Homework 2: Warm Up

Due: 10:00 PM, Sep 19, 2017

Contents

Objectives

By the end of this homework, you will understand:

1. the course missive
2. the course collaboration policy
3. `if`, `cond`, and predicates

By the end of this homework, you will know how to:

1. visit the TAs on hours

2. read messages from the course mailing list
3. carry out simple abstractions
4. write simple user-defined procedures in Racket

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. **For this homework specifically, this entails checking your CS email, going to TA hours, and answering the Communication, Course Policies, Computations, Translate Pet, Conditionals, What If?, Comparing Functions, and BNF questions.**

To hand in your solutions to these problems, you must store them in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., `communication.txt` corresponds to CS 17 Communication):

- `communication.txt`
- `course-policies.txt`
- `polygons.rkt`
- `summations.rkt`
- `translate-pet.rkt`
- `my-positive.rkt`
- `thermometer-dir.rkt`
- `what-if.rkt`
- `comparing-functions.txt`
- `bnf.txt`

For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension `.rkt`. Everything else should be a text file containing text only (no code!), and ending with extension `.txt`. All your solution files should reside in your `~/course/cs0170/homeworks/hw02` directory.

For this and every future assignment, you should also have a `README.txt` file whose first line contains only your CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

```
README.txt:
jfh@cs.brown.edu
There's nothing to say except that I'm turning in four code files plus this
README the way the instructions say that I should.
```

To hand in your solutions to these problems, you must zip your `hw02` directory into a file `hw02.zip` (instructions to do this can be found in the Homework 1 document).

Hand in this zip file using the method you learned in the first lab.

Design Recipe

Whenever we ask you to "write a procedure," you are expected to carry out each and every step of the design recipe, which will evolve over the next week or two. For convenience, here are the steps of the design recipe:

1. Data definition
2. Examples of the data
3. Type signature
4. Call structure
5. Specification
6. Test cases
7. Template
8. Code
9. Run your program

For more information, you can reference the Design Recipe post on Piazza, or the Design Recipe Guide on the course website.

Practice

On some homeworks, you will be given practice problems. You are not required to complete these, nor are you required to turn them in, but you should know *how* to solve them all.

We recommend that you use the practice problems to test your knowledge. (That's why we give you so many.) Look at a few. If they make sense, you should be fine; if not, try some more; and if you're still stumped, come to TA hours and your friendly TAs will help you work through them. We *strongly* encourage you to make sure you can do all of them.

1 Prefix (Practice)

Convert the following algebraic expressions using *infix* notation (which you know from algebra class, where the operator is between the operands, like $3 + 5$) into Racket expressions using *prefix* notation (notation where the operator precedes the operands, like `(+ 3 5)`). Follow the standard order of operations for arithmetic that you learned way back when.

Hint: No two of these expressions have quite the same translation!

- $4 \times 3 + 2$
- $4 \times (3 + 2)$

- $4 - 1 + 3 \div 2$
- $4 - (1 + 3) \div 2$
- $(4 - (1 + 3)) \div 2$
- $60 \div 5 - 6 + 2 \times 4$
- $60 \div (5 - 6) + 2 \times 4$
- $60 \div 5 - (6 + 2) \times 4$
- $(60 \div 5 - (6 + 2)) \times 4$
- $60 \div (5 - (6 + 2) \times 4)$

2 Evaluation (Practice)

Describe the output that results after each of the following expressions and definitions is entered into DrRacket. Assume they are entered sequentially in a DrRacket interactive session, so that names remain defined throughout the session. If an error is ever raised, explain why.

Note that this sequence involves interleaved definitions and expressions, and involves multiple top-level expressions rather than just one at the end, so it doesn't quite match up with the language we've been describing in class. After this exercise, we'll never again interleave definitions and top-level-expressions. As for the "multiple top-level expressions," you can treat that as a very slight generalization of what we've described: rather than processing the sole top-level expression and printing out the printed representation of its value, we do this for each top-level expression.

Finally, if one of the lines we ask you to process results in an error, you should continue processing the rest of the lines as if that line was never entered (which is what DrRacket will actually do if you enter these lines in the interaction window).

Do this exercise *without* using DrRacket!

