v.8.1

Práctica 2

Programación I

---

1 Diseñando Programas

Un programa bien diseñado es aquel que viene acompañado de una explicación de lo que hace, aquel que explica qué tipo de entradas espera y qué resultados produce. Idealmente, también debería demostrarse que realmente hace lo que se afirma.

Todo este trabajo extra es necesario ya que los/as programadores/as no construyen programas para ellos mismos. Los/as programadores/as escriben programas para que otros programadores/as puedan entenderlos. La mayoría de los programas son largos, compuestos por colecciones complejas de funciones colaborativas, y nadie puede escribir todas estas funciones en un solo día. Además, es muy frecuente que un/a programador/a se una a un proyecto, escriba algo de código y luego de un tiempo se vaya del mismo; así otros programadores/as deben retomar su labor y continuarla.

Otra dificultad es saber qué necesita realmente resolver nuestro cliente. Incluso cuando este sepa bien que problema debe resolverse, pueden ser imprecisos u omtir cierta informacion relevante.

Peor aún, construcciones lógicas complejas tales como los programas son propensas a los errores humanos. Y sí, los/as programadores/as cometen errores...

Eventualmente alguien descubre los errores y deben corregirse. Para corregirlos resulta necesario re-leer, entender y corregir programas que datan de hace más de un mes, más de un año o de hace decenas de años!

A fin de adquirir buenas prácticas de programación, se propone utilizar una receta que nos indica qué pasos hay que seguir y en qué orden para lograr un buen diseño de un programa.

1.1 La Receta

Esta receta nos indica los siguientes pasos:

1. Diseño de datos

2. Signatura y declaración de propósito

3. Ejemplos

4. Definición de la función

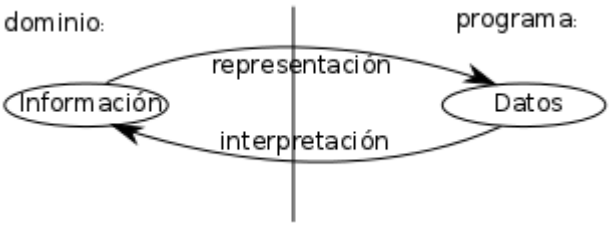
5. Evaluar el código en los ejemplos

6. Realizar modificaciones en caso que el paso anterior genere errores

Veamos más en detalle cada paso.

1. **Diseño de Datos**

   La información vive en el mundo real, y es parte del dominio del problema. Para que un programa pueda procesar información, esta debe representarse a través de datos. Asimismo, los datos producidos por un problema, deben poder interpretarse como información.



   En este paso, definimos la forma de representar la información como datos.

2. **Signatura y declaración de propósito**

   La *signatura* de una función indica qué datos consume (cuántos y de qué clase), y qué datos produce.

   Ejemplos:

```
; Number -> Number
```

   Signatura para una función que consume un número y produce un número.

```
; Number String String -> Image
```

Signatura para una función que consume un número y dos strings, y devuelve una imagen.

La *declaración de propósito* consiste en una breve descripción del comportamiento de la función. Cualquier persona que lea el programa, debe entender qué calcula la función sin necesidad de inspeccionar el código.

3. **Ejemplos**

   A partir de los dos primeros pasos, debemos estar en condiciones de predecir el comportamiento de la función para algunas entradas.

4. **Definición de la función**

   En este paso ya escribimos el código que creemos resuelve el problema planteado.

5. **Evaluar el código en los ejemplos**

   Una vez que contamos con el código de la función, la ejecutamos en los ejemplos propuestos para verificar que (al menos) funciona para esas entradas cuyas respuestas esperadas conocemos.

6. **Realizar modificaciones en caso de errores**

   Si en el paso anterior tenemos diferencias entre las respuestas esperadas y las respuestas obtenidas al ejecutar la función, debemos buscar la fuente del error. Podrían estar mal formulados los ejemplos, podría haber un error en el programa, o ambas cosas a la vez.

1.2 Ejemplo de aplicación

Para ilustrar los pasos, trabajemos con el siguiente problema:

Escribir un programa que convierta una temperatura medida en un termómetro Fahrenheit a una temperatura en Celsius.

Veamos cómo aplicar la receta para este ejemplo.

1. **Diseño de Datos**

   En este paso debemos pensar mediante qué tipos de datos representaremos la información que debe ser procesada por nuestro programa. Si bien nuestro ejemplo es sencillo, existen varias formas de representar temperaturas (por ejemplo, mediante una

enumeración que identifique *intervalos*, similar a lo visto en la práctica 1).

```
; Representamos temperaturas mediante números
```

2. **Signatura y declaración de propósito**

   La signatura se explicita en término de los datos elegidos en el paso anterior. Es de esperar entonces lo siguiente:

   ```
   ; Number -> Number
   ; recibe una temperatura en Fahrenheit, devuelve su equivalente en Celsius
   ```

   Notar que en la declaración de propósito explicamos brevemente qué transformación aplicamos a los datos de entrada para generar datos de salida.

