Lab 0

This lab is to be completed INDIVIDUALLY.

Read HtDP/2e: Chapters 1, 2, and 3 through 3.5 for Labs 0 and 1.

Goals

This week's lab is mostly concerned with **getting everyone set up and ready to work**. It will help you to practice:

- Using DrRacket's lower *Interactions Window* as a calculator
- Using DrRacket's upper *Definitions Window* to define some simple functions, and to create a file to turn in for grading
- Computer Science Vocabulary

The other goal of Lab 0 is to remind you of the **pattern matching skills** that your elementary school teacher tried to teach you. It turns out that these skills are a prerequisite for being an efficient web site administrator, computer programmer, CIA analyst, etc. Of course, other skills are needed such as reading, writing, and arithmetic.

The first two problems are "pencil and paper" problems, however, you should download Dr.Racket and use the lower interactions window as a calculator. **Provide your answers in the comments of your submitted Racket file**. In general, you will only turn in one DrRacket file for each lab assignment.

If this is the first time you've brought up Dr.Racket, you will be asked to choose a language. From the Language menu, choose ***Language→Choose Language****. . . . then push the button for* ***Teaching Languages*** *and under* ***How to Design Programs****, choose* ***Beginning Student****, and click OK. Language choice can also be accessed from the menu at the bottom left of Dr.Racket.*

Using Dr.Racket as a calculator: In the Interactions (lower) window, type in the number 7, then hit the Enter key. Dr.Racket gives you back the number 7. This is an example of an ***atomic expression***.

We can also write **compound expressions**, which consist of a left parenthesis, an ***operator***, some more expressions (either atomic or compound), and a right parenthesis. Here are some examples, each conforming to the definition of "*expression*" by repeatedly applying the two definitions given above. Type each expression into the Interactions window and make sure you understand the results returned by DrRacket.

```
7

5

(+ 7 5)

(+ 7 5 10) ;note + may consume more than two arguments

(- (+ 7 5 10) 3)

(* (- (+ 7 5 10) 3) 2)
```

(In the third **expression**, the **operator** is **+**, and the two **arguments** required by the **+** operator are two numbers, **7** and **5**.) ***There is nothing to turn in for this part.***

The upper window is called the **Definitions** window. The content of the definitions window can be saved to a file. *As always, save early and often.* Dr.Racket does not interpret (execute) the definitions window until you hit the **Run** button, then it interprets every expression in the file, from top to bottom.

In contrast, the content of the **Interactions** window is not saved, and Dr.Racket interprets anything you type into the interactions window as soon as you hit the return key. Think of it as your "scratch pad" for performing simple calculations you only need to do once.

Problem 1: (no programming)

A student is retained by a neighborhood association to distribute fliers, collect dues, and do miscellaneous chores. Just to make sure they are around when needed, they get 25 dollars a month (they don't have to work for that). For every hour they work, they get $7.

boy works *h* hours	0	1	2	…	5	…	10	…	*h*	hours
he earns	$25	$32	??		??		??		??	dollars

Create a mathematical formula for calculating how much the student earns per month if they work *h* hours. Use the formula and the Interactions Window in DrRacket to find out how much the student earns when a major event forces work for 40, 110, or 134 hours in a single month.

Write your answers as a ***comment*** in the Definitions Window (top window). Place all of your answers to the lab in the Definitions Window, and save them: this is your Racket file that you will (eventually) hand in for grading using Sakai. We have provided a Racket file for you that has space for your solutions to this lab to get you started.

A **comment** is a line of text that starts with a semicolon [;] and is for humans to read, as comments are ignored by DrRacket. For example:

```
;; Lab 0, by Sally Student
;;
;; Problem 1
;;
;; Formula: pay(h) = sin(h) + (5 * h * h)
;; pay(0) = 25
;; pay(1) = 32
;; pay(40) = ??
;; pay(110) = ??
;; pay(134) = ??
```

[10 points: 5 for the formula, 5 for the examples]

Problem 2: (no programming)

A car accelerates at 5.6 meters per second-squared like this:

after *t* seconds	0	1	2	3	4	…	10	…	15	…	*t*	seconds
car has traveled	0	2.8	11.2	25.2	44.8		??		??		??	meters

Guess a formula for calculating how far the car has gotten in *t* seconds *[hint: if you never took physics, it depends on t^2]*. Check the formula for the first five table entries. If it doesn't work, guess more.

Once the formula works for the first five entries, use it and DrRacket's Interactions Window to compute the value for 10 and 15 seconds in the above table.

Write your answer in the same Definitions Window file as a comment, similar to what you did for Problem 1.

```
;; Problem 2
;;
;; Formula: car-distance(t) = 7t + 12.3
;; car-distance(10) = 123.56
;; car-distance(15) = 457.123
```

[10 points: 5 for the formula, 2.5 for each of the new values]

Problem 3: (yay! a very short program with one function!)

