Tema II:
Introducción al Lenguaje Funcional

Características de Haskell

- Lenguaje funcional de o. superior fuertemente tipificado.
- Clasifica los entes de un programa en:
  - **objetos** (constantes y funciones) y
  - **tipos**: cada objeto debe tener un tipo.
- Dispone de objetos y tipos predefinidos.
- Permite diversas declaraciones de objetos (monomorfos y polimorfos):
  - **funciones constructoras** y
  - **funciones definidas.**
- Permite diversas declaraciones de tipos (monomorfos y polimorfos):
  - **tipos sinónimos** y
  - **tipos algebraicos**
- Aplica:
  - un sistema de inferencia de tipos basado en el sistema de **Hindley-Milner**,
  - una estrategia de **reducción perezosa**.

Principios de Haskell

- **Completitud sintáctica**

  Si en una expresión sintácticamente correcta se cambia una subexpresión por otra expresión correcta del mismo nivel (y tipo) la expresión resultante es igualmente correcta.

- **Completitud semántica**

  A todos los entes de cada nivel se les puede dar el mismo tratamiento:
  - pueden recibir nombre, y
  - pueden figurar en expresiones.

- Estos principios dan lugar a una sintaxis sin "casos especiales" de la que se deriva la posibilidad de utilizar:
  - **funciones de orden superior** y
  - **funciones currificadas**

Constructores de tipo predefinidos

- **Tipos simples** (ctes de tipo)

```
Bool  Int Integer  Float Double  Char
```

- **Tipos estructurados** (funciones de tipo)

```
[Tipo] (Tipo,Tipo{,Tipo}*)  Tipo->Tipo
```

Permiten construir:

- **Tipos monomorfos**

```
Int  [Int]  ([Char],Bool,Int)
[Char]->Int  [Char->Float]
```

- **Tipos polimorfos**

```
[a]  (a,[b])  a->Int
```

(Los nombres de tipo comienzan con mayúscula y los nombres de objetos comienzan con minúscula)

Definición de sinónimos de tipo

- Los sinónimos son tipos que se introducen para dar nombre a otras expresiones de tipo

```
type Tipo{parT} = expresión
```

```
type Entero = Integer
type Cadena = [Char]
type Pila a = [a]
type ListaArcos a b = [(a,b)]
```

Definición de tipos algebraicos

- Los tipos algebraicos son tipos:
  - cuyos valores se generan libremente a partir de unas funciones constructoras como suma directa

```
data Tipo{parT} = constr{|constr}*
  constr ::= funCon{ Tipo}*
```

  - Permiten definiciones de tipos por **enumeración**, **unión disjunta** y por **recursión**

```
data Dia = Lu|Ma|Mi|Ju|Vi|Sa|Do
data Temp = Cent Float | Fahr Float
data Punto a = Pt a a
data Arbol a = Hoja a | Nodo (Arbol a)(Arbol a)
```

Funciones constructoras y patrones

- En las declaraciones de tipos algebraicos se declaran también **funciones constructoras**:

```
Lu :: -> Dia, ...
Cent :: Float -> Temp, ...
Pt :: a -> a -> Punto a
Hoja :: a -> Arbol a,
Nodo :: Arbol a -> Arbol a -> Arbol a
```

- Las funciones constructoras comienzan por mayúscula, son irreducibles y sirven para representar/etiquetar los valores de los tipos que definen.
- Se utilizan como **patrones** para los análisis de casos en las definiciones de funciones.

