

Introducción a los paradigmas de programación

Esp. Ing. Viviana A. Santucci
AlA Federico Castoldi
Ing. V. Franco Matzkin
Ing. Cecilia Serafini

Tecnologías de la Programación

1 - Introducción y conceptos generales

“Considero a los paradigmas como realizaciones científicas universalmente reconocidas que, durante cierto tiempo, proporcionan modelos de problemas y soluciones a una comunidad científica”. Kuhn, T. S. (2019). La estructura de las revoluciones científicas. Fondo de cultura económica.

Un resumen del concepto de paradigma según Kuhn es el siguiente:

- Lo que se debe observar y escrutar.
- El tipo de interrogantes que se supone que hay formular para hallar respuestas en relación al objetivo.
- Cómo tales interrogantes deben estructurarse.
- Cómo deben interpretarse los resultados de la investigación científica.

1 - Introducción y conceptos generales

De manera más simple, se puede definir al paradigma como:

- Un modelo o ejemplo a seguir en una comunidad científica, de los problemas que tiene que resolver y del modo como se van a dar las soluciones,
- Determina una manera especial de entender el mundo, explicarlo y manipularlo.

1 - Introducción y conceptos generales

Paradigma de programación:

Un enfoque particular o filosofía para la construcción de software.

- No existe un paradigma uno mejor que otro sino que cada uno tiene ventajas y desventajas.
- Existen situaciones donde un paradigma resulta más apropiado que otro.
- Está constituido por los supuestos teóricos generales, las leyes y las técnicas para su aplicación.

1.1 – Programas

Un programa es un conjunto de códigos, instrucciones, declaraciones, proposiciones, etc. que describen, definen o caracterizan la realización de una acción en la computadora.

Hoy los programas se diseñan según un paradigma de programación y se escriben usando algún lenguaje de programación asociado.

1.2 – Paradigmas

Un paradigma es una colección de modelos conceptuales que juntos modelan el proceso de diseño, orientan la forma de definir los problemas y, por lo tanto, determinan la estructura final de un programa.

Los paradigmas son **la forma de pensar y entender un problema y su solución**, y por lo tanto, de enfocar la tarea de la programación.

1.2 – Paradigmas

Para resolver un determinado problema, deberíamos conocer cuál paradigma se adapta mejor a su resolución, y a continuación elegir el lenguaje de programación apropiado.

En teoría cualquier problema podría ser resuelto por cualquier lenguaje de cualquier paradigma. Sin embargo, algunos paradigmas ofrecen mejor soporte para determinados problemas que otros.

1.2 – Paradigmas

Los paradigmas de programación ayudan a construir códigos más legibles y organizados. Además, ofrecen las técnicas más adecuadas para cada tipo de aplicación, aumentando la productividad diaria del desarrollador. Ser capaz de entender idiomas de manera más amplia e incluso entender entre líneas de códigos.

Eligir un paradigma adecuado para un proyecto, posibilita que las aplicaciones sean desarrolladas con mayor productividad, posibilitando la singularidad en la orientación de la escritura del código entre el equipo, haciéndolo más legible y facilitando el mantenimiento a lo largo de su existencia.

1.2 – Paradigmas

La comprensión de los paradigmas de programación hace que los proyectos sean más profesionales y organizados.

Antes de reflexionar sobre la solución de un problema, se pensará en la modelización de esa solución y en el paradigma a utilizar.

1.4 – Programación

Es el proceso de diseñar, codificar, depurar y mantener el código fuente de programas computacionales. El código fuente es escrito en un lenguaje de programación. El propósito de la programación es crear programas que exhiban un comportamiento deseado. El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas.

La existencia de diferentes paradigmas de programación, implica que también haya diversos conceptos de “programa”. Por lo tanto se los puede comparar para descubrir su especificidad, su dominio de aplicación, sus ventajas y limitaciones, tanto para poder elegir la mejor solución como para combinarlos.

