
TECPRO

Lucas Fanchin
UNL - Ingeniería en Informática

ÍNDICE - TECPRO

Introducción a los paradigmas.....	2
Definición de Paradigma.....	2
Paradigma de Programación.....	2
Principales Paradigmas de Programación.....	3
Primera clasificación.....	3
Segunda clasificación.....	4
Lenguajes de programación.....	4
Lenguajes VS Paradigmas.....	5
Paradigma Orientado a Objetos.....	6
Conceptos introductorios.....	6
Características de la POO.....	6
Los objetos en el paradigma.....	7
Clases.....	8
Relaciones entre clases.....	8
Reutilización.....	9
Pecados capitales del diseño.....	9
Objetos bien formados.....	10
Principios SOLID.....	10
Principios del Diseño de Software.....	11
Ley de Demeter.....	11
Tell don't ask.....	11
Único punto de salida.....	12
Paradigma Funcional.....	13
Conceptos Introductorios.....	13
Características.....	13
Funciones.....	14
Paradigma Lógico.....	15
Características.....	15
Un Programa en Prolog.....	15
Elementos de un Programa en Prolog.....	15

Principio de Resolución o Regla de Inferencia.....	16
La Unificación.....	16
Búsqueda de soluciones.....	16
Backtracking.....	16
Predicado de Corte.....	17
Predicado de Fallo.....	17
Recursividad.....	18
Representar información y relaciones en Prolog.....	18

Introducción a los paradigmas

Definición de Paradigma

Un paradigma es *una colección de modelos conceptuales que juntos modelan el proceso de diseño, orientan la forma de definir los problemas*. Dado que un paradigma es un modelo o ejemplo a seguir de los problemas que tiene que resolver y del modo como se van a dar las soluciones. El paradigma tiene más relación con el proceso mental que se realiza para construir un programa que con el programa resultante. Determina una manera especial de entender el mundo, explicarlo y manipularlo.

Paradigma de Programación

Los paradigmas de programación:

- Representan un enfoque particular o filosofía para la construcción del software.
- No es mejor uno que otro sino que cada uno tiene ventajas y desventajas.
- Hay situaciones donde un paradigma resulta más apropiado que otro.
- Está constituido por los supuestos teóricos generales, las leyes y las técnicas para su aplicación.

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.

Paradigmas

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. 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.

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.*

Criterios de la buena programación

- Plantear modelos cercanos a la realidad.
- Diseñar implementaciones que puedan ser extendidas y modificadas.
- Dar flexibilidad a las soluciones para que puedan ser reutilizadas.
- Desarrollar un código claro, simple y compacto.
- Construir soluciones genéricas.
- Focalización de las funcionalidades y componentes del sistema.

Principales Paradigmas de Programación

Primera clasificación

→ 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*.

→ Declarativo

Se construye 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

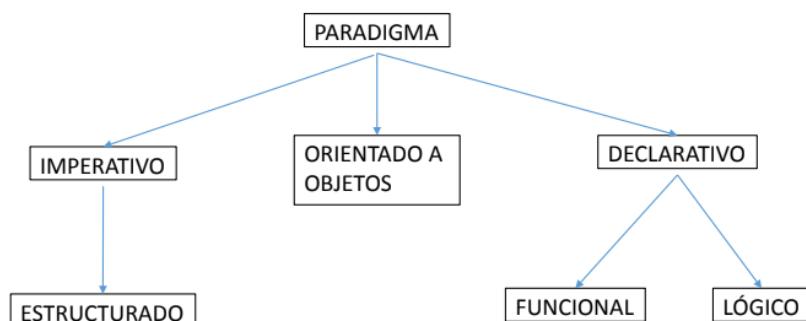
Las diferencias son que en la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema, pero no las instrucciones necesarias para solucionarlo y en la programación imperativa se describe paso a paso un conjunto de instrucciones, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema.

→ Orientado a Objetos

$$\text{OBJETOS} + \text{MENSAJES} = \text{PROGRAMA}$$

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. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo, y encapsulamiento. Su principal ventaja es ser un paradigma muy adecuado al diseño de aplicaciones relacionadas con todo lo que sea gestión.