Objetos predefinidos

(constantes y funciones)

- **Objetos simples:**
  - `True, False :: Bool`
    `(&&), (||) :: Bool -> Bool -> Bool`
    `not :: Bool -> Bool`
  - `..., -1, 0, 1, ... :: Int;`
    `(+),(-),(*),(/),(^) :: Int -> Int -> Int;`
    `div,mod,gcd,lcm :: Int -> Int -> Int;`
    `negate, abs, signum ::Int -> Int;`
    `odd, even :: Int -> Bool;`
  - `..., -1.65,...,0.0,...,1.65,... :: Float;`
    `(+),(-),(*),(/) :: Float -> Float -> Float;`
    `(e) :: Float -> Int -> Float;`
  - `'a',...,'0',...,'Z',... :: Char;`
    `ord :: Char -> Int; chr :: Int -> Char;`

Objetos predefinidos
(constantes y funciones)

- **Objetos compuestos:**
  - **Listas**

    ```
    [], [objeto{,objeto}*], objeto{:objeto}*:[]
    ```

  - **Cadenas de caracteres**

    ```
    "car{car}*"
    ```

  - **Tuplas**

    ```
    (),   (objeto,objeto{,objeto}*)
    ```

  - **Función para mensajes de error (polimorfa)**

    ```
    error :: String -> a
    ```

Definición de objetos (I)

Los objetos que no son funciones constructoras se definen mediante ecuaciones o declaraciones de ligadura:

nom_fun { patrón}* método [declaración local]

- En cada una de estas declaraciones se establece una ligadura entre un nombre de función y un método de cálculo de dicha función para una forma determinada de sus parámetros.
- Para una misma función se pueden utilizar varias declaraciones.

Definición de objetos (II)

- El **nombre de la función**, nom_fun, debe ser una cadena de caracteres alfabéticos, dígitos, subrayado o apóstrofe, comenzando por una letra minúscula.
- Cada **patrón** tendrá una forma que dependerá del tipo correspondiente al parámetro que represente:

`x , _`	patrones irrefutables (con nombre y sin nombre),
`0, (n+1), (n+3)`	patrones para números enteros (0, >= 1, >= 3)
`[], (x:xs)`	patrones para listas vacías y no vacías;
`[_, x, _]`	patrón que fija el número de elementos de una lista;
`(x,y)`	patrones para pares (tuplas, en general)

  - Las funciones constructora se puede utilizar para construir patrones:

```
Lu,  Cent x, Hoja x, Nodo x y
```

  - Los patrones se pueden anidar:

```
([],x), (x:xs, n+1)
```

Definición de objetos (III)

El **método** puede ser:

- Una denotación simple de objeto
- Una aplicación de función
- Una secuencia de expresiones condicionadas
- Una casuística basada en la estructura de un objeto
- Una alternativa simple
- Una lambda abstracción o expresión lambda

Método (I)

- **Denotación simple de objeto**:

  = valor

```
pi = 3.1415927
id x = x
```

- **Aplicación de una función**:

  = fun arg1 ... argk

```
areaR x y = x * y ,

areaRbase3  = areaR 3
```

Método (II)

- **Expresiones condicionadas**:

  | expresión lógica = expresión
  {; | expresión lógica = expresión} *

```
max x y |x >= y = x
        |x < y  = y

min x y |x > y = y
        | True = x
```

Se utilizan cuando el método de cálculo depende de alguna condición (expresión lógica).

Las condiciones se evalúan siguiendo el orden textual y se calcula la expresión correspondiente a la primera condición válida.

Método (III)

- **Casuísticas**:

  = case objeto of { alternativa
  {; alternativa} *}

  Se utilizan cuando el método de cálculo depende de la estructura de algún objeto relacionado con los parámetros.

  Las alternativas son de alguna de las formas:

  patrón -> objeto

  patrón | objeto lógico -> objeto
  {; | objeto lógico -> objeto}*

```
longitud xs = case xs of
                  []    -> 0
                  y:ys -> 1+ longitud ys
```

Método (IV)

- **Alternativa simple**:

  = **if** exp. lógica **then** expresión1 **else** expresión2

  Se utiliza cuando hay dos formas para calcular una función dependiendo del valor de una expresión lógica.

