211lab00

Getting Started – DrScheme, Design Recipe

Original Author: Dr. John Greiner

The main goals of this lab are to ensure that you can

- use the basic functionality of DrScheme and
- write simple programs in a systematic manner.

Outside of lab, don't forget to get a homework partner, if you haven't already. If you're having trouble getting a partner, try using the sign up sheet. Partners are encouraged, but not required, to attend the same lab section.

DrScheme Basics

1. Open a terminal.
2. Make a new folder (directory) called `Comp211`, for this course. The command is for making a new folder on Linux is `mkdir`.
3. Make a sub-folder in `Comp211` called `Labs`.
4. Make a sub-folder in `Labs` called `Lab01`. Save all your Lab01 work to this folder. For submitting homeworks, see the Owlspace page for Comp 211.
5. Start DrScheme by typing the command `drscheme`.
   #If this is the first time you've run DrScheme, it will ask you your preferred language, and the Scheme language level; the latter should be "Beginning Student with List Abbreviations". This language level provides the expected set of primitive operations and enables DrScheme to reject "legal" Scheme constructions that are not part of our introductory dialect.

The DrScheme window is divided into two halves. The lower half, or *interaction window*, is the Scheme "calculator". The top half, or *definition window* is just a text editor which is smart about indenting Scheme, and such. The *execute* button sends the contents of the definition window down to the interaction window, where they are actually evaluated.

We suggest that you download DrScheme to your home computer/laptop. Versions are available for all major platforms.

For our running example, we'll look at a common college problem: how to divide the cost of a pizza. For simplicity, we'll assume that a pizza costs $12 and is cut into 8 slices.

Simple example of using DrScheme

1. Type (or copy/paste) the following expressions into the interactions window

```
(+ 2 7)
```

```
(* (+ 2 7) 3)
```

2. Type (or copy/paste) the following blocks of code into the definition window, including:

```
;; contract:
;; owe: nat -> num
;; purpose:
;; (owe n) returns how much money you owe for eating n slices of pizza.
;; definition:
(define (owe slices)
   (* (/ 12 8) slices))
```

   Observe that DrScheme helps you with the indentation if you use the Return key in appropriate places. It also "bounces" the cursor to visually match the closing parenthesis with the corresponding opening parenthesis.
3. Use the definition of `owe` by typing in the interaction window some Scheme expressions using those definitions, *e.g.*,

```
(owe 5)

(owe 7)
```

4. We included *comments* before each function definition, explaining what it is supposed to do. DrScheme ignores the comments, but you, your grader, your boss, etc., can read them. Soon, we'll provide more guidance on what to say in your comments.

```
; One form of comments is anything following a semicolon
; up to the end of the line.
;; However, you'll often see two semicolons starting a comment.

#|
  Another form of comments is anything between these
  two matching markers.
|#
```

   You can also go to the *Special* menu, choose *Insert comment box*. You are welcome to do this, but one caveat: if DrScheme saves a file which includes a fancy comment box, the file will not be saved in plain-text format. (This is only an issue if you want to open or print the file with a different program later.)
5. Edit your definitions or comments some. You can use the mouse, the arrow keys, backspace, and standard keyboard shortcuts to move the cursor around. Also, the PC, Mac, and Unix versions of DrScheme each use their respective standard keyboard shortcuts. (You can look in the Edit menu under Keybindings for the complete list.)
6. Load the definitions into the interaction window by clicking on the *Run* button. If DrScheme detects any syntactic errors, it will give you an error message in the interactions window and highlight the error in the definitions window.
7. Save your work in your `Comp211\Labs\Lab01` directory. Be sure to use a filename that has a `.ss` extension. You don't need to turn in your lab work. In fact, you don't have to save it, but we recommend it so that you can look at it again.
8. Finally, intentionally introduce an error or, what you *think* will be an error, press *Run*, and see what error message is given. We'll go around the room to see what different error messages people get. Do the error messages tend to make sense?

DrScheme has lots of other features, including a complete manual. We'll explore more of DrScheme in later labs, and we encourage you to explore, but there's no need to learn all its features. Read the DrScheme Tips and Traps page for helpful information on common mistakes, error messages, etc.

Step-by-step Evaluation

While DrScheme evaluates our Scheme programs for us, it is also important for us to understand the process of evaluation. The details of evaluation will be covered in class. Here in lab, we want to explore two useful tools to help us:

- !DrScheme's stepping tool that illustrates evaluation.
- A technique for the programmer to illustrate and check evaluation "by hand".

Using DrScheme's stepper

1. Place an example use or two of your functions in the definitions window, after the appropriate definitions. For example,

```
(owe 10)
```

2. Click on the *Step* button. This brings up a new stepper windows which will show *how* DrScheme calculates the answer.
3. Use the stepper's button to look through the evaluation. At each step, it shows what part of the code needs to be evaluated next, and what the result is.

Illustrating hand-evaluation

"Hand-evaluation" is just *you* doing the same thing as DrScheme's stepper. It is useful to convince yourself that you know what is supposed to happen, independently of having DrScheme help you.