1.4.1 – Criterios de la buena programación

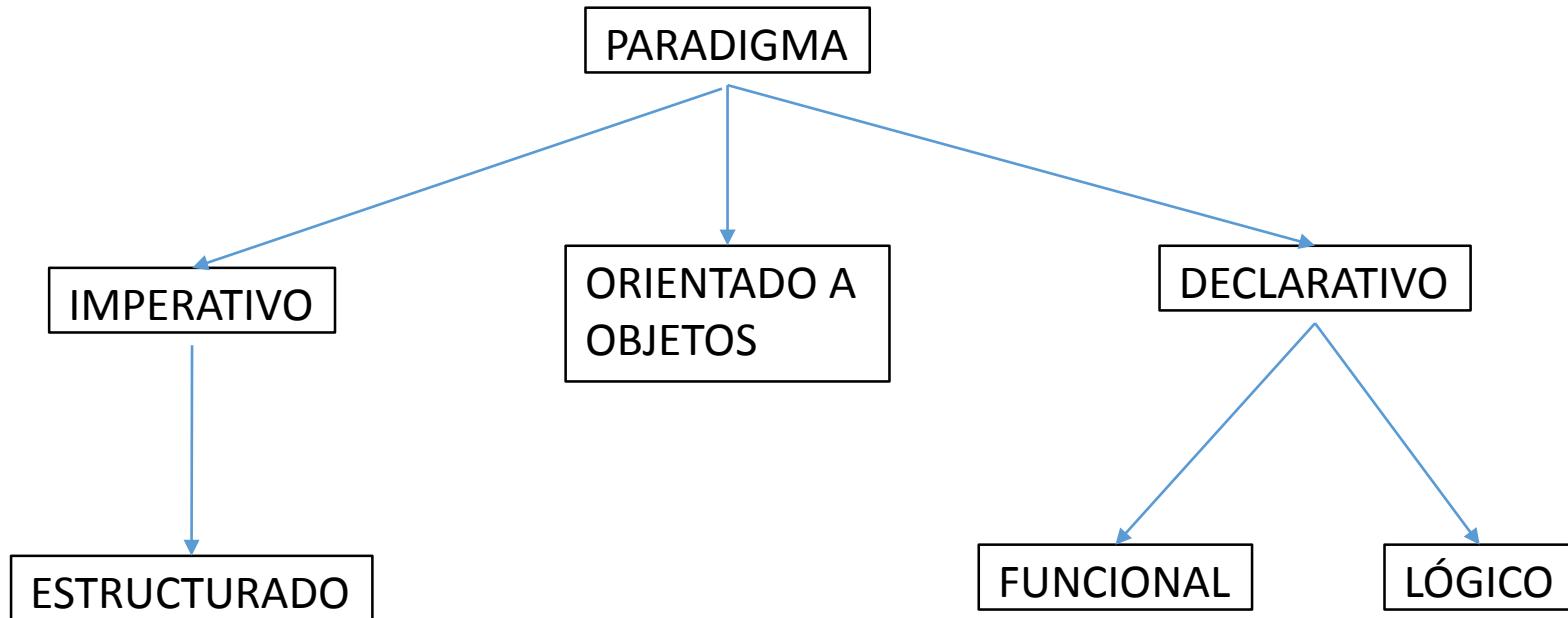
- Plantear **modelos cercanos a la realidad**
- Diseñar implementaciones de manera que puedan ser **extendidas y modificadas**
- Dar flexibilidad a las soluciones para que puedan ser **reutilizadas**
- Diseñar una **articulación funcional** adecuada
- Desarrollar un **código claro**, simple y compacto
- Construir **soluciones genéricas** que permitan abstraerse de las particularidades propias de cada tipo de entidad de software
- **Focalización** de las funcionalidades y componentes del sistema para poder trabajar sobre eficiencia.

