Winter 2024
Victoria Sakhnini

Assignment:	1
Due:	Tuesday, January 23, 2024 9:00 pm
Coverage:	Slide 25 of Module 02
Language level:	Beginning Student
Files to submit:	`functions.rkt`, `conversion.rkt`, `grades.rkt`, `bonus-a01.rkt`

Assignment policies for this and all subsequent assignments

- The deadline for submission without penalty is stated above. More details are listed on the course website's assignment policies page.
- The work you submit must be entirely your own.
- Do not look up or generate 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.

Grading

- **You will not receive marks for *Ax* (where *x* is 01, 02, ...) unless you have earned full marks for A00 before the *Ax* due date.** Assignments may be submitted as often as desired up to the due date.
- Your solutions for assignments will be graded on both correctness and on craft[1].
- For this assignment, you do not need to do the design recipe for any question.

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. An automated email will also be sent to your UWaterloo email address with your basic test results.
- If you do not get full marks on the basic tests, then your submission may not be markable by our automated tools, and you will very likely receive a low mark 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.**

[1] the result of applying knowledge, skill, and care into creating a product

Craft

- Your "craft" will increase as you learn more about programming. We won't always be spelling it out like this. For now, "craft" consists of using DrRacket's "Reindent All" feature to provide consistent indentation, choosing meaningful identifiers, using constants when applicable, and providing some comments.
- See Sections 1 and 2 of the style guide for more details.

Here are the assignment questions you need to submit.

1. **(20%):** In Module 02 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 userid and password. Once you are logged in, you will see three sets of exercises related to Module 2: one on expressions, one on functions, and one on constants. Note the "Show instructions" link at the bottom of each problem. **Read the instructions!**

   There are both practice questions (which you do not need to complete for marks, but should complete for your understanding) and required questions (which you do need to complete for marks for this assignment). On the practice problems, there is a "Hint" button that is enabled; use it if you get stuck. The required questions do not have a "Hint" button.

   Complete the three required stepping problems in Module 2a, the two required problems in Module 2b, and the three required problems in Module 2c using the substitution rules given in class and in Module 02.

   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 exams will definitely have step-by-step evaluation questions.

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.

The Basic Tests will include a list of stepper questions that have not been completed. After completing more steppers, you'll need to request another Basic Test to see that list updated.

2. **(30%):** Computers are helpful tools in many fields of human endeavour. Below are functions that are useful to compute, taken from a variety of areas.

   Implement the function definitions below in Racket, using the names given. Place your solutions in the file `functions.rkt`.

   Note that when you are asked to **implement** a function, it is best practice for it to be a direct implementation, though it can be any equivalent implementation. For example, when asked to implement $(a+b)$, the direct implementation is `(+ a b)`, but the implementation `(+ b a)` would be a valid equivalent implementation. When implementing x^2, the direct implementation is `(sqr x)`, but `(* x x)` would also be a valid equivalent implementation.

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

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

   you could submit:

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

or:

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

   But the one on the left is more direct and would be the better choice.

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

$$\text{euclidean-distance}(x1, y1, x2, y2) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

(b) An example from music (*equal temperament*):

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

(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 *"direct implement"* the given function, not improve it.)

(c) Another example from geometry (*surface area of right circular cone*):

$$cone\text{-}area(r,h) = \pi r(r + \sqrt{h^2 + r^2})$$

Use the built-in value `pi`.

(d) An example from physics (*Lorentz Transformation Factor*):

$$B(v,c) = \sqrt{1 - \frac{v^2}{c^2}}$$

(e) An example from mathematics (*the partition function*):

A partition is a way of writing n as a sum of positive integers. For example $5+2+1+1$ is a partition of 9. Mathematicians Hardy and Ramanujan developed the following formula to approximate the number of partitions for the positive integer n:

$$\text{partition-size-approximation}(n) = \frac{1}{4n\sqrt{3}} \cdot e^{(\pi\sqrt{\frac{2n}{3}})}$$

Refer to the documentation for the difference between the Racket functions `exp` and `expt`.

(f) An example from finance (*investment returns*):

The future value of an investment of principal p with compound interest rate r after t years compounded n times per year:

$$\text{future-value}(n,p,r,t) = p \times (1 + \frac{r}{n})^{nt}$$

3. **(30%):** As you may know from physics, the constant 9.80665 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![2]

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

(b) You may feel some pressure in this course. The International System of Units (SI) uses Pascals (yes, like the triangle) as the unit of pressure, which is actually $\frac{\mathrm{N}}{\mathrm{m}^2}$. In other words, one Pascal is one Newton of force applied to one square metre of area.

The Imperial Standard unit is pounds per square inch (PSI), which is more accurately pounds-force per square inch (lbf/in^2). One pound-force (lbf) is 4.4482 Newtons. One foot can be viewed as 0.3048m, or also as 12 inches.

Write a function `psi->pa` that consumes a measure of pressure in PSI and produces the same pressure in Pascal units. Your solution should use the three constants listed above in it's calculations. As a good test case, you should ensure that your function produces $6894\frac{11674}{16129}$ when it is called with 1.

(c) You might also feel slightly torqued in this course too. Similarly to the previous part, torque can be measured in two different units: the pound foot (lbf· ft) or in the newton metre (N · m). Write a function `lbf-ft->Nm` that consumes a measure of torque in pound feet and produces the same measure of torque in newton metres. You must reuse some constants from the previous part of this question. As a good test case, your function should produce 67.790568 when it is called with 50.

(d) 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.

4. **(20%):** Place your solutions to this question in the file `grades.rkt`.

(a) The end of the term has come and you decide to use Racket to calculate your final grade in CS 135. Your final grade depends on your participation marks and 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):

[2] https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf, p.16

1. the class participation grade,
2. the 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. Assume that we follow the "in-person" mark allocation for the term. Although it is not necessarily true in general, for this part, 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:

1. the class participation grade,
2. the midterm grade, and
3. the overall assignments grade.

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 in part (a), assume that the in-person marking scheme is followed.

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.

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

5. **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 02. 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 1.6 of this page:
   `https://docs.racket-lang.org/htdp-langs/beginner.html`

   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. Course staff will not provide any help for bonus questions.

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 *Intermediate Student with Lambda*.

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. (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.