- *Reusabilidad*: Realizar un diseño adecuado de la solución (clases), es factible. Utilizar diferentes partes de la solución del programa en otros proyectos.
- *Mantenibilidad*: Se logra gracias a la facilidad de abstracción de los problemas, más fáciles de entender. Permite ocultar detalles de implementación dejando visibles sólo aquellos aspectos más relevantes.
- *Fiabilidad*: Posibilidad de dividir el problema en partes más pequeñas, es posible testearlas de manera independiente y aislarlas para encontrar posibles errores.



Segunda clasificación

→ Estructurado

ALGORITMO + ESTRUCTURA DE DATOS = PROGRAMA

La programación imperativa se define por el estado del programa y las instrucciones que lo modifican. Es una forma de escribir programación de computadora de forma clara, para ello utiliza únicamente tres estructuras: secuencial, selectiva e iterativa.

- El conjunto de instrucciones, suele ser un código 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 hay suficiente documentación adjunta.

→ 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.

- *Altos niveles de abstracción*: Mayor énfasis en “¿qué hace?”
- *Fácil de formular matemáticamente*: Programas más cortos y sencillos de entender.
- *Rapidez en la codificación de los programas*
- *Administración automática de memoria*: El programador no es responsable de reservar la memoria.

→ 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. Este paradigma puede deducir nuevos hechos a partir de otros hechos conocidos.

- 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.

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

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

Lenguajes híbridos y puros

Los *lenguajes 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.

- Lenguajes puros: Tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial. Permiten realizar programación orientada a objetos
- Lenguajes híbridos: Son menos dogmáticos que los puros, al admitir conceptos tomados del paradigma imperativo, como la secuencia de instrucciones o la asignación de variables. Permiten mezclar programación orientada a objetos con la programación estructurada básica.

Lenguajes VS Paradigmas

El Paradigma de Programación condiciona la forma en que se expresa la solución a un problema. En cambio el Lenguaje de Programación 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 procedimental (*imperativo*)
 - Tenemos secuencia.
 - Decimos cómo resolver un problema.
 - Tenemos mayor control sobre el algoritmo.
- Cuando la solución tiende a lo *declarativo*
 - Expresamos características del problema.
 - Alguien termina resolviendo el algoritmo para la máquina.
 - Tenemos menor control sobre el algoritmo.

Paradigma Orientado a Objetos

- Comprender los fundamentos del paradigma con orientación a objetos utilizados por los lenguajes de programación.
- Conocer el modelo formal o semiformal subyacente del paradigma y la forma en que el mismo es incorporado en un lenguaje de programación.
- Resolver problemas a través del modelo de programación orientado a objetos.

Conceptos introductorios

“Un programa es un conjunto de objetos que colaboran entre sí e enviándose mensajes”

$$\text{OBJETOS} + \text{MENSAJES} = \text{PROGRAMA}$$

- **OBJETO:**

Los objetos responden a los pedidos interactuando con los otros objetos que conoce.

Un objeto es una abstracción conceptual del mundo real que se puede traducir a un lenguaje computacional o de programación. Un objeto tiene características y comportamiento A través de mensajes se relaciona con otros objetos

Características de la POO

→ Abstracción

Denota las *características esenciales de un objeto* donde se capturan sus comportamientos. Permite *identificar comportamientos comunes* para definir nuevos tipos de entidades en el mundo real. Por medio de ella podemos llegar a armar un conjunto de clases que permitan modelar el problema.

→ Modularidad

Permite *subdividir una aplicación* en partes más pequeñas, cada una de las cuales debe ser tan independiente como sea posible. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos.

→ Encapsulamiento

Permite reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción.

→ Principio de ocultación

El aislamiento *protege a las propiedades de un objeto* contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas.

→ Cohesión y acoplamiento

La *cohesión* mide las responsabilidades asignadas a cada objeto, mientras que el *acoplamiento* mide las relaciones entre los objetos. En POO se busca tener ALTA cohesión y BAJO Acoplamiento.

→ Herencia

Las clases se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes.

→ Polimorfismo

Es un comportamiento diferente asociados a objetos distintos pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Nos da la habilidad de implementar el mismo mensaje en formas diferentes en objetos diferentes.