2 – Paradigmas fundamentales

Los principales paradigmas que tienen vigencia, tanto por su desarrollo conceptual y su importancia en las ciencias de la computación, como por su presencia significativa en el mercado, son:

- Paradigma Lógico
- Paradigma Funcional
- Paradigma Imperativo o procedural
- Paradigma de Objetos

2 – Paradigmas fundamentales



2.1 – Clasificación y evolución histórica

Partiendo de los principios fundamentales de cada paradigma en cuanto a las orientaciones sobre la forma para construir las soluciones y teniendo en cuenta su evolución histórica, se pueden distinguir mayores o menores similitudes entre los paradigmas que permiten organizarlos esquemáticamente en subgrupos y relacionarlos entre sí.

Los primeros lenguajes de programación, conocidos como **imperativos o procedurales**, establecieron las bases para los paradigmas de programación modernos. Estos lenguajes se centran en procedimientos que ejecutan operaciones para modificar datos en memoria, siguiendo algoritmos detallados con estructuras de control. Permiten la organización del código en módulos y el uso de tipos de datos simples y estructuras de datos. A pesar de la evolución y complejización de los lenguajes, los principios del paradigma imperativo siguen influyendo en la programación actual debido a su **simplicidad, versatilidad y robustez**, manteniendo su relevancia en las ciencias de la computación.

2.1 – Clasificación y evolución histórica

Los paradigmas declarativos, como los lógicos y funcionales, surgieron como alternativas a la programación imperativa, **enfocándose en el "qué" de la solución** en lugar del "cómo". Estos paradigmas utilizan declaraciones para describir problemas y soluciones, dejando los detalles de implementación a mecanismos internos. Aunque la declaratividad es una característica distintiva, no es exclusiva de estos paradigmas y puede manifestarse de diversas formas en otros enfoques de programación.

Los lenguajes de programación orientados a objetos evolucionaron del paradigma imperativo, integrando elementos que permiten una programación más cercana al mundo real. Estos lenguajes **se centran en "objetos" que interactúan mediante mensajes, encapsulando datos y comportamientos**, y utilizando conceptos como herencia y polimorfismo para crear sistemas modulares y extensibles. Esta aproximación permite representar entidades del mundo real y sus interacciones de manera más natural y organizada.

2.2 – Paradigma imperativo

Describe paso a paso un conjunto de instrucciones que deben ejecutarse para cambiar el estado del programa y hallar la solución, es decir, un algoritmo en el que **se describen los pasos necesarios para solucionar el problema**.

Dentro de este paradigma surgieron diferentes etapas, primero surgió el paradigma lineal donde los programas eran un solo bloque, con instrucciones para:

- Ejecutar un conjunto de instrucciones, encabezadas por una etiqueta (label)
- Ejecutar un rango de instrucciones definido por etiquetas desde/hasta.
- Derivar el flujo de control a determinada etiqueta. (Go to, go to .. depending).
- Realizar el uso intensivo de banderas para la detección de eventos anteriores.
- Entre otros.

Luego apareció el paradigma estructurado

2.2.1 – Paradigma estructurado

ALGORITMO + ESTRUCTURA DE DATOS = PROGRAMA

La programación imperativa se define por el estado del programa y las instrucciones que lo modifican. Los computadores, diseñados según el paradigma de las Máquinas de Turing, ejecutan código de máquina imperativo. Los lenguajes imperativos de alto nivel, aunque más complejos, siguen este enfoque, similar al de recetas o listas de revisión, donde cada paso altera el estado del sistema.

Lenguajes asociados: Fortran, C, Pascal

2.2.1 – Paradigma estructurado

Ventajas:

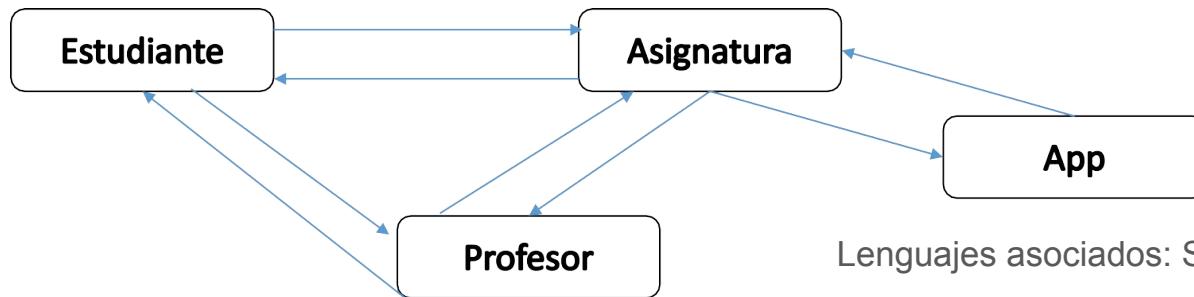
- El conjunto de instrucciones del programa es más cercano al conjunto de instrucciones de código de máquina, por consiguiente el código es más directo y es más rápida su ejecución.
- Si el programa está bien modularizado, es más fácil de corregir los errores de ejecución y su mantenimiento.
- La estructura del programa puede ser clara si:
 - Se hace hincapié en la modularización del programa, existen niveles, puede distribuirse la codificación siguiendo un diseño Top Down. (La codificación distribuida en módulos va de un nivel general al de máximo detalle).
 - La cantidad de niveles que usamos para modularizar es la adecuada (No está limitada por los lenguajes).
 - Los nombres de los módulos (Procedimientos, funciones), definidos por el programador, son representativos de lo que el módulo realiza.
 - Hay suficiente documentación adjunta al código.

2.3 – Paradigma orientado a objetos

OBJETOS + MENSAJES = PROGRAMA

Este paradigma está basado en la idea de encapsular estado y operaciones en objetos. Cada objeto no está aislado de los demás. Los objetos colaboran entre sí para lograr un objetivo en común.

Esta colaboración se realiza a través de mensajes. Objeto es toda entidad que posee un estado y que puede realizar actividades. Mensaje es el medio por el cual se comunican los objetos. Su principal ventaja es ser un paradigma muy adecuado al diseño de aplicaciones relacionadas con todo lo que sea gestión.



Lenguajes asociados: Smalltalk, Java, C++, Python

2.3 – Paradigma orientado a objetos

Ventajas:

- Reusabilidad: en este contexto al realizar un diseño adecuado de la solución (clases), es factible
- utilizar diferentes partes de la solución del programa en otros proyectos.
- Mantenibilidad: esto se logra gracias a la facilidad de abstracción de los problemas en este paradigma, es por esto que los programas orientados a objetos son más sencillos de leer y comprender. Ya que permite ocultar detalles de implementación dejando visibles solo aquellos aspectos más relevantes.
- Fiabilidad: en este paradigma al tener la posibilidad de dividir en problema en partes más pequeñas (clases) es posible testearlas de manera independiente y aislar de manera más simple los posibles errores que puedan surgir.

2.3 – Paradigma orientado a objetos

Limitaciones:

- La implementación de algunos de los mecanismos soportados por el paradigma como la herencia, hace que los programas sean más extensos lo que provoca como consecuencia que su ejecución sea más lenta en algunos casos.

Ejemplo: en un sistema con demasiados niveles de herencia, como un modelo de ERP (Enterprise Resource Planning), puede ser complicado entender las relaciones entre clases, lo que dificulta la mantenibilidad.

- La curva de aprendizaje cuando se conoce otras formas de programación suele ser más lenta.

Ejemplo: Alguien acostumbrado a lenguajes imperativos puede tener dificultades para entender conceptos como polimorfismo o encapsulación en Java o Python.

