Homework: Simple functions and conditionals

COMP 50

Fall 2013

This homework is due at **11:59PM on Monday, September 16**.
If all goes well, you will download an extension to DrRacket that enables you to submit your homework direct from DrRacket. If not all goes well, Plan B will be announced on Piazza.

All the exercises should be done using the *Beginning Student Language* on the "choose language" option DrRacket's Language menu. You can use the same menu to add teachpacks; when you do, the display in the lower left should read `Beginning Student custom`.

Finger Exercises

For this homework, I am recommending the following finger exercises:

- Exercises 2.3.1 and 2.3.2 in the book.
- Write function `years->months` which converts a number of years to a number of months.
- Write function `radians->degrees` which converts an angle in radians to an angle in degrees.
- Write a function that takes an image and returns the string "landscape" if the image is wider than it is tall, "portrait" if the image is taller than it is wide, and "square" if the image is as equally tall as it is wide. You will need to add the `image.rkt` teachpack from the `2htdp` teachpacks.
- Write a function that is given the value of $\cos\ \theta$ and that returns the great-circle distance in *meters* between two points on the Earth's surface that are separated by the angle θ.

  Domain knowledge: According to the official "geoid datum", the Earth's radius is deemed to be 6378137.0 meters. The Earth is official not a sphere, but you should treat it as such.

  Racket knowledge: using the Racket Documentation button on the Help menu, search for and understand the function `acos`.

Problems to submit

1. Define function `dd/mm/ss->radians` which converts an angle in degree-minute-second form ($DD^\circ\ MM$" $SS.SSS$') to radians.
2. Since the time of the ancient Greeks, people have used geographic latitude as an indicator of climate. The old latitude models have been superseded by more sophisticated models that take account of terrain, winds, and ocean currents, but they are still not a bad approximation. Using the following table, define a function that converts a latitude in degrees to the name of a climate:

Latitude range	Expected climate
below 23.5	tropical
23.5 to 35	subtropical
35 to 50	temperate
50 to 70	subpolar
above 70	polar

3. Century Bank in Medford offers certificates of deposit at the following interest rates:

Term	Rate
3 month up to 6 months	0.25%
6 months up to 1 year	0.35%
1 year to 2 years	0.50%
2 years to 3 years	0.80%
3 years to 5 years	0.90%
5 years	1.00%
36 months (*for new customers only*)	1.34%
48 months (*for new customers only*)	1.83%
60 months (*for new customers only*)	2.03%

   Define two functions, one for regular customers and one for new customers, each of which takes a given term in months and returns the interest rate available for a certificate of deposit with that term.

4. Using teachpack `guess.rkt`, available from the Language menu (Add Teachpack), do Exercise 5.1.2 in the textbook.

5. A Labrador retriever will eat anything. And if a piece of orange chicken is dropped on the floor, within the first 100 milliseconds, the Lab will close 10% of the distance between itself and the chicken.

   Using the `2htdp/image` teachpack, design a *program* that illustrates the first step taken by the Labrador retriever:

   - The retriever and the chicken should be located in an empty room that measures 300 by 300 pixels.
   - The main function should take two `posn` values called `dog` and `food`. It should produce an image in which
     - The dog is represented by a solid black circle.
     - The chicken (`food`) is represented by an orange dot.
     - The expected trajectory of the retriever is shown as a green line.
     - The location where the retriever will be in 100 milliseconds, which is 10% of the distance along its trajectory, is shown as an outlined black circle.

   Here is a sample output:

   Domain Knowledge (Geometry): To find the black circle between the dog and the food, compute 10% of the *signed* distance in each dimension and add it to the coordinates of the dog. To be sure you get the correct sign of the distance, you'll want to consider several configurations.