- `(+ 1 1)`
- `((+ 1 1))`
- `(+ 1 true)`
- `(+ 1 one)`
- `(define 1 one)`
- `(define one "1")`
- `(+ 1 one)`
- `(define one 16)`
- `(+ 1 one)`
- `(define a (+ 3 4))`
- `(define b (+ 2 a))`

Problems

3 CS 17 Communication

Task: There is a post to the mailing list with the subject line "Magic Phrase." Copy and paste the contents of this e-mail into `communication.txt`.

Task: Before this homework is due, go to TA hours and introduce yourself to one of the TAs. Tell them about any pets you may have! This is an opportunity for you to meet the TAs and learn what hours are like.

Note: It is your responsibility to make sure the TA you talk to checks you off on the spreadsheet of students who have attended hours.

Also, if you own a laptop, you can ask the TAs to help you to set up working from your laptop while you're home.

4 CS 17 Course Policies

Task: The CS 17 course missive, collaboration policy, and guides to TA hours and pair programming can all be found on the course website. For each of the following situations involving fictional CS 17 students, decide whether their actions are appropriate per the course policies, and explain why or why not.

Note: The response for the first situation is provided to give you an example of the level of explanation expected in your responses.

1. To get closer to her classmate Steve the monkey, Snowball the gerbil decides to work on a lab with him. As they get settled, Steve mentions that he'd prefer to drive. "What a coincidence!" Snowball exclaims, as she says that she prefers navigating because her paws are tiny and she is a slow typer as a result. The two know it's important to get experience in both roles, so they decide to spend 30 minutes in their preferred role and 15 in the other as they pair program the lab.

   Solution: This violates the collaboration policy. The expectation is that partners pair program as outlined in the pair programming handout. This includes spending an equal amount of time in both roles.

2. Carey the dog and her friend Bella the cat partner up for the Rackette project. They enjoy working on the design check together, and decide to pair program the code. Carey then sees an open gate on a walk and bolts for it, gets distracted by squirrels, and then gets horribly lost. She misses a few meetings. She then misses another scheduled meeting because a turkey dinner was left too close to the edge of the counter. Bella doesn't want to fall behind so she codes a few parts of the project herself. Carey makes it to their last two meetings, and they finish up and submit their project together.

3. Suki the chinchilla and Amir the frog partner up for Lab 06. Suki missed a recent lecture because of an appointment with her veterinarian, so she asks Amir if he minds if they look over the notes together before working on the assignment. After five minutes or so, they get

to work. When Suki is driving, she takes Amir's suggestions and flips to the lecture notes as necessary to make sure she understands everything. When Suki is navigating, she asks Amir to explain code that's unclear.

4. Sonya the horse was having a bad week because it rained every day, and she couldn't go outside. She had put off working on HW 8 until the very last minute . . . but then she remembered she had a horse show to attend. The next day, she finished the homework and turned it in late, hoping for partial credit and figuring that one day late is better than never.

5. José the fish is having some serious problems with this week's homework. While sitting in the Sun Lab aimlessly browsing through directories, he happens to discover that Shelly the turtle has world-readable permissions set on his course folder. Excited that he can now read the contents, José finds that Shelly hasn't yet completed the homework, but has completed the first problem, and written some notes about possible approaches to the remaining problems in code comments. José exits out of the folder and does the homework on his own.

6. Elijah the snake is having trouble with a homework problem, so he asks Sophie the gecko for help. Sophie offers to meet him in the SciLi study center. They meet up and, with Sophie leading most of the discussion, come up with a solution on the white board. Afterwards, they erase the white board and head to their own rooms to write up "their" solution.

7. Kitty the cat bumps into Michael the parakeet at the V-Dub during the final exam. Kitty mentions that the last problem is a real pain. Michael tells her that he disagrees, but that they're not supposed to talk about the exam and he'd rather not discuss it further.

8. Buster the pug has had a difficult time trying to get help at hours this week, because everyone there keeps laughing at his squishy face and wheezing. He sees a TA at the Ratty during lunch and asks her a question about the homework, figuring that he should take the opportunity to ask for help without having to wait in line and endure the ridicule.

9. Mahdi the mouse, hiding in the walls of Spike's home, loses track of what day it is. He comes out of the wall and realizes that his first project is due extremely soon, but it's way too late to work on it with his partner. He thinks about emailing Spike for an extension, but decides to email the TA Staff to ask instead because he feels bad about hiding in Spike's walls.

