Lab 4: Style, Recursion, Runtimes
12:00 PM, Sep 30, 2018

Contents

1 Style By Fire! 1

2 Testing Testing 2

3 Chutes and Ladders 2

4 Long Multiplication 3

5 Long Numbers, Long Lists 3

6 Long Runtimes 3

1 Style By Fire!

HTA Cody, realizing he’s behind on his work for CS17, wrote the solutions for all of the labs, homeworks and projects last night. Unfortunately, in his frantic coding, Cody forgot about CS17’s style guidelines! Help him fix this code before the course falls apart!

Task: Look through the code below and find all of the style mistakes you can.

Hint: The mistakes might not just be in the code; make sure to look at the design recipe too!

; input: a list of numbers
; output: a list containing all of the elements of aloi incremented
;
; input: (1 2 3)
; recursive input: (2 3)
; recursive output: (3 4)
; output: (2 3 4)
;
; input: ()
; recursive input: ()
; recursive output: ()
; output: ()
(define increment-all
  (lambda (aloi)
    (cond
      ((equal? (equal? aloi (quote ())) true) (quote ()))
      ((equal? (equal? aloi (quote ())) false)
       (cons (+ (car aloi) 1) (increment-all (cdr aloi))))))))
; input: a list of data, alod
; output: the item of aloi at index n
;
; input: 2 (1 2 3)
; recursive input: (2 3)
; recursive output: 3
; output: 3
;
; input: 1 (1 2 3)
; recursive input: 0 (2 3)
; recursive output: 2
; output: 2
(define get
  (lambda (n alod)
    (cond (= n 0) (car alod))
      (= n 1) (car (cdr alod)))
    (> n 1) (get (- n 1) (cdr alod)))
    (< n 0) (quote none))))

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

2 Testing Testing

In addition to ignoring the style guide, Cody has also forgotten to test any of his code! He has no idea if any of it works, and he needs your help!

**Task:** For each of the following procedure specifications, write a test suite. Make sure to test any “edge cases” or special cases, and to have a wide variety of tests to make sure the code works under many circumstances.

1. **list-max**, a procedure that takes in a non-empty list of numbers and outputs the maximum element from the list.

2. **all-greater**, a procedure that takes in a number, n, and a list of numbers, aloi, and outputs a list containing all of the elements of aloi that are greater than n.

3. **count-zeros**, a procedure that takes in a list of numbers and outputs the number of zeros in the list.

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

3 Chutes and Ladders

You and your friends are having a blast playing chutes and ladders. However, soon you realize the game has become too easy. You decide to challenge yourself by only following paths of alternating chutes and ladders. As a further challenge, your friend requires that the path always end in a chute.
Task: Write a procedure, chutes-and-ladders, that takes as input a non-negative integer num and returns a list of symbols containing num alternating dashes (-) and tildes (~), ending with a tilde. Make sure to fill out the design recipe and write tests, too!

Examples:

(chutes-and-ladders 5) => (~ - ~ - ~)
(chutes-and-ladders 4) => (- ~ - ~)
(chutes-and-ladders 0) => empty

Hint: You may find the built-in procedure even? useful.

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

4 Long Multiplication

In this problem, you will write a simple but very slow multiplication procedure, long-long-mult, that takes as input one non-negative integer, a, and another integer, b, and outputs their product. Your procedure should use an algorithm that computes $a \cdot b$ by adding $b$ to itself $a$ times.

Task: Write the long-long-mult procedure.

Task: Write a recurrence relation for the number of operations executed by long-long-mult. “Let $f(a)$ be the worst-case number of operations performed when the first argument is $a$...”

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

5 Long Numbers, Long Lists

Task: Write a procedure, n-element-digit-list, that takes a nonnegative integer n and outputs an n-element list consisting of as many repetitions as necessary of the sequence of integers 0 9 8 7 6 5 4 2 1 in that order. The last element of the list must be 1. The first element can be any integer.

