Homework 2: Warm Up

Due: 11:59 PM, Sep 18, 2018

Contents

1 Talk to a TA

2 Evaluating expressions in the context of an environment

3 Computations

3.1 Polygons

3.2 Summations

4 Translate Candy

5 If to Cond

6 Write your own if

7 atan2

8 Comparing Functions

Objectives

By the end of this homework, you will understand:

1. how to write input/output specifications
2. how to perform computation using Racket
3. lambda expressions and procedure definitions
4. the purpose of quote form
5. if and cond special forms
6. how to compare functions

How to Hand In (important)

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:
All Racket files you submit should run. That is, when hitting Run in Dr. Racket, your submitted code should not result in an error.

For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension .rkt. If you choose to do or save your work on the department machines, 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 Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
B1234567
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, upload them to Gradescope. Do not zip or compress them.

Design Recipe

In this course, you’re going to start to hear about a "Design Recipe." We want you to be considerate of the type of data your programs handle, and thoughtful about what your programs do. The design recipe will be built up over time, but to start, it’s just going to involve one step — writing a specification for each procedure you write.

A procedure specification lists what the procedure accepts as input, and what it produces as output. We’ve provided some examples below:

;;; input: n, a number
;;; output: the sum of n and 1, a number
(define my-add1 (lambda (n) (+ n 1)))

;;; input: x, a number
;;; y, a number
;;; z, a number
Problems

1 Talk to a TA

Task: Before this homework is due, go to TA hours and introduce yourself to one of the TAs. Ask them a question about the content or the course. If you don’t have any, tell them about any board games you like, pets you may have, classes you’re taking, or really anything! 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.

2 Evaluating expressions in the context of an environment

For each problem part, we give you some bindings and a possible expression. (We use the symbol \( \leadsto \) to mean “is bound to”.)

Task: Find and report the value of the (possible) expression in the context of the environment consisting of those bindings. If it is not an expression, or the expression does not evaluate, say why. If there is no text whose denotation is the value of the expression, describe the data object with words.

Note: You’re encouraged to do this by hand, rather than using Dr. Racket. If you choose to check your work in Dr. Racket, change your language mode to Intermediate Student with Lambda, by going to Language, then Choose Language.

1. Bindings: \( a \leadsto 2, b \leadsto 1 \)
   Possible Expression: \((* (+ a b) a)\)

2. Bindings: \( foo \leadsto 7, bar \leadsto 10 \)
   Possible Expression: \((foo + bar)\)

3. Bindings: \( a \leadsto 1, c \leadsto 0 \)
   Possible Expression: \((* (+ a b) a)\)

4. Bindings: \( a \leadsto 3, b \leadsto 1, z \leadsto 1 \)
   Possible Expression: \((\text{lambda } (x) (+ x a))\)

5. Bindings: \( a \leadsto 3, b \leadsto 1, z \leadsto 1 \)
   Possible Expression: \((((\text{lambda } (x) (+ x a)) z)\)

;; output: the sum of x+y+z, a number
(define add-three-numbers (lambda (x y z) (+ x y z)))
6. Bindings: \(a \sim 3, b \sim 1, z \sim -1\)
   Possible Expression: \(((\lambda (x y) (+ x a)) z (+ a 1))\)

7. Bindings: \(a \sim 3, b \sim 1, z \sim -1\)
   Possible Expression: \(((\lambda (x y) (* y (+ x a))) z (+ a 1))\)

8. Bindings: \(a \sim \text{the value of } (\lambda (x) (* 3 x)), b \sim 7\)
   Possible Expression: \((- (a b) b)\)

9. Bindings: \(a \sim \text{the value of } (\lambda (x) (* 3 x)), b \sim 7\)
   Possible Expression: \((- (a b b) b)\)

10. Bindings: \(a \sim \text{the value of } (\lambda (x) (+ 2 x)), b \sim 7\)
    Possible Expression: \((- ((\lambda (y) (* y 2)) (a b)) b)\)

11. Bindings: \(a \sim \text{the value of } (\lambda (x) (+ 2 x)), b \sim 7\)
    Possible Expression: \((- ((\lambda y (* y 2)) (a b)) b)\)

12. Bindings: \(a \sim \text{the value of } (\lambda (x) (+ 2 x)), b \sim 7\)
    Possible Expression: \((- ((\lambda () (* a a)) b))\)

3  Computations

3.1  Polygons

Task: Write a procedure \texttt{sum-angles} with the call structure \((\texttt{sum-angles } n)\) 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 \(180(n - 2)\). You are not expected to, nor should you, define special behavior for when \(n < 3\).