10. During Lab 02, Gabriela the canary and Sarah the guinea pig work together and become good friends, bonding over their shared ex-boyfriend, Christopher the fox. They meet up the next day in the Ratty and reminisce about how great of a partnership they had. Gabriela understands Sarah in ways that Kaylah the rabbit, her partner during Lab 01, never could! They can't wait until the next lab session, when they will have a chance to work together again! When their next lab finally rolls around, they make a beeline for the first open computer and start working together.

Turn in a signed collaboration policy, if you have not already done so in lab! We will not grade any of your further assignments until we have your signed collaboration policy.

5 Computations

5.1 Polygons

Task: Write a procedure `sum-angles` with the call structure `(sum-angles n)` that, when given an integer $n \geq 3$, calculates the sum, in degrees, of the interior angles in an n-gon (i.e., a polygon with n sides). The formula for this is $180(n-2)$.

Before you write this procedure, figure out what values it should output on some sample inputs, say 3, 8, and 11. You can use any trusted means other than your procedure itself to compute these outputs. (In fact, you cannot use your procedure even if you tried, because you have not written it yet!) After you write your procedure, run it on those inputs and verify that the output is correct.

Here's a hint to get you started.

```
;; Example Data:
;; int: 3, 5, 8

;; sum-angles : int -> int

;; Input:  a positive integer, n, greater than 2
;; Output: the num of degrees in an n-sided polygon

(define (sum-angles n)
        ...)

;; Test cases for sum-angles
(check-expect (sum-angles 3) 180)
;; Write more tests here!
```

5.2 Summations

Task: Write a procedure that, for an integer $n \geq 1$, computes the sum of all numbers from 1 to n, inclusive, i.e., $1 + 2 + \ldots + n$, for which there's a formula (Gauss figured it out):

$$1 + 2 + \ldots n = \frac{n(n+1)}{2},$$

which is the formula you'll want to use in your procedure.

Before you write your procedure, pick some sample input values (maybe 1, 6, and 1000), and compute the output you expect on these inputs. Then, after writing your procedure, make sure your procedure outputs what you expect.

Hint: The design recipe for this problem should be very similar to the problem just before.

6 Translate Pet

Task: Write a procedure `translate-pet` that, when given the string "dog", "cat", or "fish", returns the equivalent Spanish word. The input "dog" should return "perro", "cat" should return "gato", and "fish" should return "pez".

Note: When determining the equality of two strings, you should compare using the `string=?` procedure (this is required by the style guide!)

Hint: Part of the point of this question is to give you practice with using the design recipe for different types of data. Consider your domain. Do you need to restrict it? And if so, remember that you don't need to test outside of those restrictions.

7 Conditionals

Write the following procedures using `cond`, `and`, `or`, and `not`, as appropriate.

- `my-positive?`, with call structure `(my-positive? n)`, which returns a boolean indicating whether the integer `n` is positive or not. (**Note:** 0 is neither positive nor negative.)
- `thermometer-dir`, with call structure `(thermometer-dir t1 t2)`, which takes in two floats: the starting temperature `t1` and the ending temperature `t2`. It returns a string from among `"rising"`, `"falling"`, and `"steady"` according to whether the temperature is rising, falling, or holding steady.

8 What if?

A close cousin of `cond` is another Racket construct, `if`. Like `cond`, an `if` expression is used to evaluate different subexpressions depending on the result of some predicate. An `if` expression has the following shape, where `condition` must be an expression whose value is a boolean, but `result-t` and `result-f` can be arbitrary Racket expressions:[1]

```
(if condition
  result-t
  result-f)
```

In fact, every `if` expression is functionally equivalent to a special kind of a `cond` expression:

```
(cond [condition result-t]
      [else result-f])
```

By "functionally equivalent," we mean that we would get the same result by evaluating either one.

An `if`-expression has the shape

`(if condition expression1 expression2)`

where `condition`, `expression1`, and `expression2` are all expressions.

An `if`-expression is evaluated by:

(a) Evaluate `condition`, whose value must be a boolean.