  Tiene el comportamiento habitual

```
par n = if (n mod 2)==0
          then True
          else False
```

Método (V)

- **Expresión lambda**

  Es una denotación anónima de función de la forma:

  \ parámetro { parámetro}* -> expresión

  Estas expresiones pueden ir en cualquier sitio donde se requiera una función.

```
cuadrado  = \x -> x*x
aplicar =
    \f xs -> case xs of
                  [] -> []
                y:ys -> f y : aplicar f ys
```

Definición de objetos (IV)

- La **declaración local** se utiliza para definir objetos (valores y funciones) de uso privado dentro del método.

  where {definición de obj {**;** definición de obj}*}

  Sirve para definir funciones auxiliares:

```
inversa xs = inv_ac xs [] where
                  inv_ac [] ys = ys
                  inv_ac (x:xs) ys = inv_ac xs (x:ys)
```

  y para introducir denotaciones simples para expresiones

```
coc_res x y |x>=0 && y>0 = if x<y then (0,x)
                                  else (1+c,r)
            |otherwise = error "argtos incorrectos"
                       where (c,r) = coc_res (x-y) y
```

Operadores

Son una variante sintáctica de las funciones con dos argumentos; se caracterizan porque se aplican de manera infija: `x op y`

- Existen operadores predefinidos:

  `(&&)` `(||)` `(+)` `(-)` `(*)` `(/)` `(^)` `(:)`

- Se pueden definir operadores en las
  - declaraciones de funciones constructoras de tipos algebraicos:

    ```
    data ArbolInt = H Int | ArbolInt :^: ArbolInt
    ```

  - declaraciones de ligadura de funciones definidas:

    ```
    (.) :: (b -> c) -> (a -> b) -> (a -> c)
    f . g = \x -> f (g x)
    ```

Normas para la declaración de operadores

- Para los nombres de operadores se puede utilizar cualquier combinación de los símbolos siguientes:

  : ! # $ % & * + . / < = > ? @ \ ^ | -

  siempre que no coincidan con símbolos predefinidos.
- Los nombres de operadores que sean funciones constructoras deben comenzar por '**:**'
- Los nombres de operadores que correspondan a funciones definidas deben ir entre paréntesis en su declaración de tipo.

Asociatividad y precedencia (I)

- Las declaraciones de operadores se pueden completar con declaraciones de asociatividad y de precedencia (que deben preceder a la declaración del operador) de la forma:

  asociatividad precedencia nom_op {, nom_op }*

- La **asociatividad** sirve para resolver situaciones de ambigüedad entre dos aplicaciones de un mismo operador, como: `x op y op z`
- La **precedencia** sirve para resolver situaciones de ambigüedad entre aplicaciones sucesivas de distintos operadores, como: `x op y op' z`

Asociatividad y precedencia (II)

- La asociatividad puede tomar los valores siguientes:

  `infixl` - se asocia por la izquierda

  `infixr` - se asocia por la derecha

  `infix` - no se asocia

- La precedencia debe ser un valor del rango 0..9 y marca la "fuerza de atracción" del operador sobre los argumentos.
- Los valores por defecto para cualquier operador son: **`infix`** y 9.
- Las funciones prefijas tienen mayor precedencia que los operadores.

Declaraciones de los operadores predefinidos

- infixr 2 ||
- infixr 3 &&
- infix 4 ==, /=, <, <=, >=, >
- infixr 5 ++, :
- infixl 6 +, -
- infix 7 /, ‘div‘, ‘rem‘, ‘mod‘
- infixl 7 *
- infixr 8 ^

1+2-3 es **(1+2)-3**

x == y || z es **(x==y) || z**

x : xs ++ ys es **x : (xs++ys)**

12 / 6 / 3 es ambigua

3 * 4 + 5 es **(3*4) + 5**

Operadores y funciones prefijas

- Los operadores se pueden utilizar como funciones prefijas escribiéndolos entre paréntesis:

```
x op y  =>  (op) x y
```

- Las funciones prefijas se pueden utilizar como operadores escribiéndolos entre comillas simples abiertas:

```
f x y   =>  x `f` y
```

- Con las funciones prefijas se pueden producir aplicaciones parciales sobre los primeros argumentos:

```
f x y z  => f, f x, f x y
```

- Con los operadores se pueden producir aplicaciones parciales sobre cualquiera de los dos argumentos:

```
x op y => (x op), (op y)
```