Paradigmas de Programación (M)

Introducción a la materia

2do cuatrimestre de 2024

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Docentes (Turno Mañana)

Profesores

- Hernán Melgratti

Jefes de trabajos prácticos

- Christian Cossio Mercado

Ayudantes de primera

- Malena Ivnisky

Ayudantes de segunda

- Rafael Romani
- Lucas Di Salvo
- Francisco Demartino

Días y horarios de cursada

- Martes de 9:00 a 14:00 — generalmente práctica
- Viernes de 9:00 a 14:00 — generalmente teórica

Vías de comunicación

Docentes → alumnxs
Avisos a través del campus.

Alumnxs → docentes
Lista de correo: `plp-docentes@dc.uba.ar`
(para consultas administrativas)

Discusión entre estudiantes fuera del horario de la materia
Servidor de Discord: `https://tinyurl.com/plpdiscord`
(con eventual participación de docentes)

Modalidad de evaluación

Parciales

- Primer parcial — 8 de Octubre
- Segundo parcial — 26 de Noviembre
- Recuperatorio del primer parcial — 3 de diciembre
- Recuperatorio del segundo parcial — 10 de diciembre

Trabajos prácticos

- TP 1 (con su recuperatorio)
- TP 2 (con su recuperatorio)

Examen final

La materia **es promocionable**: Leer cuidadosamente la normativa.

Objeto de estudio de esta materia

Programación

Proceso de escribir instrucciones que una computadora puede ejecutar para resolver algún problema.

Programa

Un conjunto de instrucciones que una computadora sigue para realizar una tarea específica.

¿Cómo se escriben esas instrucciones o (programas)?

Lenguajes de Programación

Lenguaje de Programación

Un formalismo artificial en el que se pueden describir computaciones.

Luego

La materia **Paradigmas de Programación** tiene como objetivo **el estudio de los lenguajes de programación**.

Algunos Lenguajes de Programación I

- **1940 Plankalkul**: considerado el primer lenguaje de programación (Konrad Zuse)
- **1949 Lenguaje de Ensamblador** y **Shortcode**
- **1957 FORTRAN** Primer lenguaje de programación **de alto nivel** (John Backus)
- **1958 LISP** (List processor) Computación simbólica para IA (John McCarthy)
- **1969 ALGOL 60**: primer uso de la estructura de bloques, procedimientos recursivos, notación BNF, paso de parámetros por valor y por nombre, y varias otras características muy comunes en los lenguajes actuales
- **1959 COBOL** (Common Business Oriented Language)) Aplicaciones bancarias (Grace Hopper)
- **1967 Simula 67** Simulación, corutinas y clases.

Algunos Lenguajes de Programación II

- **1968 ALGOL 68** Estructuras de datos definidas por el usuario.
- **1970 PL/1** Primer intento a gran escala de un lenguaje con un amplio espectro de áreas de aplicación (IBM)
- **Principios de los 70 BASIC** (Beginners All-Purpose Symbolic Instruction Code) Fácil de aprender.
- **Principios de los 70 PASCAL** Fácil de aprender (Niklaus Wirth)
- **1972 C** Originalmente para programación de sistemas, Unix (Dennis Ritchie)
- **1972 Smalltalk** Programación orientada a objetos (Alan Kay, Adele Goldberg y Dan Ingalls)
- **1972 SQL** (Structured Query Language) Para bases de datos (Raymond Boyce y Donald Chamberlain)

Algunos Lenguajes de Programación III

- **Principios-mediados de los 70 PROLOG** Programación lógica (Alain Colmerauer y Philippe Roussel)
- **Mediados de los 70 Scheme** Dialecto de LISP, funciones como ciudadanos de primera clase.
- **Finales de los 70 ML** Propósito general, (principalmente) funcional (Robin Milner et al.)
- **1980 ADA** Sistemas críticos, encapsulación, abstracción de datos, excepciones, genericidad.
- **1983 C++** Ampliación de C con clases, plantillas, funciones virtuales (Bjarne Stroustrup)
- **1985 Eiffel** Orientado a objetos, Diseño por contrato (Bertrand Meyer)
- **1986 Erlang** Sistemas escalables, tiempo real, telecomunicaciones, modelo de actores (Joe Armstrong, Robert Virding, Mike Williams)