[1]Although Racket does not enforce this requirement, it is good practice to make sure that *result-t* and *result-f* are of the same type, i.e., that both evaluate to numbers, or that both evaluate to strings, or that both evaluate to procedures, etc.

(b) If that boolean is **true**, then evaluate `expression1` to get a value v; the value of the **if**-expression is then v.

(c) If that boolean is **false**, then evaluate `expression2` to get a value v; the value of the **if**-expression is then v.

Notice that in case "b", `expression2` is not evaluated, and in case "c", `expression1` is not evaluated. These non-evaluations are the reason that **if** cannot be a procedure: in the course of evaluating a procedure-application expression, all the expressions get evaluated. Therefore, **if** is a keyword, and these rules for **if**-expression evaluation get added to our general rules of evaluation.

A more concise (but equivalent) way to evaluate an **if** expression is to say that the value of such an expression, when the condition evaluates to **true**, is the value of the first result, and the value of the expression, when the condition evaluates to **false**, is the value of the second result.

Here are some more examples:

```
(if true "apple" "banana")
=> "apple"

(if (= 17 15) "apple" "banana")
=> "banana"
```

Why would you use **if** instead of **cond**? Just to make your code easier to read in certain situations. After you have written a procedure, if you notice that you have a **cond** expression with only one condition and an **else**, then you might choose to rewrite it as an **if**. There are, however, situations where even with just two cases, a **cond** is *required* by the style guide. Refer to the style guide for a discussion on when you should use **cond** vs. **if**.

Task: Follow the design recipe to write a procedure that, given an integer, returns the string:

```
"This number is zero."
```

if the number is 0; and, otherwise it returns the string:

```
"This number is not zero."
```

Call your procedure `is-zero?`, and be sure to write it using an **if** expression.

9 Comparing Functions

This problem is about functions from the positive integers to the reals, i.e., functions we write in the form

$$f : \mathbb{Z} \to \mathbb{R} : x \mapsto \ldots$$

If f and g are two such functions, and we compare $f(1)$ to $g(1)$, $f(2)$ to $g(2)$, and so on, we may notice that eventually all the f-values are larger than the corresponding g-values. Perhaps $f(1) < g(1)$ and $f(5) < g(5)$, but for $n = 6, 7, 8, \ldots$, we find $f(n) > g(n)$. In this situation, we'll say that "f is *eventually larger than* g". The number 6 in this example is completely arbitrary. Maybe for some other pair of functions, which I'll again call f and g, f and g trade off which is higher for numbers less than 113, but then for all numbers larger than 113, we have $f(n) > g(n)$. We still say that f is *eventually larger than* g.

The formal definition is this:

If $f, g : \mathbb{N} \to \mathbb{R}$ are functions, and there's some number M with the property that whenever $n \geq M$, we have $f(n) > g(n)$, we say that "f is *eventually larger than* g".

This is a new definition of a new term; we've taken an ordinary English phrase and given it a very specific meaning, and for the remainder of this problem, we'll abandon whatever other meaning we might have thought it had in the past. "Eventually larger than" means the thing in the previous paragraph.

Let me give a concrete example. Suppose that

$$f : \mathbb{Z} \to \mathbb{R} : n \mapsto 3n - 1$$

and

$$g : \mathbb{Z} \to \mathbb{R} : n \mapsto n + 7.$$

Then I claim that f is eventually larger than g. To convince you of this, I have to show you a number M with the property that whenever $n \geq M$, we have $f(n) > g(n)$, i.e., we have

$$3n - 1 > n + 7.$$

To do this, I'm going to graph the two of them, using a great online graphing tool called Desmos. I navigate to `https://www.desmos.com/calculator` and enter in the formulas for f and g, making sure to use x, rather than n so that the functions will actually appear on the graph.

