CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 01

Due:	Tuesday, September 17, 2019 9:00 pm
Language level:	Beginning Student
Files to submit:	`translations.rkt`, `bmi.rkt`, `grades.rkt`, `bonus-a01.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

Notes:

- Become familiar with the CS 135 assignment policies on the course website.
- Frequently scan the OFFICIAL A01 post on Piazza for answers to frequently asked questions, *especially before you post to Piazza*.
- 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. Read about plagiarism on the course web site.
- **If you haven't completed Assignment 0 with full marks, you'll receive 0 for this assignment.**
- A significant proportion of your marks will be for good coding style. You should be familiar with the style guide and apply relevant parts to each assignment. Appropriate constant usage is particularly important for A01. **You do *not* need to include the design recipe for A01.**
- Submit early and often. If you don't know what this means, talk to your prof or an ISA.

Here are the assignment questions you need to submit. All functions required to complete these questions can be found in section 1.5 of the documentation for the *Beginning Student* language.

1. In Module 01 we made a big deal about how there may be many possible substitutions when evaluating an expression but that we all need to agree on exactly one in any given situation. This will continue to be a big deal in the course as we introduce more and more programming constructs.

   To help reinforce the substitution rules we have an online tool, known as the "Stepper" at

   `https://www.student.cs.uwaterloo.ca/~cs135/assign/stepping/`

   Note: the use of `https` is important; that is, the system will not work if you omit the `s`.

   You will need to authenticate yourself using your Quest/WatIAM ID and password. Once you are logged in, click on the first question under "Function application". Note the "Show instructions" link at the bottom of each problem. **Read the instructions!**

Complete the 5 stepping problems under "Function application", using the informal substitution rules given in class. We will be formalizing these rules in Module 3.

The "Hint" button is enabled for the first problem. Use it if you get stuck.

You can re-enter a step as many times as necessary until you get it right, so keep trying until you completely finish every question. All you have to do is complete the questions online—we will be recording your answers as you go, and there is no file to submit. The basic tests for this assignment will tell you whether or not we have a record of your completion of the stepper problems. **Note that you are not done with a question until you see the message** `Question complete!` You should see this once you have arrived at a final value and clicked on "simplest form" (or "Error," depending on the question).

You are stepping through the given expressions assuming that constant definitions that appear above the expression exist, and have been fully processed as though you have pressed Run in DrRacket. **This means that you are not required to do any simplification of the constant definitions as part of the stepping.**

You should **not** use DrRacket's Stepper to help you with this question for several reasons. First, DrRacket's evaluation rules are slightly different from the ones presented in class, but we need you to use the evaluation rules presented in class. Second, in an exam situation, you will not have DrRacket's Stepper to help you, and there will definitely be step-by-step evaluation questions on at least one of the exams.

Note: If you get stuck on a stepping question, **do not post to Piazza requesting the next step.** This is a violation of the academic integrity policy. Review the substitution rules carefully to try to solve the problem yourself. If you still cannot find your error, then you are encouraged to ask a question in person during office hours. As a last resort, you may make a private post to Piazza describing where you are stuck. Course staff will provide guidance directing you to the next step, but they will not give you the answer.

2. **Translate** the function definitions into Racket, using the names given. 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*). When translating fractions, treat them as though they had brackets surrounding them. For example $5 \cdot \frac{x}{2}$ is translated to (* 5 (/ *x* 2)) not (/ (* 5 *x*) 2). However, fractional numeric literals such as $\frac{1}{3}$ should be written as 1/3.

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

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

   you would submit:

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

One way that *mean* could be tested is to have the definition in the top pane of DrRacket's window (the *definitions pane*). Click the 'run' button. Assuming no errors are shown, type (*mean* 3 5) in the bottom pane (the *interactions pane*) and hit 'return'. It should print '4'. Similar actions can help you test the code you submit. We'll learn better testing methods when we look at the Design Recipe early in Module 03.

Place your solutions for the following functions in the file `translations.rkt`.

