Changes to the design recipe to accommodate mutation

Mutation allows for functions to do more than algebraic manipulation of their arguments. In particular, a function can change the value of a variable, or can have other side effects.

This is explored in some depth in Section 36.4 of *How to Design Programs*. These are the elements of the modified design recipes for programs using mutations:

- **State variables** are the way that we store the values that may change over the execution of a program. For these variables, if they exist, define them and give them a value.
- The **contract** should describe the consumed value(s) and the produced value(s) of the function. Use `(void)` to indicate that the function has no consumed values, and use `(void)` to indicate the function has no produced values.
- The **purpose** of a function should describe what the function consumes and produces.
- **Effects** are added to explain the side effects of the function, that is, how calling a function will have other effects besides computing the returned value. You should also explain what variables are changed after calling the function, how these variables change, how the consumed values, if any, are involved with the change.
- **Examples** must be revised to account for time's passage: the key thing is that a variable may have a "before" and an "after" value. For example, you need to provide sample values of variables before the function call, indicate parameter values, and indicate new value of variables after the function call. You should provide at least 2 examples per function. For question with unique input and output values, one example is usually sufficient.
- **Function body** will use `begin`, `set!` and/or structure field mutator functions where appropriate.
- **Tests:** Use `equal?` To ensure that a variable has the desired result before and after a call is made. Use `begin` to join together all parts of a single test. As the first part of the `begin` statement, you may assign the state variable; the second part of the statement usually involves mutating state variables (by calling the function); the third part of the statement uses `equal?` To verify that mutation worked as expected. Finally, use `check-expect` to compare the last expression of the being statement with the expect value produced by the `begin` statement.

The following solution illustrates how to provide the modified design recipes for *`change-x-and-y`* function.

```
;; State variables
(define x 100)
(define y 0)

;; change-x-and-y: num  -> (void)
;; Purpose: consumes a number, ch, and produces (void).
;; Effects: Changes the values of x or y as follows:
;;    * if ch < 0, then x is decreased by 1,
;;    * if ch > 0, then x is increased by 1,
;;    * if ch = 0, then x is not changed, but y is
;;      set to the current value of x.
;; Examples:
;; if x and y are 0, after calling the function
;;    (change-x-and-y 5), x will be 1 and y will be 0.
;; if x is 100 and y is 3, after calling the function
;;    (change-x-and-y -5), x will be 99 and y will be 3.
;; if x is -5 and y is 7, after calling the function
;;    (change-x-and-y 0), x and y will both be -5.

(define (change-x-and-y ch)
  (cond [(< ch 0) (set! x ( x 1))]
        [(> ch 0) (set! x (+ x 1))]
        [else     (set! y x)]))

;; Test with ch > 0
(check-expect (begin (set! x 5)
                     (set! y 12)
                     (change-x-and-y -3)
                     (and (= x 4) (= y 3)))
              true)

;; Test with ch < 0
(check-expect (begin (set! x 33)
                     (set! y -12)
                     (change-x-and-y 7)
                     (and (= x 34) (= y -12)))
              true)

;; Test with ch = 0
(check-expect (begin (set! x -5)
                     (set! y 2)
                     (change-x-and-y 0)
                     (and (= x -5) (= y -5)))
              true)
```