# Scheme Exercises

1. Define a procedure `(cube n)` that takes a number as input and computes its cube.

2. Define a procedure `(sphere-volume radius)` that takes the radius of a sphere as input and computes its volume using the formula $V = (4/3)\ \pi\ r^3$ where $\pi$ is the value 3.14159. Use your `cube` procedure from the previous exercise as a "helper" function.

3. According to the *Joy of Cooking*, candy syrups should be cooked 1 degree cooler than listed in the recipe for each 500 feet of elevation above sea level. Define a procedure `(candy-temp degrees elevation)` that takes two values as input: the recipe's temperature in degrees and the elevation in feet. It should calculate the temperature to use at that elevation. The recipe for Chocolate Caramels calls for a temperature of 244 degrees; suppose you wanted to make them in Denver, the "mile high city". (One mile equals 5280 feet.) Use your procedure to find the temperature for making the syrup.

4. Candy thermometers are usually calibrated only in integer degrees, so it would be handy if your `candy-temp` procedure would give an answer rounded to the nearest degree. Rounding can be done using the predefined procedure called `round`. For example, `(round 7/3)` and `(round 5/3)` both evaluate to 2. Insert an application of `round` at the appropriate place in your procedure definition and test it again.

5. Remember the good old quadratic formula from high school math, which can be used to figure out the value of $x$ in the equation $ax^2 + bx + c = 0$? Here it is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

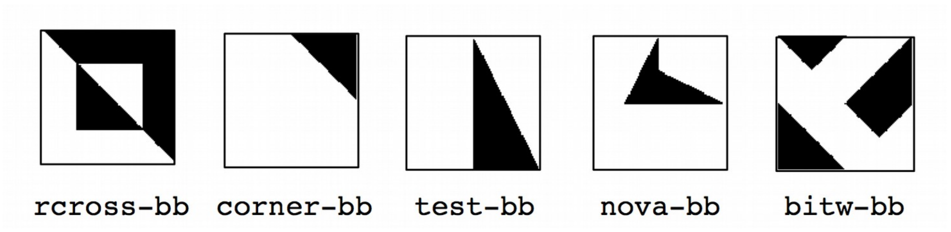
   Actually, there are always two values of $x$ (called "roots") that will satisfy the above equation. The symbol $\pm$ in the formula means that you can calculate the first root by using addition in the formula and the second root by using subtraction. Write a procedure `(quadratic a b c)` that takes the values $a$, $b$, and $c$ as input and calculates the *first* root of the equation (the one using addition). Remember that Scheme has a built-in procedure for computing square roots called `sqrt`. Some examples are shown below:

```
> (quadratic 2 3 1)
-1/2
> (quadratic 5 6 2)
-3/5+1/5i
```

6. Define a procedure `(absolute n)` that takes a number n as input and returns its absolute value. Use a `cond` expression to test whether `n` is less than zero, and if it is, return `(- n)`. Otherwise just return `n`. Test your procedure on several inputs to make sure it works.

7. The U.S. tax code uses what is called a *marginal tax rate*. This policy means roughly that the tax rate used depends on the level of income. For example, suppose that the first \$10,000 of a person's income is not taxed at all, but the amount above \$10,000 is taxed at 20 percent. If you earned \$12,500, the first \$10,000 would be untaxed, but the amount over \$10,000, namely \$2,500, would be taxed at 20 percent, yielding a tax bill of 20/100 × \$2,500 = \$500. Write a procedure `(tax income)` that takes a person's income as input and calculates the tax using these assumptions.

8. The next few exercises use the predefined graphics images shown below:



IMPORTANT: to use these images, you need to have installed the “concabs” library by following the instructions on our class web page. You must also include the following line at the top of your code:

```
(require (lib "fungraph.ss" "concabs"))
```

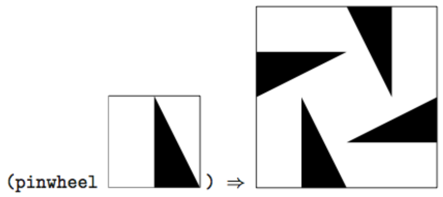
Try evaluating the following expressions, which use the operations `stack` and `quarter-turn-right` to combine images into more complex images in various ways:

```
(stack rcross-bb corner-bb)

(quarter-turn-right test-bb)

(stack (stack rcross-bb corner-bb) test-bb)

(stack (stack rcross-bb corner-bb)
       (stack (quarter-turn-right test-bb) test-bb))
```