2.3 – Paradigma orientado a objetos

Aplicaciones:

- Ampliamente usado en aplicaciones de negocios y tecnología Web
- Modelado y simulación
- Bases de datos orientadas a objetos
- Programación en paralelo
- Sistemas grandes que requieren la administración de estructuras complejas

2.4 – Paradigma declarativo

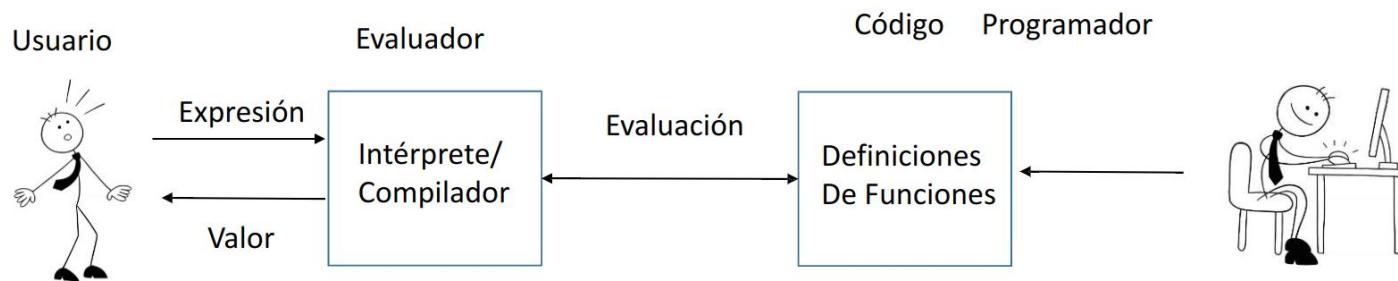
Está basado en el desarrollo de programas especificando o "declarando" un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución.

La solución se obtiene mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan sólo se le indica a la computadora que es lo que se desea obtener o que es lo que se está buscando).

2.4.1 – Paradigma funcional

FUNCIONES + CONTROL = PROGRAMA

El paradigma funcional ve la computación como la evaluación de funciones matemáticas y se enfoca en la aplicación y composición de funciones, en lugar de en la mutación de estados y la ejecución secuencial de instrucciones. Esto permite una solución más elegante de problemas y evita los efectos secundarios típicos de otros paradigmas de programación.



Lenguajes asociados: Lisp, Haskell, Scheme (dialecto de Lisp), Scala

2.4.1 – Paradigma funcional

Ventajas:

- Altos niveles de abstracción: ya que en la codificación del programa se hace mayor énfasis en el “¿qué hace?” en lugar del “¿cómo se hace?”.
- Fácil de formular matemáticamente: debido a los altos niveles de abstracción los programas suelen ser más cortos y sencillos de entender.
- Rapidez en la codificación de los programas: esto es posible gracias a la “expresividad”. La programación funcional simplifica el código de manera significativa. Por ejemplo mediante el uso de funciones de orden superior entre otros recursos soportado por este paradigma.
- Administración automática de memoria: el programador no es responsable de reservar la memoria utilizada por cada objeto y liberarla; la implementación del lenguaje la realiza en forma automática como la gran mayoría de los lenguajes vigentes en la actualidad.
- La evaluación perezosa: esta estrategia de evaluación permite realizar cálculos por demanda, evitando un gasto computacional innecesario.

2.4.1 – Paradigma funcional

Limitaciones

- Son limitados en cuanto portabilidad, riqueza de librerías, interfaces con otros lenguajes y herramientas de depuración.

Ejemplo: Lenguajes como Haskell tienen menor compatibilidad con sistemas existentes, dificultando la integración con bibliotecas de otros paradigmas.

- El modelo funcional al estar alejado del modelo de la máquina de von Newmann, la eficiencia de ejecución de los intérpretes de los lenguajes funcionales no es comparable con la ejecución de los programas en otros paradigmas como por ejemplo el imperativo.

Ejemplo: Un programa funcional que utiliza "evaluación perezosa" puede consumir más memoria al mantener múltiples cálculos pendientes, lo que impacta el rendimiento en sistemas embebidos.

2.4.1 – Paradigma funcional

Aplicaciones:

- Programas de Inteligencia Artificial (IA) y aprendizaje automático:

Por qué destaca: Su énfasis en las funciones puras y la manipulación de datos lo hace adecuado para escribir algoritmos complejos, como redes neuronales y árboles de decisión.

