Programación I

Unidad I: La receta

Para diseñar programas, es fundamental desarrollar varias habilidades:

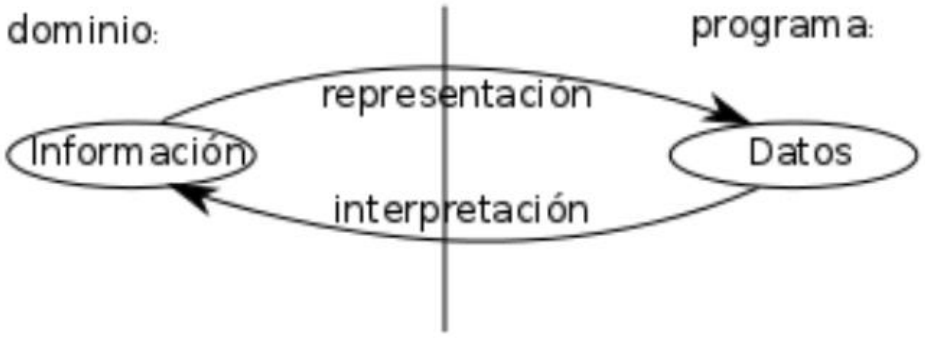
- Analizar el enunciado de un problema,
- extraer y poder expresar su esencia, de manera abstracta y con ejemplos concretos,
- esquematizar, planear y desarrollar una solución basándose en los puntos previos,
- evaluar los resultados en comparación con los esperados,
- realizar revisiones cuando los resultados no son los esperados.

Para programar necesitamos un **lenguaje,** del cual debemos conocer su vocabulario, su gramática (o sintaxis), y el significado de sus frases (o semántica). Además del lenguaje, necesitamos la habilidad para transformar el enunciado de un problema en un programa. Necesitamos, para el mismo, preguntarnos:

- ¿Qué partes del enunciado son relevantes y cuáles no?
- ¿Qué información recibe el programa? ¿Qué produce? ¿Cómo se relacionan la entrada y la salida?
- ¿Qué herramientas me provee el lenguaje y sus bibliotecas para resolver el problema?
- Ya tengo el programa, ¿resuelve realmente el problema? ¿Cómo encuentro errores?, y ¿Cómo los elimino?

Pasos para diseñar programas

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 que produce un problema, deben poder *interpretarse como información.*



2) *Signatura y declaración de propósito:* La signatura de una función indica qué datos consume (cuántos y de qué clase), y qué datos produce. La declaración de propósito es un breve comentario que

describe el comportamiento de la función. Sirve para que cualquiera sepa qué hace cada función sin necesidad de leer su código.

3) *Ejemplos:* Ayuda a deducir una regla general para el problema y permite validar la solución general una vez definida.

4) *Definición de la función:* Template de la función, parte creativa. Se debería poder definir el código, siendo ayudados tal vez por los ejemplos.

5) *Validación en los ejemplos:* Evaluar la expresión en los ejemplos para verificar que, al menos en esos casos, la respuesta es la esperada.

6) *Realizar modificaciones en caso de error.* Buscar la fuente del error, el resultado esperado difiere del que obtengo pueden darse tres situaciones: los ejemplos fueron mal calculados, la función calcula de forma incorrecta el resultado u ocurren ambas.

Proposiciones y Condicionales

Una *proposición* es una expresión sobre la cual se puede afirmar que es verdadera o falsa.

Leyes de reducción

Explicitando sus leyes de reducción, indican cómo se evalúa una expresión. En el caso de las expresiones if, tenemos dos leyes de reducción,

- (if #true a b)

== definición de if (ley 1)

- a
- (if #false a b)

== definición de if (ley 2)

- b

Condicionales múltiples

Soluciona la desventaja de las expresiones *if* que sólo es posible elegir entre dos posibilidades. Entonces, la expresión condicional está formada por una lista de parejas. La primera parte de cada pareja (Condición-X) es una expresión que debe evaluar a un valor booleano. La segunda parte de la pareja se conoce como el resultado de la condición correspondiente, y puede evaluar a cualquier tipo de dato.

DrRacket evaluando condicionales múltiples. Primero se evalúa la primer condición (Condición-1). Si esta reduce a #true, entonces el resultado de toda la expresión es el que se obtiene de evaluar Resultado-1. Caso contrario, se elimina la primer pareja de la expresión condicional, y DrRacket procede con la segunda pareja del mismo modo que con la primera. Si todas las condiciones evalúan a #false, entonces se produce un error.

Unidad II: Programas Interactivos

Un *programa* es un conjunto de definiciones constantes, funciones y expresiones que calculan valores. Existen, sin embargo, dos grandes categorías: los *programas por lotes* (ej. compresor de archivos), que una vez lanzado el proceso no necesita ningún tipo de interacción con el usuario; y los *programas interactivos* (ej. cajero automático), que una vez iniciados esperan la intervención del usuario para llevar a cabo sus funciones.

Eventos y manejadores de eventos

Los programas reciben información del entorno a través de *eventos*, es decir, la ocurrencia de alguna situación para la que el programa está preparado y debe tomar una determinada acción.