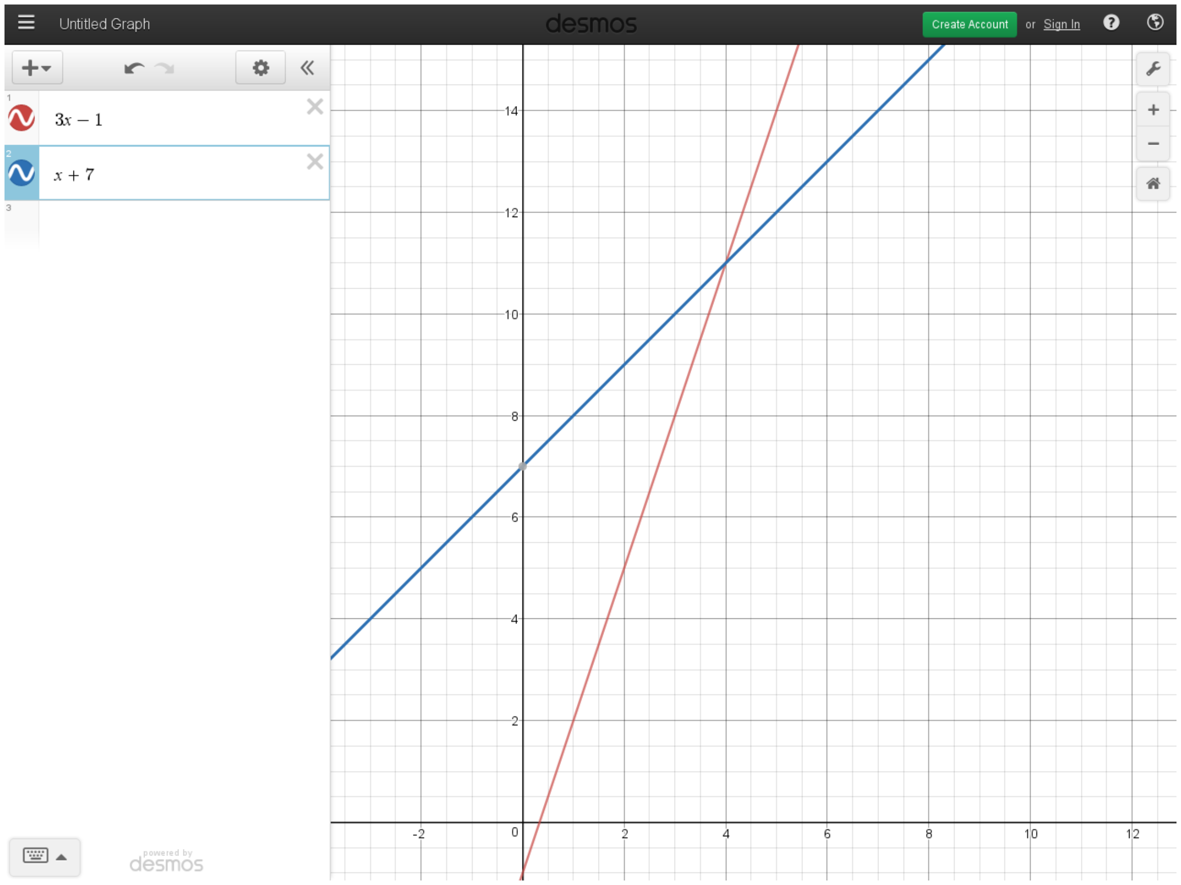
When I follow these instructions, I see that the graph of f rises above the graph of g just after $n = 4$. For any $n \geq 5$, we can see that $f(n) > g(n)$. So to show that f is eventually greater than g, I'm going to say "the number M = 5 proves it!"

Let me pause and make two observations here:

- This isn't an actual proof. To do a real proof would require a bit of algebra. But it's probably pretty convincing to you, and for this homework, I'm going to say that this is good enough.
- I didn't have to pick $M = 5$. If I'd said "Look at $M = 10$, folks!" you would have agreed that for numbers $n = 10$ and larger, we certainly have $f(n) > g(n)$. We also have that inequality for some smaller numbers, but the definition doesn't care about those, so exhibiting $M = 10$ is just as good as exhibiting $M = 5$. You might think that $M = 5$ is "the best" answer, but it's really not any better that any of the other possible choices, except that it doesn't require you to graph as much of the function.