Algunos Lenguajes de Programación IV

- **1987 Perl** Procesamiento de texto y datos (Larry Wall)
- **1989 Coq** Programación funcional con tipos dependientes, asistencia para pruebas (Thierry Coquand et al.)
- **1990 Haskell** Programación funcional, inferencia de tipos.
- **1999 Agda** Programación funcional con tipos dependientes, asistencia para pruebas (Catarina Coquand)
- **1991 Python** Propósito general (Guido Van Rossum)
- **1993 Ruby** Aplicaciones web (Yukihiro Matsumoto)
- **1995 Java** Propósito general, (basado en clases) Orientado a objetos (James Gosling)
- **1995 JavaScript** Aplicaciones web, (basado en prototipos) Orientado a objetos (Brendan Eich)
- **1995 PHP** Aplicaciones web (Rasmus Lerdorf)
- **2003 Scala** Integra programación funcional y orientada a objetos (Martin Odersky)

Algunos Lenguajes de Programación V

- **2009 Go** Concurrencia, comunicación por paso de mensajes (Robert Griesemer, Rob Pike y Ken Thompson)
- **2010 Rust** Programación de sistemas, multiparadigma, propiedad y préstamo (Graydon Hoare)
- **2014 Solidity** Smart contracts en Ethereum (Gavin Wood et al.)

¿Cómo vamos a tratar con la gran diversidad?

- Existen muchos lenguajes de programación:
  - La lista precedente contiene sólo a algunos de los lenguajes mas populares, o históricamente significativos.
- Sin embargo, hay características en ellos que son comunes.

> Vamos a estudiar distintos enfoques o estilos para la construcción y estructuración de programas en un lenguaje de programación

Paradigma

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*

Paradigma de Programación

Paradigma 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

Paradigmas de programación

- Paradigmas clásicos:
  - Imperativo
  - **Funcional**
  - **Lógico**
  - **Orientado a Objetos**
- Existen otros: concurrente, event-driven, quantístico, probabilístico, etc.

¿Qué estudiamos de un Lenguaje de Programación?

Differentes Aspectos

- Gramática
- Semántica
- Pragmática
- Implementación

Gramática

Gramática

El nivel que responde a la pregunta **"¿Qué frases son correctas?"**

- Establece
  - el **alfabeto**
    - Ejemplo: alfabeto latino de 22 o 26 letras, alfabeto cirílico, alfabeto chino, etc.
  - las **palabras** (o *tokens*): secuencia válida de símbolos
  - la **sintaxis**: las secuencias de palabras que son frases legales.

Semántica

Semántica

El nivel que responde a la pregunta **"¿Qué significa una frase correcta?"**

- La semántica asigna un significado a cada frase correcta.

Pragmática

Pragmática

El nivel que responde a la pregunta **"¿Cómo usamos una frase significativa?"**

- Las frases con el mismo significado pueden usarse de diferentes maneras.
- Diferentes contextos pueden requerir diferentes frases:
  - algunas son más elegantes,
  - algunas son más eficientes,
  - algunas son más dialectales, etc.

Implementación

Implementación

El nivel que responde a la pregunta **"¿Cómo ejecutar una frase correcta, de manera que respetemos la semántica?"**

- Esto generalmente no es necesario para el usuario (programador)
- Es fundamental para los diseñadores e implementadores del lenguaje.

¿Qué aspectos estudiamos en Paradigmas de Programación?

- **Pragmática**: Vamos a usar tres lenguajes
  - Haskell, Prolog, Smalltalk
- **Semántica**: Estudiamos distintas alternativas para definir el significado de un programa de **manera formal**.
- **(Algunos aspectos de) Implementación**: Interpretación e inferencia de tipos.

Sobre sintaxis

Vamos a hacer uso, pero este aspecto es objeto de estudio de **Lenguajes Formales, Autómatas y Computabilidad**

Cronograma

Programación funcional	2 semanas
Razonamiento ecuacional	1 semana
Sistemas deductivos	1 semana
Cálculo-λ	2 semanas
(Repaso)	
Primer parcial	
Interpretación e inferencia	1 semanas
Lógica de primer orden	1 semana
Resolución	1 semana
Programación lógica	1,5 semanas
Programación orientada a objetos	1 semana
(Repaso)	
Segundo parcial	

Bibliografía (no exhaustiva)

Lógica proposicional y de primer orden

Logic and Structure
D. van Dalen.

