v.6.11

Resolución práctica 2

A continuación presentamos soluciones a ejercicios seleccionados de la Práctica 2.

Como siempre, en el proceso de resolución de un problema, primero debemos dedicar un tiempo al análisis del mismo para asegurarnos de entender correctamente qué se pide. Luego, pasamos a la resolución propiamente dicha.

Recomendamos usar este documento como un elemento de ayuda y referencia. Es importante que no lean la solución sin antes haber leído y entendido el problema y sin haber dedicado un tiempo a pensar cómo resolverlo.

Por otro lado, no hay una única forma correcta de escribir una solución.

1 Diseñando funciones

En esta práctica en particular hay un enfoque mucho mayor en plasmar el análisis que venimos haciendo en el código, mediante el uso de la receta. Por esto, si bien los ejercicios no plantean una dificultad mayor a las prácticas anteriores, el eje de estos ejercicios recae en cómo expresamos el análisis que hacemos en papel antes de encarar un ejercicio.

1.1 Ejercicio 4

En este ejercicio debemos hacer una función `area-cubo` que calcule el área de un cubo dada la longitud de su arista.

Nuestro primer paso es el **diseño de datos**. La longitud de una arista es un número natural, por lo que la podemos representar con un número.

```
Representamos la longitud de la arista con un número.
```

Luego, debemos proseguir con la **signatura y declaración de propósito**.

Como dijimos anteriormente, la arista es un número natural, y el área también, por lo que la signatura sería:

```
area-cubo: Number -> Number
```

Como declaración de propósito podemos decir:

```
Recibe un valor que representa la longitud de una arista y calcula su área.
```

Luego, pensemos algunos **ejemplos**:

El área de un cubo es la arista al cubo. Por lo tanto algunos casos posibles serían:

```
Entrada: 5,Salida: 125
Entrada: 4,Salida: 64
Entrada: 12,Salida: 1728
```

Y luego pasamos a **definir** la función en sí. Dado que ya sabemos que debemos tomar un número natural y calcular su cubo, solo nos queda expresar eso en Racket.

```
; Representamos la longitud de la arista con un número.
; area-cubo: Number -> Number
; Recibe un valor que representa la longitud de una arista y calcula su área.
```

```
; entrada: 5, salida 125
; entrada: 4, salida 64
; entrada: 12, salida 1728

(define
    (area-cubo n)
    (expt n 3))
```

Como siempre, debemos **evaluar** los casos de prueba para corroborar que den los resultados esperados.

> Recordar que solo porque el resultado sea el esperado en los casos que pensamos NO significa que la función sea correcta. Además, ¿tiene sentido calcular el área de lado cero?

```
> (area-cubo 5)
125
> (area-cubo 4)
64
> (area-cubo 12)
1728
> (area-cubo 0)
0
```

1.2 Ejercicio 6

En este ejercicio debemos diseñar una función que dada una cadena no vacía de caracteres, retorne la cadena sin su último caracter.

> El ejercicio pide que la cadena no sea vacía, ¿por qué?

Haremos el proceso en forma similar al ejercicio anterior.

Inicialmente comenzamos por el **diseño de datos**. Dado que nuestra entrada es una cadena de caracteres, la representaremos con un string.

`Representamos la cadena de caracteres con un string.`

Luego, seguiremos con la **signatura y declaración de propósito**.

Como dijimos anteriormente, la cadena es un string y el resulado sigue siendo una cadena, por lo tanto:

`string-last: String -> String`

Luego, como declaración de propósito podemos decir:

`Recibe un cadena no vacía y retorna la misma sin su último caracter.`

Pensemos algunos **ejemplos**:

```
Entrada: "Hola mundo",Salida: "Hola mund"
Entrada: "Estres",Salida: "Estre"
Entrada: "Supercalifragilisticoespialidoso",Salida: "Supercalifragilisticoespialidos"
```

Pasamos a **definir** la función.

Sabemos que el objetivo es quitar el último caracter de un string. Para resolver esto, podemos utilizar la función `substring`. `(substring str start end)` retorna un string que contiene los mismos caracteres que `str`, desde la posición `start` inclusive, hasta la posición `end` sin incluir.

