CMPU 101 § 55 · Computer Science I · Spring 2019

Assignment 4

Due March 27, 9:00 a.m.

Problem 1

Define a deeply recursive Scheme procedure `replace*` that takes three arguments: a datum (e.g., a symbol) `old`, a datum `new`, and a possibly nested list `lst`. It should return a new expression that results from replacing every occurrence of `old` in `lst` with `new`.

```
(replace* 'a 'x '(a b () c a))           ⇒  (x b () c x)
(replace* 'a 'x '((a c a) (a b) a))      ⇒  ((x c x) (x b) x)
(replace* '() 'b '(a (() b) () c))       ⇒  (a (b b) b c)
(replace* '(1 2) 'n '((1 2) (w (1 2))))  ⇒  (n (w n))
(replace* 'a 'b '())                     ⇒  ()
```

Problem 2

Define a deeply recursive Scheme predicate `all-greater*?` that takes two arguments: a number `n` and a possibly nested list of numbers `lst`. It should return `#t` if every number in `lst` is greater than `n`. Otherwise it should return `#f`. You won't be penalized for leaving it as a `cond`, but writing it without using `if` or `cond` would build character.

```
(all-greater*? 3 '())           ⇒  #t
(all-greater*? 3 '(4 5 6))      ⇒  #t
(all-greater*? 3 '(() ()))      ⇒  #t
(all-greater*? 2 '(1 (3 1) 5))  ⇒  #f
(all-greater*? 0 '(4 (3 1) 5))  ⇒  #t
```

Problem 3

Write a Scheme procedure called `up-to-by-two` that takes a positive integer `n` as input. If `n` is even, it should return a list of positive even integers from `2` to `n`. If `n` is odd, it should return a list of positive odd integers from `1` to `n`. The returned list should be in increasing order from left to right, as shown below. Use the accumulator method in your solution.

```
(up-to-by-two 0)   ⇒  ()
(up-to-by-two 1)   ⇒  (1)
(up-to-by-two 5)   ⇒  (1 3 5)
(up-to-by-two 10)  ⇒  (2 4 6 8 10)
```

Problem 4

Define a function that satisfies the following contract:

```
;; REM-DUPES
;; ---------
;; INPUTS: LST, any list
;; OUTPUT: A list containing the same elements as LST, but without any
;;   duplicates
```

The order of the elements in the output list does not really matter, but try to preserve as much of the order of elements in `lst` as possible. Here are some examples:

```
> (rem-dupes '(1 1 1 1 1))
(1)
> (rem-dupes '(a b r a c a d a b r a))
(a b r c d)
> (rem-dupes '(1 2 3 1 2 3 1 2 3 4 3 2 1))
(1 2 3 4)
```

Use an accumulator-based helper function, `rem-dupes-acc`, that satisfies the following contract.

```
;; REM-DUPES-ACC
;; -------------
;; INPUTS: LST, any list
;;         ACC, a list accumulator
;; OUTPUT: When called with ACC = (), the output is a list that contains
;;   the same elements as LST, but without any duplicates.
```

Hint: In the recursive case, use the built-in `member` function to decide whether or not the first element of `lst` already appears in the accumulator. Use that information to decide whether or not to accumulate it now.

Problem 5

Write a Scheme procedure called `partition` that takes as input `lst`, a list of numbers and `pivot`, a number. The procedure will return a list of three lists, which contain the elements below, equal to, and above the pivot, respectively. The order of numbers in each list doesn't matter, but each element in the output must appear the same number of times that it appears in the input.

```
(partition '() 36)                ⇒  (() () ())
(partition '(1 2 3 4 5) 3)        ⇒  ((1 2) (3) (4 5))
(partition '(8 -2 4 37 16 2 2) 2) ⇒  ((-2) (2 2) (8 4 37 16))
(partition '(9 4 7 8) 3)          ⇒  (() () (9 4 7 8))
```

Problem 6

Define a function called `print-and-sum-n-tosses` that satisfies the following contract:

```
;; PRINT-AND-SUM-N-TOSSES
;; ----------------------
;; INPUTS: N, a non-negative integer
;; OUTPUT: The sum of N random tosses of a six-sided die
;; SIDE EFFECT: Prints out the tosses along the way
```

Here are some sample interactions:

```
> (print-and-sum-n-tosses 5)
2 2 6 6 2 : 18
> (print-and-sum-n-tosses 5)
3 3 5 3 5 : 19
> (print-and-sum-n-tosses 5)
3 1 5 1 2 : 12
```

Note that the numbers to the left of the `:` are side-effect printing, whereas the numbers to the right of the `:` are output values.

Hints: Define `print-and-sum-n-tosses-acc` as a tail-recursive helper function that takes an accumulator `acc` as an extra input. In the recursive case, store the current toss in a local variable before printing it and making the tail-recursive function call. Then you can define `print-and-sum-n-tosses` as a wrapper function that calls the helper with appropriate inputs.

Submitting

Don't forget to submit your work using the `submit101` command!

```
submit101 g-asmt04 asmt04
```

(If the name of your directory is different from `asmt04`, change `asmt04` to whatever the name of your directory is.)