CS 135
Winter 2014
Troy Vasiga

Assignment: 1

Due:	Tuesday, January 14, 2014 9:00 pm
Language level:	Beginning Student
Files to submit:	`constants.rkt`, `functions.rkt`, `speed.rkt`, `grades.rkt`, `stars.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

For this and all subsequent assignments the solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources. Please read the course Web page for more information on assignment policies and how to submit your work. Make sure to follow the style and submission guide available on the course web page when preparing your submissions. Your solutions for assignments in this course will be graded both on correctness and on readability, meaning that, among other things, you should use constants and parameters with meaningful names.

Note that for this assignment only, you do not need to include the design recipe in your solutions. A well-written function definition is sufficient.

You will not be able to submit this assignment (or any later one) before you have first received full marks in Assignment 0.

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.

Be sure to check your public test results after each submission! If you do not get full marks on the public test, then your assignment will not be markable by our automated tools, and you will receive a low mark (probably 0) for the correctness portion of the assignment. On the other hand, getting full marks on the public 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 from you for each assignment.**

Here are the assignment questions you need to submit.

1. Translate the following constant definitions into Scheme. Place your solutions in the file `constants.rkt`. For example, if we asked you to translate the definition of the mean of three numbers given by

$$mean = \frac{n_1 + n_2 + n_3}{3}$$

with the specific values

$$mean = \frac{7 + 8 + 9}{3}$$

you would submit:

(**define** *mean* (/ (+ 7 8 9) 3))

Which would give a value 8 as the correct output (when you type *mean* in the Scheme interpreter).

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

$$distance = |x1 - x2| + |y1 - y2|$$

$$distance = |9 - 5| + |3 - 6|$$

(b) An example from geometry (*the cosine law*):

$$a = \sqrt{b^2 + c^2 - 2 \cdot b \cdot c \cdot \cos(t)}$$

$$a = \sqrt{52^2 + 16^2 - 2 \cdot 52 \cdot 16 \cdot \cos(115)}$$

(c) An example from physics (*Einstein's equation — full form*):

$$c = 299,792,458$$

$$E = \sqrt{m^2 \cdot c^4 + p^2 \cdot c^2}$$

$$E = \sqrt{1^2 \cdot c^4 + 0^2 \cdot c^2}$$

2. Translate the following function definitions into Scheme. Place your solutions in the file `functions.rkt`.

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

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

(Hint: recall from Assignment 0 the Scheme name for the built-in exponential function e^x.)

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

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

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

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

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

3. 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 measurements of speed between units. Place your solutions in the file `speed.rkt`.

   (a) The unit of speed most often used in physics is meters per second (m/s). A common imperial unit of speed is miles per hour (mph). Write a function *mph→m/s* which consumes a speed in the units of mph and produces the same speed in units of m/s. It may be helpful to know that one mile is exactly $1609.344\ m$. (Remember that in your function name, → is typed as `->`.)

   (b) A 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→S/nc* which consumes a speed in units of mph and produces the same speed in units of S/nc.

4. The end of term has come, and you decide to use Scheme 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):

   (a) the final exam grade,
   (b) the first midterm grade,
   (c) the second midterm grade, and

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

(d) 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 input marks are percentages and are given as integers between 0 and 100, inclusive.

Now write a function *final-cs135-exam-grade-needed* that consumes three numbers: the first midterm grade, and the second midterm grade, and the overall assignments grade (in that order). This function produces the minimum mark needed on the final exam to obtain 60% in the course. 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.)

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

This concludes the list of questions for which you need to submit solutions. Don't forget to always request a public test 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.

5. **5% Bonus**: Remember that on this assignment, you are to use only the techniques presented in Module 01. In particular, even if you happen to know them, or have been reading ahead, you may not use **cond**, **if**, lists, recursion, etc. You may, however, use any mathematical function available in Beginning Student.

   A *starstep pattern of size* n is a pattern of stars like this (when $n = 6$, for example):

   ```
   * *
   * * * *
   * * * * * *
   ```

   That is, the last row has n stars, and each higher row has 2 fewer stars than the row below it. The first row will have either 1 or 2 stars.

   Write a function *starstep-stars* that consumes a positive integer n and produces the total number of stars in the starstep pattern of size n. Place your solutions in the file `stars.rkt`.

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 Scheme, 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 3 above are exact numbers. (What is the 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 3, 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 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.