En nuestro caso, necesitamos el substring que va desde la posición 0, hasta la posición final, sin incluir. La posición del último caracter la podemos obtener restándole 1 a la longitud del string.

Finalmente, en Racket, el diseño nos queda como se muestra a continuación.

```
; Representamos la cadena recibida con un string
; string-last: String -> String
; Recibe un cadena no vacía a la cual le retira el último caracter y devuelve la cadena restante
; Entrada: "Hola mundo", Salida: "Hola mund"
; Entrada: "Estres", Salida: "Estre"
; Entrada: "Supercalifragilisticoespialidoso", Salida: "Supercalifragilisticoespialidos"
(define
    (string-last s)
    (substring s 0 (- (string-length s) 1)))
```

Finalmente nos queda **evaluar** los casos de prueba y corroborar que den los resultados esperados.

```
> (string-last "Hola")
"Hol"
> (string-last "Estres")
"Estre"
> (string-last "Supercalfragilisticoespialidoso")
"Supercalfragilisticoespialidos"
> (string-last "LosGatosSonClaramenteMejoresQueLosPerros")
"LosGatosSonClaramenteMejoresQueLosPerro"
```

1.3 Ejercicio 9

Leamos atentamente el enunciado buscando comprender cuál es el problema que vamos a resolver, con qué datos contamos para encontrar una solución y de qué forma vamos a mostrar la resolución del mismo. Esto nos ayudará a la hora de aplicar la receta para el diseño

Primero nos encargamos del diseño de datos. Algunas preguntas que nos pueden ayudar en esta etapa son:

- ¿Con qué datos contamos para iniciar la resolución del problema?
- ¿De qué forma se pueden transformar esos datos?
- ¿Qué datos nos pide que devolvamos como resolución al problema?

El enunciado nos dice que vamos a contar con la cantidad de personas que se están anotando al instituto y la cantidad de meses que abonan todos así que podemos empezar describiendo cómo vamos a representar esos datos:

```
; Representamos cantidades de personas mediante números.

; Representamos cantidades de meses mediante números.
```

Naturalmente trabajaremos con montos de dinero:

```
; Representamos montos de dinero mediante números.
```

También vamos a trabajar con descuentos. Optamos por utilizar un valor entre 0 y 1 para respresentar un descuento:

> ¿Podríamos haber elegido otra representación para los descuentos? ¿Esta representación afecta la forma en la que vamos a escribir nuestras funciones?

```
; Representamos los descuentos como un valor numérico entre 0 y 1
; que representan 0% de descuento y 100% de descuento,
; respectivamente.
```

Por el momento con esos datos parece bastar. Si en el futuro necesitamos manipular otros datos o transformarlos simplemente agregamos descripciones para las nuevas representaciones elegidas.

A continuación, definimos algunos de los parámetros del problema como constantes simbólicas. Estos valores pueden llegar a cambiar en el futuro y al definirlos como constantes podemos introducir los cambios de forma sencilla y sin manipular la lógica del programa.

```
; Valor original de la cuota mensual.
(define VALOR_CUOTA_MENSUAL 650)

; Descuento máximo aplicable a la cuota mensual.
(define DESCUENTO_MAXIMO 0.35)

; Descuento correspondiente a la promoción por personas,
; cuando se anotan 3 o más.
(define DESCUENTO_PERSONAS_3_O_MAS 0.2)

; Descuento correspondiente a la promoción por personas,
; cuando se anotan 2.
(define DESCUENTO_PERSONAS_2 0.1)

; Descuento correspondiente a la promoción por meses,
; cuando se cancelan 3 o más.
(define DESCUENTO_MESES_3_O_MAS 0.25)

; Descuento correspondiente a la promoción por meses,
; cuando se cancelan 2.
(define DESCUENTO_MESES_2 0.15)
```

Podemos comenzar definiendo algunas funciones que nos ayuden a calcular los descuentos de las distintas promociones disponibles. Podemos observar las siguientes propiedades de las promociones:

- La primera, que depende únicamente de la cantidad de personas que se anotan a la clase.
- La segunda, que depende únicamente de la cantidad de meses juntos que se abonan al momento de pagar.