Ejemplo: Haskell y Lisp se utilizan para implementar sistemas de procesamiento de lenguaje natural y motores de razonamiento lógico.

2.4.1 – Paradigma funcional

Aplicaciones:

- Sistemas distribuidos de control (NCS) tolerante a fallos de software (por ejemplo, control de tráfico aéreo, mensajería instantánea, servicios basados en Web)

Por qué destaca: La inmutabilidad y la ausencia de estado compartido facilitan el paralelismo y la ejecución distribuida.

Ejemplo: Erlang (influido por la programación funcional) es utilizado en sistemas como WhatsApp para gestionar millones de conexiones simultáneamente, garantizando alta disponibilidad y tolerancia a fallos.

2.4.1 – Paradigma funcional

Aplicaciones:

- Aplicaciones matemáticas y científicas.

Por qué destaca: Su capacidad para expresar cálculos matemáticos de manera declarativa y clara.

Ejemplo: Haskell se utiliza para resolver ecuaciones diferenciales y modelos matemáticos en simulaciones científicas.

2.4.1 – Paradigma funcional

Aplicaciones:

- Procesamiento de Datos y Big Data

Por qué destaca: La programación funcional permite escribir algoritmos eficientes para manipular grandes cantidades de datos gracias a herramientas como funciones de orden superior y evaluación perezosa.

Ejemplo: Spark, una herramienta de procesamiento masivo de datos, utiliza Scala, que combina funcionalidad y paradigmas funcionales.

2.4.1 – Paradigma funcional

Aplicaciones:

- Telecomunicaciones

Por qué destaca: Su capacidad para manejar eventos concurrentes de manera eficiente.

Ejemplo: Erlang es ampliamente usado en sistemas de telecomunicaciones para garantizar el procesamiento en tiempo real sin errores.

2.4.1 – Paradigma funcional

Aplicaciones:

- Interfaces de Usuario Reactivas

Por qué destaca: Conceptos funcionales como programación reactiva son ideales para sistemas interactivos.

Ejemplo: React.js, una biblioteca de JavaScript basada en componentes funcionales, es muy utilizada en el desarrollo de interfaces web dinámicas.

2.4.1 – Paradigma funcional

Aplicaciones:

- Interfaces de Usuario Reactivas

Por qué destaca: Conceptos funcionales como programación reactiva son ideales para sistemas interactivos.

Ejemplo: React.js, una biblioteca de JavaScript basada en componentes funcionales, es muy utilizada en el desarrollo de interfaces web dinámicas.

2.4.1 – Paradigma funcional

Aplicaciones:

- Educación y Proyectos Académicos

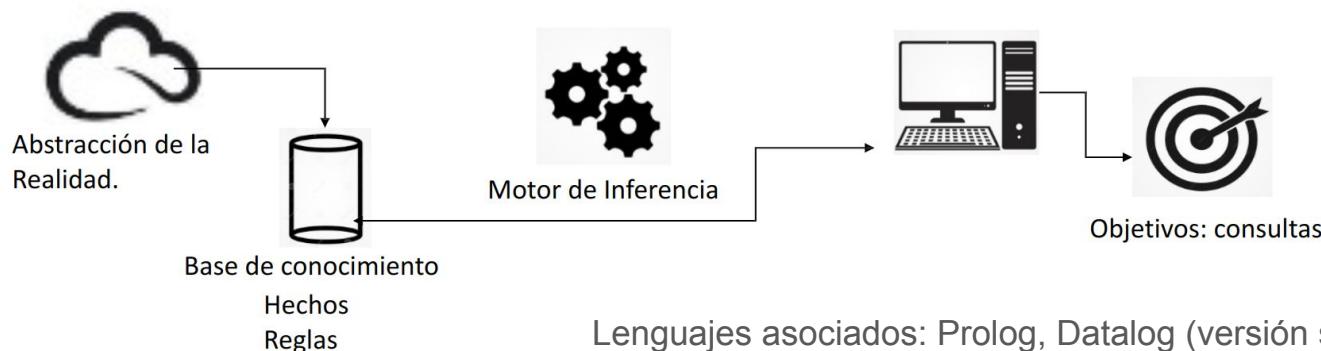
Por qué destaca: Su enfoque en la claridad y la lógica lo convierte en una herramienta poderosa para enseñar fundamentos de programación y matemáticas.

