midterm grade,
3. the final exam grade, and
4. the overall assignments grade.

This function should produce the final grade in the course (as a percentage, but not necessarily an integer). You may need to review the mark allocation in the course. You can assume that all argument values are percentages and are given as integers between 0 and 100, inclusive. (Hint: where does this assumption go in the design recipe?)

(b) Now write a function *cs135-final-exam-grade-needed* that consumes three integers: the first midterm grade, the second midterm grade, and the overall assignments grade (in that order). As above, these argument values are each in the range of 0–100 inclusive. This function produces the minimum grade needed on the final exam to obtain 60% in the course (recall that 60% is the minimum grade a student needs to earn in CS 135 in order to advance to CS 136). Note that this function might produce values outside the range 0–100, and might produce non-integer values. (It should always produce an exact value, however.) As above, you should assume full participation marks.

Place your solutions to this question in the file `grades.rkt`.

This concludes the list of questions for you to submit solutions (but see the following pages as well). Don't forget to always check the basic test results after making a submission.

On each assignment, we will list extra practice exercises. You don't need to submit these either. You can do them at any time after completing the assignment to solidify your understanding, or as part of studying for an exam. This week's extra practice exercises are HtDP exercises 3.3.2, 3.3.3, and 3.3.4. From Assignment 2 on, look for these just below the warmup exercises at the top of the assignment.

Assignments will sometimes have additional questions that you may submit for bonus marks.

4. **5% Bonus**: Reimplement the *final-cs135-grade* function above, but this time, you *must* take into account the rule about passing the exam and assignment components of the course separately.

   In particular, if *either* the assignment component *or* the weighted exam component of the course grade is less than 50%, then the final course grade will be either the course grade as normally computed, or a grade of 46%, whichever is *smaller*.

   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 non-Boolean functions in Section 1.5 of this page:
   `http://docs.racket-lang.org/htdp-langs/beginner.html`

   Hint: you may find the *min* and *sgn* functions particularly useful.

   Place your solution in the file `bonus.rkt`. Note that bonus questions are typically "all or nothing". Incorrect or very poorly designed solutions may not be awarded any marks. You are **not** required to submit a design recipe for this bonus question, but we encourage you to use it anyway in order to design a correct solution!

Challenges and Enhancements

Each assignment in CS 135 will continue with challenges and enhancements. We will sometimes have questions (such as the one above) that you can do for extra credit; other questions are not for credit, but for additional stimulation. Some of these will be fairly small, while others are more involved and open ended. One of our principles is that these challenges shouldn't require material from later in the course; they represent a broadening, not an acceleration. As a result, we are somewhat constrained in early challenges, though soon we will have more opportunities than we can use. You are always welcome to read ahead if you find you want to make use of features and techniques we haven't discussed yet, but don't let the fun of doing the challenges distract you from the job of getting the for-credit work done first. On anything that is not to be handed in for credit, you are permitted to work with other people.

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 `1609.344` 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 `1609.344`, they no longer have an exact answer, perhaps because their calculators don't treat it as exact.

But try it in DrRacket: `(/ 2 1609.344)`. 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 `125/100584`.

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 argument values that give different answers from these?

If you go on to take further CS courses like CS 251 or CS 370, 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.