Chapter 15

Local Variables, Local Environments

Problem 15.1

This problem has two parts. The first part can be implemented without using `let`*; the second part is best implemented using* `let`*.*

(a) Define a function called `print-n-tosses` *that satisfies the following contract:*

```
;;  PRINT-N-TOSSES
;; ---------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints the results of N random tosses of a
;;    six-sided die in the Interactions Window.
```

Here are some examples:

```
> (print-n-tosses 10)
5 2 6 4 5 6 3 2 3 1
> (print-n-tosses 10)
5 2 6 1 5 6 2 5 5 3
> (print-n-tosses 10)
6 2 6 4 5 3 1 2 5 3
```

(b) Define a function called `print-and-sum-n-tosses` *that satisfies the following contract.*

```
;;  PRINT-AND-SUM-N-TOSSES
;; ------------------------------
;;  INPUT:   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 an accumulator-based, tail-recursive function called `print-and-sum-n-tosses-acc` *that takes an accumulator* `acc` *as an extra input. (You may wish to review Section 14.3.) In the recursive case, store the current toss in a local variable before printing it and making the tail-recursive function call. Afterward, define* `print-and-sum-n-tosses` *as a* wrapper *function that calls* `print-and-sum-n-tosses-acc` *with appropriate inputs. (You may wish to review Section 14.4.)*

Problem 15.2

Define a function, called `sum-the-even-tosses`*, that satisfies the following contract:*

```
;;  SUM-THE-EVEN-TOSSES
;; ------------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;  OUTPUT:  The sum of the even tosses out of N random tosses
;;  SIDE EFFECT:  Print out the tosses along the way.
```

Here is an example of its use:

```
> (sum-the-even-tosses 5)
3 6 2 1 5                      ⟵ side-effect printing: the 5 tosses
8                              ⟵ output: the sum of the even tosses
> (sum-the-even-tosses 7)
2 4 1 6 6 3 2
20
```

Problem 15.3

Define a function called `num-occurs-in-n-tosses` *that satisfies the following contract.*

```
;;  NUM-OCCURS-IN-N-TOSSES
;; ------------------------------------------------------------------
;;  INPUTS:  TARGET, an integer from 1 to 6, inclusive
;;           N, a non-negative integer
;;  OUTPUT:  Reports the number of times the TARGET number showed
;;           up when tossing a six-sided die N times.
;;  SIDE EFFECT:  Prints out the random tosses along the way.
```

Here are some examples of it in action:

```
> (num-occurs-in-n-tosses 3 20)
4 3 3 1 6 3 5 6 4 1 6 5 5 4 3 5 3 3 5 2 ... 6
> (num-occurs-in-n-tosses 3 20)
4 6 5 1 6 5 3 2 3 4 2 4 2 4 4 5 3 6 3 5 ... 4
> (num-occurs-in-n-tosses 3 20)
5 4 5 1 4 2 4 3 5 3 1 1 2 5 5 1 6 4 2 3 ... 3
```

Notice that the numbers to the left of the dot-dot-dots are side-effect printing, whereas the numbers to the right of the dot-dot-dots are output values.

Hint: In the recursive case, use a `let` *special form to store the value of the toss of a die. Then print it out and decide whether you hit the target number or not.*

Note: You may choose to implement this function using tail recursion or not, as you wish. If using tail recursion, you should name your tail-recursive helper function `num-occurs-in-n-tosses-acc`*. It will need an extra input—an accumulator—that accumulates the number of occurrences of the target number over all the tosses. After your accumulator-based helper function is working properly, you should then define a wrapper function, called* `num-occurs-in-n-tosses`*, that simply calls the accumulator-based function with appropriate inputs. Of course, you may wish to implement both versions!*

Problem 15.4: Flipping coins

When flipping coins, any occurrence of n consecutive coin flips that come out the same (i.e., all `H` *or all* `T`*) may be called a* streak of length n*. For this problem, you must define a function, called* `max-streak-in-n-flips`*, that satisfies the following contract:*

