Programación funcional con Haskell T5

Paradigma Funcional

Es uno de los paradigmas que tiene la *programación declarativa*. La cual describe un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.

Está basado en el modelo matemático de *composición funcional*. El resultado de un cálculo es la entrada del siguente, y así sucesivamente hasta que una composición produce el valor deseado. En programación funcional (como en matemática) las funciones son elementos
(valores).

Un programa en un lenguage funcional es un conjunto de ecuaciones orientadas que definen una o más funciones.

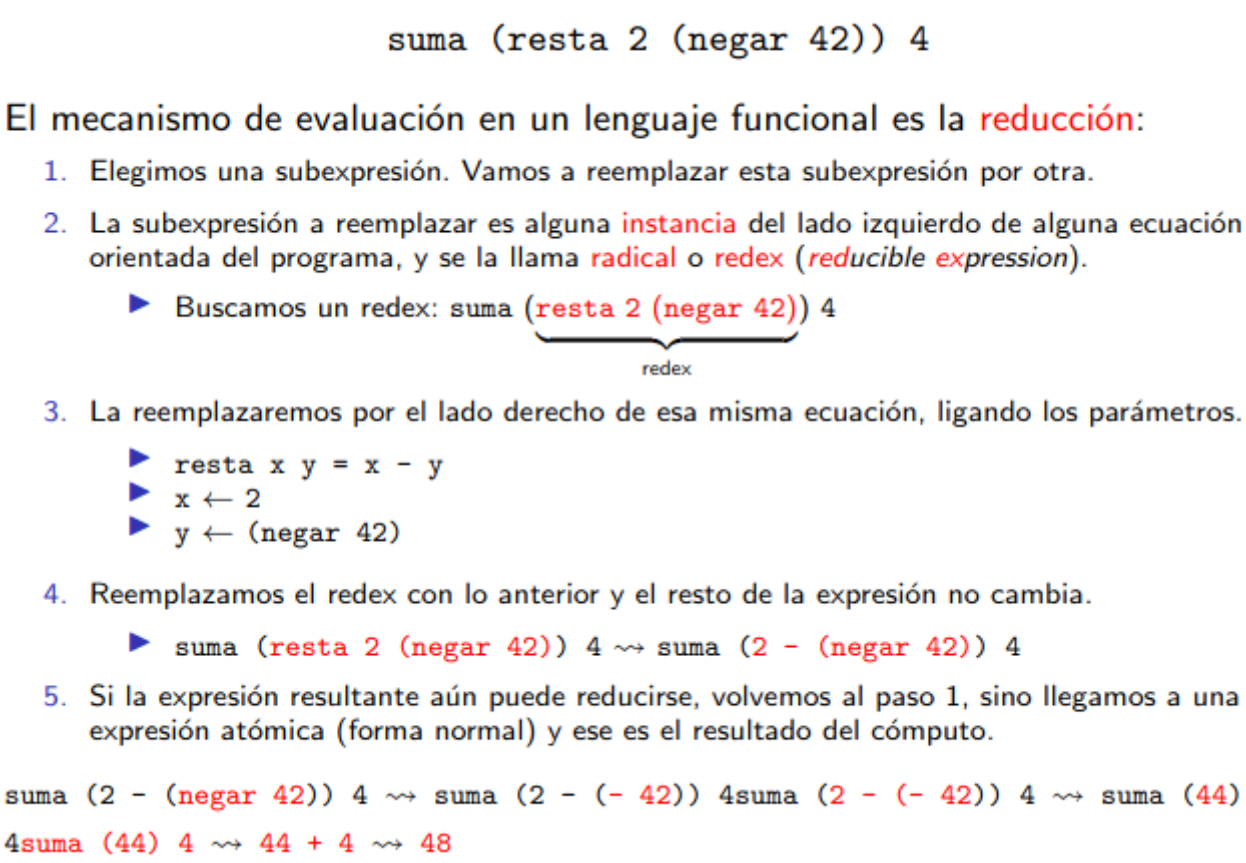
La ejecución de un programa en este caso corresponde a la *evaluación de una expresión*, habitualmente solicitada desde la consola del entorno de programación. La expresión se evalúa usando las ecuaciones definidas en el programa, hasta llegar a un resultado.
Las ecuaciones orientadas junto con el mecanismo de reducción describen algoritmos.

- **Ecuaciones orientadas**
  Lado izquierdo: expresión a definir.
  Lado derecho: definición.
  Cálculo del valor de una expresión: reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho.
- **Transparencia referencial**
  Es la propiedad de un lenguaje que garantiza que el valor de una expresión depende exclusivamente de sus subexpresiones. Cada expresión del lenguaje representa siempre el mismo valor en cualquier lugar de un programa.
- **Formación de expresiones**
  Expresiones atómicas (formas normales): denotan un valor y no se puede reducir más. Ej: 2, False.
  Expresiones compuestas: se construyen combinando expresiones atómicas con operaciones.

Para saber si una expresión está bien formada, aplicamos:
-Reglas sintácticas.
-Reglas de asignación o inferencia de tipos (algoritmo de
Hindley-Milner)

En Haskell toda expresión denota un valor, y ese valor pertenece a un tipo de datos y no se puede usar como si fuera de otro tipo distinto.

- **Evaluación y reducción**



Haskell tiene un orden de evaluación normal o **lazy (perezoso)**: se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que **primero se evalúa la función y después los argumentos (si se necesitan)**.

Otros lenguajes de programación (C, C++, Pascal, Java) tienen un orden de evaluación eager (ansioso): primero se evalúan los argumentos y después la función.

Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** (⊥).

- **Clasificación de funciones**
  Funciones totales: nunca se indefinen.
  Funciones parciales: hay argumentos para los cuales se indefinen.
- **Definición de funciones por casos**

GUARDAS
Las guardas son una forma de comprobar si alguna propiedad de un valor (o varios de ellos) es cierta o falsa.
Las guardas se indican con barras verticales que siguen al nombre de la función y sus parámetros. Normalmente tienen una sangría y están alineadas. Una guarda es básicamente una expresión booleana. Si se evalúa a True, entonces el cuerpo de la función correspondiente es utilizado. Si se evalúa a False, se comprueba la siguiente guarda y así sucesivamente.

Muchas veces la última guarda es otherwise. otherwise está definido simplemente como otherwise = True y acepta todo.

-Podemos usar guardas para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1
    | n /= 0 = 0
```

Palabra clave "si no".

```
f n | n == 0 = 1
    | otherwise = 0
```

Prestar atención al orden de las guardas. ¡Cuando las condiciones se solapan, el orden de las guardas cambia el comportamiento de la función!

```
f4 n | n >= 3 = 5
     | n <= 9 = 7
```

```
f5 n | n <= 9 = 7
     | n >= 3 = 5
```

PATTERN MATCHING (ajuste de patrones)

Un ajuste de patrones consiste en una especificación de pautas que deben ser seguidas por los datos, los cuales pueden ser deconstruidos permitiéndonos acceder a sus componentes. Podemos usar el ajuste de patrones con cualquier tipo de dato: números, caracteres, listas, tuplas, etc.

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "¡Uno!"
sayMe 2 = "¡Dos!"
sayMe 3 = "¡Tres!"
sayMe 4 = "¡Cuatro!"
sayMe 5 = "¡Cinco!"
sayMe x = "No entre uno 1 y 5"
```

Tener en cuenta que si movemos el último patrón (el más general) al inicio, siempre obtendríamos "No entre uno 1 y 5" como respuesta, ya que el primer patrón encajaría con cualquier número y no habría posibilidad de que se comprobaran los demás patrones.

Cuando utilizamos patrones siempre tenemos que incluir uno general para asegurarnos que nuestro programa no fallará.

WHERE

Las variables que definamos en la sección where de una función son solo visibles desde esa función, así que no nos tenemos que preocupar de ellas a la hora de crear más variables en otras funciones. Si no alineamos la sección where bien y de forma correcta, Haskell se confundirá porque no sabrá a que grupo pertenece.

Las secciones where también pueden estar anidadas. Es muy común crear una función y definir algunas funciones auxiliares en la sección where y luego definir otras funciones auxiliares dentro de cada uno de ellas.

```
cantidadDeSoluciones a b c | discriminante > 0 = 2
                           | discriminante == 0 = 1
                           | otherwise = 0
                          where discriminante = b^2 - 4*a*c
```

- **TIPOS DE DATOS**

  Un conjunto de valores a los que se les puede aplicar un conjunto de funciones. Podemos declarar explícitamente el tipo de datos del dominio y codominio de las funciones. A esto lo llamamos dar la **signatura** de la función. No es estrictamente necesario hacerlo (Haskell puede inferir el tipo), pero suele ser una buena práctica.

Ejemplos:

1. $\texttt{Int} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}, \{+, -, *, /\})$ es el tipo de datos que representa a los racionales, con la aritmética de punto flotante.
3. $\texttt{Char} = (\{\text{'a'}, \text{'A'}, \text{'1'}, \text{'?'}\}, \{\texttt{ord}, \texttt{chr}, \texttt{isUpper}, \texttt{toUpper}\})$ es el tipo de datos que representan los caracteres.
4. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\&\&, ||, \texttt{not}\})$ representa a los valores lógicos.

- **Funciones**

Una función es un valor. La operación básica que podemos hacer con ese valor es la *aplicación* (aplicar la función a un elemento para obtener un resultado). Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara _ _ True = True
funcionRara x y False = x >= y
```

La **variable anónima** _ (guión bajo) funciona, en términos de matcheo, como una variable común. Esta opción es mejor, porque ayuda a la expresividad: si la variable no se usa en la devolución (a la derecha del igual, ó a la derecha de la guarda) entonces mejor ni ponerle nombre.

Ej:

```
anioActual = 2014

edad (nombre, anioNac, cantPerros) = anioActual - anioNac

edad ( _, anioNac, _) = anioActual - anioNac
```

Errores comunes con la variable anónima
Muchas veces pensamos que no necesitamos una variable, cuando en realidad sí la usamos a la derecha del igual. Por ejemplo, si queremos tomar esa definición de “persona” como una tupla de tres, y agregarle perros a la persona, debemos devolver a una persona, pero cambiando sólo la cantidad de perros.

Error:
`agregarPerros cant (_, _, cantPerros) = (_, _, cantPerros + cant)`

Si quiero usarlas a la derecha del igual (ó sea, si quiero que sean parte de lo que devuelvo) entonces tengo que usar variables comunes:

`agregarPerros cant (nombre, anioNac, cantPerros) = (nombre, anioNac, cantPerros + 1)`

- **Polimorfismo**

Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla). Se usa cuando el comportamiento de la función *no depende paramétricamente* del tipo de sus argumentos. EJ: la función identidad. En Haskell los polimorfismos se escriben usando *variables de tipo* y conviven con el tipado fuerte.

Variables de tipo
Son parámetros que se escriben en la signatura usando variables minúsculas. En lugar de valores, denotan TIPOS. Cuando se invoca la funci´on se usa como argumento el tipo del valor. Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

Clases de tipos
Son conjuntos de tipos a los que se le pueden aplicar un conjunto de funciones. Un tipo puede pertenecer a distintas clases. EJ: Los Float son números (Num), con orden (Ord), de punto flotante (Floating), etc.

Algunas clases:

1. `Integral := ({ Int, Integer, ... }, { mod, div, ... })`
2. `Fractional := ({ Float, Double, ... }, { (/), ... })`
3. `Floating := ({ Float, Double, ... }, { sqrt, sin, cos, tan, ... })`
4. `Num := ({ Int, Integer, Float, Double, ... }, { (+), (*), abs, ... })`
5. `Ord := ({Bool, Int, Integer, Float, Double, ... }, { (<=), compare })`
6. `Eq := ({ Bool, Int, Integer, Float, Double, ... }, { (==), (/=) })`

Las clases de tipos se describen como restricciones sobre variables de tipos:
`(Floating t, Eq t, Num u, Eq u) => ...` significa que:

-La variable t tiene que ser de un tipo que pertenezca a Floating y Eq
-La variable u tiene que ser de un tipo que pertenezca a Num y Eq

Tuplas de tipos

Tuplas

- Dados tipos $A_1, \ldots, A_k$, el tipo *k*-upla $(A_1, \ldots, A_k)$ es el conjunto de las *k*-uplas $(v_1, \ldots, v_k)$ donde v_i es de tipo A_i

```
(1, 2)            :: (Int, Int)
(1.1, 3.2, 5.0)   :: (Float, Float, Float)
(True, (1, 2)) :: (Bool, (Int, Int))
(True, 1, 2)    :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

Funciones de acceso a los valores de un par en `Prelude`

- `fst :: (a, b) -> a` Ejemplo: `fst (1 + 4, 2)` $\rightsquigarrow$ 5
- `snd :: (a, b) -> b` Ejemplo: `snd (1, (2, 3))` $\rightsquigarrow$ `(2, 3)`

Ejemplo: suma de vectores en $\mathbb{R}^2$

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```

Podemos usar *pattern matching* para acceder a los valores de una tupla

```
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)
```

- **Currificación**
  Para declarar una función separamos los tipos de los parámetros con una flecha. La notación se llama currificación, evita el uso de varios signos de
  puntuación (comas y paréntesis).

El poner un espacio entre dos cosas es sencillamente aplicar una función. El espacio es una especie de operador y tiene el orden de preferencia mayor.

```
promedio1 (promedio1 (2, 3), promedio1 (1, 2))
promedio2 (promedio2 2 3) (promedio2 1 2)
```

- **Funciones binarias**

Notación prefija: función antes de los argumentos. EJ: `suma x y`
Notación infija: función entre argumentos. EJ: `x + y` .

La notación infija se permite para funciones cuyos nombres son operadores. El nombre real de una función definido por un operador • es (•). Se puede usar el nombre real con notación prefija. EJ: `(+) 2 3`.

Una función binaria f puede ser usada de forma infija escribiendo ‘f‘.

```
(>=) : : Ord a => a -> a -> Bool
(>=) 5 3 --evalua a True
(==) : : Eq a => a -> a -> Bool
(==) 3 4 --evalua a False
(^) : : (Num a , Int b) => a -> b -> a
(^) 2 5 --evalua 32.0
mod : : ( Integral a) => a -> a -> a
5 ‘mod‘ 3 --evalua 2
div : : ( Integral a) => a -> a -> a
5 ‘ div ‘ 3 --evalua 1
```