3. **Ejemplos**

   Por lo explicado en clase de teoría, es de esperar que podamos predecir al menos dos resultados para nuestra función. También conociendo que la relación entre ambas escalas es lineal, no es difícil obtener un tercer ejemplo:

   ```
   ; entrada: 32, salida: 0
   ; entrada: 212, salida: 100
   ; entrada: 122, salida: 50
   ```

4. **Definición de la función**

   ```
   (define
       (far->cel t)
       (* 5/9 (- t 32)))
   ```

5. **Evaluar el código en los ejemplos**

   ```
   > (far->cel 32)
   0
   > (far->cel 212)
   100
   > (far->cel ( 122))
   50
   ```

   Observamos que los resultados coinciden con los esperados.

6. **Realizar modificaciones en caso de errores**

   Para este ejemplo concreto no es necesario realizar modificaciones ya que las evaluaciones en los ejemplos funcionaron tal como se

esperaba.

Como mencionamos anteriormente, en el caso de que ocurran discrepancias entre los valores esperados y los obtenidos, deberíamos revisar nuestro código en busca de errores. Los errores se pueden presentar en muchas formas y lo obtenido en la ejecución debe usarse como indicio para buscarlo.

Luego de ejecutar cada uno de los pasos de la receta, tenemos como resultado un programa bien diseñado.

El resultado final para este ejemplo concreto es el siguiente:

```
; Representamos temperaturas mediante números
; far->cel : Number -> Number
; recibe una temperatura en Fahrenheit, devuelve su equivalente en Celsius
; entrada: 32, salida: 0
; entrada: 212, salida: 100
; entrada: 122, salida: 50
(define
  (far->cel t)
  (* 5/9 (- t 32)))
```

1.3 Diseñemos funciones simples

Los siguientes ejercicios sirven como práctica para internalizar el proceso de diseño.

Algunos de estos ejercicios son casi copias de ejercicios ya resueltos en prácticas anteriores, salvo que antes usábamos la palabra "defina" y ahora utilizamos "diseñe". Lo que se espera es que trabajen utilizando la receta para crear sus funciones y las soluciones reflejen todas las etapas del diseño.

Ejercicio 1. Diseñe una función `distancia-origen`, que recibe dos números x e y, devolviendo como resultado la distancia al origen del punto (x,y).

Ejercicio 2. Diseñe una función `distancia-puntos`, que recibe cuatro números x1, y1, x2 e y2 y devuelve la distancia entre los puntos (x1, y1) y (x2, y2).

Ejercicio 3. Diseñe la función `vol-cubo` que recibe la longitud de la arista de un cubo y calcula su volumen.

Ejercicio 4. Diseñe la función `area-cubo` que recibe la longitud de la

arista de un cubo y calcula su área.

Ejercicio 5. Diseñe la función `string-insert`, que consume un string y un número i e inserta "-" en la posición i-ésima del string.

Ejercicio 6. Diseñe la función `string-last`, que extrae el último caracter de una cadena no vacía.

Ejercicio 7. Diseñe la función `string-remove-last`, que recibe una cadena y devuelve la misma cadena sin el último caracter.

1.4 Testeando funciones

Es fácil testear programas pequeños usando el área de interacción, pero la tarea se complica a medida que aumenta la complejidad del código. El testeo de funciones puede convertirse rápidamente en una tarea pesada para el/la programador/a.

Mientras más complejo sea el código de un programa, más necesidad tenemos de testearlo para detectar errores.

Por lo tanto, resulta útil contar con un mecanismo para automatizar el testeo de programas. *DrRacket* provee una funcionalidad de testeo a través de la función `check-expect`.

Recordemos el diseño que obtuvimos de la función `far->cel` utilizando la receta:

```
; Representamos temperaturas mediante números
; far->cel : Number -> Number
; recibe una temperatura en Fahrenheit, devuelve su equivalente en Celsius
; entrada: 32, salida: 0
; entrada: 212, salida: 100
; entrada: 122, salida: 50
(define (far->cel f)
        (* 5/9 (- f 32)))
```

Podemos automatizar el testeo de los ejemplos propuestos agregando las siguientes líneas de código en el área de definiciones de *DrRacket* :

```
(check-expect (far->cel 32) 0)
(check-expect (far->cel 212) 100)
(check-expect (far->cel 122) 50)
```

Una vez agregadas estas líneas podemos presionar el botón EJECUTAR, y veremos que *DrRacket* reporta que el programa ha

pasado con éxito las tres pruebas. Además de ayudar a automatizar el testeo, la función `check-expect` provee otra ventaja en caso de que falle alguna prueba. Para ver cómo funciona, cambie alguno de los casos, por ejemplo:

```
(check-expect (far->cel 122) 60)
```

Cuando presione el botón EJECUTAR, verá abrirse una ventana adicional. El texto en esta ventana explicará que uno de los tres casos falla. Para dicho caso se mostrarán: la entrada utilizada para el cálculo (`122`), el resultado obtenido al ejecutar la función (`50`) y el resultado esperado (`60`); y un link al texto de dicho caso de prueba.