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.

```scheme
;;; input: a positive integer, \(n\), greater than 2
;;; output: the num of degrees in an \(n\)-sided polygon
(define sum-angles (lambda (n) ...))

;;; sample calls (optional for now)
(sum-angles 3)
(sum-angles 4)
(sum-angles 5)
;;; add more if you want to!
```
3.2 Summations

**Task:** Write a procedure `sum`, 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. You are not expected to, nor should you, define special behavior for when \( n < 1 \).

You should 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.

Here’s a template to get you started.

```
;; input: ...
;; output: ...
(define sum ...)  
;; sample calls (optional for now) ...
```

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

4 Translate Candy

**Task:** Write a procedure `translate-candy` that, when given the symbol `candy`, `cake`, or `mint`, returns the equivalent Spanish symbol. That is:

- `candy` should return `caramelo`
- `cake` should return `pastel`
- `mint` should return `menta`

Here’s a template to get started.

```
;; input: ...
;; output: ...
(define translate-candy ...)  
;; sample calls (optional for now) ...
```

**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 input. Do you need to restrict it?
5 If to Cond

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:

\[
(\text{if } \text{condition} \\
\text{result-t} \\
\text{result-f})
\]

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

\[
(\text{cond} \ (\text{condition} \ \text{result-t}) \\
(\text{else} \ \text{result-f}))
\]

By “functionally equivalent,” we mean that we would get the same result by evaluating either one. An if-expression has the shape

\[
(\text{if } \text{condition} \ \text{expression1} \ \text{expression2})
\]

where condition, expression1, and expression2 are all expressions.

An if-expression is evaluated by:

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

(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.

\[
(\text{if true } "\text{apple}" \ "\text{banana}"
\Rightarrow "\text{apple}"

(\text{if } (= 17 15) "\text{apple}" "\text{banana}"
\Rightarrow "\text{banana}

---

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.
Determining when to use an if instead of cond is often stylistic. In the problem below, you’ll see how an if can be written as a cond in a far cleaner matter. There will also be times where a cond expression can be written as an if expression in a far cleaner format.

Task: Change the following program fragment into an equivalent one which uses cond instead of if (and is somewhat more comprehensible). Since you’re not writing a procedure, a design recipe is not needed.

Note: Above your cond, define x to be some number such that your code can run.

```scheme
(if (< x -1)
    (/ -1 x)
  (if (< x 0)
      (- x)
    (if (> x 1)
        (/ 1 x)
      x)))
```

6 Write your own if

Now that you know you can translate code using if into code using cond, you should have no trouble writing your own version of if, although it won’t have all the functionality of the built-in if.

Task: Write the procedure my-if satisfying the following spec:

```scheme
;; input: condition, expression that evaluates to a Boolean
;;   then-expr, expression to evaluate if condition evaluates to true
;;   else-expr, expression to evaluate if condition evaluates to false
;; output: the value of the then-expr if the condition evaluates to true or
;;   the value of the else-expr if the condition evaluates to false
(define my-if
  (lambda (condition then-expr else-expr)
    ...))
```

Task: Show that your procedure works on the if statement in the previous problem. Put this example under the sample calls comment. Define x to be some number such that your code can run.