→ Recolección de basura

Técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria.

Los objetos en el paradigma

Un objeto debe tener:

- **Abstracción:** Representación computacional de entes de la realidad y tiene dos características:
 - 1- *Esenciales*: Hacen que el ente sea lo que es.
 - 2- *Accidentales*: Si el ente no lo tiene o se cambia no deja de ser lo que es.
- **Identidad:** Es la existencia única del objeto en tiempo y espacio.
- **Comportamiento:** Conjunto de mensajes que el objeto puede responder.
- **Capacidad de inspección:** Visualizar en todo momento quienes son sus colaboradores y eventualmente cambiarlo.

Colaboradores internos

Es el conjunto de variables que va a tener un objeto. Una “variable” es el nombre de un objeto. En resumen es el conjunto de variables que contiene, y se define al programar el objeto.

Encapsulamiento

Permite cambiar la implementación de un objeto y que el usuario del objeto no se entere. El usuario del objeto no ve cómo el objeto lo implementa (métodos y estado interno). El encapsulamiento consiste en combinar datos y comportamiento en un paquete y ocultar los detalles de la implementación del usuario del objeto.

Interpretación Colaboraciones

Un contrato entre emisor y receptor. El emisor debe escribir bien el mensaje y enviar un colaborador externo que sepa hacer cosas que requiere el receptor. Por su lado el receptor debe devolver un objeto que sepa hacer cosas que requiere el emisor.

Polimorfismo

Dos o más objetos son polimórficos respecto de un conjunto de mensajes si todos ellos pueden responder de manera semánticamente equivalente, aún si su implementación es distinta.

El paradigma orientado a objetos está basado en la encapsulación de estado y comportamiento en entidades llamadas objetos. Emplea herencia para reutilizar código.

Clases

Se define a una **clase** como el descriptor para un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y comportamiento.

Podemos pensar en una clase como un molde o plantilla para un objeto. Cada objeto creado a partir de esa clase puede tener sus propios valores para las variables de instancia de esa clase.

Cada objeto es una **instancia** de una clase exactamente. Las instancias de una clase entienden y tienen la capacidad de responder los mensajes para los cuales hay métodos definidos en la clase.

Relaciones entre clases

→ Asociación (conexión entre clases)

Diremos que dos clases tienen una relación de asociación cuando una de ellas tenga que requerir o utilizar alguna de las propiedades o métodos de las otras.

Esta relación permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Las asociaciones pueden tener un nombre que es una frase verbal, nombres de roles en alguno de sus extremos y una multiplicidad que restringe el número de objetos de una clase en la relación.



→ Agregación/Composición (relaciones de pertenencia)

Agregación: implica una composición débil, si una clase se compone de otras y quitamos alguna de ellas, entonces la primera seguirá funcionando normalmente.

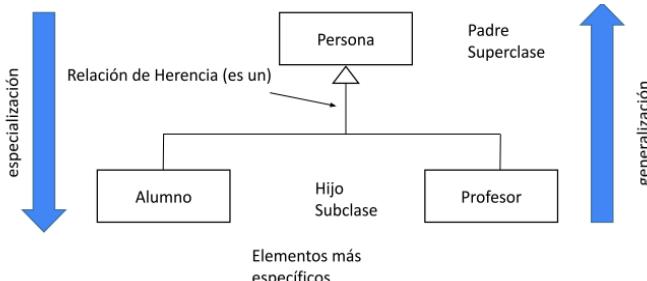


Composición: Es una forma fuerte de composición, donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora. Los componentes constituyen una parte del objeto compuesto.



→ Generalización/especialización (relaciones de herencia)

De todas las relaciones posibles entre las distintas clases y objetos, hay que destacar por su importancia la relación de **herencia**, ésta es una relación entre clases que comparten su estructura y el comportamiento.



Reutilización

Existen dos mecanismos para construir clases utilizando otras clases.