El código `check-expect` puede ubicarse tanto antes como después de las definiciones que se deseen testear. Sin embargo, de acuerdo a la metodología de diseño presentada en esta práctica, **siempre escribiremos los ejemplos antes del código de la función.** En el resto del curso, y salvo casos excepcionales, preferiremos usar `check-expect` en lugar de escribir los casos de test como comentarios.

Ejercicio `8`. Para cada una de las funciones diseñadas en la sección anterior, agregue las líneas de código necesarias para automatizar los casos de prueba.

1.5 Más ejercicios

> Recuerde que quizás le resulte útil definir funciones auxiliares para contruir su solución.

Ejercicio `9`. Un Instituto de Portugués decide lanzar las siguientes promociones buscando aumentar la cantidad de alumnos:

- Si se anotan 2 amigos, cada uno obtiene un 10% de descuento sobre el valor de la cuota; mientras que si se anotan 3 o más el descuento alcanza el 20%
- Si al momento de pagar se decide abonar 2 meses juntos se recibe un descuento del 15%; en caso de cancelar 3 o más meses a la vez la reducción es del 25%

Las promociones son combinables, pero nunca pueden superar el 35% de descuento. El valor original de la cuota mensual es de $650.

La administración del Instituto nos solicitó diseñar la función `monto-`

`persona`, la cual recibe la cantidad de personas que se están anotando y la cantidad de meses que abonan (para que se aplique la promoción deben pagar la misma cantidad de meses), y devuelve el monto que el Instituto debe cobrarle a cada uno. Para desarrollar `monto-persona` es conveniente definir ciertas constantes, ya que los precios pueden variar con el tiempo.

Ejemplos:

- Supogamos que Pedro y Juan deciden anotarse al curso de Portugués pagando 2 meses juntos, obtendrán un descuento del 25%, debiendo pagar $975 cada uno.
- Si Pedro y Juan también invitan a Paula y cancelan 3 meses juntos, recibirán una reducción del 35%, debiendo abonar $1267.50 cada uno.
- Si José se anota solo, pero paga 5 cuotas juntas, entonces deberá abonar $2437.5

Ejercicio `10`. Tomando como base los resultados obtenidos en un laboratorio de análisis clínicos, un médico determina si una persona tiene anemia o no, lo cual depende de su nivel de hemoglobina en la sangre y de su edad.

Si el nivel de hemoglobina que tiene una persona es menor que el valor mínimo que le corresponde de acuerdo a su edad, el resultado del análisis es "anemia positivo" y en caso contrario es "anemia negativo".

El médico se basa en los siguientes valores mínimos para cada grupo de edades:

- edad <= 1 mes: nivel mínimo de hemoglobina normal 13 g/dl
- 1 mes < edad <= 6 meses: nivel mínimo de hemoglobina normal 10 g/dl
- 6 meses < edad <= 12 meses: nivel mínimo de hemoglobina normal 11 g/dl
- 1 año < edad <= 5 años: nivel mínimo de hemoglobina normal 11.5 g/dl
- 5 años < edad <= 10 años: nivel mínimo de hemoglobina normal 12.6 g/dl
- 10 años < edad: nivel mínimo de hemoglobina normal 13 g/dl

Diseñe una función `anemia` que recibiendo la edad de una persona expresada en meses y la hemoglobina en sangre expresada en g/dl devuelva `#true` si la persona está anémica, `#false` en caso contrario.

Ejercicio `11`. Decimos que una terna de números *a,b,c* es *autopromediable* si uno de sus valores concide con el promedio de los otros dos.

Por ejemplo, la terma *(7,5,9)* es autopromediable puesto que 7 es el promedio entre 5 y 9.

Diseñe una función que dados tres números, devuelva el producto de ellos en caso que formen una terna autopromediable, y la suma de los mismos en caso contrario.

Defina todas las constantes y funciones auxiliares que crea convenientes para obtener un buen diseño.

Ejercicio `12`. El consumo promedio de una Chevrolet Zafira modelo 2010 es de 8km/l en ciudad y 11km/l en ruta. Es decir, ese modelo de auto utiliza un litro de nafta para recorrer 8km en ciudad, pero en ruta la misma cantidad de combustible alcanza para recorrer 11km.

Estos valores fueron calculados utilizando nafta grado 2 (conocida como "súper"). Al cargar combustible con nafta grado 3 (conocida como "premium"), el rendimiento mejora un 10%. Por lo tanto, con cada litro de combustible grado 3 se puede recorrer un 10% más de distancia de la especificada en el párrafo anterior.

Diseñe una función `autonomía`, que dados los siguientes argumentos:

- La cantidad de litros restantes en el tanque de combustible, y
- La clase de combustible que se está utilizando,

devuelva un string indicando la autonomía del auto, tanto en ciudad como en ruta.

Por ejemplo, si quedan 20 litros de nafta grado 2 en el tanque de combustible, se espera que el string que devuelva la función `autonomía` sea

```
"Autonomía en ciudad: 160km. Autonomía en ruta: 220km."
```

En cambio, si quedan 20 litros de nafta grado 3, se espera que el string sea:

```
"Autonomía en ciudad: 176km. Autonomía en ruta: 242km."
```

Utilice constantes y todas las funciones auxiliares que crea conveniente para lograr un buen diseño.