7 atan2

Often in computer science one needs a procedure to compute the angle $\theta$ between the positive $x$-axis and the ray from $(0,0)$ through a given point $(x,y)$. The professor used such a procedure this past summer in constructing a graph representation of the boundaries of census blocks. The traditional name for this procedure is atan2. See this figure:
Those of you who remember trigonometry (ugh) will recall that \( \theta \) is an angle such that \( \frac{y}{x} = \frac{\sin \theta}{\cos \theta} \). The expression \( \frac{\sin \theta}{\cos \theta} \) is \( \tan \theta \), so the procedure’s should return a number \( \theta \) such that \( \tan \theta = \frac{y}{x} \). The functional inverse of the tan function is called \( \arctan \), and is written \( \text{atan} \). Thus one would think that \( \text{atan}(y/x) \) would suffice. However, this is not true; \( \text{atan}(y/x) \) returns the same value for, say, \( x = 1 \) and \( y = 1 \) as for \( x = -1 \) and \( y = -1 \), whereas the angle \( \theta \) should be \( \pi/4 \) in radians (45 degrees) in the first case and \( 5\pi/4 \) (225 degrees) in the second case. For this reason, the traditional \( \text{atan2} \) takes two arguments, \( y \) and \( x \) (in that order).

In this problem, you are to write the procedure \( \text{atan2} \). To make things simpler, you are allowed to assume that \( x \) and \( y \) are nonnegative (but not necessarily positive). You may also assume that at least one of \( x \) or \( y \) is positive.

**Task:** Define the procedure \( \text{atan2} \) using the following spec:

```scheme
;;; input: x, y, two nonegative numbers where at least one is positive
;;; output: an angle theta, measured in radians, such that theta is the
;;;         angle between the positive x-axis and the range from (0,0)
;;;         through (x,y)
;;;         (define atan2
;;;           (lambda (y x)
;;;             ...))
;;; sample calls (optional for now)
```

You will need to use \( \text{if} \) in your code. Obviously you should not use any built-in equivalent of \( \text{atan2} \) but you can use other built-in trigonometric procedures, such as \( (\sin u) \), \( (\cos u) \), and \( (\tan u) \), and their inverses, \( (\text{asin} u) \), \( (\text{acos} u) \), and \( (\text{atan} u) \).

**Task:** Substitute my-\( \text{if} \) for \( \text{if} \) in your code. What goes wrong and why? In particular for what input(s) does the new code fail to give the same results as the old code? Describe the difference between my-\( \text{if} \) and \( \text{if} \) that accounts for the failure.

**Note:** For your final hand-in, be sure to use \( \text{if} \) rather than my-\( \text{if} \). Write your answer to this task in a comment at the botton of the \( \text{atan2}.rkt \) Racket file.

### 8 Comparing Functions

This problem is not about Scheme or programming. It’s about mathematical functions. In this course, we will reason about univariate (single-input) functions that take nonnegative integers as
inputs and produce real numbers as outputs. To indicate that a function $f$ is such a function we write

$$f : \mathbb{Z}^+ \to \mathbb{R}$$

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 greater 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 $g$, and take turns being greater ($f(1) \geq g(1)$ and $f(2) \leq g(2)$ and $f(3) \geq g(3)$ and so on for inputs less than 113, but then for all numbers larger than or equal 113, we have $f(n) > g(n)$. We still say that $f$ is eventually larger than $g$.

For most of the functions we consider in this case, it is simpler—one function will start out being bigger and then the other will take over.

The formal definition is this:

If $f, g : \mathbb{Z}^+ \to \mathbb{R}$ 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 greater 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 greater 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, for every value of $n$ that is at least $M$, we have $f(n) > g(n)$, i.e., we have

$$3n - 1 > n + 7.$$
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. $f : \mathbb{Z} \rightarrow \mathbb{R} : n \mapsto 3n$
   $g : \mathbb{Z} \rightarrow \mathbb{R} : n \mapsto 2n + 5$

b. $f : \mathbb{Z} \rightarrow \mathbb{R} : n \mapsto 5 \sin(n) + \frac{2}{3}$
   $g : \mathbb{Z} \rightarrow \mathbb{R} : n \mapsto 1 + \ln n$

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

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

**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 nonnegative integers*, 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.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback