Tecnologías de Programación

Resumen 2025

Paradigmas de Programación

Un **paradigma** es un modelo que determina una manera especial de entender problemas y sus soluciones.

Un **paradigma de programación** es un enfoque particular para la construcción de software. Cada paradigma tiene ventajas y desventajas y, en teoría, cualquier problema debería poder ser resuelto por cualquier paradigma *pero existen situaciones donde un paradigma es más apropiado que otro*. Por esto, para resolver un determinado problema debemos conocer cuál paradigma se adapta mejor a su resolución

Programación

Un **programa** es un conjunto de código que describen, definen o caracterizan las acciones que puede ejecutar una computadora. Como la estructura final de un programa es determinada por el paradigma que se utilizó para su diseño existen diferentes conceptos de "programa"para cada paradigma.

El proceso de **programación** incluye diseñar, codificar, depurar y mantener el código.

Criterios de la buena programación:

- Modelos cercanos a la *realidad*
- Implementaciones que puedan ser *extendidas* y *modificadas*
- Flexibilidad para la *reutilización*
- *Código claro* y compacto
- Soluciones *genéricas*
- *Focalización* de las funciones

Paradigmas fundamentales

Los primeros lenguajes de programación eran **imperativos** o **procedurales**, se centran en operaciones que modifican datos en memoria y el estado del programa a través de algoritmos que detallan *el paso a paso* con estructuras de control. Se suele usar este paradigma cuando hay una secuencia clara y queremos tener mayor control sobre el algoritmo. (**Paradigma estructurado**)

Los paradigmas **declarativos** surgieron como alternativa, enfocándose en el *qué* de la solución en vez del *cómo*. Utilizan declaraciones (condiciones, proposiciones, restricciones, etc) para describir el problema, dejando los detalles de la implementación a mecanismos internos. Se suele usar este paradigma cuando se expresan características del problema y queremos tener menos control sobre el algoritmo. (**Paradigmas funcional y lógico**)

El paradigma **orientado a objeto** agregan el concepto de objeto, que encapsula datos y comportamientos.

Paradigma Estructurado

ALGORITMO + ESTRUCTURA DE DATOS = PROGRAMA

Se define por el estado del programa y las instrucciones que lo modifican.

Lenguajes asociados: Fortran, C, Pascal

Ventajas	Desventajas
• Código directo y de rápida ejecución • De fácil corrección y mantenimiento (*De estar bien modularizado*)	

Paradigma Funcional

FUNCIONES + CONTROL = PROGRAMA

Se enfoca en la aplicación y composición de funciones, permitiendo soluciones más elegantes.

Lenguajes asociados: Lisp, Haskell, Scheme, Scala.

Aplicaciones: IA y aprendizaje automático, sistemas de control tolerantes a fallos de software (ej. tráfico aéreo, mensajería instantánea, etc), matemáticas y científicas, procesamientos de datos, telecomunicaciones, interfaces de usuario reactivas.

Ventajas	Desventajas
• Altos niveles de abstracción • Cortos y sencillos de entender (*Por el nivel de abstracción*) • Rapidez en la codificación • Administración automática de memoria. • Evita gastos computacionales innecesarios (*Por la evaluación perezosa*)	• Portabilidad y herramientas limitadas. • Baja eficiencia de ejecución. (*Por estar alejado al modelo de la máquina de von Neumann*)

Paradigma Logico

LÓGICA + CONTROL = PROGRAMA

Basado en la definición de reglas lógicas para responder, mediante un motor de interferencias lógicas, preguntas planteadas al sistema.

Lenguajes asociados: Prolog, Datalog.

Aplicaciones: Sistemas expertos, sistemas de soporte de decisiones (DSS), sistemas de reglas de negocio (Drools), sistemas de conocimiento y aprendizaje, robótica.

Ventajas	Desventajas
• Sencillez y potencia en la búsqueda de soluciones. • Sencillez en la implementación de estructuras complejas.	• Poco utilizado en gestión. • Pocas herramientas de depuración. • Pocos eficientes por lentitud de ejecución.

Paradigma Orientado a Objetos

OBJETOS + MENSAJES = PROGRAMA

Se basa en la idea de *encapsular* estado y operaciones en objetos los cuales no están aislados entre sí, sino que colaboran para lograr un único objetivo.
La colaboración se realiza a través de *mensajes*.
Lenguajes asociados: Smalltalk, Java, C++, Python.
Aplicaciones: Tecnología web, modelado y simulación, bases de datos, programación en paralelo, sistemas que requieren administración de estructuras complejas.

Ventajas	Desventajas
• Reusabilidad. • Mantenibilidad (*Fácil abstracción*) • Fiabilidad (*Fácil aislación de problemas*)	• Algunos de los mecanismos (*Como la herencia*) provocan una ejecución más lenta. • Curva de aprendizaje más grande.

Ejemplo

Calcular promedio. Mismo problema, diferentes paradigmas.

Imperativo

```python
numeros = [10, 20, 30, 40, 50]
suma = 0

for numero in numeros:
    suma += numero

promedio = suma / len(numeros)
print("promedio:", promedio)
```

Python.
Define explícitamente cómo sumar los números y cómo calcular el promedio mediante iteraciones y asignación de variables.

Orientado a objetos

```python
class CalculadoraPromedio:
    def __init__(self, numeros):
        self.numeros = numeros
    def calcular_promedio(self):
        suma = sum(self.numeros)
        return suma / len(self.numeros)

numeros = [10, 20, 30, 40, 50]

calculadora = CalculadoraPromedio(numeros)
print("Promedio:",
calculadora.calcular_promedio())
```

Python.
Modela objetos y sus interacciones.

Funcional	
(define (promedio numeros) (define suma(apply + numeros)) (/suma (length numeros)) (define numeros '(10 20 30 40 50)') (display ln (promedio numeros))	Racket. La función promedio no depende de ningún estado ni modifica variables globales-
Lógico	
promedio(Numeros, Promedio):- suma(Numeros, Suma) length(Numeros, Len) Promedio is Suma / Len. suma ([],0). suma ([H|T], Suma) :- suma (T, ST), Suma is H + ST	Prolog. Define las relaciones lógicas entre los elementos en lugar de dar instrucciones explícitas.