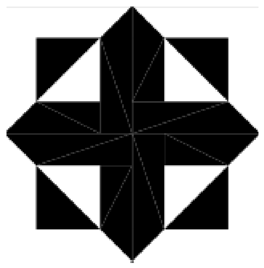
9. Define procedures `half-turn` and `quarter-turn-left` that do as their names suggest. Both procedures take a single argument, namely, the image to turn. You will naturally need to use the built-in procedure `quarter-turn-right`.

10. Define a procedure `side-by-side` that takes two images as arguments and creates a composite image having the first image on the left and the second image on the right. Hint: use the `quarter-turn-left` procedure you defined above to help you out.

11. We can construct a new image by joining together four copies of a basic block, each facing a different way. We call this operation *pinwheeling* the basic block; here is an example of this operation performed on the image `test-bb`:



Define the `pinwheel` procedure and use it to make a cross as shown below:



Now try pinwheeling the cross—you should get larger pattern reminiscent of a “quilt”, with four dark crosses. If you pinwheel that, how big is the quilt you get? Try making other pinwheeled quilts in the same way, but using the other basic blocks. What do the designs look like?

12. All five of the basic blocks can be produced using two primitive graphics procedures. The first of these, `filled-triangle`, takes six arguments, which are the `x` and `y` coordinates of the corners of the triangle that is to be filled in. The coordinate system runs from -1 to 1 in both dimensions. For example, here is the definition of `test-bb`:

    ```
    (define test-bb (filled-triangle 0 1 0 -1 1 -1))
    ```

    The second of these procedures, `overlay`, combines images. To understand how it works, imagine having two images on sheets of transparent plastic laid one on top of the other so that you see the two images together. For example, here is the definition of `nova-bb`, which is made out of two triangles:

    ```
    (define nova-bb
       (overlay (filled-triangle 0 1 0 0 -1/2 0)
                (filled-triangle 0 0 0 1/2 1 0)))
    ```

    Use these primitive graphics procedures to define the other two basic blocks, `rcross-bb` and `corner-bb`.

13. Now that you know how it is done, be inventive. Come up with some basic blocks of your own and make pinwheeled quilts out of them. You might find it interesting to try experiments such as overlaying rotated versions of an image on one another.

14. The next few exercises give you practice with recursive procedures. Using `factorial` as a guide, write a recursive procedure `(power` *`x n`*`)` that raises a number `x` to the power of `n`. By definition, `x` raised to the power of 0 is 1. For example, `(power 2 0)` should give 1, and `(power 2 5)` should give 32. Hint: think about how you would compute the correct answer for `(power 2 5)` if you knew that calling `(power 2 4)` would return 16.

15. Write a recursive procedure `(sum-of-first` *`n`*`)` that takes an input value *n* and adds up all of the numbers from 1 to *n*. For example, `(sum-of-first 5)` should give 1 + 2 + 3 + 4 + 5 = 15, and `(sum-of-first 100)` should give 5050. Hint: if you know that `(sum-of-first 99)` gives 4950, how would you use this to compute the correct answer for `(sum-of-first 100)`?

16. Write a recursive procedure `(stack-copies` *`n image`*`)` that stacks up *n* copies of a given image. Hint: to create a stack of height *n*, just create a stack of height $n-1$ and then stack one more copy of the image on top of that.

17. Write a recursive procedure `(fibo` *`n`*`)` that computes the *n*th Fibonacci number. By definition, the 1st and 2nd Fibonacci numbers are 1. After that, the *n*th Fibonacci number can be computed by simply adding the $(n-1)$th and $(n-2)$th Fibonacci numbers together. Check your results by comparing your procedure's output to the Fibonacci values given on page 135 of *GEB*.

18. Write recursive Scheme procedures called `G`, `H`, `F`, `M`, and `Q`, which compute the functions described in *GEB* on pages 137-138. What is the 25th Q number?