Task: For each of the following pairs, however, I want you to find the *smallest* possible *integer* value of M that's good enough to stand as a witness to show that f is eventually greater than g.

a. $f : \mathbb{Z} \to \mathbb{R} : n \mapsto 3n$
$g : \mathbb{Z} \to \mathbb{R} : n \mapsto 2n + 5$



b. $f : \mathbb{Z} \to \mathbb{R} : n \mapsto 5\sin(n) + \frac{n}{3}$
   $g : \mathbb{Z} \to \mathbb{R} : n \mapsto 1 + \ln n$

c. $f : \mathbb{Z} \to \mathbb{R} : n \mapsto n^{1.08}$
   $g : \mathbb{Z} \to \mathbb{R} : n \mapsto n + 2$

d. $f : \mathbb{Z} \to \mathbb{R} : n \mapsto 2^n - 100$
   $g : \mathbb{Z} \to \mathbb{R} : n \mapsto n$

Note: Desmos is a great plotter, but it's not really plotting the functions that we've described. Our functions take, as arguments, i.e., elements of the domain, *only natural numbers*, so their graphs consist of lots of disconnected points. Desmos assumes that the domain is all real numbers, so it draws continuous lines for the graphs and shows the plot for negative as well as positive elements of the domain. The distinction has no impact on this problem, but is worth remembering.

10 BNF

For this problem, we ask you to practice your skills with Backus Naur Form (BNF).

Here's a current BNF description of the part of Racket we've encountered in lecture so far. Each of the following represent one or more named entities (an entity is one of the things in the angle-brackets) in this BNF. For instance, (**define** a 3) is a definition (defn), but it's also a name-definition (name-defn).

```
        <program> ::=  <defn>* [<top-level-expr>]
        <defn> ::= <name-defn> | <proc-defn>
        <top-level-expr> ::= <expr>
        <expr> ::=  <number> | <string> | <name> | <proc-app-expr>
        <name-defn> ::= (define <name> <expr>)
        <proc-defn>  ::= (define (<name> <name>*) <expr>)
        <proc-app-expr> ::= (<expr> <expr>*)
```

Task: For each piece of program text, say the most specific entity it matches in the BNF. A typical answer might be "name-defn", as with the example above.

a. "CS17"

b. (+ 1 3)

c. (**define** (add3 num)(+ num 3))

11 Challenge problem (not graded): BNF and types

In writing type-signatures, we've told you to only use certain types: `int, float, string, bool`, which are not really Racket types, but they're close enough for our purposes, and they *are* types in OCaml, a language in which writing type-signatures is particularly important.

I've told you can combine these by putting asterisks between them, so that a function that combines an `int` and a `string` to produce a `bool` gets described by

```
int * string -> bool
```

And this illustrates one more part of this notation: the arrow, formed by a hyphen followed by a greater-than sign.

Pretty soon, we'll have something called *lists*, and a list full of integers is called an `int list`. There's a small ambiguity that arises. Does

```
string * int list -> bool
```

indicate a procedure that consumes a string and an int-list to produce a bool, or does it indicate a procedure that consumes a list full of string-int pairs and produces a bool? To disambiguate, we'll allow the use of parentheses around any type, so we can write `(int list)` or `(int)* (string)`, although that last one means the same thing as `int * string`.

Task: Think of everything you've encountered so far in Racket, and try to write down a type-description of each thing, like "I saw the number 4, and it's an `int`, or maybe a `float`; I saw the string "A", and it's a `string`, ..." You only have to do this once per type (i.e., you don't have to say that the number 3 was also an `int`), and you don't have to worry about `list` yet, because it's not part of what you've seen, except the brief mention in this problem.

Note: You've encountered functions like the builtin addition procedure that can take 0 or 1 or 2 or ... any number of argumetns. For the sake of this problem, let's pretend that all arithmetic procedures take exactly two arguments (while some predicates, like `zero?`, take just a single argument).

Task: Try to write down a BNF description that allows all of these type-descriptions, and only such type-descriptions. (In other words, formally describe the syntax of the type-description language, as well as you can). This should include type-descriptions for functions (i.e., their type-signatures, but only the part after the colon). So if you have

```
my-func: int * string -> string
```

then `int * string -> string` should be an allowable 'type-description' according to your BNF.

Task: Consider the following procedure:

```
(define (weird-proc p x y)
  (p x y))

(check-expect (weird-proc + 2 3) 5)
```

It consumes

- a proc that operates on two ... let's say two *int*s, and produces an int
- a first int
- a second int

and from these produces an `int`.

What's the type-signature for `weird-proc`? Does the BNF you wrote allow this type-description? If not, try to modify it so that it does.