- **Composición:** Una clase posee objetos de otras clases (tiene un). Se puede reutilizar los atributos y métodos de otras clases, a través de la invocación de los mensajes.
- **Herencia:** Se pueden crear clases nuevas a partir de clases preexistentes (es un). Se puede reutilizar los atributos y métodos de otras clases como si fueran propios. Las clases se *organizan jerárquicamente*, y una nueva clase puede reutilizar la estructura y comportamiento de otras previamente definidas.
La herencia habilita el *polimorfismo*, que es que una clase base contenga un método polimórfico, pueda ser redefinido en las clases derivadas. Esto nos permite diseñar sistemas más sencillos que se pueden acomodar más fácilmente el cambio porque le permite tratar objetos diferentes de la misma forma.

Pecados capitales del diseño

1. Rigidez

Tendencia del software a ser difícil de cambiar, aún ante cambios simples.

2. Fragilidad

Tendencia del software a romperse en varios lugares cuando se hace un cambio.

3. Inmovilidad

Contiene partes que podrían ser útiles en otros sistemas, pero el esfuerzo y riesgo de separarlas del sistema original es muy grande.

4. Viscosidad

Un software viscoso es aquel que su diseño es difícil de preservar.

5. Complejidad innecesaria

Contiene elementos que no son actualmente útiles

6. Repetición innecesaria

Cortar y pegar puede ser desastroso para operaciones de edición de código.

Cuando el mismo código aparece una y otra vez, en ligeramente distintas formas, se está necesitando una abstracción.

Cuando hay código redundante en el sistema, los cambios en el sistema pueden ser arduos.

7. Opacidad

Tendencia de un módulo a ser difícil de entender. El código puede ser escrito de una manera clara y expresiva, o de una manera compleja y opaca. A medida que el código evoluciona en el tiempo, llega a ser más y más opaco.

Objetos bien formados

Los objetos cumplen con ciertos principios y pautas de diseño que los hacen coherentes y mantenibles. Algunos criterios comunes son:

- Encapsulación: Los objetos deben encapsular su estado interno y ocultarlo de otros objetos. Deben proporcionar una interfaz clara y consistente para interactuar con ellos.
- Cohesión: Los objetos deben tener una única responsabilidad claramente definida. Deben estar enfocados en hacer una cosa y hacerla bien.
- Acoplamiento: Los objetos deben estar acoplados de manera débil, lo que significa que deben depender lo menos posible de otros objetos. Esto promueve la flexibilidad y la reutilización del código.
- Cumplimiento de los principios SOLID: Los objetos bien formados suelen cumplir estos principios.

Principios SOLID

En el paradigma orientado a objetos, existen varias reglas y principios de diseño que ayudan a crear un código limpio, modular y que se pueda mantener. Algunos de los principales son:

1. Principio de Responsabilidad Única (**SRP**)
2. Principio de Abierto/Cerrado (**OCP**)
3. Principio de Sustitución de Liskov (**LSP**)
4. Principio de Segregación de Interfaces (**ISP**)
5. Principio de Inversión de Dependencia (**DIP**)

→ Principio de Responsabilidad Única (**SRP**)

Una clase debe estar diseñada para hacer una sola cosa y hacerla bien. No debe tener múltiples responsabilidades o tareas, ya que esto puede hacer que la clase sea más difícil de entender, mantener y probar.

→ Principio de Abierto/Cerrado (OCP)

Las entidades de software deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que se deben poder agregar nuevas funcionalidades o comportamientos sin modificar el código existente.

→ Principio de Sustitución de Liskov (LSP)

Las clases derivadas deben poder ser sustituidas por sus clases base sin alterar el correcto funcionamiento. Esto garantiza que los objetos de una clase base puedan ser reemplazados por objetos de una clase derivada sin generar errores ni comportamientos inesperados.

→ Principio de Segregación de Interfaces (ISP)

Los clientes no deben depender de interfaces que no utilizan. Este principio promueve la creación de interfaces específicas y cohesivas, evitando así la dependencia de funcionalidades innecesarias.

→ Principio de Inversión de Dependencia (DIP)

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Este principio promueve la dependencia hacia interfaces, lo que facilita la flexibilidad y el cambio en el diseño.