Lenguajes de programación

Un **lenguaje de programación** es una notación para escribir programas y darle órdenes para la realización de un proceso al Hardware.

Lenguajes Puros

Se apegan rigurosamente a las normas de un paradigma. Tienen una mayor potencia expresiva.

Lenguajes: Smalltalk, Eiffel, HyperTalk y Acto.

Lenguajes Hibridos

Integran aspectos distintivos de diferentes paradigmas. Son menos dogmáticos.

Lenguajes: Objetive C, Objetive Pascal, Java, C#.

Paradigma Orientado a Objetos

Está basado en la encapsulación del estado, con **atributos**, y el comportamiento, **métodos**. Procura favorecer un buen diseño modular, conformado por pocas interfaces pequeñas y explícitas.

Ventajas	Desventajas
• Reusabilidad. • Fácil mantenimiento y modificación. • Estructura modular clara. • Buen marco para GUIs. • Acopla bien a bases de datos.	• Limitaciones del programador. • Múltiples formas de resolver problemas. • Requiere amplia documentación.

Para el paradigma,

*Un programa es un conjunto de **objetos** que colaboran entre sí enviándole mensajes.*

Características de la POO

Abstracción: Seleccionar las características esenciales de un objeto a modelar e identificar comportamientos comunes.

Modularidad: Subdividir una aplicación en partes más pequeñas (módulos) que deben ser lo más independientes posibles de la aplicación en sí y las partes restantes. Se pueden compilar por separado.

Encapsulamiento: Reunir todos los elementos pertenecientes a una misma entidad al mismo nivel de abstracción.

Principio de ocultación: Aislar cada objeto del exterior y exponer una interfaz (protocolo) a otros objetos que específica cómo interactuar con él. Protege las propiedad de un objeto de modificaciones que no sean por métodos propios, eliminando interacciones inesperadas.

Cohesión y acoplamiento: Ofrecer alta cohesión (Cada objeto se responsabiliza por *una sola* cosa) y bajo acoplamiento (poca o nula interdependencia)

Herencia: Formar una jerarquía de clasificación. Los objetos heredan propiedades y comportamientos de todas las clases a las que pertenecen.

Polimorfismo: Implementar el mismo mensaje en formas diferentes en objetos diferentes.

Recolección de basura: Técnica de destrucción automática de objetos.

Objetos

Los **objetos** son una unidad que encapsula estado y comportamiento. Son instancias de una clase.
Los objetos deben tener:
Abstracción: Representación computacional de entes (con sus características y comportamientos) de la realidad.

- **Esencial:** Hacen lo que el ente sea lo que es.
- **Accidental:** Si el ente no las tiene o se cambia no deja de ser lo que es.

Identidad: Existencia única de un mismo objeto.
Comportamiento: Conjunto de mensajes que puede responder (Protocolo) y mensajes que puede entender (Vocabulario)
Inspección: Visualizar en todo momento a sus colaboradores y poder cambiarlos.

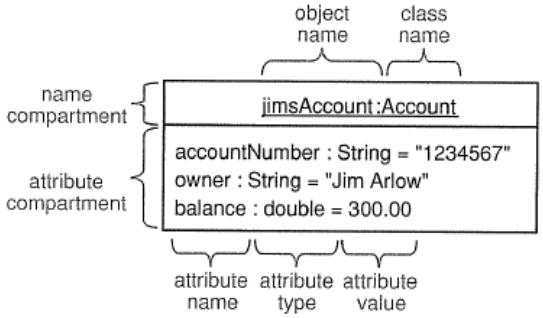
Colaboraciones

Los objetos se mandan mensajes para ejecutar las funciones del sistemas. Esto se llama **colaboración**.

Los **métodos** son el conjunto de colaboraciones que un objeto tendrá con otros. Dictan qué hacer con cada mensaje. Implementan el comportamiento del objeto.
El protocolo o interfaz es público, la definición de métodos y colaboradores es privada.

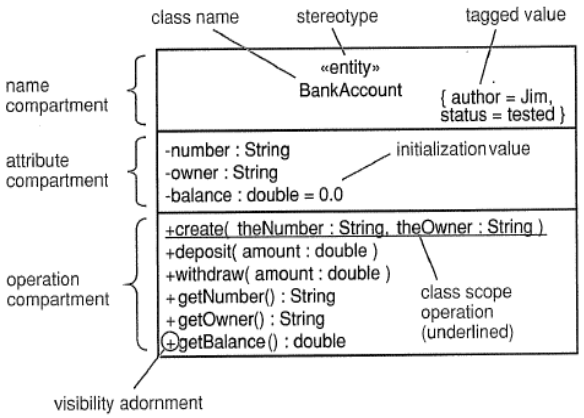
Cuando se envía un mensaje hay un **emisor**, un **receptor** y un **nombre** que identifica el mensaje (puede tener, además, *parámetros*)
Puede ocurrir que un objeto reciba un mensaje que no entienda, esto puede ocurrir por dos razones: (1) se manda al objeto correcto un mensaje equivocado o (2) se manda un mensaje correcto al objeto equivocado.

Hay dos tipos de colaboradores: **habituales** (internos) que son el conjunto de variables que va a tener un objeto y **eventuales** (externos) que son el conjunto de métodos públicos.
En UML;



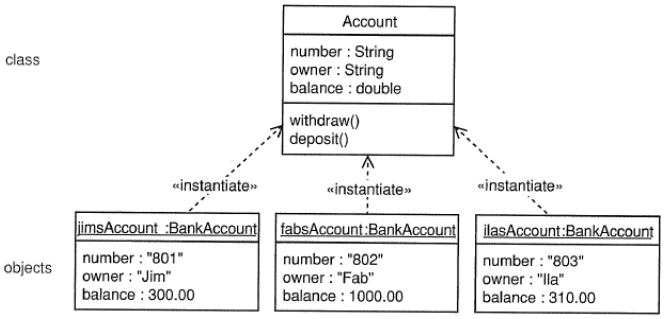
Clases

Las **clases** son plantillas que describen la estructura de los objetos.
En UML;



Instancias

Las relación entre clases y objetos es de <<instantiate>>. La **instanciación** es la creación de nuevas instancias a partir de un elemento modelo.
En UML,

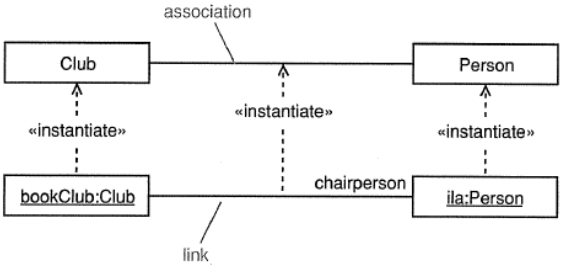


Relaciones entre clases

Una **relación** es una conexión semántica significativa entre elementos modelo.

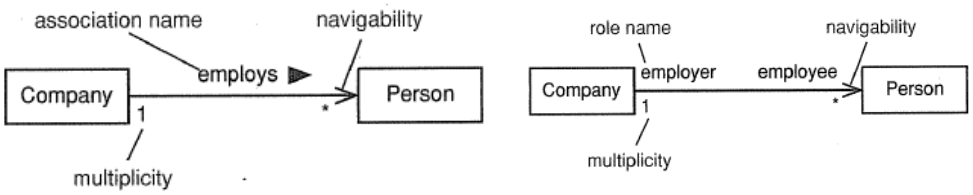
Asociación

Una clase *usa* a otra clase.
Los objetos se mandan mensajes mediante conexiones llamadas **enlaces** (links). Los enlaces son instancias de una asociación.

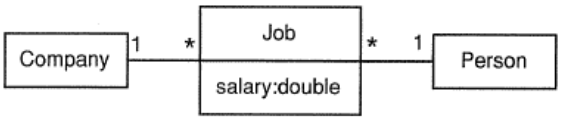


No es una relación fuerte. El tiempo de vida de un objeto no depende del otro.

Las asociaciones tienen: un nombre de asociación o roles, multiplicidad (número de objetos de una clase que puede estar involucrado en una relación) y navegabilidad (hacia donde los mensajes pueden ser mandados).



Clase asociación: Clase intermedia que representa una relación muchos-muchos mediante entradas que tienen relación 1 a muchos. No solo conecta dos clases sino que también define un conjunto de características propias de la relación.



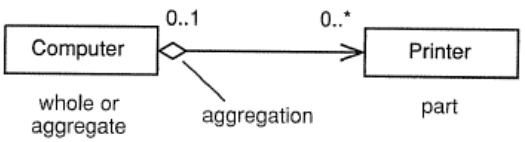
Agregación

Una clase *es parte* de otra clase.

La agregación es un tipo de relación *todo-parte*, donde un objeto (el todo) usa servicios de otro objeto (la parte)

El todo puede a veces existir independientemente de sus partes pero las partes pueden siempre existir independientemente del todo y es posible que varios objetos todos compartan un mismo objeto parte.

La agregación es **transitiva** (Si un objeto C es parte de B y B es parte de A, C es también parte de A) y **asimétrica** (Un objeto jamás puede ser parte de sí mismo)

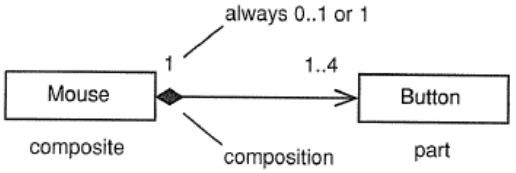


Composición

Más fuerte que agregación.

En este caso las partes no tienen independencia del todo y cada parte le puede pertenecer a máximo un solo todo.

El todo tiene la responsabilidad de manejar el ciclo de vida de las partes y si es destruido, las partes tienen que ser eliminadas (o su responsabilidad pasada a otro objeto)



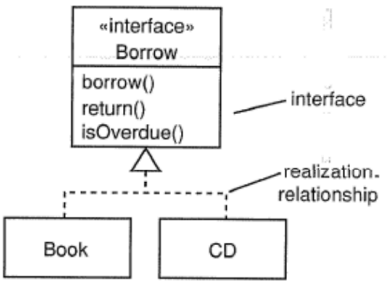
Otros conceptos

Polimorfismo

Invocar métodos distintos con el mismo nombre desde distintos objetos

Interfaces

Las **interfaces** especifican un conjunto de operaciones. El propósito de las interfaces es separar la *especificación* de la funcionalidad de la *implementación*.



En C + + se aplica utilizando clases abstractas.

Principios SOLID

Principio de Responsabilidad Única

Los módulos/clases deben tener una sola responsabilidad y una única razón para cambiar. El SRP está relacionado con la **cohesión**. No aplicarlo puede provocar dependencias innecesarias.

Una clase con muchas responsabilidades puede ser replanteada en varias clases y distribuir sus responsabilidades.

Principio de Abierto/Cerrado

Los módulos deben estar *abiertos* para la extensión (adaptarse a nuevos requerimientos o cambiar el dominio del problema) pero *cerrados* a la modificación (alterar la lógica del código actual). No aplicar el OCP puede provocar que un solo cambio afecte a toda la lógica del programa.

Este principio está relacionado con el uso efectivo del **polimorfismo**.

Principio de Sustitución de Liskov

Las clases hijas deben poder ser sustituidas por la clase base sin alterar el funcionamiento del programa.

LSP está relacionado con el **polimorfismo** y la **herencia**.

Si existen métodos en la clase base que las clases hijas no pueden usar correctamente, se pueden reescribir para que si puedan.

Principio de Segregación de Interfaces

Las interfaces deben ser pequeñas. En Python las interfaces se definen utilizando las clases abstractas (a partir del módulo abc).
El ISP está relacionado con la **cohesión**.
Si una interfaz provee muchos métodos, es mejor dividirla en múltiples interfaces que contengan menos métodos.

Principio de Inversión de Dependencia

Las abstracciones tienen que ser pensadas de tal forma que la implementación concreta no dependa de la abstracción.

Otros conceptos

Ley de Demeter

“Habla solo con tus amigos”
Un objeto solo puede invocar métodos suyos y de los objetos miembros de su clase.
Evitar cadena de llamados.

Tell don’t ask

En lugar de preguntarle a un objeto su estado y tomar decisiones basadas en su respuesta, se debe simplemente "decirle" al objeto qué hacer y permitirle que gestione internamente su estado.
Reduce acoplamiento y encapsulamiento.

Único punto de salida

Un programa debe tener un punto de entrada principal y bien definido donde inicia la ejecución e, idealmente, solo un punto de salida donde la ejecución del programa termina.

Patrones de diseño

Base para la búsqueda de soluciones a problemas comunes en el desarrollo de software.
Su uso no es obligatorio pero sí aconsejable. Forzar el uso de los patrones puede ser un error.

Ventajas:

- **Evitar** la reiteración en la búsqueda de soluciones a problemas conocidos y solucionados.
- **Formalizar** un vocabulario común entre diseñadores
- **Estandarizar** el modo que se realiza el diseño.
- **Facilitar** el aprendizaje condensando conocimiento ya existente.

Componentes

- **Nombre:** Describe en pocas palabras un problema.
- **Problema:** Indica cuándo aplicar el patrón (Requerimientos a cumplir, restricciones a considerar, propiedades deseables de la solución)

- **Solución:** Brinda una descripción abstracta del problema y cómo resolverlo.
- **Consecuencias:** Resultados en términos de ventajas e inconvenientes.

Otros tipos

Según el nivel de abstracción se distinguen tres patrones de diseño.

- **Patrones arquitectónicos:** Definen una estructura fundamental sobre la organización del sistema.
- **Patrones de diseño:** Describen una estructura recurrente y común de componentes comunicantes.
- **Patrones de codificación:** Implementan aspectos particulares del diseño en un lenguaje de programación específico.

Catálogos

Los patrones definen subsistemas o componentes de software. Abstraen el proceso de instanciación, así el sistema es independiente de cómo se crean, componen y representan los objetos.

Patrones Creacionales

Estos patrones se encargan de manejar la instanciación de objetos.
Características:

- **Instanciación genérica:** Permite que los objetos sean creados sin especificar clases concretas.
- **Simplicidad:** Facilitan la creación de objetos evitando código complejo.
- **Restricciones creacionales**: Ayudan a establecer restricciones sobre la creación.

Patrón Singleton

Asegura que una determinada clase sea instanciada *una sóla vez*, proporcionando un único punto de acceso global a ella.

Problemas:

(1) Garantizar única instancia: Pensado para controlar el acceso de algún recurso compartido (Imposible con un constructor normal)

(2) Único punto de acceso: Pensado para evitar que otro código reescriba el estado de la instancia.

Solución:

(1) Hacer privado al constructor: Evita el uso del operador new.

(2) Crear un método de creación estático: Invoca al constructor privado para crear un objeto y lo guarda en un campo estático.

Ventajas	Desventajas
• Certeza de única instancia y punto de acceso • Inicialización cuando se requiera	• Vulnera el Principio de Responsabilidad Única. • Puede enmascarar mal diseño

	• Dificil de simular

Patrones Estructurales

Estos patrones explican cómo combinar objetos y clases en estructuras más grandes.

Composite

Compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.

Problema:

(1) El modelo central puede ser pensado como un árbol.

Solución:

(1) Trabajar desde una interfaz común.

Estructura:

(1) Interfaz componente: Describe operaciones comunes entre elementos simples y complejos del árbol.

(2) Hoja: Elemento sin suplementos. Terminan haciendo la mayoría del trabajo, ya que no se lo delegan a nadie más.

(3) Contenedor/Composite: Elemento con sub-elementos. Delega su trabajo a los subelementos, procesa los resultados intermedios y devuelve el resultado final.

(4) Cliente: Manipula los objetos en la composición a través de la interfaz componente.

Ventajas	Desventajas
• Trabajar con comodidad estructuras de árbol complejas • Principio abierto/cerrado	• Difícil proporcionar una interfaz común única para clases cuya funcionalidad difiere demasiado.

Patrones de Comportamiento

Se focalizan en el flujo de control y asignación de responsabilidades. Pueden estar basados en clases (en donde utilizan la herencia para distribuir el comportamiento) o en objetos (en donde utilizan la composición)

Observer

Brinda un mecanismo que permite transmitir mensajes a aquellos objetos que estén interesados en él cuando este se actualice.

Problema:

(1) Un objeto está interesado en el estado de otro: Cómo está interesado debe de saber su estado en todo momento, pero preguntarlo en todo momento es ineficiente ya que este probablemente no cambie su estado constantemente.

(2) El objeto interés desconoce quienes están interesados en él: Por otro lado, el objeto interés *no* puede estar mandando él cuando se actualiza porque la mayoría de esos mensajes llegarían a objetos que no están interesados en él.

Solución:

(1) Agregar un mecanismo de suscripción: Los objetos que quieren conocer los cambios de estado (suscriptores) en el objeto interés (notificador) se *suscriben* al objeto. Cuando el notificador tiene un evento importante, recorre sus suscriptores y llama al método de notificación.

Estructura:

(1) Notificador: Envía eventos de interés (cambio de estado o ejecución de comportamientos) a otros objetos. Tiene una estructura que permite agregar y quitar suscriptores.

(2) Suscriptores: Realizan acciones en respuesta a las notificaciones emitidas por el notificador mediante una interfaz.

(3) Interfaz suscriptora: Declara la interfaz de notificación.

(4) Cliente: Crea objetos tipo notificador y suscriptor por separado.

Ventajas	Desventajas
• Principio abierto/cerrado • Establecer relaciones entre objetos durante tiempo de ejecución	• Los suscriptores son notificados en orden aleatorio.