Ejemplo: Haskell es ampliamente utilizado en universidades para enseñar paradigmas de programación y algoritmos.

2.4.2 – Paradigma lógico

LÓGICA + CONTROL = PROGRAMA

Basado en la definición de reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas



2.4.2 – Paradigma lógico

Ventajas:

- Simplicidad.
- Cercanía a las especificaciones del problema realizada con lenguajes formales.
- Sencillez y potencia en la búsqueda de soluciones.
- Sencillez en la implementación de estructuras complejas.

Limitaciones:

- Poco utilizado en aplicaciones de gestión.
- Existen pocas herramientas de depuración efectivas.
- Los programas en este paradigma se consideran pocos eficientes por la lentitud de ejecución de los intérpretes, como ocurre de forma similar con el paradigma funcional

2.4.2 – Paradigma lógico

Aplicaciones:

- Sistemas expertos.
- Sistemas de soporte a decisiones (DSS).
- Definiciones de reglas de negocio (Drools).
- Sistemas de conocimiento y aprendizaje (Bases de datos de conocimiento).
- Creación de lenguajes de consulta para análisis semántico (por ejemplo, Web semántica).
- Procesamiento de lenguaje natural.
- Robótica.
- Compilación de lenguajes funcionales.
- Especificar semántica a lenguajes imperativos.
- Formalismo para definir otras teorías

3 – Mismo problema y diferentes paradigmas.

PROBLEMA

calcular el promedio de una lista de números.

3 – Mismo problema y diferentes paradigmas.

Paradigma imperativo se basa en dar instrucciones paso a paso para modificar el estado del programa. Implementación en Python:

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

for numero in numeros:
    suma += numero

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

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

3 – Mismo problema y diferentes paradigmas.

Paradigma orientado a objetos, el enfoque está en modelar entidades (objetos) y sus interacciones.

```
# Programa orientado a objetos para calcular el promedio de una lista de números

class CalculadoraPromedio:
    def __init__(self, numeros):
        self.numeros = numeros

    def calcular_promedio(self):
        suma = sum(self.numeros) # Se usa sum para mayor simplicidad
        return suma / len(self.numeros)

# Lista de números
numeros = [10, 20, 30, 40, 50]

# Crear instancia de la clase y calcular el promedio
calculadora = CalculadoraPromedio(numeros)
print("Promedio:", calculadora.calcular_promedio())
```

3 – Mismo problema y diferentes paradigmas.

El paradigma funcional se enfoca en el "qué" (el resultado deseado) en lugar del "cómo" (los pasos para llegar allí). Ejemplo en Racket:

```
#lang racket

(define (promedio numeros)
  (define suma (apply + numeros)) ; Suma todos los elementos de la lista
  (/ suma (length numeros))) ; Divide la suma por la cantidad de elementos

; Ejemplo de uso
(define numeros '(10 20 30 40 50)) ; Lista de números
(displayln (promedio numeros)) ; Imprime el promedio
```

3 – Mismo problema y diferentes paradigmas.

- Uso de funciones puras:

La función promedio no depende del estado externo ni modifica variables globales.

apply se usa para aplicar la función + a todos los elementos de la lista, calculando su suma.

- Evaluación declarativa:

Se describe directamente lo que se quiere calcular (la suma de los números dividida por su longitud) en lugar de dar instrucciones paso a paso.

- Expresividad:

La solución es compacta y clara, aprovechando funciones de orden superior como apply.

3 – Mismo problema y diferentes paradigmas.

El paradigma lógico se basa en reglas y hechos para describir el problema. Un ejemplo simplificado en Prolog::

```
% 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.
```

Define las relaciones lógicas entre los elementos (suma, length) en lugar de dar instrucciones explícitas.