(a) Write a function, *volume*, to compute the volume of a sphere, as given by

$$\text{volume}(r) = \frac{4}{3}\pi r^3$$

where r is the sphere's radius.

(b) The *Fibonacci numbers* are one of the most famous number sequences in all of mathematics. They are defined by letting $F_1 = F_2 = 1$, and calculating all subsequent values via $F_{n+2} = F_n + F_{n+1}$. The first few elements are $1, 1, 2, 3, 5, 8, 13, 21, 34\ldots$

There are many different ways to compute the Fibonacci numbers, which all serve as good demonstrations of elementary computer programming principles. For now, we will simply use the following closed-form solution:

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}.$$

Here, φ is the golden ratio $\frac{1+\sqrt{5}}{2}$. Name it *phi* (the name of the Greek character).

Read the Racket documentation carefully to understand the difference between the built-in functions *exp* and *expt*.

Write a function *fib* that consumes an integer $n > 0$ and computes F_n using the formula above. Your answer should be an inexact approximation and should not attempt to round the result (for example, (*fib* 10) produces `#i54.9999999` instead of `55`).

(c) The speed needed by a rocket to escape a planet's gravity is called the *escape speed*:

$$\text{escape}(m, r) = \sqrt{\frac{2Gm}{r}}$$

where m is the mass of the body (e.g. planet) to escape and r is the body's radius. G is the gravitational constant $6.674 \times 10^{-11}\ \text{N} \cdot (\text{m}/\text{kg})^2$. This can be written in Racket as `6.674e-11`.

Notes:

- The units for the gravitational constant are provided for your information; they are not required as part of the function definition.
- The gravitational constant is always written as an uppercase G to distinguish it from g which is used to describe acceleration due to gravity.

Write the function *escape*.

(d) "Loudness" is frequently measured in terms of decibels (dB). For example, a TV might be measured at 60 dB, while a busy street might be closer to 90 dB. We hear sound because it exerts pressure on the ear drum; the decibel level is measured in terms of the ratio of this pressure to a reference level corresponding to the threshold of human hearing. Given a pressure P, the loudness L is given by

$$L = 20 \log_{10} \left(\frac{P}{P_{\text{ref}}} \right),$$

where P_{ref} is the reference level of 2×10^{-5} Pascals. (Recall that for any base k, $\log_k x = \log x / \log k$.)

Write a function *pressure→loudness* that consumes a sound pressure in Pascals and produces a measurement of loudness as given by the formula above. Note that we use $\rightarrow$ to look pretty on the page but you should use ->.

3. Body Mass Index (BMI) is a frequently used (albeit disputed) approximate measure of body fat in terms of mass (in kilograms) and height (in meters).

$$BMI(m,h) = \frac{m}{h^2}$$

(a) Write a function *body-mass-index* that consumes two numbers representing mass and height (in that order) and produces the corresponding BMI.

(b) For people who haven't gone metric, write an alternate function *body-mass-index-imperial* that calculates BMI from three numbers: weight (in pounds) and height (in feet and inches). For example, a person weighing 180lbs and measuring $5'10''$ would write (*body-mass-index-imperial* `180 5 10`). You are encouraged to write helper functions to convert pounds to kilograms (one pound is exactly 0.45359237 kilograms) and feet/inches to metres (one inch is exactly 0.0254 metres; there are 12 inches in 1 foot). You can then rely on your original BMI function.

Good style is often a matter of judgement. Here, it seems excessive to have both a named constant for 0.45359237 and a conversion function to convert pounds to kilograms. In this case, just write the conversion functions with well-chosen names.

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

4. 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.

There are a lot of "magic numbers" in this problem. Think carefully about how to handle them.

(a) 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 between 0 and 100, 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.

(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 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. This is intended to be a personal challenge for students in the course if they wish to try. Course staff will not answer questions about the Bonus either in person or on Piazza.

5. **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 (a number between 0 and 100 but not necessarily an integer). 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.

Place your solution in the file `bonus-a01.rkt`.

Note that bonus questions are typically "all or nothing". Incorrect or very poorly designed solutions may not be awarded any marks.

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 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 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.