Semántica y fundamentos de la implementación

Introduction to the Theory of Programming Languages
J.-J. Lévy, G. Dowek. Springer, 2010.

Types and Programming Languages
B. Pierce. The MIT Press, 2002.

Programación funcional

Introduction to Functional Programming using Haskell
R. Bird. Prentice Hall, 1998.

Programación lógica

Logic Programming with Prolog
M. Bramer. Springer-Verlag, 2013.

Programación orientada a objetos

Smalltalk-80 the Language and its Implementation
A. Goldberg, D. Robson. Addison-Wesley, 1983.

Programación funcional

Programa y modelo de cómputo

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

Programa

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

Expresiones

Las **expresiones** son secuencias de símbolos que sirven para representar datos, funciones y funciones aplicadas a los datos. (Recordemos: las funciones también son datos).

Una expresión puede ser:

1. Un **constructor**:

   `True  False  []  (:)  0  1  2  ...`

2. Una **variable**:

   `longitud  ordenar  x  xs  (+)  (*)  ...`

3. La **aplicación** de una expresión a otra:

   ```
   ordenar lista
   not True
   not (not True)
   ```

4. También hay expresiones de otras formas, como veremos. Las tres de arriba son las fundamentales.

Tipos

Tipos

- El universo de las expresiones 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
```

Tipos

Ejemplo

```
99          :: Int
not         :: Bool -> Bool
not True    :: Bool
(+)         :: Int -> Int -> Int
((+) 1) 2   :: Int
```

Tipos

Hay secuencias de símbolos que no son expresiones bien formadas (sintaxis).

Ejemplo

```
1,,2      )f x(
```

Hay expresiones que están bien formadas pero no tienen sentido (semántica).

Ejemplo

```
True + 1
0 1
[[], (+)]
```

Tipos

Condiciones de tipado

Para que un programa esté **bien tipado**:

1. Todas las expresiones deben tener tipo.
2. Cada variable se debe usar siempre con un mismo tipo.
3. Los dos lados de una ecuación deben tener el mismo tipo.
4. El argumento de una función debe tener el tipo del dominio.
5. El resultado de una función debe tener el tipo del codominio.

Sólo tienen sentido los programas bien tipados.

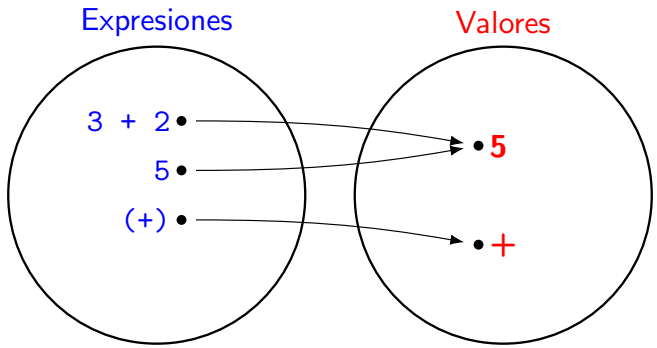
Evaluación

Valor

- 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`

Modelo de cómputo

Dada una expresión, se computa su *valor* usando las ecuaciones:



Hay expresiones bien tipadas que no tienen valor. Ej.: `head []`.
Decimos que dichas expresiones se indefinen o que tienen valor $\bot$.

Modelo de cómputo

Un programa funcional está dado por un conjunto de ecuaciones.
Más precisamente, por un conjunto de **ecuaciones orientadas**.

Una ecuación `e1 = e2` se interpreta desde dos puntos de vista:

1. **Punto de vista denotacional.**
   Declara que `e1` y `e2` tienen el mismo significado.
2. **Punto de vista operacional.**
   Computar el valor de `e1` se reduce a computar el valor de `e2`.

Modelo de cómputo

El lado *izquierdo* de una ecuación no es una expresión arbitraria.
Debe ser una función aplicada a **patrones**.

Un patrón puede ser:

1. Una variable.
2. Un comodín `_`.
3. Un constructor aplicado a patrones.

El lado izquierdo no debe contener variables repetidas.

Ejemplo

¿Cuáles ecuaciones están sintácticamente bien formadas?