Particularidades de Python

Métodos privados

Python no tiene verdaderos métodos privados (aquellos que solo pueden ser usados internamente de una clase) pero se sigue una convención de nomenclatura para identificarlos: dos guiones bajos (__) como prefijo en el nombre del método

Main

Si bien puede no ser definido para que algunos códigos funcionen, lo mejor es que se haga.

Se declara como `if __name__ = "__main__":` .

Tipado

Si bien no es necesario definir el tipo de variable, se puede explicitar con el tipeado `nombre : str`

Funciones Útiles

`isinstance(objeto, Clase)` : Determina si un objeto es una instancia de otra clase.

`issubclass(objeto, Clase)` : Determina si un objeto es una subclase de otra clase.

`super()` : Accede a métodos y atributos de la clase base desde una subclase.

Paradigma Funcional

Es un **paradigma declarativo** que se basa en un modelo matemático de composición de funciones.
No existe el concepto de celda de memoria que es asignada o modificada. El resultado de un cálculo es la entrada del siguiente y funcionan como valores intermedios.

Características

Transparencia referencial: El valor que devuelve una función está únicamente determinado por el valor de sus argumentos. Los valores resultantes son *inmutables.* No existe el concepto de cambio de estado.

Utilización de tipos de datos genéricos: Aumenta flexibilidad y permite unidades de software genéricas (Implementación de polimorfismo)

Recursividad: Basado en el principio de inducción.

Posible trato de funciones como datos: La definición de funciones de orden superior permite un gran nivel de abstracción y generalidad en las soluciones.

No gestión de memoria: Se releva al programa la gestión de memoria. El almacenamiento se asigna e inicializa implícitamente.

Ventajas	Desventajas
• Fácil de formular matemáticamente • Administración automática de memoria • Simplicidad en el código • Rapidez en la codificación	• Difícil de escalar. • Difícil de integrar a otras aplicaciones. • No es recomendable para modelar lógica de negocios.

Áreas de aplicación: Demostración de teoremas, creación de compiladores, resolver demostraciones por inducción, resolver problemas matemáticos complejos, centros de investigaciones y universidades.

Funciones

Una **función** es una regla de asociación entre un conjunto llamado **dominio** y uno llamado **imagen.**

$$f\colon A \to B$$

La definición de funciones es una **estructura jerárquica**, en la cual las funciones más simples aparecen en la definición de las más complejas. Además, al poder referenciar una función por su nombre, la misma puede ser empleada sin conocer su estructura interna.

Las funciones se denominan **expresiones** y se pueden representar con las siguientes sintaxis.

Expresiones atómicas: `<variable>, <constante`

Expresiones compuestas:

Abstracciones funcionales: `(<variable>, … <variable>) <expresión>`

Aplicaciones funcionales: `<expresión> (<expresión>, …, <expresión>)`

La **abstracción funcional** consiste en construir una expresión tal que pueda instanciarse con valores particulares.

Funciones de Orden Superior

Es una función donde uno de sus argumentos es una función que devuelve una función como resultado.

Lambda-Cálculo

Es un sistema formal diseñado para investigar la definición de función.

Consiste de una regla de transformación simple (substitución de variables) y un esquema simple para definir funciones.

El lambda-cálculo es universal porque cualquier función computable puede ser expresada y evaluada a través de él.

Los términos **lambda expresión** son una forma alternativa de escribir funciones donde

$$f(t) = t \Rightarrow x \cdot t$$

Evaluación perezosa

Los lenguajes funcionales utilizan la **evaluación perezosa** que consiste en no evaluar un argumento hasta que no se necesita.

Esto permite la posibilidad de manipular estructuras de datos (*potencialmente*) infinitos según las necesidades de evaluación.

Racket

Datos atómicos

Los **datos atómicos** son el bloque de construcción básico de cualquier lenguaje. Son los tipos de datos elementales como números y cadenas de textos.

Las palabras claves, variables y símbolos son llamados **identificadores.** *No son case sensitive.*

Los identificadores se evalúan con su valor asociado. Para evitar que se evalúe un identificador no vinculado hay que prefijar con un apóstrofe

Sintaxis

Las **estructuras** y **listas** se encierran entre paréntesis.
Los **booleanos** se definen como `#t` (verdadero) y `#f` (falso)
Los **vectores** se encierran con paréntesis y comienzan con un asterisco: `#(ejemplo)`
Los **strings** se encierran entre comillas: `"ejemplo"`
Los **caracteres** se prefijan por un asterisco y la barra: `#\e`

La **conversión** entre tipos se escribe como `tipo1->tipo2`

Se utiliza la **notación prefija:** `(<operador><operando>)`

Listas

Principal estructura no atómica.

```
(list 1 2 3)
```

Las listas pueden tener otras listas.

```
(list 1 (2 3) 4 5)
```

Hay dos tipos de listas:

- **Propias**: Una lista vacía y toda lista cuyo CDR sea una lista propia es una lista propia.
- **Impropias**: Compuestas por pares donde se marca la separación de elementos por puntos.

Una palabra clave que puede agregarle a una lista es Quote, que evita que los identificadores sean evaluados dentro de la lista. Así los identificadores son tratados como símbolos y no como variables.

```
(list 1 2 3 pi)
>(1 2 3 3.15179)

(quote (3 1 4 pi))
>(1 2 3 pi)
```

Operador de listas

CAR: Retorna el primer elemento de la lista.

```
> (car '( a b c)) => a
```

CDR: Retorna la cola de la lista.

```
> (cdr '( a b c)) => (b c)
```

CONS: Construye pares. Recibe dos argumentos.

```
> (cons '(a b) '( c )) => ((a b) c )
> (cons 'a (cons 'b (cons c '()))) => (a b c)
```

Una **lista propia** es una lista vacía y toda lista cuyo cdr sea una lista propia es una lista propia.

Las **listas impropias** están compuestas por pares donde se marca las separación de elementos por puntos.

Expresiones-Let

La expresión let no crea una variable, sino que introduce un nuevo *enlace* entre un nombre y un valor. El enlace es válido dentro del cuerpo de la expresión.

Características:

Inmutabilidad: El valor al que se enlaza un nombre no puede ser modificado. Si se intenta reasignar un valor se crea un nuevo enlace.