La idea fundamental de un programa interactivo es que ante cierto *evento* el programa llevará a cabo cierta acción, que queda establecida a partir de la definición de una función llamada *manejador de eventos*.

Diseñar un sistema reactivo requiere diseñar *manejadores de eventos* para los que necesita estar preparados y una *función principal* que comunique estas asociaciones al sistema operativo.

Manejadores de eventos -> son funciones que toman como argumento el estado actual del sistema y una descripción del evento ocurrido, y devuelven siempre el nuevo estado. ⟶

Stop-when -> sirve para que el programa decida dejar de manejar eventos y termine su ejecución. Es una función cuyo tipo es: `Estado -> Bool`⟶

Estado

Durante la ejecución de un programa, ante ciertos *eventos* los correspondientes *manejadores de eventos* llevan a cabo acciones que *cambian propiedades o valores* dentro del programa. Entonces, al iniciar, estará en un estado particular, el cuál cambiará ante la aparición de un evento.

1. "¿Qué cambia?" es la pregunta cuando se necesita determinar cómo será el estado del programa.

Big-bang

Se define con anterioridad cómo representa e interpreta el estado del programa, y elegir un valor inicial para el mismo. Recibe el estado inicial y una lista de parejas. Sólo una es obligatoria (to-draw …) que interpreta el estado actual.

El comportamiento de la expresión es el que sigue:

1. La función asociada a to-draw es invocada con el estado inicial como argumento, y su resultado se muestra por pantalla.
2. El programa queda a la espera de un evento.
3. Cuando un evento ocurre, el manejador asociado a dicho evento (si existe) es invocado, y devuelve el nuevo estado.
4. En caso de estar presente, se aplica el predicado asociado a la cláusula stop-when al nuevo estado. Si devuelve #true, el programa termina, si no,
5. la función asociada a to-draw es nuevamente invocada con el nuevo estado, devolviendo la nueva imagen o escena.
6. El programa queda a la espera de un nuevo evento (volvemos al paso 2).

Unidad III: Estructuras

```
(define-struct Nombre [Campo1 … CampoN])
```

Define-struct indica la definición de nuevo tipo de datos, indicado por el nombre (*Nombre*) y finalmente una lista con los nombres de los campos que incluye la estructura. Incorpora funciones nuevas:

- *Constructor:* permite crear elementos en el nuevo tipo. (`make-Nombre`)
- *Selector:* uno por cada campo, permiten observar el valor de cada uno. (`Nombre-CampoN`)
- *Predicado:* distingue instancias de la clase creada de otros objetos. (`Nombre?`)

Las leyes de evaluación para estructuras relacionan el constructor con los selectores.

Unidad IV: Listas

`‘(), empty` -> Expresiones de lista vacía

Cuando se agrega un elemento a una lista, se está construyendo otra lista -> `(cons "Jorge" (cons "María" (cons "Juan" '())))`

Definición autoreferenciada:

```
; List(X) es:
; - ‘()
; - (cons X List(X))
; interpretación: List(X) representa una lista con elementos del
; tipo X.
```

Operaciones sobre listas:

Operador	*Tipo de Operador*	*Función*
'()	Constructor	Representa la lista vacía.
empty?	Predicado	Reconoce únicamente la lista vacía.
cons	Constructor	Agrega un elemento a una lista.
first	Selector	Devuelve el primer elemento.
rest	Selector	Devuelve lista sin su primer elemento.
cons?	Predicado	Reconoce listas no vacías.

Aplicando una transformación a los elementos de la lista

```
; map : (X -> Y) List(X) -> List(Y)

; (cons (f (first l)) (map f (rest l)))
```

Dada una función que transforma objetos de X en objetos de Y, y una lista con objetos en X, devuelve una lista con objetos en Y.

- [a0, a1, …., an] -> [f(a0), f(a1), …, f(an)]

Clasificando los elementos de una lista

```
; filter : (X -> Boolean) List(X) -> List(X)

; (if (p (first l)) (cons (first l) (filter p (rest l))) (filter p (rest l)))
```

Dado un predicado p y una lista l con objetos en X, devuelve una lista con aquellos objetos de l para los cuales p evalúa en #*true*

Operando los elementos de una lista

```
; (f a0 (f a1 (… (f an c))))

; (f (first l) (fold f c (rest l)))
```

La función *fold* recibe tres argumentos:

- La función f con la que se quiere operar los elementos de la lista,
- Un valor c, que es el resultado esperado para la lista vacía,
- La lista l a transformar.

Unidad V: Números Naturales

```
; Un Natural es:

; - 0

; - (add1 Natural)

; interpretación: Natural representa los números naturales.
```

El 0 es un número natural. Si n es un número natural, entonces n + 1 también lo es.

Operador	***Tipo de Operador***	***Función***
`0`	Constructor	Constante usada para representar el primer número natural.
`add1`	Constructor	Calcula el sucesor de un número natural.
`sub1`	Selector	Devuelve el predecesor de un número natural positivo.
`zero?`	Predicado	Reconoce al natural 0.
`positive?`	Predicado	Reconoce naturales construidos con add1.