Define a function, named `pay`, that computes the mathematical function you defined in problem 1. When we **call,** or **apply** the function, we will provide the *hours h* we want as an **argument** to the function. When we **define** the function, we will need to reserve 1 empty place (1 **parameter**) that will hold the one **argument** (hours) we provide when we **call** the function. Here is the **contract/signature** and **purpose statement** for our function:

```
;; pay: Number --> Number                    [This is the contract/signature]
;; consumes a positive number hours representing
;; the number of hours worked this month,    [This is the purpose statement]
;; and produces the payment for the month in dollars
```

Copy the signature and purpose statement into the Definitions (top) window of Dr.Racket. Now write below the function definition for `pay` (Hint: here is the template for the function definition, with parameter name `hours`. Note that the function definition is NOT written as a comment, so that DrRacket will compute with it.)

```
(define (pay hours)                  [This is the header of the function definition]
  ...)  [This is where you write the body]
```

Hit the **Run** button. If DrRacket reports any errors, try to figure out what's wrong, make corrections, and hit the Run button again. In the ***Interactions*** (bottom) window, **call** the function `pay` to compute the elements in the table you built in question 1.

Earlier, when you filled out the table in problem 1, what you did was create your **examples**. Now let's turn you examples into **unit tests**. Unit tests test a single "unit" of a larger program, that is, they test one individual function.

Back in the definitions window, underneath your definition of `pay`, create a set of unit tests corresponding to your examples in problem 1, like this:

```
(check-expect (pay 0)   25)
(check-expect (pay 1)   32)
(check-expect (pay 40)  ??)          [These are the Unit Tests]
(check-expect (pay 110) ??)
(check-expect (pay 134) ??)
```

Hit the Run button, and make sure your tests work. You might want to intentionally make an incorrect test, just to see what happens. Then you can change it back. Congratulations on your first program using the Design Recipe!!

We'd like to discourage you from having ***"magic numbers"*** in your programs like 50 and 8.25, which may change often. Instead, rewrite ("**re-factor**") this program to use two **constants** that you have defined like this:

```
(define BASE-PAY 50)
(define HOURLY-RATE 8.25)
```

*[These are **Constant definitions**]*

You only need to turn in this final version of your program for problem 3. *[11 pts: 2 points for 2 correct constant definitions, 1 pt signature, 1 pt purpose, 1 pt fn header, 1 pt body, 5 pts for 5 tests]*

Problem 4:

Define a function, named `car-distance`, that computes the mathematical function you defined in problem 2. When we **call/apply** the function, we will provide the time *t* we want as an **argument** to the function. When we **define** the function, we will need to reserve 1 empty place (1 **parameter**) that will hold the one argument we provide when we call the function. Here are the signature and purpose statement for our function:

```
;; car-distance: Number --> Number
;; consumes a positive number time, representing the number
;; of seconds since the car started
;; accelerating at 5.6 meters per second squared,
;; and produces the number of meters traveled.
```

Copy the signature and purpose into the **definitions** (top) window. Now write the function definition for car-distance (Hint: here is the template for the function definition, with parameter name `time`:)

```
(define (car-distance time)
  …)
```

Hit the **Run** button. If DrRacket reports any errors, try to figure out what's wrong, make corrections, and hit the Run button again. In the Interactions window, call the function car-distance to compute the elements in the table you built in question 2. In fact, when you filled out the table in problem 2, what you did was create your **examples**. Now let's turn you examples into **unit tests**. Back in the definitions window, underneath your definition of `car-distance`, create a set of unit tests corresponding to your examples in problem 1. This time, since you are dealing with inexact numbers (not integers), you must use `check-within` and specify a *tolerance*. Use 0.001 for the tolerance in all 7 tests:

```
(check-within (car-distance 0)  0    0.001)
(check-within (car-distance 1)  2.8  0.001)
```

```
(check-within (car-distance 2)  11.2  0.001)
(check-within (car-distance 3)  25.2  0.001)
(check-within (car-distance 4)  44.8  0.001)
(check-within (car-distance 10) ??    0.001)
(check-within (car-distance 15) ??    0.001)
```

Hit the Run button, and make sure your tests work. You might want to intentionally make an incorrect test, just to see what happens. Then you can change it back. Congrats on your second program using the Design Recipe.

Does your new function have a ***magic number***? Please re-factor your answer to use a defined ***constant*** for full credit.

[12 pts: 1 pt constant def, 4 pts for signature, purpose, header, body, and 7 pts for 7 correct tests]

Problem 5

Answer these questions directly in your Racket file using **comments** and **expressions**:

1. Write an ***atomic expression*** that does not involve the use of any numbers. *[1 pt]*
2. Define a ***constant*** called `TAX-RATE` that declares the tax rate is 5.75%. *[1 pt]*
3. Define a ***function*** named `taxes-owed` that *consumes* a number representing income and *produces* a number that is the product of income times the TAX-RATE. *Use good parameter names*. *[2 pts: one for header, one for body]*
4. In a comment, give the name(s) of any ***parameters*** you defined in the function `taxes-owed`. *[1 pt]*
5. Write three different ***function calls*** / ***function applications*** for the function `taxes-owed`. *[3 pts]*
6. In a comment, list the ***argument(s)*** in each of the three function calls in the previous part 5.5. *[3 pts]*

What to Turn In

Using Sakai, turn in a single file *lab0.rkt* containing all code and documentation (comments) for this assignment. **Your name must be listed in a comment at the top of the file**. Make sure that you receive a receipt from Sakai!!