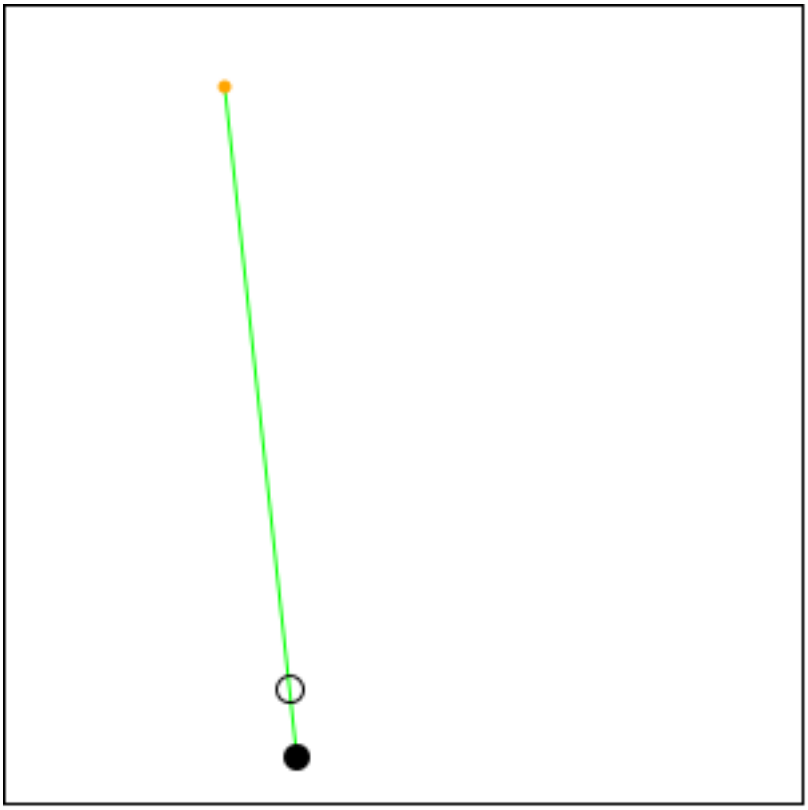

Figure 1: Sample retriever step

Domain Knowledge (Computer Graphics): By convention, in computer graphics, larger values on the y axis are *down*.

Domain Knowledge (Dogs): the shortest distance between a dog and food is a straight line.

Follow the design recipe for programs in Section 3 of the textbook. You may find it helpful to consult Section 3 of the second edition, especially Section 3.4 (From Functions to Programs).

Karma problems

A. Change the traffic-light animation from the lecture (which is very similar to an example from the second edition) so that the relative times for red, green, and yellow lights behave more like real traffic lights. You will need to use the `big-bang` function in the `2htdp/universe` teachpack. You will also need to define a *world state* that knows not only what color the light is but also how much time is left before the light is due to change.

B. Using the `big-bang` function in the `2htdp/universe` teachpack, animate the Labrador retriever going for the food. Have the retriever cover 10% of the remaining distance at each step. [1] When the dog closes to a distance within 5 pixels of the food, make the food disappear and stop the animation.

Resources

For information on the `2htdp/image` and `2htdp/universe` teachpacks, you can go directly to the Racket Documentation. (We will also do some image things in lab.) For a more leisurely introduction,

- Section 1.4 of the second edition give a thorough introduction to images.
- Section 3 of the second edition is primarily about designing programs as collections of functions, but it has a great many examples of `big-bang`.

You can develop examples any way you want, but **your submitted examples must be formulated as test cases.** Use `check-expect` or `check-within`. Put your tests in with your definitions.

How your work will be evaluated

We will evaluate your work by judging it against the table of criteria below. The table may look big and intimidating, but it boils down to these issues:

- Did you use Beginning Student Language with suitable teachpacks?
- Is the code well formed? That is, is it laid out nicely on the page?
- Are the names well chosen?
- Have you followed the design recipes?
    1. Are there data descriptions where needed? For conditional functions, does the code show that all possible situations have been enumerated?
    2. Does every function have an *appropriate* signature (called "contract" in the first edition), purpose statement, and header?
    3. Did you come up with examples?
    4. Has the function body been developed by following a template that is suited to the problem and to the class of data being consumed?

[1] As the dog closes in on the food, the dog slows down. This is not how real dogs behave.

5. In the coding step, have you carried out the purpose statement and only the one purpose statement?
6. Does your submission included test cases derived from your examples using `check-expect` and/or `check-within`? Are the test cases sufficient to handle every situation from your analysis and every clause in every relevant data description?