Ámbito Léxico: El nombre solo es visible y accesible dentro del bloque definido por let.

Beneficios:

Predecibilidad: El comportamiento se vuelve fácil de razonar, ya que los valores no cambian inesperada

Transparencia referencial: Las expresiones producen el mismo resultado siempre, lo que facilita la depuración y pruebas.

Paralelismo: Facilita la ejecución concurrente de programas.

```
> (let ([x 10])
>     (let ([x 20])
>          (+ x 5))
> x)
```

Let-bound

Se llama **let-bound** a las variables que han sido ligadas a un valor mediante una expresión let.

Definición variables y funciones

La forma principal de definir valores y funciones es mediante la palabra clave `define`.

Sintaxis variables: `(define <nombre> <valor>)`

```
> (define mensaje "hola, mundo")
```

Sintaxis funciones: `(define(<nombre> <parametro1> <parametro2> … ) <cuerpo>)`

```
> (define(sumar x y)
>     (+ y))
```

Puntos claves:

- **Ámbito de visibilidad**: las funciones realizadas con define a nivel superior son visibles en todo el programa, dentro de una función las definiciones son locales a esa función.
- **Evaluación:** Cuando se define un valor con define, el valor se evalúa antes de realizar la asociación.
- **Inmutabilidad:** Una vez que el nombre está asociado a un valor, esa asociación no cambia.

Funciones lambda

Las **funciones anónimas o lambda** son funciones que no están ligadas a un nombre, y se definen en el lugar donde se van a utilizar. En Racket, las funciones anónimas se crean con la palabra clave `lambda`.

```
> (lambda (x)(* x x))
> ((lambda (x)(* x x)) 5)     Aplica la función a 5.
```

Control de flujo

Se utilizan para ejecutar diferentes bloques dependiendo si una condición es verdadera o falsa.

Condicional if

Para condicionales simples con dos opciones.

Tiene la siguiente sintaxis:

```
(if <condición>
      <entonces-expresión>
      <si-no-expresión>)

      >(define (valor-absoluto x)
      >(if (< x 0)
            (-x); Si x es negativo, devuelve su negativo.
            x)); Si x es positivo, devuelve x.
```

Condicional cond

Proporciona una forma más general de expresar condicionales múltiples.

Tiene la siguiente sintaxis:

```
(cond
      [<condición-1> <expresión-1>]
      [<condición-1> <expresión-1>]
      …
      [else (expresión-por-defecto)])
```

```
(define (signo x)
(cond
      [(< x 0) -1]
      [(= x 0) 0]
      [(> x 0) 1])))
```

Operadores lógicos

Se utilizan para realizar operaciones booleanas.

Not

Toma un valor como argumento y devuelve su negación lógica.
Cualquier valor que no sea #F, se considera #V

```
(not #t) ; -> #f
(not "false") ; -> #f
(not #t) ; -> #t
```

Or

Toma uno o más valores como argumentos y devuelve verdadero si *al menos* uno es verdadero

```
(or) -> #f
(or #t #f) -> #t
(or #f #f) -> #f
```

And

Toma uno o más valores como argumentos y devuelve falso si *al menos uno* es falso.

```
(and) -> #t
(and #t #f) -> #f
(and #t #t) -> #t
(and #f #t) -> #f
```

Predicados

Son funciones que toman uno o más valores como entrada y devuelve un valor de verdad, indicando si cierta propiedad se cumple.
Se utilizan para tomar decisiones y controlar el flujo en condicionales, filtrado y funciones de ordenamiento.
Los predicados u operadores aritméticos de comparación son los siguientes.

- = : Son los números iguales?
- < : ¿Es el primer número menor que el segundo?
- > : ¿Es el primer número mayor que el segundo?
- =< : ¿Es el primer número menor o igual que el segundo?
- => : ¿Es el primer número mayor o igual que el segundo?

Algunos predicados terminan en ?. Es una convención de estilo, no una regla.

- `null?`: ¿Es una lista vacía?
- `qev?` : ¿Son dos valores idénticos?
- `pair?` : ¿Es un par? (elemento construido por cons)
- `symbol?` : ¿Es un símbolo?
- `number?` : ¿Es un número?
- `string?` : ¿Es una cadena de texto?

Recursión

Mecanismo principal para realizar iteraciones. La función se define en términos de sí misma y se llama a sí misma. Tiene dos "partes":

Caso base: Condición que detiene la recursión. Define el resultado para la entrada más simple posible.

Caso recursivo: Condición en la que la función se llama a sí misma con una entrada más pequeña o simplificada.

```
(define (sumaLista ls)
(if (null? ls); CASO BASE: la lista está vacía
     0
     (+ (car ls) ; CASO RECURSIVO: suma el primer elemento
     (sumaLista (cdr ls))))) ;y suma el resto de la lista
```

Letrec

Letrec se utiliza para definir funciones **recursivas locales**. Letrec permite que las variables definidas dentro de él se refieran entre sí.

```
(define (miFuncion ls)
     (letrec ([es-par? (lambda(n);
          (if (= n 0)
               #t
               (es-impar?(sub1 n)))))]
     (+ (car ls) ; CASO RECURSIVO: suma el primer elemento
     (sumaLista (cdr ls))))) ;y suma el resto de la lista
```

Como letrec tiene ámbito local, es útil para encapular funciones auxiliares.

Ventajas

- Código más conciso y elegante.
- Facilita razonamiento.

Vectores

Un **vector** es una secuencia de objetos separados por un blanco y precedidos por un # o la palabra clave `vector`.

```
> (vector 'a 'b 'c) -> #(a b c)
```

Funciones de Orden Superior

Map

Toma como primer argumento una función (la función de **transformación**) que se aplicará a cada elemento de una colección.

```
> aplicar procedimiento a una lista
```

Paradigma lógico

La **programación lógica** es un paradigma de programación **declarativo**. En lugar de decir a la computadora *cómo* resolver un problema, decimos *cuál* es el problema: se lo describe en forma de relaciones y reglas lógicas.