```
;;  MAX-STREAK-IN-N-FLIPS
;; ----------------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  The length of the longest streak of consecutive
;;    coin flips (whether all H or all T) that occur in a
;;    sequence of N random coin flips.
;;  SIDE EFFECT:  Prints out the coin flips along the way.
```

Here are some examples of its use:

```
(max-streak-in-n-flips 10) ===> T H H H H T T T H H  ...
4
(max-streak-in-n-flips 10) ===> T H T H T H T T T H  ...
3
(max-streak-in-n-flips 15) ===> T T H T H T T H T H H H T T H  ...
3
```

In the first sequence, the longest streak involves four consecutive `H`*s. In the second and third sequences, the longest streak involves three consecutive* `H`*s.*

- ⋆ *Begin by defining a separate helper function, called* `max-streak-in-n-flips-acc` *that takes additional inputs to keep track of things such as: the value of the most recent coin flip, the number of consecutive coin flips that have just come out the same, and the maximum number of consecutive coin flips that have been seen since you started flipping coins. Once you get things working properly, you should then define your wrapper function,* `max-streak-in-n-flips`*.*

  Consider the following sequence of coin flips:

  H T H T T T H T H H H H H T T H T ...

  What information would you need to keep track of along the way to solve this problem?

- ⋆ *Be sure to test your function in cases where $n = 0$ and $n = 1$.*

Problem 15.5

Define a function, `num-tosses-until-repeat`*, that satisfies the following contract:*

```
;;  NUM-TOSSES-UNTIL-REPEAT
;; -----------------------------------
;;  INPUTS:  None
;;  OUTPUT:  An integer specifying the number of random tosses of a
;;    6-sided die until two CONSECUTIVE tosses come out the same.
;;  SIDE EFFECTS:  Prints out the tosses along the way.
```

Here are some examples that illustrate the desired behavior:

```
> (num-tosses-until-repeat)
3 5 1 3 4 2 3 6 1 3 2 2 --> Got a repeat!
12
> (num-tosses-until-repeat)
2 4 4 --> Got a repeat!
3
> (num-tosses-until-repeat)
1 6 1 4 4 --> Got a repeat!
5
```

Note that, in each case, the first line of text is side-effect printing, while the second line displays the output value.

Hint: Define a helper function, `num-tosses-until-repeat-helper`*, that does most of the work. It should satisfy the following contract:*

```
;;  NUM-TOSSES-UNTIL-REPEAT-HELPER
;; ----------------------------------------
;;  INPUTS:  NUM-TOSSES-SO-FAR, a non-negative integer
;;           MOST-RECENT-TOSS,  a non-negative integer
;;  OUTPUT & SIDE-EFFECTS similar to NUM-TOSSES-UNTIL-REPEAT
```

Note that `num-tosses-so-far` *keeps track of how many tosses have been made so far. (It accumulates the number of tosses so far.) In addition, note that* `most-recent-toss` *keeps track of the value of the most recently tossed die so that a new toss can be compared to the most recent toss.*

In the body of `num-tosses-until-repeat-helper`*, you should toss one die and store its value in a local variable. Then print it out. Then compare this new toss to the most recent toss. If they are the same, then stop; otherwise, keep going—with adjusted inputs.*

The `num-tosses-until-repeat` *"wrapper" function should let the helper function do most (or all) of the work.*

Problem 15.6

Define a function, called `num-tosses-until-three`*, that satisfies the following contract:*

```
;;  NUM-TOSSES-UNTIL-THREE
;; ------------------------------------------------------------
;;  INPUTS:  None
```

```
;;  OUTPUT:  An integer representing the number of random
;;    tosses of a 6-sided die that were required before three
;;    CONSECUTIVE tosses came out the same.
;;  SIDE EFFECTS:  Prints out the tosses along the way
```

Here are some examples of it in action:

```
> (num-tosses-until-three)
5 2 1 4 4 4  --> We got three in a row!      ⟵ side-effect printing
6                                            ⟵ output value
> (num-tosses-until-three)
1 1 4 6 6 3 6 5 3 4 5 1 5 2 4 6 2 1 3 1 3 5 2 1 2 5 6 1 6 3 4 3
1 4 1 4 6 5 3 4 4 6 2 4 2 2 5 3 5 5 5  --> We got three in a row!
51
```

Hint: Define a helper function, called `num-tosses-until-three-helper`*, that does most of the work:*

```
;;  NUM-TOSSES-UNTIL-THREE-HELPER
;; ------------------------------------------------------------
;;  INPUTS:  NUM-TOSSES-SO-FAR, an integer
;;           PREV-TOSS-1, PREV-TOSS-2, the most recent tosses
;;             (or #f if just getting started)
;;  OUTPUT:  When called with NUM-TOSSES-SO-FAR = 0, and
;;    PREV-TOSS-1 = PREV-TOSS-2 = #f, outputs the number of
;;    random tosses of a 6-sided die before three consecutive
;;    tosses came out the same.
;;  SIDE EFFECTS:  Prints out the tosses along the way.
```

Problem 15.7

Define a function, called `toss-until-doubles`*, that satisfies the following contract:*

```
;;  TOSS-UNTIL-DOUBLES
;; -------------------------------------------
;;  INPUTS:  None
;;  OUTPUT:  The sum of the first occurrence of "doubles"
;;  SIDE EFFECT:  Tosses a pair of dice, printing out the
;;    results (and their sum), until doubles are encountered!
```

Here are some examples of its use:

```
> (toss-until-doubles) ===> TOSSES: 5, 2; sum = 7
TOSSES: 1, 2; sum = 3
TOSSES: 3, 6; sum = 9
TOSSES: 2, 2; sum = 4
HEY!  We got doubles!!
4
> (toss-until-doubles) ===> TOSSES: 5, 6; sum = 11
TOSSES: 2, 6; sum = 8
TOSSES: 1, 1; sum = 2
```

```
HEY!  We got doubles!!
2
```

In the first example, each pair of tosses is printed out, along with their sum, until doubles are found. (The `2` *and* `2` *count as doubles.) Then, the message "HEY! We got doubles!" is printed out. Finally,* `4` *(i.e., the sum of the recently tossed doubles) is returned as the* output *value; it is* not *displayed as side-effect printing. Similarly, in the second example, each pair of tosses is printed out until the* `1` *and* `1` *occurrence of doubles is found. In that case,* `2` *is the output value, not side-effect printing.*

Problem 15.8

Define a function, called `toss-three-dice-until-beat-target`*, that satisfies the following contract:*

```
;;  TOSS-THREE-DICE-UNTIL-BEAT-TARGET
;; ------------------------------------------------------------
;;  INPUT:  TARGET, an integer LESS THAN 18
;;  SIDE EFFECT:  Simulates the repeated tossing of three dice,
;;    printing out the tosses and their sum, until the sum is
;;    GREATER than TARGET
;;  OUTPUT:  The sum of the three dice that beat the TARGET.
```

Here are some examples of its behavior:

```
> (toss-three-dice-until-beat-target 12)
6 + 2 + 3 = 11
3 + 1 + 2 = 6
5 + 6 + 3 = 14
14
> (toss-three-dice-until-beat-target 12)
5 + 3 + 6 = 14
14
> (toss-three-dice-until-beat-target 14)
5 + 4 + 5 = 14
6 + 3 + 6 = 15
15
> (toss-three-dice-until-beat-target 14)
1 + 1 + 5 = 7
5 + 5 + 1 = 11
4 + 1 + 3 = 8
3 + 5 + 4 = 12
1 + 3 + 1 = 5
1 + 2 + 4 = 7
2 + 2 + 2 = 6
5 + 2 + 3 = 10
5 + 5 + 3 = 13
5 + 1 + 1 = 7
4 + 6 + 6 = 16
16
```

Problem 15.9

Define a function, called `toss-until-total-beats-target`*, that satisfies the following contract:*