Especially as you get started, I urge you to get a member of the course staff to examine your work *before* the deadline, to make sure you know how to follow the design recipe. You can come to office hours or post *private* questions on Piazza.

	Exemplary	**Satisfactory**	**Must Improve**
Language and libraries	• The solution uses Racket's Beginning Student Language, and where needed, the `2htdp/image` and `2htdp/universe` teachpacks.	• The solution uses additional teachpacks. • The solution uses Racket libraries that are not teachpacks.	• The solution uses the wrong language.
Form	• All code fits in 80 columns. • *Or*, almost all code fits in 80 columns, and all code fits in 90 columns. • Indentation is consistent everywhere. • All code respects the *offside rule*: any code enclosed by parentheses or brackets appears to the *right* of the opening parenthesis, even if the code is split across multiple lines. • There is never space immediately inside a parenthesis or bracket. • Except where the outside of a parenthesis is on the inside of another parenthesis, there is whitespace to the outside of every parenthesis. (The start and end of a line both count as whitespace.) • No code is commented out.	• Several lines are wider than 80 columns. • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • The code contains one or two violations of the offside rule. • Conventions regarding whitespace and parentheses are violated in more than three places.	• Many lines are wider than 80 columns. • There is a line wider than 90 columns. • The code is not indented consistently. • The code contains three or more violations of the offside rule. • Solution file contains code that has been commented out.

	Exemplary	Satisfactory	Must Improve
Names	• Significant constants, whether mentioned in a problem statement or arising elsewhere, are given names (as `define`d variables) • Each function is named either with (1) a noun describing the result it returns, (2) a verb describing the action it does to its argument, (3) a word or words describing a relationship among the arguments, (4) two nouns connected by an `->` arrow, or (5) a property followed by a question mark. • In each function definition, the name of each parameter is a noun saying what, in the world of ideas, the parameter represents. • Or the name of a parameter is the name of an entity in the problem statement, or a name from the underlying mathematics. • The name of `define`d variable is a noun saying what, in the world of ideas, the varaible represents. • Or the name of a varible is the name of an entity in the problem statement, or a name from the underlying mathematics.	• One or two important constants are not named. • Functions' names contain appropriate nouns and verbs, but the names are more complex than needed to convey the function's meaning. • The name of a parameter is a noun phrase formed from multiple words. • Although the name of a parameter is not an English noun and not a name from the math or the problem, it is still recognizable—perhaps as an abbreviation or a compound of abbreviations. • The name of a variable is a noun phrase formed from multiple words. • Although the name of a variable is not an English noun and not a name from the math or the problem, it is still recognizable—perhaps as an abbreviation or a compound of abbreviations.	• No important constants are named. • Function's names include verbs that are too generic, like "calculate", "process", "get", "find", or "check" • Helper functions are given names that don't relate to their purpose statements but that instead indicate a vague relationship with another function. • Course staff cannot identify the connection between a function's name and its purpose statement. • The name of a parameter is a compound phrase which could be reduced to a single noun. • The name of some parameter is not recognizable—or at least, course staff cannot figure it out. • The name of a variable is a compound phrase which could be reduced to a single noun. • The name of some variable is not recognizable—or at least, course staff cannot figure it out.
Data description (design recipe step 1)	• For every clause in the data description, course staff can use the clause to create examples of that kind of data. The examples are acceptable to DrRacket.	• There is one clause in one data description for which the course staff are unable to come up with examples.	• The solution contains more than one data-description clause for which the course staff are unable to come up with examples.