Principios del Diseño de Software

Ley de Demeter

La Ley de Demeter es un principio de diseño de software que promueve la encapsulación y reduce la dependencia entre las clases. La idea principal es que un objeto debe tener conocimiento limitado sobre otros objetos y solo debe interactuar con sus objetos cercanos. Por lo que promueve que no se debe llamar a métodos de los objetos devueltos por otros métodos.

La Ley de Demeter ayuda a maximizar la encapsulación, por lo que ayuda a reducir el acoplamiento entre clases, lo que hace que el código sea más fácil de mantener, entender y probar.

Por ejemplo, si tienes un objeto A que tiene una referencia a un objeto B, y quieras llamar a un método de un objeto C que es miembro de B, en lugar de hacerlo a través de A (A->B->C), deberías hacerlo directamente desde A si es posible (A->C).

Tell don't ask

Es un principio de diseño de software que promueve la encapsulación y la responsabilidad adecuada de las clases y objetos. Se debe proporcionar al objeto los comandos necesarios para que realice las acciones requeridas.

Esto ayuda a reducir el acoplamiento y la dependencia entre objetos. Facilita el mantenimiento y la extensibilidad del código, ya que los cambios en el estado interno de un objeto no afectarán a otras partes del sistema.

Fomenta un diseño más cohesivo y modular, donde los objetos son capaces de gestionar su propio estado y realizar acciones basadas en los comandos que se les envían, en lugar de ser manipulados externamente

Único punto de salida

Establece que un programa debe tener un punto de entrada principal y un único punto de salida donde la ejecución del programa termina. Esto promueve la claridad, la simplicidad y el mantenimiento del código, y facilita la gestión de recursos y la comprensión del flujo de ejecución del programa.

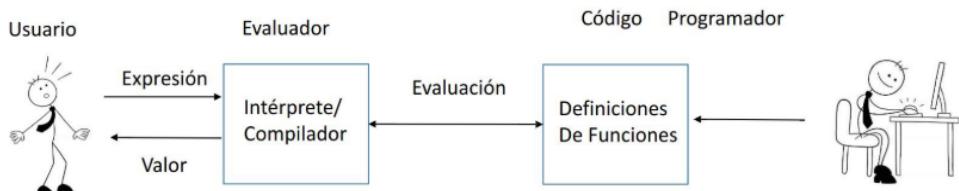
Esto significa que las diferentes rutas o flujos de ejecución dentro del programa deben converger en un único punto de salida.

HACER RESUMEN U2-CLASE4 (Patrones, en finales entra)

Paradigma Funcional

Conceptos Introductorios

Es un paradigma declarativo que se basa en un modelo matemático de composición de funciones. En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que se produce el valor deseado, tal como sucede en la composición de funciones matemáticas.



Expresan mejor QUÉ hay que calcular. No especifican tanto CÓMO realizar el mencionado. (nos enfocamos en el resultado y no en el procedimiento)

"Los cálculos se ven como una función matemática que hacen corresponder entradas y salida"

Algunos detalles:

- No existe el concepto de celda de memoria que es asignada o modificada.
- Existen valores intermedios que son el resultado de cómputos intermedios, los cuales resultan en valores útiles para los cálculos subsiguientes.
- Todas las funciones tienen transparencia referencial.
- La repetición se modela utilizando la recursividad ya que no existe el valor de una variable.

Características

Transparencia referencial → Garantiza que una función siempre devuelva el mismo valor para los mismos argumentos, lo que se traduce en inmutabilidad de los valores resultantes. Esto significa que no hay cambios de estado ni efectos colaterales en las funciones; cada expresión produce un resultado predecible basado exclusivamente en sus entradas.

Utilización de tipos de datos genéricos → Permite trabajar con diferentes tipos de datos, para que el código sea más flexible y reutilizable (forma de implementar el polimorfismo).

Recursividad → Permite trabajar con estructuras de datos que se definen en términos de sí mismas, como las listas y funciones recursivas que las operan.

Funciones de Orden Superior → Podemos tratar las funciones como si fueran datos, lo que significa que pueden ser pasadas como argumentos a otras funciones o devueltas como resultados. Esto permite crear funciones más abstractas y generales, mejorando la estructura y modularidad de los programas.