Podemos calcular el descuento obtenido a través de la primera promoción, que depende sólo de la cantidad de personas. Para el caso en que la promoción no aplique, elegimos devolver un descuento del 0%. Escribimos una descripción del funcionamiento de la función, algunos casos de uso y luego la implementación:

> Es importante que la declaración de propósito y los casos de uso se escriban antes de comenzar a escribir la implementación. ¿Por qué?

```
; Number -> Number
; Dada una cantidad de personas, calcula el descuento aplicable
; según la promoción vigente.
(check-expect (descuento-personas 3) 0.2)
(check-expect (descuento-personas 2) 0.1)
(check-expect (descuento-personas 1) 0)
(define (descuento-personas personas)
  (cond
    [(>= personas 3) DESCUENTO_PERSONAS_3_O_MAS]
    [(= personas 2) DESCUENTO_PERSONAS_2]
    [else 0]))
```

Procedemos de forma similar para calcular el descuento proporcionado por la otra promoción:

```
; Number -> Number
; Dada una cantidad de meses, calcula el descuento aplicable
; según la promoción vigente.
(check-expect (descuento-meses 3) 0.25)
```

```
(check-expect (descuento-meses 2) 0.15)
(check-expect (descuento-meses 1) 0)
(define (descuento-meses meses)
  (cond
    [(>= meses 3) DESCUENTO_MESES_3_O_MAS]
    [(= meses 2) DESCUENTO_MESES_2]
    [else 0]))
```

Dado que las promociones son combinables elegimos ahora diseñar una función que se encargue de combinar los beneficios de ambas, teniendo en cuenta la limitación impuesta por el problema. Podemos combinar las funciones que definimos previamente en una nueva:

> ¿Por qué esta función utiliza la función `min`? ¿Podríamos haber utilizado otra función para hacer lo mismo?

```
; Number Number -> Number
; Dadas una cantidad de personas y una cantidad de meses, calcula
; el descuento total considerando las promociones vigentes y teniendo
; en cuenta el descuento máximo.
(check-expect (descuento-total 2 1000) 0.35)
(check-expect (descuento-total 2 2) 0.25)
(check-expect (descuento-total 1 1) 0)
(define (descuento-total personas meses)
  (min (+ (descuento-personas personas) (descuento-meses meses)) DESCUENTO_MAXIMO))
```

El enunciado nos pide calcular el monto que el instituto debe cobrarle a cada nuevo alumno luego de aplicarse los descuentos que correspondan por las promociones. Para llegar a esto, podemos definir primero una función que calcule el monto bruto a pagar (es decir, el monto sin descuentos). Esto depende únicamente de la cantidad de meses:

```
; Number -> Number
; Dada una cantidad de meses, calcula el total a pagar sin aplicar
; descuentos.
(check-expect (monto-bruto 1) 650)
(check-expect (monto-bruto 6) 3900)
(check-expect (monto-bruto 2) 1300)
(define (monto-bruto meses)
  (* VALOR_CUOTA_MENSUAL meses))
```

Luego, podemos hacer una función que a un monto de dinero le aplique un descuento:

> ¿Habría cambiado la definición de esta función si nuestra representación de los descuentos fuera distinta?

```
; Number Number -> Number
; Dados un monto de dinero y un descuento, calcula el monto de dinero
; luego de aplicar el descuento.
(check-expect (aplicar-descuento 1000 0) 1000)
(check-expect (aplicar-descuento 1000 0.25) 750)
(check-expect (aplicar-descuento 1000 0.5) 500)
(define (aplicar-descuento monto descuento)
  (* monto (- 1 descuento)))
```