3 – Mismo problema y diferentes paradigmas.

- H (Head): Representa la cabeza de la lista, es decir, el primer elemento. Por ejemplo, si tienes una lista [10, 20, 30], el H en esta lista sería 10.
- T (Tail): Representa la cola de la lista, es decir, todos los elementos restantes después de la cabeza. En el ejemplo [10, 20, 30], el T sería [20, 30].
- ST (Suma Tail): Es una variable que guarda la suma acumulada de la cola (T) en la recursión. Básicamente, en cada paso de la recursión, se calcula la suma de la cola, que luego se usa para calcular el total de la lista.

4 – Lenguajes Vs. Paradigmas

El Paradigma de Programación condiciona la forma en que se expresa la solución a un problema.

El Lenguaje de Programación (que se encuadra en un determinado paradigma) es la herramienta que permite expresar la solución a un problema.

¿Cómo elegir un paradigma en base a un problema?

Cuando la solución tiende a lo procedural (imperativo)

- tenemos secuencia
- decimos cómo resolver un problema
- tenemos mayor control sobre el algoritmo (decidimos más cosas definimos más cosas)

Cuando la solución tiende a lo declarativo

- expresamos características del problema
- alguien termina resolviendo el algoritmo para la máquina (necesitamos de alguna “magia”, algún mecanismo externo)
- tenemos menor control sobre el algoritmo (pero hay que pensar en menos cosas)

5 – Lenguajes de programación

Un lenguaje de programación es una notación para escribir programas, a través de los cuales podemos comunicarnos con el Hardware y dar así las órdenes adecuadas para la realización de un determinado proceso.

Un lenguaje viene definido por una gramática o conjunto de reglas que se aplican a un alfabeto constituido por el conjunto de símbolos utilizados.

CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación se pueden clasificar atendiendo a varios criterios, los principales son:

- Según el nivel de abstracción
- Según la forma de ejecución
- Según el paradigma de programación

5.1 – Reseña histórica y evolución

Los lenguajes de máquina, que representan la etapa más temprana de la programación, estaban intrínsecamente ligados al hardware. Dado que el desarrollo del hardware precedía al del software, cada computadora requería un lenguaje único, lo que hacía que la programación fuera una tarea ardua y específica para cada dispositivo.

El progreso inicial en este campo incluyó la creación de herramientas automáticas para la generación de código fuente.

Sin embargo, con la evolución continua de las computadoras y la creciente complejidad de las tareas informáticas, emergieron en la década de 1950 los primeros lenguajes de programación de alto nivel, marcando un hito en la abstracción y la eficiencia en la escritura de software.

5.2 – Tipos de lenguaje híbridos y puros

Al diseñar un lenguaje de programación, es posible adherirse estrictamente a las características de un paradigma específico o, alternativamente, enriquecerlo con elementos de otros paradigmas. Los lenguajes considerados puros son aquellos que se apegan rigurosamente a las normas de su paradigma original.

Por otro lado, los lenguajes híbridos se caracterizan por integrar aspectos distintivos de diferentes paradigmas.

En lenguajes funcionales puros: tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido. Ejemplos Haskell y Miranda.

Lenguajes funcionales híbridos: son menos dogmáticos que los puros, al admitir conceptos tomados del paradigma imperativo, como la secuencia de instrucciones o la asignación de variables. Ejemplos: Scala, Lisp, Scheme, Standard ML.

El Paradigma de Programación Orientado a Objetos plantea un modelado de la realidad que reviste una complejidad no soportada completamente por algunos lenguajes de programación.

- Lenguajes puros: los que solo permiten realizar programación orientada a objetos (Smalltalk, Eiffel, HyperTalk y Actor)
- Lenguajes Híbridos: los que permiten mezclar programación orientada a objetos con la programación estructurada básica. Estos incluyen C++, Objective C, Objective Pascal, Java, C#.