(n-element-list 10) => (0 9 8 7 6 5 4 3 2 1)
(n-element-list 11) => (1 0 9 8 7 6 5 4 3 2 1)
(n-element-list 12) => (2 1 0 9 8 7 6 5 4 3 2 1)

Task: Write a procedure, n-digit-number, that takes a nonnegative integer n and outputs an n-digit nonnegative integer of the form 12345678901234567890123... That is, the sequence of digits starts with 1, 2, 3, continuing up to 9, then 0, then repeating the pattern. Your procedure should not use n-element-list.
You've reached a checkpoint! Please sign up to get a lab TA to review your work.

6 Long Runtimes

Now, let’s measure the runtimes of a few different procedures. In addition to long-long-mult, we’ll measure the runtime of my-reverse from the last lab and parity from lecture. The code for parity can be found below:

```
(define parity
  (lambda (n)
    (cond
      ((zero? n) (quote even))
      ((equal? (parity (- n 1)) (quote even)) (quote odd))
      ((equal? (parity (- n 1)) (quote odd)) (quote even))))
```

Let’s see how long these implementations take! To do this, we’re going to use the built-in procedure time. time times how long an expression takes to evaluate. Try typing `(time (n-digit-number 300))` into the interpreter.

Notice how it prints out three different numbers for time. For our purposes, we only have to worry about the first number. Notice also how, after printing out the time information, it printed out the entire number we generated. With the procedures we’re currently dealing with, we don’t actually care about the outputs, just the time taken to find them. For this reason, we’ve written another procedure for you, suppress-output. suppress-output takes an expression to be evaluated and, after evaluating the expression, returns 0. The code for suppress-output is as follows:

```
(define suppress-output
  (lambda (x) 0))
```

Try running `(time (suppress-output (n-digit-number 300)))`. See how much cleaner that is?

Now, the moment we’ve been waiting for.

**Task:** Run the following procedures with the following inputs, and record how long they take. You should run each a few times, since the amount of time it takes Racket to evaluate an expression can vary.

**long-long-mult** For the following, bind an identifier to these values using `define` outside of the call to `time`, so the time it takes to make the list isn’t included in the runtime.

1. `a = (n-digit-number 4), b = (n-digit-number 4)`
2. `a = (n-digit-number 5), b = (n-digit-number 5)`
3. \( a = (\text{n-digit-number 6}), b = (\text{n-digit-number 6}) \)
4. \( a = (\text{n-digit-number 7}), b = (\text{n-digit-number 7}) \)

my-reverse As above, make sure to define these values before calling my-reverse.

1. alod = (n-element-list 500)
2. alod = (n-element-list 1000)
3. alod = (n-element-list 2000)
4. alod = (n-element-list 4000)

parity
1. \( n = 20 \)
2. \( n = 30 \)
3. \( n = 40 \)
4. \( n = 50 \)

Now, let’s test out the built-in \(*\) procedure. Like in my-reverse, you’ll want to save the following numbers as variables before you test the runtime. Test \(*\) on the following inputs:

1. \( a = (\text{n-digit-number 5000}), b = (\text{n-digit-number 5000}) \)
2. \( a = (\text{n-digit-number 25000}), b = (\text{n-digit-number 25000}) \)
3. \( a = (\text{n-digit-number 50000}), b = (\text{n-digit-number 50000}) \)
4. \( a = (\text{n-digit-number 100000}), b = (\text{n-digit-number 100000}) \)
5. \( a = (\text{n-digit-number 200000}), b = (\text{n-digit-number 200000}) \)

**Task:** Make a log-log plot of runtime with relation to input size for each of the preceding procedures using the data you’ve collected. A log-log plot of data \((a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\) is a plot of the points \((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\) where \(x_1 = \log a_1, y_1 = \log b_1, x_2 = \log a_2, y_2 = \log b_2, \ldots, x_k = \log a_k, y_k = \log b_k\). You can use \((\log x)\) in Racket to find the logarithm of \(x\) with base e (2.718281828...).

**Note** The input size of an integer is the number of digits making up the integer.
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](http://cs.brown.edu/courses/csci0170/feedback)