

**Programación imperativa:** define un control de flujo; describe programas con instrucciones, condiciones y pasos que modifican el estado de un programa al permitir la mutación de variables, para llegar a un resultado. En este tipo de programas se coloca un valor en una variable, se lo muta con un algoritmo definido con un orden de ejecución específico (el orden suele afectar al resultado) y escribe el valor en una salida.

**Programación declarativa:** define la lógica; proposiciones y afirmaciones sin especificar los pasos para resolver el problema. El estado no puede ser modificado ya que todos los tipos de datos son inmutables. Este tipo de programas se basan en evaluar funciones matemáticas puras. Una **función** es **pura** cuando su único valor accesible es su retorno (resultado), y éste depende solamente de las entradas, es decir, no hay “**side effects**” (efectos colaterales; dependencia al estado global como puede ser el cambio de variables globales por otras partes del código), como en la programación imperativa. Otra característica de estos es la **transparencia referencial**, que ocurre cuando "en un programa, una expresión E del lenguaje puede ser sustituida por otra de igual valor V, resultando en un programa cuya semántica no va a diferir de la del original". Básicamente, quiere decir que el resultado de evaluar una expresión compuesta depende únicamente del resultado de evaluar las subexpresiones que la componen y de nada más; no depende de la historia del programa en ejecución ni del orden de evaluación de las subexpresiones que la componen. Desanima la mutación de variables y el uso de loops como for y while, prefiriendo la recursión.

**Diferencia sobre “variables” en prog. Imperativa y prog. Declarativa:** en lenguajes imperativos, una variable es una forma de identificar una ubicación en memoria que puede contener valores distintos durante diferentes momentos del programa. Por ende, examinar esa ubicación potencialmente puede devolver resultados diferentes cada vez. En la programación declarativa, una variable le da un nombre a una expresión. Una vez que se asocia esa variable con una expresión en particular, su valor no cambia. Se puede escribir el nombre de la variable en cualquier momento, y siempre se obtiene el mismo resultado.

**Sobre Haskell**

Hay 3 características importantes sobre el sistema de tipos de Haskell:

* **Tipado fuerte**: garantiza de que el programa va a estar libre de cierta clase de errores, como, por ejemplo, una función que espera recibir un int, pero al momento de probar el programa recibe una string. Estos errores no ocurren en Haskell porque el casteo debe ser explícito.
* **Tipos estáticos**: el compilador conoce el tipo de cada valor y expresión en tiempo de compilación (antes de que sea ejecutado). Un compilador, o intérprete de Haskell va a detectar cuando intentemos usar expresiones cuyos valores no coincidan, lo rechazará y emitirá un mensaje de error. Por ejemplo, ghci dará un mensaje de error si se trata de ejecutar `True && “hola”` ya que la función `&&` tiene que recibir dos expresiones/valores de tipo Bool.
* **Inferencia de tipo**: un compilador de Haskell puede deducir automáticamente casi todos los tipos de las expresiones en un programa. Permite explícitamente declarar el tipo de un valor, pero la inferencia de tipo hace que esto sea, casi siempre, opcional.

**Tipo de dato compuesto**: está construido usando otros tipos de datos. Los más comunes son **listas** y **tuplas**. Una lista debe tener todos valores del mismo tipo internamente y puede tener cualquier longitud, en cambio, una tupla tiene una longitud fija, pero puede tener distintos tipos de valores. Tupla de un elemento = Unidad, dos elementos = par, etc. Ejemplo: lista – [1, 2, 3] o [a, b, c], pero no [1, a, b]. tupla – (a.b), (1, True, “hola”), etc.

**Evaluación vaga**: En un lenguaje que usa evaluación estricta, el argumento de la función es evaluado antes que la función sea aplicada. Haskell adopta la evaluación no estricta, es decir que no evalúa una expresión hasta que no la necesite. Por ende, si al final no se necesita usar el resultado de la expresión, esta no fue evaluada, ahorrando cálculos.

Esto es ventajoso, por ejemplo, a la hora de utilizar la función or (||). En un lenguaje con evaluación estricta, se le suele dar un trato especial a este operador, pero haskell lo resuelve con la evaluación vaga. Esto es porque en Haskell se evalúa primero la condición de la izquierda (las funciones en Haskell son asociativas de izquierda a derecha), y en caso de que ésta devuelva True, no analiza la expresión de la derecha.

Otro ejemplo: [1..] es una expresión con un resultado infinito, que no se podría evaluar, ya que quería atrapado un loop. Pero la evaluación vaga permite hacer operaciones como `True || length[1..] > 0`.

Definir nuevos tipos de datos:

`data BookInfo = Book Int String [String]

deriving (Show)`

type constructor - BookInfo - constructor de tipo

value constructor - Book, se usa para crear un valor del tipo BookInfo. Constructor de valor

`type BookID = Int

type BookName= String

type BookPair = (BookID, BookName)

data BookInfo = BookInfo BookID BookName [String]

deriving (Show)`

type synonym - BookID, BookName, BookPair. C usa acceso mediante nombre (estruct.nombredato), haskell no lo tiene, se maneja de forma ordinal (según posición). Sirve para hacer el código más descriptivo, similar al typedef

Los TIPOS DE DATOS ALGEBRAICOS pueden tener más de un value constructor y permiten distinguir 2 cosas estructuralmente iguales, serian parecido a un tipo de clase abstracta en c++

a = (1,2)

b = (3,5)

son tuplas del mismo tipo (Int, Int)

`data num1 = num1 Int Int

data num2 = num2 Int Int

c = num1 1 2

d = num2 3 5`

c y d son tipos distintos ya que tienen distinto nombre, no importa que tengan la misma estructura. Esto serviría tmb, por ej, para distinguir si usamos coord polares o cartesianas.

data Bool = True | False

Es un tipo de dato algebraico, porque tiene 2 constructores de valor distintos (true y false)

Matcheo de patrones

Permite saber que constructor de valor se usó para crear una instancia de un tipo de dato algebraico (pq tienen mas de un constructor) y extraer los valores de los campos del constructor. Esto lo hace usando la deconstrucción.

Por ejemplo:

`myNot True = False

myNot False = True`

La función myNot está definida como 2 ecuaciones (ambas lineas son parte de la función). Se evaluará primero la condición de arriba, y después las de abajo sucesivamente.

Lo que está antes del = es el nombre de la función, seguido de un constructor de valor (True es uno de los constructores del tipo Bool). La función lo que hace es tomar el dato que ingreso de entrada, y lo compara con esta parte, para ver si fue construida con ese constructor (matcheo de patrones). Si la entrada fue construida con ese constructor, hubo un matcheo, y la función devuelve el resultado de la expresión a la izquierda del =.

Por ejemplo, al hacer:

`let a = False

myNot a`

Primero la función myNot toma el valor de a (False) y lo compara con la primera ecuación de la función:

`myNot True`

El patrón no matchea, porque el valor de a fue construido con False, entonces pasa a la ecuación de abajo.

`myNot False = True`

El patrón matchea; a fue construido con el constructor False. Entonces, la función devuelve lo que está a la derecha del =, True. Es decir, finalmente myNot a devuelve como resultado True.

También se puede usar el comodín ( \_ ), queriendo decir que no interesa un valor en específico.