```
sumaPrimeros (x : y : z : _) = x + y + z
predecesor (n + 1) = n
iguales x x = True
```

Modelo de cómputo

Evaluar una expresión consiste en:

1. Buscar la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
2. Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
3. Continuar evaluando la expresión resultante.

La evaluación se detiene cuando se da uno de los siguientes casos:

1. La expresión es un constructor o un constructor aplicado.
   `True` `(:) 1` `[1, 2, 3]`
2. La expresión es una función *parcialmente* aplicada.
   `(+)` `(+) 5`
3. Se alcanza un *estado de error*.
   Un estado de error es una expresión que no coincide con las ecuaciones que definen a la función aplicada.

Modelo de cómputo

Ejemplo: resultado — constructor

```
tail :: [a] -> [a]
tail (_ : xs) = xs
```

`tail (tail [1, 2, 3])` $\rightsquigarrow$ `tail [2, 3]` $\rightsquigarrow$ `[3]`

Modelo de cómputo

Ejemplo: indefinición — error

```
head :: [a] -> a
head (x : _) = x
```

```
head (head [[], [1], [1, 1]]) ⇝ head []
                              ⇝ ⊥
```

Ejemplo: indefinición — no terminación

```
loop :: Int -> a
loop n = loop (n + 1)
```

```
loop 0 ⇝ loop (1 + 0)
       ⇝ loop (1 + (1 + 0))
       ⇝ loop (1 + (1 + (1 + 0)))
         ...
```

Modelo de cómputo

Ejemplo: evaluación no estricta

```
indefinido :: Int
indefinido = indefinido
```

```
    head (tail [indefinido, 1, indefinido])
⇝ head [1, indefinido]
⇝ 1
```

Modelo de cómputo

Ejemplo: listas infinitas

```
desde :: Int -> [Int]
desde n = n : desde (n + 1)
```

```
    desde 0
⇝ 0 : desde 1
⇝ 0 : (1 : desde 2)
⇝ 0 : (1 : (2 : desde 3)) ⇝ ...
```

```
   head (tail (desde 0))
⇝ head (tail (0 : desde 1))
⇝ head (desde 1)
⇝ head (1 : desde 2)
⇝ 1
```

Modelo de cómputo

Nota. En Haskell, el orden de las ecuaciones es relevante.
Si hay varias ecuaciones que coinciden se usa siempre la primera.

Ejemplo

```
esCorta (_ : _ : _) = False
esCorta _           = True
```

```
esCorta []     ⇝ True
esCorta [1]    ⇝ True
esCorta [1, 2] ⇝ False
```

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?

```
id x = x

id :: a -> a
```

Donde a es una variable de tipo.

Polimorfismo

Hay expresiones que tienen más de un tipo.
Usamos *variables de tipo* a, b, c para denotar tipos desconocidos:

```
id   ::  a -> a
[]   ::  [a]
(:)  ::  a -> [a] -> [a]
fst  ::  (a, b) -> a
snd  ::  (a, b) -> b
```

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)`

Tipo de funciones

Convenimos en que "->" es **asociativo a derecha**:

```
a -> b -> c       ≡  a -> (b -> c)    ≢  (a -> b) -> c
a -> b -> c -> d  ≡  a -> (b -> (c -> d))
```

Ejemplo

```
suma4 :: Int -> Int -> Int -> Int -> Int
suma4 a b c d = a + b + c + d
```

Se puede pensar así:

```
suma4 :: Int -> (Int -> (Int -> (Int -> Int)))
(((suma4 a) b) c) d = a + b + c + d
```

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
```

Función flip (alto orden)

Flip

```
flip f x y = f y x
```

¿Qué tipo tiene `flip`?

```
   flip (:) [2, 3] 1
=  (:) 1 [2, 3]
≡  1 : [2, 3]
=  [1, 2, 3]
```

Composición

Definamos la composición de funciones ("g . f").

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

Otra forma de definirla (usando la notación "lambda"):

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

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`

    dobleL

    ```
    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)
```