CIS 130 - Intro to Programming - Spring 2010
Homework #1
DUE: Wednesday, February 3rd, 5:00 pm

Each student should work individually on this assignment

When you are done with the following problems:

- save your resulting Definitions window contents in a file with the suffix **.ss** or **.scm**
- transfer/save that file to a directory on **nrs-labs**
- use **ssh** to connect to **nrs-labs**
- **cd** to the directory where you saved it (`cd 130hw1` for example)
- use **~st10/130submit** to submit it
- make sure that ~st10/130submit shows that it submitted your homework .ss or .scm file
- (we will be practicing the above in class on Monday, February 1st -- ASK ME if this is not clear after that, or if you have any problems with submission!)

0. Start up DrScheme, setting the language to **Beginning Student** and adding in the **universe.ss** and **fabric-teachpack.scm** teachpacks. (You saw how to do this during the class on Monday of Week 2 -- that class's posted example includes some notes to remind you how to do it, if you'd like a refresher.)

   In the definitions window (the top one) type in a comment-line containing your name, followed by a comment-line containing CIS 130 - HW 1, followed by a comment-line with no other text in it --- that is,

```
; type in YOUR name
; CIS 130 - HW 1
;
```

1. Below what you typed in #0 above, type the comment lines:

```
; Problem 1
;
```

   Now type the expressions specified below in the Scheme definitions window, each starting on its own line. Run (push the Run button) and look in the interactions window (in the lower window) to see if you get the expected results for each.

   (a) a simple expression of type number representing the number 47
   (b) a compound expression of type number representing the sum of 13 and 47
   (c) a simple expression of type string that includes your name
   (d) a simple expression of type string that represents a color of your choice
   (e) a compound expression of type image that is a rectangle or a fabric image (your choice!) of the color of your expression from (d), of width 75 pixels and height 45 pixels

2. Next, in your definitions window, type the comment lines:

```
;  Problem 2
;
```

There is a special operation called `check-expect` -- it expects two expressions of any type, and it tests to see if the second has the same value as the first. What is unusual about this particular operation, though, is that it doesn't produce a boolean value -- instead, it has several side-effects. If the values of its expressions are the same for all of the `check-expect` expressions in your Definitions window, it simply causes a message saying how many tests were passed to be displayed in the Interactions window, at the end. But if the values of the expressions are not the same, a separate window is displayed, giving how the expressions differed, and with links to each `check-expect` expression that "failed".

Figure out (by yourself) what you think the value is for each of the following Scheme expressions.

- Are any using **incorrect** syntax? If so, type them within a comment in your Definitions window.
- For each of those using **correct** syntax, write a `check-expect` expression including the expression below and what you think its value should be.

Then push the Run button, and see if your tests -- your `check-expect` expressions -- pass. If any do not, feel free to tweak them until they do -- but try to figure out why their values differed from what you expected. The important thing here is to think about what **should** happen, and then compare it to what **really** happens.

(a) `(* (- 3 5) 20)`
(b) `(* pi (* 10 10))`
(c) `(+ 73 true)`
(d) `(/ 13 0)`
(e) `(or (< 1 2) (> 5 8))`
(f) `(and (< 1 2) (> 5 8))`
(g) `(< (image-width chili) (image-width hat))`

3. Next, in your definitions window, type the comment lines:

```
;  Problem 3
;
```

Scheme happens to have a built-in operation for finding the maximum of some set of numbers, called **max**. Write expressions in your definitions window, each on its own line, using **max** to see if it works:

* with 2 values,
* with more than two values,
* with just one value, and
* with no values.

If any of these do not work, **COMMENT OUT** that expression in your definition window (put a **;** in FRONT of it) and re-run (so the subsequent expressions will get a chance to execute!).

4. Next, in your definitions window, type the comment lines:

```
;   Problem 4
;
```

The following are currently **not** "proper" Scheme expressions (expressions that follow Scheme's syntax rules). **Correct** each, and then type each (now-"legal"-syntax) Scheme expression in your definitions window (starting on its own line).

(a) `(chili)`
(b) `image-width chili`
(c) `(* (/ 7 3) (8) (image-height flower))`

5. Next, in your definitions window, type the comment lines:

```
;   Problem 5
;
```

Write a compound expression of type image, using any of the operations we have discussed thus far.

Now write another compound expression of type image, using any of the operations we have discussed thus far, **except** make sure that the width and height of the resulting image are **greater than** those for the image resulting from your first expression.

6. Next, in your definitions window, type the comment lines:

```
;   Problem 6
;
```

You should recall the following operations from Week 2's class:

* ; **rectangle**: number number string string -> image
  ; (rectangle *width length mode color*)
  ; purpose: expects a numeric *width* in pixels,
  ; a numeric *length* in pixels,
  ; a string *mode* (either "solid" or "outline"),
  ; and a *color* given as a string,
  ; and produces a rectangle image (either filled in or not) of that size and color.

* ; **circle**: number string string -> image
  ; (circle *radius mode color*)
  ; purpose: expects a numeric *radius* in pixels,
  ; a string *mode* (either "solid" or "outline"),
  ; and a *color* given as a string,
  ; and produces a circle image (either filled in or not) of that size and color.

* Note that images have something called a **pinhole** -- in circles and rectangles this pinhole is in the center of the image. When you combine such images using the operation **overlay**, it overlays them on their pinholes.

  (You can move an image's pinhole, too - type **pinhole** in DrScheme, type the F1 key while the cursor is anywhere on the word **pinhole**, and click on any **pinhole** operation in the resulting window that says it is from teachpack/htdp/image. You'll then get a page

of documentation telling you more about pinholes and more pinhole operations.)

* ; **overlay**: image image ... -> image
; (overlay *image1 image2* ...)
; purpose: expects as many images as you'd like,
; and produces a new image that is the result of overlaying *image1*, *image2*, and so
; on on their pinholes.
; (Note: the pinhole of the new image is in the same place as the pinhole of *image1*)

Now, here's a new operation:

* ; **add-line**: image number number number number string -> image
; (add-line *image begin-x begin-y end-x end-y color*)
; purpose: expects an image, beginning x and y coordinates,
; ending x and y coordinates, and a color (given as a string),
; and produces a new image that is the result of adding a line of color *color*
; starting at (*begin-x, end-x*) and ending at (*end-x, end-y*)
; to the top of *image*.
; (NOTE: (0, 0) is considered to be the current pinhole of *image*;
; x's higher than 0 are to the RIGHT of this,
; y's higher than 0 are BELOW this.)

Write one or more compound expression(s) so that:

* you use **each** of the operations:

- `rectangle`
- `circle`
- `overlay` and
- `add-line`

...**at least once**,

* ...**each time** using them **together** with **ONE or more** of the operations:

- `create-solid-fabric`
- `add-horiz-stripe`
- `add-vertical-stripe`
- `add-print`
- `image-width` or
- `image-height`

(You can do this with as many as four compound expressions, or with as few as one compound expression.)

Optional class challenge: (**not** for a grade, but I reserve the right, if I have time, to display the best one(s) to the class): See how many of these operations you can successfully use within a **single** compound expression.