Paradigmas de lenguajes de programación

Departamento de Computación, FCEyN, UBA

1er Cuatrimestre 2020 (modalidad virtual)

Docentes

- Gonzalo Fernández Florio
- Carolina González
- Malena Ivnisky
- Daniela Marottoli
- Hernán Melgratti (Prof)
- Facundo Ruiz
- Gabriela Steren (JTP)

Modalidad (sujeta a cambios)

Herramientas de comunicación

- Zoom (aula 9)
- Discord (Link en la página principal de la materia)
- Meet (si fuese necesario enviaremos coordenadas a la lista)
- **listas de email (docentes y alumnos)**

Franjas horarias de actividades interactivas

- Martes de 17.30 a 21.00
- Jueves de 17 a 19.30

Modalidad

- Variarán y serán comunicadas según corresponda
- Requerirá un rol activo

Recursos

Bibliografía

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

Página web

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

Mailing list

- ¡Hacer todas las preguntas y consultas que quieran!

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

- 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

Cómo seguimos? Actividad para realizar individualmente

- Repaso de Haskell.
  - Índice de temas en las transparencias que siguen.
  - Recursos:
    - Graham Hutton. *Programming in Haskell* (Desde 1 a 7.2 y (1st Ed.) 10.1 a 10.4 o (2nd Ed.)8.1 a 8.4
    - Transparencias de Programación Funcional (de Fidel) (hasta filter sobre listas inclusive)
    - Video G. Hutton sobre funciones currificadas (`https://www.youtube.com/watch?v=psmu_VAuiag`)
- Instalar GHC (Glasgow Haskell Compiler).

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.
- 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

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
```

Expresiones lambda

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

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

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 a b = f (a,b)
```

Uncurry

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

Listas

Listas

Tipo algebraico 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)
```

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
```

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 no esté en forma normal.
- 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).

Map

Map

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