Finalmente, el resultado pedido por el ejercicio se desprende fácilmente de combinar funciones anteriores. Simplemente calculamos el total sin descuentos `monto-bruto` y le aplicamos el descuento obtenido de ambas promociones `descuento-total`. Los ejemplos de uso se obtienen del enunciado:

```
; Number Number -> Number
; Dadas una cantidad de personas y una cantidad de meses, calcula
; el monto que el instituto
; debe cobrarle a cada persona.
; Ejemplo 1 (Pedro, Juan; 2 meses)
(check-expect (monto-persona 2 2) 975)
; Ejemplo 2 (Pedro, Juan, Paula; 3 meses)
```

```
(check-expect (monto-persona 3 3) 1267.5)
; Ejemplo 3 (José; 5 meses)
(check-expect (monto-persona 1 5) 2437.5)
(define (monto-persona personas meses)
  (aplicar-descuento (monto-bruto meses) (descuento-total personas meses)))
```

1.4 Ejercicio 11

Este ejercicio nos pide definir una función que, dados tres números, devuelva el producto entre ellos en caso de que formen una terna autopromediable y, en caso contrario, la suma de los mismos. Por lo tanto, el resultado final será una función que llamaremos `ejercicio11`.

Para definir dicha función, primero nos concentramos en resolver cómo determinar si tres números conforman una terna autopromediable. ¿Qué necesitamos para esto? Veamos la definición.

Tres números conforman una *terna autopromediable* si uno de sus valores concide con el promedio de los otros dos.

Por lo tanto, necesitaremos calcular el promedio de dos números. Resolvamos este pequeño problema en una función auxiliar, utilizando la receta.

```
; Diseño de datos:
; Es trivial
; Declaración de propósito:
; Dados dos números, determina el promedio
; Signatura:
; Number Number -> Number
; Casos de prueba:
(check-expect (promedio 2 3) 2.5)
(check-expect (promedio -2 8) 3)
(check-expect (promedio -5 -4) -4.5)

(define (promedio a b) (/ (+ a b) 2))
```

Volvamos a pensar en la definición de terna autopromediable. Dados tres números `a`, `b` y `c` que conforman una terna, sabemos que ni el máximo ni el mínimo de ellos puede ser el promedio de los restantes (a menos que los tres valores sean iguales). ¿Por qué es cierta esta afirmación?

Habiendo pensado esto, podemos definir una función auxiliar `promediable?`, que compare por igualdad el promedio entre el valor mínimo y el valor máximo de la terna, con el valor medio.

```
; Diseño de datos:
; Es trivial
; Declaración de propósito:
; Determina si tres números conforman una terna autopromediable o no.
; Signatura:
; Number Number Number -> Bool
; Casos de prueba:
(check-expect (autopromediable? 7 5 9) true)
(check-expect (autopromediable? -15 -15 -15) true)
(check-expect (autopromediable?  -0.5 12 -13) true)
(check-expect (autopromediable? 15 12 30) false)
(check-expect (autopromediable? -15 -12 -30) false)

(define (autopromediable? a b c) (= (promedio (max a b c) (min a b c))
                                    (- (+ a b c) (+ (max a b c) (min a b c)))))
```

Una vez resuelto este problema, es mucho más sencillo definir la función final. Para hacerlo, debemos sumar o multiplicar los valores de la terna, en función de si ésta es autopromediable o no.

```
; Diseño de datos:
; Es trivial
; Declaración de propósito:
```

```
; Dados tres números, devuelve el producto de ellos en caso que formen una terna
; autopromediable, y la suma de los mismos en caso contrario.
; Signatura
; Number Number Number -> Number
; Casos de prueba:
(check-expect (ejercicio11 7 5 9) 315)
(check-expect (ejercicio11 -7 -5 -9) -315)
(check-expect (ejercicio11 15 12 30) 57)
(check-expect (ejercicio11 -15 -12 -30) -57)

(define (ejercicio11 a b c) (if (autopromediable? a b c) (* a b c) (+ a b c)))
```

El diseño de datos se consideró trivial a lo largo de todo el ejercicio dado que el enunciado pide diseñar una función que toma tres números. La representación de los números, naturalmente, se hace a través de valores numéricos.