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 which may change over the execution of a program. For these variables, if they exist, define them and give them a value. (The state variables used in Assignment 5 are provided in the interface file.)
- 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 questions with unique input and output values (such as a5q1), one example is usually sufficient.
- **Function body** will use **set!** or other 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 variables; 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 **begin** statement with the expect value produced by the **begin** statement.

Suppose we write a function *change-x* which consumes a number *ch* and changes the value of a previously defined variable *x* in the following ways. 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.

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

```
;; State variable
;; x: num
;; initial value of x
(define x 100)
;; Contract
;;change-x: num -> (void)
;;
;; Purpose: this function consumes a number, and produces (void).
;;
;; Effects: may change the value of a defined variable x
;; 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.
;;
;;Examples
;; if x is 0, after calling the function (change-x 5), x will be 1.
;; if x is 100, after calling the function (change-x -5), x will be 99.
;; if x is -5, after calling the function (change-x 0), x will still be -5.
;;
;; Function body
;;
(define (change-x ch)
  (cond
    [(< ch 0) (set! x (- x 1))]
    [(> ch 0) (set! x (+ x 1))]
    [else (void) ]))
;;
;; Tests
;;
;; one test to check the third cond expression
(check-expect (begin
                (set! x -5)
                (change-x 0)
                (equal? x -5))
              true)
;;
;; one test to check the second cond expression
(check-expect (begin
                (set! x 3)
                (change-x 5)
                (equal? x 4))
              true)
```

```
;;
;; one test to check the first cond expression
(check-expect (begin
                (set! x -2)
                (change-x -5)
                (equal? x -3))
              true)
;;
;; You may also do the following type of tests:
;; producing the value from the begin statement and
;; compare it with the expect value.
(check-expect (begin
                (set! x -4)
                (change-x 5)
                (change-x -7)
                x)
              -4)
```