- **Cuerpo de conocimiento**: consiste en una colección de hechos y reglas.
- **Lógica formal:** Utiliza un lenguaje formal.
- **Motor de inferencia:** La "ejecución" de un programa consiste en probar si una afirmación es verdadera.
- **Intérprete de comandos:** Permitir la posibilidad de responder consultas.

Sintaxis

Símbolos lógicos

Los **conectores lógicos** combinan elementos para crear nuevos elementos más complejos.

¬	Negación
∧	Conjunción
∨	Disyunción
→	Implicación
↔	Equivalencia

Los **cuantificadores** expresan la validez de una sentencia.

∀	Universal	"Para todo"
∃	Existential	"Para algún"

Fórmulas

Una **fórmula** tiene la forma

$$p(t_0, t_1, \dots, t_n)$$

Donde p es un símbolo de predicado de aridad n y $t_0, t_1, \dots, t_n$ son términos.

Si F y G son fórmulas y X una variable, también son fórmulas:

- $(\sim F)$
- $(F \wedge G)$
- $(F \vee G)$
- $(F \rightarrow G)$
- $(F \leftrightarrow G)$
- $(\forall x\, F)$
- $(\exists x\, F)$

Semántica

Marco para entender el significado y el valor de verdad en la lógica formal.
Componentes:

- Interpretación: Una función asigna elementos no lógicos (constantes variables, predicados, funtores) y lógicos (conectores y cuantificadores) del lenguaje a un dominio (conjunto no vacío de objetos).
- Valor de verdad: En fórmulas atómicas está determinado si la relación se cumple en los objetos, en fórmulas compuestas se define por tablas de verdad estándar.
- Modelo: Una interpretación I es un modelo de una fórmula F *si y sólo si* F es verdadera en I.

Prolog

Fuentes adicionales: Basic Concepts, Guide to Prolog Programming y Learn Prolog NOW!
Programar en prolog consiste en:

1. Declarar **hechos**
2. Definir **reglas**
3. Hacer **preguntas**

Hechos y reglas

Los hechos y reglas, en conjunto, se conocen como **cuerpo de conocimiento**.
Las **reglas** se utilizan para expresar que un hecho depende de uno o más hechos. Representa la implicación lógica.

```
head :- body.
```

Si el cuerpo de la regla es verdad, entonces lo será la cabeza.
El cuerpo puede estar compuesto de más de un ítem o *goal* y para que sea verdadero todos los ítems dentro del cuerpo tienen que serlo.

```
head :-
     hecho1,
     hecho2.
```

Además puede expresarse dos reglas con la misma cabeza y diferentes cuerpos, entendiéndose que que la cabeza puede ser verdadera si es verdadero cualquiera o ambos cuerpos.

```
head :-
     hecho1,
     hecho2.
head :-
     hecho3.
```

Un **hecho** es una regla que *siempre* es verdad, simbólicamente:

```
head :- true.
```

Los hechos representan propiedades y relaciones. Las relaciones se llaman también **predicados**.

```
nombre_propiedad(objeto).
nombre_relacion(objeto1, objeto2, … ).
```

Cada predicado tiene una **aridad** (número de argumentos) y al instanciarse evalúa un valor de verdad. Un predicando con el nombre `nombre` y `n` argumentos se denomina `nombre/n`
El nombre de las reglas y hechos van con minúscula.
Ej.

```
estudia(ana)

aprueba(ana):-
      estudia(ana).

aprueba(ana):-
      tienemuchasuerte(ana).
```

```
?- aprueba(ana).
true //ojalá…
```

Variables

En las consultas, las **variables** permiten obtener *todos* los objetos que cumplen una propiedad o relación.

Prolog unifica desde arriba a abajo, para obtener todos las expresiones que puede instanciar se utiliza el punto y coma desde la terminal. Además se puede preguntar por objetos que cumplen más de una propiedad o relación usando la coma. Van con mayúscula.

Ej.

```
estudia(ana).
estudia(mateo).
estudia(abril).

mujer(abril).

amigos(ana, mateo).
amigos(ana, abril).
```

```
?- estudia(X).
X = ana ;
X = mateo ;
X = abril.

?- amigos(ana, X).
X = mateo ;
X = abril.

?- amigos(Y, X).
Y = ana,
X = mateo ;
Y = ana,
X = abril.

?- amigos(ana, X), mujer(X).
X = abril.
```

Las variables también se usan en el cuerpo de conocimiento para describir reglas generales.

Ej.

```
amigos(ana, mateo).
amigos(ana, abril).

amigos_en_comun(X,Y):-
      amigos(Z,X).
      amigos(Z,Y).
```

```
?- amigos_en_comun(abril,
mateo).
true
```

Recursividad

Los predicados pueden ser definidos mediante la **recursividad**. Un predicado es recursivo si una o más de sus reglas en su definición *"se llama a sí mismo"*.
En una definición recursiva se consideran siempre dos casos.

Caso básico: Momento en el que se detiene el proceso.
Caso recursivo: Reducción del problema. Dónde está la llamada a sí mismo.

Listas

Una **lista** es una secuencia finita de elementos. En Prolog se encierra los elementos entre corchetes y separados por coma. Internamente, se representa con con un árbol binario.

```
[ítem_0, ítem_1, ítem_2, … , ítem_n]
```

La **cabeza** y la **cola** se pueden separar mediante el símbolo `|`.

```
[a,b,c]
[a|[b,c]]
[[a,b]|c]
```

Trabajar con lista suele implicar trabajar con recursividad. Algunas de las operaciones con listas más comunes incluyen:

Buscar pertenencia.

```
miembro(X,[X|_]).
miembro(X,[_|Cola]) :-
     miembro(X,Cola).
```

```
caso base.
     La cabeza es igual a X
     -> true
caso recursivo.
     X no es la cabeza.
     Quitar la cabeza.
     Seguir buscando.
```

```
?- miembro(4, [1,2,3]).
false

?- miembro(4, [1,2,4,3]).
true
```

Concatenar dos listas.

```
concatenar([], L, L).
concatenar([X|L1], L2, [X|L3])
:-
     concatenar(L1, L2, L3).
```

```
?-concatenar([1,2,3],[4,5,6],L
)
```

caso base:
Concatenar la lista vacía con cualquier otra lista L, da la lista L.

caso recursivo:

Consultas

Las **consultas** determinan si una pregunta es consecuencia lógica del programa: se considera todo lo que se puede probar por la base de conocimiento como `true` y todo lo que no, `false`.
Una llamada concreta a un predicado se denomina **objetivo** *(goal)*. Todos los objetivos tienen un resultado de éxito o fallo tras su ejecución.