	Exemplary	Satisfactory	Must Improve
Signature, purpose statement, and header (design recipe step 2)	• Each function comes with a signature (called a "contract" in the first edition) that is consistent with the function's header. • Whenever a function's signature refers to a kind of data (sometimes called a *class* of data), that data is either atomic data built into BSL or it is explained in an explicit data description included in the solution. • From each function's purpose statement, it is easy to see how each parameter affects the result. • Each function's purpose statement fits in 90 characters or less. • Each function's purpose statement says only *what* the function does, not *how* • Function signatures and purpose statements are sufficiently precise that course staff can independently create examples of what the function is supposed to do.	• Each function comes with a signature, which always has the correct number of parameters and is usually but not always consistent with the function's header. • Some functions have a signature that uses a different name from the function's header. • A function's signature mistakenly refers to atomic data built into BSL, but a more restrictive data description is needed. • A function's purpose statement mentions every parameter, but the course staff cannot easily see how each parameter affects the result. • Some function's purpose statement is over 90 characters, but it fits in 140 characters. • Some function's purpose statement contains information about *how* the function works, not just what it does. • Function signatures and purpose statements have a little bit of wiggle room, but course staff can narrow down potential examples to just a few possibilities.	• Not every function has a signature. • Some functions have signatures with the wrong number of parameters. • Less than half the functions have signatures (**serious fault**). • A function's signature refers to a class of data that is not atomic data built into BSL, and there is no data description. • A function's purpose statement does not mention every parameter. • Some function's purpose statement is over 140 characters. • Some function's purpose statement narrates a sequence of events that occurs when the function is called. • Course staff are unable to use function signatures and purpose statements to create independent examples.

	Exemplary	Satisfactory	Must Improve
Examples and tests (design recipes steps 3 and 6)	• Every function, including image-building functions, is covered by tests using `check-expect` or `check-within`. • Every function should be covered by tests, but not every function is covered because some of the tests are written using plain calls instead of `check-expect` or `check-within` (minor deduction) • For every function that uses `cond`, for every alternative within every `cond`, there is a test case using `check-expect` or `check-within` that exercises the alternative. • Test cases using `check-expect` or `check-within` are placed either immediately before or immediately after the definitions of the functions they test. • Submitted test cases pass.	• Every function, except possibly some image-building functions, is covered by tests using `check-expect` or `check-within`. • Functions include examples in informal English that could be expressed using `check-expect` or `check-within`, but aren't. • One `cond` is not fully tested. • Test cases appear before the signature (aka contract) and purpose statement of a function. • One or more submitted test cases fails because of small inaccuracies in arithmetic.	• Some functions are defined without examples or tests. • There are multiple `cond`s that are not fully tested. • All test cases are gathered together into one lump that is separated from the functions they test. • One or more submitted test cases fails. • *Or*, test cases (or examples) appear to be present, but they don't use `check-expect` and friends. • Racket `Run` does not report passing any tests, and the course staff cannot find anything that looks like a test case (**serious fault**).

	Exemplary	Satisfactory	Must Improve
Function bodies (design recipes stesp 4 and 5)	• Solutions to large problems are tackled by decomposition that uses one or more helper functions. • Each function body is small and simple enough that the course staff can easily verify whether the body meets the obligations set out in the purpose statement. • Each function is about 5 to 9 lines of Beginning Student Langauge. The line count is achieved through simple structure, not by cramming a lot of stuff on one line. • *Or*, some functions are larger than 5 to 9 lines, but the extra size is forced on the function because of a data description that has many alternatives. • The body of every function follows the template(s) determined by the *input data* of that function. • Every `cond` contains a case with an explicit test for every alternative. An alternative may be a situation in a problem statement (as described in the design recipe for conditionals) or it may be an alternative clause in a data description.	• Solutions to large problems do not use helper functions, but they are so clean that the staff can follow them anyway. • Course staff have to work to tell whether the code is correct or incorrect. • Some functions of ten or more lines, but course staff can follow them. • There is a function whose body almost follows the template determined by the input data, but the course staff sees where one or more shortcuts have been taken. • A `cond` uses `else` instead of explicit tests.	• Solutions to large problems do not use helper functions. • From reading a function body, course staff cannot tell whether it is correct or incorrect. • From reading a function body, in the context of its signature and purpose statement, course staff cannot easily tell what it is doing. • One or more functions of 20 or more lines. • So much stuff is crammed into a small number of lines that the course staff have a hard time following the code. • The course staff had a hard time following your code (staff will say what they saw and where) • There is a function whose body does not follow the template determined by the input data, and the course staff cannot see the relationship. • A `cond` mixes up multiple situations from a problem statement or multiple clause from a data description. • *Or*, problem situations and data-description clauses are mixed up in a single `cond` • *Or*, there is a `cond` in the code that is not justified by an analysis of input data.