1. Hand-evaluate a few example expressions, and type in each of the small steps you would make to calculate the result. Type these in the definitions window after your definitions. For example,

```
;; Hand-evaluation:

(owe 10)

(* (/ 12 8) 10)

(* 3/2 10)

15
```

This is just high school algebra, successively simplfying *expressions*, until you reach a final, simple *value*.
    - What do you hope to see, when you *Run* all these expressions? What would you see if there were a mistake?
    - While you are doing this, it is often helpful to copy the previous step, and edit this copy for the current step. This can save you some typing, and it helps eliminate mistakes. You'll want the *Copy* and *Paste* features in the *Edit* menu.
2. Verify your steps by clicking *Run*.
3. Once you've used this to verify your steps, put the steps in comments, rather than deleting them. (Later, you might change the code, and you want to re-use these as test cases.)

Design Recipe

We will discuss these in class this coming week, but here's a preview. Programming methodology is a very important component of this course, and you will be required to follow these ideas, so the earlier you get into the habit of using them, the better.

When writing programs, there are lots of things we need to think about. It helps if we have some guidelines to follow that remind us to do these things in the proper order. While following some strict rules can seem annoying at first, in the end it will save you lots of errors and grief. These guidelines will be our *design recipes*.

Working with unstructured data, like the numbers that we've seen, is relatively simple. With more complicated data and program styles, we will add to the following steps.

1. **Write the function's contract, purpose, and header.** The *contract* specifies the function name and what kind of values the function will use as input and output. The *purpose* is a short statement describing *what* (not how) the function will calculate. The *header* is the part of the function definition specifying the function and parameter names. Type these in the definition window. Put the contract and purpose in comments, as in the following examples:

```
;; contract:
;; owe : nat -> num
;;
;; purpose:
;; (owe n) returns how much money you owe for eating n pizza slices.
;;
;; header:
;; (define (owe slices) ...)
```

We won't use DrScheme to verify that our contracts are correct, although that is a very useful thing to do. Look for that in COMP 212.
2. **Make some examples of what the function should compute.** You need to make sure you understand what the function is supposed to do before you define it. When applicable, use some of the example data. We recommend the following form for these examples:

```
;; Examples:
;; (owe 0) => 0
;; (owe 10) => 15
```

The first line provides an explanation of the following, making it more readable. (We haven't introduced symbols yet officially -- we will see them in class soon.) The next two lines each give an example, asking DrScheme to compare (with the numeric equality function =) the actual result with the expected result. You should pick enough examples to test the function adequately. We'll say more about that later, but this should include any interesting inputs like zero.
3. **Test the function.** i.e., make sure the previous examples really work. If you wrote the examples as suggested above, you can use them to test the function. Add a test section after your examples, like this:

```
;; Tests:
(check-expect (owe 0) 0)
(check-expect (owe 10) 15)
```

All of these tests should pass , assuming that the code for owe is correct. The more elaborate your test, the more errors you'll catch sooner and the less time it will take to write a correct function. Note that at this point owe has not been written yet. Still we write the test code first! This is one of the key steps in modern software development: test before coding. It is called *test-driven development*. This is where we deviate from the design recipe describe in HTDP (the textbook). We write the test code first!

4. **Write the function body.** Soon, we will have *lots* to say about this step. For now, for programs on "unstructured" data, this is very straightforward, because we are typically given an equation like …

```
;; definition:
(define (owe slices)
   (* (/ 12 8) slices))
```

For sets of intervals or other conditional functions, there should be exactly one condition clause per option.

Using the design recipe, define a function calculating the area of a right isosceles triangle.

Simple Data: Booleans and Symbols

We've seen that DrScheme has

- numbers: `17` , `-42.89` , `3/17` , ...
- Booleans: `true` , `false`
- symbols: `'central-daylight-time` , `'galatea2.2` , `'mary` , ...

Numbers are old hat, so let's explore Booleans and symbols some.

Exercises: Using built-in data

1. Try calling some the built-in operations like <=, =, `and`, `or`, `not`, `equal?` on various data values, *e.g.*, is 17 times 18 bigger than 256?
2. What are the contracts of these built-in functions?
   *Choose any two, and write down the contract.
   *Verify your guess by asking the labby
   or by using DrScheme's Help Desk to look at the
   manual for the Beginning Student language.
   *Practice writing a function which returns a Boolean:
   Following the design recipe,
   write

```
within-two?
```

   which takes in two numbers
   `m` and `n`, and returns true if
   `m - n` and `n - m` are both
   less than `2`.
   Otherwise, it should return false.
   (For learning purposes only, don't use `abs`,
   which computes the absolute value. Also, don't use
   `cond` or `if`.)
   Your examples/tests can look like

```
(check-expect (within-two? 99.8 101) true)
(check-expect (within-two? 5 -5)     false)
```

Sample solution

Other Basics

We assume that you already know how to use email and a Web browser. If you have any trouble, ask a labbie during office hours, or contact the staff of the information desk in Mudd.

You'll also need to know how to use SVN. On the SVN Turnin Page we have basic instructions for using SVN to turn in your homeworks.

Always "log out" from your account when you are done and leaving. Otherwise, someone can use your account, e.g., to copy or delete your assignments. *Be sure to save everything before you log out!*