```
;;  TOSS-UNTIL-TOTAL-BEATS-TARGET
;; ------------------------------------------------------------
;;  INPUT:  TARGET, an integer
;;  SIDE EFFECT:  Simulates the tossing of a die, printing out
;;    all tosses along the way, until the sum of all dice tossed
;;    is *greater* than TARGET.
;;  OUTPUT:  The total of the dice that finally beat the target.
```

Here are some examples of its behavior:

```
> (toss-until-total-beats-target 10)
5 4 4  ... 13
> (toss-until-total-beats-target 10)
4 5 6  ... 15
> (toss-until-total-beats-target 20)
1 3 4 1 4 4 5  ... 22
> (toss-until-total-beats-target 20)
5 1 6 3 4 3  ... 22
```

Problem 15.10: Using `let*` to create a fuel report

Define a function, called `fuel-report`*, that satisfies the following contract:*

```
;;  FUEL-REPORT
;; ----------------------------------
;;  INPUTS:  STARTING-MILES, non-negative number representing
;;             the starting reading of the odometer of a car
;;           ENDING-MILES, non-negative number representing
;;             the ending reading of the odometer of a car
;;           COST-PER-GALLON, cost of gas purchased
;;           NUM-GALLONS, number of gallons purchased
;;  OUTPUT:  none
;;  SIDE EFFECT:  Prints out a fuel report including the number
;;    of miles traveled, the miles per gallon, the amount of
;;    money spent (in dollars), and the cost per mile (in
;;    dollars per mile).
;;  NOTE:  miles-per-gallon = num-miles-traveled / num-gallons
;;         dollars-spent = cost-per-gallon * num-gallons
;;         cost-per-mile = num-dollars-spent / num-miles-traveled
```

Here are some examples of its desired behavior:

```
> (fuel-report 0 100 5.0 10)
Miles traveled: 100, miles-per-gallon: 10
Dollars spent: 50.0, cost-per-mile: 0.5
> (fuel-report 25 75 4.0 3.0)
Miles traveled: 50, miles-per-gallon: 16.666666666666668
Dollars spent: 12.0, cost-per-mile: 0.24
```

Note that it does not generate any output; all of the text is side-effect printing.
The purpose of this problem is to practice using the `let*` *special form to simplify a sequence of computations. Thus, you should use a single* `let*` *to create a sequence of local variables with the following names:* `miles-traveled`, `miles-per-gallon`, `dollars-spent` *and* `cost-per-mile`. *Note that the value of each variable depends only on the values of variables defined before it. For example, the value of* `miles-per-gallon` *depends only on* `miles-traveled` *and* `num-gallons`. *Similarly, the value of* `miles-traveled` *depends only on the inputs* `starting-miles` *and* `ending-miles`.

Problem 15.11

Mimicking the structure of `facty` *and* `facty-acc` *from Example 15.6.3, define a function called* `sum-cubes` *that uses* `letrec` *to define an accumulator-based, tail-recursive local helper function called* `sum-cubes-acc`. *The* `sum-cubes` *function should satisfy the following contract:*

```
;;  SUM-CUBES
;; ------------------------------------------------------
;;  INPUTS:  N, a positive integer
;;  OUTPUT:  The sum:  1*1*1 + 2*2*2 + ... + N*N*N
```

Here are some examples of its desired behavior:

```
> (sum-cubes 3)
36
> (sum-cubes 4)
100
```

Problem 15.12

Same as Problem 15.1b, except that you should use `letrec` *to define the recursive helper function as a local function.*

Problem 15.13

Same as Problem 15.3, except that you should use `letrec` *to define the recursive helper function as a local function.*

Problem 15.14

Same as Problem 15.5, except that you should use `letrec` *to define the recursive helper function as a local function.*