Características

- Los objetivos se ejecutan secuencialmente.
- Si un objetivo falla, los siguientes objetivos no se ejecutan y la conjunción falla.
- Si un objetivo tiene éxito, las variables ligadas dejan de estar libres para el resto de la secuencia.
- Si todos los objetivos tienen éxito, la conjugación tiene éxito.

Prolog utiliza el **principio de resolución**, una regla de inferencia que permite la demostración automática de un teorema, y las **cláusulas de Horn**, disyunciones (OR) de literales donde hay *como máximo* un literal positivo, durante la evaluación.

Unificación

La **unificación** es el mecanismo en el cual las variables toman valor en Prolog.
Se dice que dos términos unifican cuando

1. Son iguales.
2. Las variables pueden ser instaladas de forma tal de que los términos sean iguales.

Para que dos términos unifiquen *deben* tener el mismo functor y ariedad.
Cuando un objetivo tiene éxito las variables libres que aparecen en los argumentos quedan ligadas y son los valores que hacen cierto al predicado. Si el objetivo falla, no ocurren ligaduras.

Backtracking

El **backtracking** es una técnica que consiste en recorrer sistemáticamente todos los caminos posibles: si un camino no conduce a la solución, se retrocede al paso anterior para buscar un nuevo camino.

La búsqueda con el backtracking es el siguiente:

1. Al ejecutar un objetivo, Prolog sabe de antemano cuantas alternativas o **puntos de elección** puede tener.
2. Se anotan todos los puntos de elección en una pila.
3. Se escoge el primer punto de elección, se ejecuta el objetivo y se elimina de la pila.
   3.1. Si el objetivo tiene éxito -> Se continúa al siguiente objetivo.
   3.2. Si el objetivo falla -> Prolog da “marcha atrás” buscando los objetivos que anteriormente tuvieron éxito (Comienza el backtracking).
4. El proceso se repite mientras haya objetivos y puntos de elección anotados.

Las etapas por la que pasa el algoritmo pueden representarse mediante un árbol de expansión en el que cada nivel representa una etapa.

Ej.

```
alumno (jose).
alumno (ana).
alumno (juan).
aprobo_primer_parcial(jose).
aprobo_primer_parcial(juan).
aprobo_primer_parcial(ana).
aprobo_segundo_parcial(jose).
aprobo_segundo_parcial(ana).

alumno_regular(X) :-
      alumno(X),
      aprobo_primer_parcial(X),
      aprobo_segundo_parcial(X).
```

```
// Esto NO aparece en la terminal. Es el funcionamiento interno del
backtracking.

alumno_regular(X)
      X = jose
      alumno_regular(jose)
            alumno(jose) -> true
            aprobo_primer_parcial(jose) -> true
            aprobo_segundo_parcial(jose) -> true
      alumno_regular(jose) -> true

      X = ana
      alumno_regular(ana)
            alumno(ana) -> true
            aprobo_primer_parcial(ana) -> true
            aprobo_segundo_parcial(ana) -> true
      alumno_regular(ana) -> true

      X = juan
      alumno_regular(juan)
            alumno(juan) -> true
            aprobo_primer_parcial(juan) -> true
            aprobo_segundo_parcial(juan) -> false
      alumno_regular(juan) -> false
```

El backtracking se puede controlar usando dos predicados: el predicado de fallo y el predicado de corte.

El **predicado de corte** es un predicado predefinido que no recibe argumentos. Se representa con un signo de admiración (`!`). Siempre que se cumple en una ejecución, falla en el proceso de backtracking, dejando al objetivo con una única *aparente* solución.

El **predicado de fallo** es otro predicado predefinido sin argumentos. Siempre falla forzando el backtracking para generar más soluciones.

Sintaxis

Llamamos **términos** a los objetos del dominio, los bloques de construcción básicos. *Toda la data* está representada por términos.
Estos **elementos del lenguaje** son la estructura base de Prolog, los hechos, reglas y preguntas son los tipos de sentencias que se pueden formar con estos elementos.
Clasificamos los objetos entre **simples** y **estructuras**.

Objetos simples

Constantes: Representan objetos o relaciones específicas e inmutables. Hay dos tipos.

> **Átomos:** Nombres simbólicos.
> Pueden estar formados por letras y dígitos o por signos (`maria, juan`). También son átomos los símbolos especiales que Prolog usa para detonar preguntas y reglas (‘`?-`’, ‘`:-`’)
> **Números:** Valores numéricos.

Variables: Representan objetos no especificados o que puedan variar.

> **Normales:** Variables con nombre. Son objetos que no podemos/queremos nombrar. Comienzan con mayúscula o guión bajo. (`X, Y, Una_variable_muy_larga`)
> **Anónimas:** Variable cuyo valor no importa. Se utilizan para saber si *algún objeto* (cualquiera) cumple el predicado. Único guión bajo.

Objetos estructuras

Una **estructura** es un objeto que consta de una colección de otros objetos llamados **componentes**. Está formada por un **functor** (el nombre de la estructura) y una secuencia de argumentos. Los functores *deben* ser átomos, pero los argumentos pueden ser de cualquier tipo.

```
functor(arg1, arg2, …, argN)
```

Una forma de interpretar a las estructuras es mediante **árboles**, en los cuales cada functor es un nodo y los componentes son ramas.

Declarativo y procedural

Prolog se basa en la lógica de primer orden que es **declarativa** pero su estrategia de ejecución (depth-first search) es **procedural**, lo que lo vuelve dependiente del orden.