Lab 2: Scheme On
12:00 PM, Sep 16, 2018

Contents

1 The Game of Life: Millionaire Tycoon or Bankrupt? 1
2 Ands and Ors 1
3 What Are Good Tests? 2
4 cons, car and cdr 2
5 More Lists 3
6 Optional: Even More Lists! 4

1 The Game of Life: Millionaire Tycoon or Bankrupt?

One of the most important aspects of programming is understanding and being able to describe the inputs and outputs of the procedures you write.

Task: In this problem, you will write a procedure am-i-a-millionaire which can only take the symbols millionaire or bankrupt. It should return the symbol no if it is given the symbol bankrupt and the symbol yes if it is given the symbol millionaire. First, write the input-output spec for this procedure (see the document Input/Output Specifications in CS17 under Documentation on the course web page). Next, write the procedure and try it out.

2 Ands and Ors

HTA Cody has been too busy going to concerts, and suddenly realized he forgot to sign up to hold TA Hours! After checking his schedule, he realized there were only four slots he could make it to: 12-2 on Mondays, 2-4 on Mondays, 12-2 on Tuesdays, or 2-4 on Tuesdays. Believing himself to be something of a computer scientist, he decided to write a Racket program to determine whether he could hold hours at a given slot. He wrote a procedure, can-cody-come?, which takes in two inputs, day and slot. It can be seen here:

```racket
(define can-cody-come?
  (lambda (day slot)
    (cond
      ((and (equal? day (quote monday))
            (equal? slot (quote 12-2)))
       true)
      ((and (equal? day (quote monday))
            (equal? slot (quote 2-4)))
       true)
      ((and (equal? day (quote tuesday))
            (equal? slot (quote 12-2)))
       true)
      ((and (equal? day (quote tuesday))
            (equal? slot (quote 2-4)))
       true)
      true))
```
While this procedure works correctly, Cody has failed to realize that this code is far longer than it needs to be!

**Task:** Using **and** and **or**, write **can-cody-come** without using any of: **cond, true, false**.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

### 3 What Are Good Tests?

When you write code in CS17, you’ll also write tests to make sure your code does what it is supposed to do. The way we do this is by using the built-in procedure **check-expect**. **check-expect** takes in two arguments and checks to see if their value is the same. The test will pass if the values are the same and fail if not. For example, `(check-expect (+ 1 2) 3)` would check if `(+ 1 2)` evaluates to the same value as 3.

Not all tests are good tests, though. Let’s say we have a procedure called **positive-half** that takes in `n`, a number greater than 0, and returns half of that number. `(check-expect (positive-half 8) 4)` would be a good test, since it takes in a valid input and checks that it is equivalent to the correct output. However, `(check-expect (positive-half 3) 3)` would not be a good test, since 3 is not half of itself. Furthermore, `(check-expect (positive-half -2) -1)` would also be a bad test, because while -1 is half of -2, **positive-half** is only supposed to take in positive numbers, so it may not behave in predictable ways on negative numbers. In addition, two different implementations of **positive-half** might work perfectly well on the positive numbers, but return different values for -2!

Now consider the **sum-angles** procedure from the homework. This procedure takes in an integer greater than or equal to three, `n`, and returns the sum, in degrees, of the interior angles in an `n`-gon. The formula for this is $180(n - 2)$.

**Task:** Determine which of the follow **check-expects** are good tests and which are bad.

```
(check-expect (sum-angles 3) 180)
(check-expect (sum-angles 5) 540)
(check-expect (sum-angles 10) 1800)
(check-expect (sum-angles 4) 360)
(check-expect (sum-angles 0) -360)
(check-expect (sum-angles 2) 0)
(check-expect (sum-angles 6) 720)
```
4 cons, car and cdr

Task: We'll jump ahead of the lectures and play around with built-in procedures that have not yet been covered in class.

1. Type the following expressions into DrRacket and record the output.

   1. empty
   2. (cons 1 empty)
   3. (cons 1 (cons 2 empty))
   4. (cons 1 (cons 2 (cons 3 empty)))
   5. (cons 1 (cons 2 (cons 3 (cons 4 empty))))
   6. (cons (cons 1 empty) (cons (cons 2 empty) empty))
   7. (cons 1 (cons (cons (cons 2 empty) (cons (cons 3 empty) empty)) empty))

2. It should be clear that the symbol cons is bound to a procedure. Carefully describe what the procedure does. (What input(s) does it expect and how does the output relate to the input?)

3. Type an expression without using quote whose value is the list denoted by (1 (2 3) 4 (5) 6).

4. Type the following expressions into DrRacket and record the output.

   1. (car (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
   2. (cdr (cons 1 (cons 2 (cons 3 (cons 4 empty)))))

5. Explain what car and cdr appear to do.

5 More Lists

In this section of the lab, you will learn how to implement procedures that take in lists with finite lengths in Racket. For these procedures, you’ll need to know about two more built-in procedures: empty? takes in a list and returns true if the list contains 0 elements and false otherwise.

cons? takes in a quoted list and returns true if the list contains 1 or more elements and false otherwise.

Both return false if the input is not a quoted list.
Task: Write a procedure length/many which takes in a quoted list and outputs the number of items in the list if this number is less than five. If there are five or more elements in the list, instead output the symbol many.

Task: Write a procedure short-reverse which takes in a list with at most three elements and outputs a list with the same elements in the input list in reverse order.

Hint: Feel free to use the length/many procedure you implemented in the previous task.

Task: Write a procedure shuffle which takes in two lists each with exactly two elements and outputs a list of four elements where the first element is the first element of the first list, the second element is the first element of the second list, the third element is the second element of the first list, and the fourth element is the second element of the second list.

6 Optional: Even More Lists!

Task: Write a function called index, whose inputs are $L$, a list of at most three items, and $k$, a positive integer of at most the number of items in $L$. The output of index should be the $k^{th}$ item in $L$.

Task: Write a procedure called select whose inputs are $L$, a list of at most three integers, and $K$, a list of at most two positive integers, none of which is greater than the length of $L$. The output should be a list of the same length as $K$; the $i^{th}$ item of the output should be $(\text{index } L k)$, where $k$ is the $i^{th}$ item of $K$.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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.