Paradigmas de programación

Departamento de Computación, FCEyN, UBA

2do Cuatrimestre 2022

Docentes

- Mariana Milicich (Ay 2)
- Malena Ivnisky (Ay 1)
- Gustavo Lado (Ay 1)
- Daniela Marottoli (JTP)
- Gabriela Steren (JTP)
- Hernán Melgratti (Prof)

Modalidad Mixta

Horarios

- Martes de 17.30 a 21.00
- Jueves de 17 a 19.30
- Martes de 17.00 a 17.30 (o 18.00)

Modalidad

- Prácticas y talleres: Presencial (Aula 1113, Labos a confirmar)
- Teóricas: algunas presenciales (Aula 1113), otras virtuales (dc.aula04)
- Consultas: Martes de 17.00 a 17.30

Herramientas de comunicación

- Discord (Link en la página principal de la materia)
- Anuncios en el campus
- **listas de email (docentes y alumnos)**

Recursos

Bibliografía

- Textos: no hay un texto principal, se utilizan varios, referencias en página web
- Publicaciones relacionadas
- Diapositivas de teóricas y prácticas

Campus

- Información al día del curso, consultar periódicamente y leer al menos una vez todas las secciones

Mailing list

- Preguntas y consultas.

Paradigma

Marco filosófico y teórico de una escuela científica o disciplina en la que se formulan teorías, leyes y generalizaciones y se llevan a cabo experimentos que les dan sustento

Fuente: Merriam-Webster[1]

[1] *A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated*

Lenguajes de Programación

- lenguaje usado para comunicar instrucciones a una computadora
- instrucciones describen cómputos que llevará a cabo la computadora
- computacionalmente completo si puede expresar todas las funciones computables

Paradigmas de Lenguaje de Programación

Marco filosófico y teórico en el que se formulan soluciones a problemas de naturaleza algorítmica

- Lo entendemos como
  - un estilo de programación
  - en el que se escriben soluciones a problemas en términos de algoritmos
- Ingrediente básico es el modelo de cómputo
  - la visión que tiene el usuario de cómo se ejecutan sus programas

Objetivos del curso

Conocer los *pilares conceptuales* sobre los cuales se erigen los lenguajes de programación de modo de poder

- Comparar lenguajes
- Seleccionar el más adecuado para una determinada tarea
- Usar las herramientas adecuadamente
- Prepararse para lenguajes/paradigmas futuros

Nos centraremos en

En esta y en tleng el objeto de estudio

- los paradigmas:
  - *imperativo*
  - funcional
  - orientado a objetos
  - lógico
- Hay otros: concurrente, eventos, continuaciones, probabilístico, quantum.
- ¡La distinción a veces no es clara!

Enfoque del curso

1. Conceptos
   - Introducción informal de conceptos a través de ejemplos.
   - Ilustración en lenguajes concretos (Haskell, Prolog y JavaScript)
2. Fundamentos
   - Introducir las bases rigurosas (lógicas y matemáticas) sobre las que se sustentan cada uno de los paradigmas o parte de los mismos

Programación funcional

Programa y modelo de cómputo

- Programar = Definir funciones.
- Ejecutar = Evaluar expresiones.

Programa

- Conjunto de ecuaciones `doble x = x + x`

Expresiones

- El significado de una expresión es su valor (si está definido).
- El valor de una expresión depende sólo del valor de sus sub-expresiones.

transparencia

- Evaluar/Reducir una expresión es obtener su valor.
  `doble 2` $\rightsquigarrow$ `4`
- No toda expresión denota a un valor: `doble true`

Tipos

Tipos

- El universo de valores está particionado en colecciones, denominadas tipos
- Un tipo tiene operaciones asociadas

Tipos básicos (primitivos)

```
Int
Char
Float
Bool
```

Tipos Compuestos

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

Expresiones y tipo

Tipado Fuerte

- Toda expresión bien-formada tiene un tipo
- El tipo depende del tipo de sus subexpresiones

Tipos elementales

```
1           :: Int
'a'         :: Char
1.2         :: Float
True        :: Bool
[1,2,3]     :: [Int]
(1, True)   :: (Int, Bool)
succ     :: Int -> Int
```

Si no puede asignarse un tipo a una expresión, no se la considera bien formada.

Funciones

Se puede pensar que solo te

Definición

```
doble :: Int -> Int
doble x = x + x
```

Guardas

```
signo :: Int -> Bool
signo n | n >= 0    = True
        | otherwise = False
```

Funciones

Definiciones locales

```
f (x,y) = g x + y
        where g z = z + 2
```

g solamente esta en el

Expresiones lambda

```
\x -> x + 1
```

Sirve para definir funciones ano

Polimorfismo paramétrico

¿Cuál es el tipo de `id`?

```
id x = x
```

```
id :: a -> a
```

Donde `a` es una variable de tipo.

Alto orden

Las funciones son ciudadanas de primera clase

```
id :: a -> a
id id
```

- pueden ser pasadas cómo parámetros
- pueden ser el resultado de evaluar una expresión
- pueden formar parte de estructuras de datos: `[id]`

Currificación

Definición de suma

```
suma :: ??
suma x y = x + y


suma' :: ??
suma' (x,y) = x + y
```

Tipos

```
suma :: Int -> (Int -> Int)


suma' :: (Int, Int) -> Int
```

Currificación

Currificación

- Mecanismo que permite reemplazar argumentos estructurados por una secuencia de argumentos "simples".
- Ventajas:
  - Evaluación parcial: `succ = suma 1`
  - Evita escribir paréntesis (asumiendo que la aplicación asocia a izquierda): `suma 1 2 = ((suma 1) 2)`

curry y uncurry

Curry

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
curry f x y = f (x, y)
```

Uncurry

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y
```

Tipos Algebraicos

- Se introducen con la cláusula `data`

Día

```
data Dia = Lunes | Martes | Miercoles | Jueves |
Viernes | Sabado | Domingo
```

- donde se define
  - el nombre del tipo
  - los nombres de los constructores
  - los tipos de los argumentos de los constructores

Figura

```
data Figura = Circulo Float | Rectangulo Float Float
```

- `Lunes :: Dia`
- `Circulo 1.0 :: Figura`
- `Circulo :: Float -> Figura`

Pattern matching

- Un mecanismo para
  - comparar un valor con un patrón
  - si la comparación tiene éxito se puede deconstruir un valor en sus partes
- Una función se puede definir por casos de acuerdo con la forma de los parámetros

area

```
area :: Figura -> Float
area (Circulo radio) = pi * radio^2
area (Rectangulo l1 l2) = l1 * l2
```

- El patrón aparece a la izquierda de la ecuación
  - constructor
  - variables (todas distintas)

El patrón debe ser lineal

- Lineal : una variable debe aparecer una única vez a la izquierda

esCuadrado

```
esCuadrado :: Figura -> Bool
esCuadrado (Rectangulo x y) | (x==y) = True
esCuadrado _ = False
```

- _ : coincide con cualquier forma
- los casos se evalúan en el orden en que están escritos
- Se pueden definir funciones parciales

radio

```
radio (Circulo r) = r
```

- `radio :: Figura -> Float`
- `radio (Rectangulo 1 2) :: Float`
- Evaluar `radio (Rectangulo 1 2)` alcanza la excepción `Non-exhaustive patterns`

Tipos Recursivos

- la definición de un tipo A puede tener uno o más parámetros de tipo A

Naturales

```
data Natural = Zero | Succ Natural
```

Listas

Listas

Tipo algebraico paramétrico y recursivo con dos constructores:

- `[] :: [a]`
- `(:) :: a -> [a] -> [a]`

Pattern matching

```
vacia :: [a] -> Bool
vacia [] = True
vacia _  = False
```

Recursión

Longitud

```
long :: [a] -> Int
long []       = 0
long (x:xs)   = 1 + long xs
```

Evaluación Lazy

Modelo de cómputo: Reducción

- Se reemplaza un *redex* por otra utilizando las ecuaciones orientadas. Un redex (reducible expresion) es una sub-expresión que puede reescribirse
- El redex debe ser una instancia del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las variables correspondientes ligadas.
- El resto de la expresión no cambia.

Para seleccionar el redex: Orden Normal, o también llamado Lazy

- Se selecciona el redex más externo para el que se pueda conocer que ecuación del programa utilizar.
- En general: Primero las funciones más externas y luego los argumentos (sólo si se necesitan).

No terminación y orden de evaluación

No terminación

```
inf1 :: [Int]
inf1 = 1 : inf1
```

Evaluación no estricta

```
const :: a -> b -> a
const x y = x
const 42 inf1 ⇝ 42
```

Ejercicios

- Definir
  - `dobleL :: [Float] -> [Float]` tal que `dobleL xs` es la lista que contiene el doble de cada elemento en `xs`

dobleL

```
dobleL [] = []
dobleL (x:xs) = (doble x) : (dobleL xs)
```

  - `esParL :: [Int] -> [Bool]` tal que la lista `esParL xs` indica si el correspondiente elemento en `xs` es par o no

esParL

```
esParL [] = []
esParL (x:xs) = (even x) : (esParL xs)
```

  - `longL :: [[a]] -> [Int]` tal que `longL xs` es la lista que contiene las longitudes de las listas en `xs`

longL

```
longL [] = []
longL (x:xs) = (length x) : (longL xs)
```

Map

Map

```
map ::(a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs)= (f x):(map f xs)
```

Ejercicios

- Definir
  - `negativos :: [Float] -> [Float]` tal que `negativos xs` contiene los elementos negativos de `xs`

negativos

```
negativos [] = []
negativos (x:xs) | x < 0 = x : (negativos xs)
negativos (x:xs) = negativos xs
```

  - `noVacias :: [[a]] -> [[a]]` tal que la lista `noVacias xs` contiene las listas no vacías de `xs`

noVacias

```
noVacias [] = []
noVacias (x:xs) | length x>0 = x:(noVacias xs)
noVacias (x:xs) = noVacias xs
```

Filter

Filter

```
filter ::(a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)= if (p x) then x:(filter p xs)
                 else (filter p xs)
```

Vemos que negativos (o cualq otro ejemplo) cu

Clases de tipos (Type classes)

¿Cuál es el tipo de `maximo`?

```
maximo x y | x > y = x
maximo _ y = y
```

Podría ser cualquier tipo que provea la operación >

- Una clase de tipo es como una interface, define un conjunto de operaciones
  - `Eq`: `(==)`, `(/=)`
  - `Ord`: `(<)`, `(<=)`, `(>=)`, `(>)`, `max`, `min`, `compare`

¿Cuál es el tipo de `maximo`?

```
maximo :: Ord a => a -> a -> a
```

Instancia de una clases de tipos

Deriving

```
data Figura = Circulo Float | Rectangulo Float Float
deriving Eq
```

Definir una instancia

```
instance Ord Figura where
  (<=) = \x -> \ y-> area x <= area y
```