Los lenguajes funcionales simplifican la gestión de memoria del programa → La asignación y la inicialización de la memoria se realizan de forma implícita, y la memoria no utilizada se recupera automáticamente mediante un recolector de basura.

Ventajas

- Fácil de formular matemáticamente.
- Administración automática de la memoria.
- Simplicidad en el código.
- Rapidez en la codificación de los programas.

Limitaciones

- No es fácilmente escalable.
- Difícil de integrar con otras aplicaciones.
- No es recomendable para modelar lógica de negocios o para realizar tareas transaccionales.

Funciones

Función

Una función es una regla que asocia elementos de dos conjuntos. En términos matemáticos, una función f de un conjunto A a un conjunto B (escrita como $f : A \rightarrow B$) asigna a cada elemento de un subconjunto de A (el dominio de f) exactamente un elemento de un subconjunto de B (la imagen de f).

1. **Relación Única:** Cada elemento del dominio tiene una única imagen en el conjunto de llegada.
2. **Jerarquía de Funciones:** Las funciones simples se usan para definir funciones más complejas.
3. **Referencia por Nombre:** Se referencian por su nombre, facilitando su uso.
4. **Funciones Primitivas:** Se comienza con funciones primitivas y se construyen funciones más complejas.
5. **Diseño Modular:** Se aplican conceptos de diseño modular, descomponiendo problemas en partes más pequeñas.
6. **Descomposición de Problemas:** Cada problema puede ser un problema primitivo o necesitar una descomposición adicional.
7. **Abstracción:** Es posible usar una función sin conocer su estructura interna gracias a la referencia por nombre.
8. **Refinamientos Sucesivos:** Se puede diseñar de manera incremental, refinando sucesivamente.
9. **Modularidad Compleja:** La estructura de un módulo complejo puede definirse sin necesitar definir inmediatamente sus componentes internos.

Funciones de Orden Superior

Son funciones que reciben como argumento otra función o devuelven una función como resultado.

- Son útiles porque permiten capturar esquemas de cómputo generales (abstracción).
- Son más útiles que las funciones normales ya que permiten especificar parte del comportamiento al utilizarlas, lo que aumenta su flexibilidad y aplicabilidad.

Paradigma Lógico

Este paradigma está centrado en el razonamiento lógico y la inferencia automática. Se basa en la declaración de hechos y reglas sobre el problema y deja que el motor de inferencia lógica derive las conclusiones y soluciones pertinentes (cuyo objetivo es responder consultas).

Características

Ventajas

- Simplicidad en la expresión del problema: Permite expresar problemas de manera natural y concisa.
- Automatización de la inferencia: El motor de inferencia deduce las respuestas, simplificando el desarrollo.

Desventajas

- Rendimiento: Los motores de inferencia pueden ser lentos y consumir muchos recursos en problemas complejos.
- Curva de aprendizaje: Puede ser más difícil de aprender para programadores acostumbrados a paradigmas imperativos o funcionales.

Campos de aplicación

Sistemas Expertos: Emulan el comportamiento de un experto humano en un dominio específico de conocimiento.

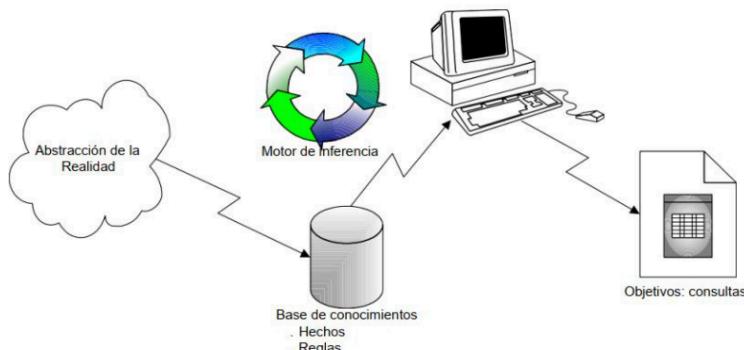
Demostración Automática de Teoremas: Verificación formal de programas informáticos para demostrar propiedades de seguridad, corrección y rendimiento.

Inteligencia Artificial: Puede considerarse parte de la inteligencia artificial debido a su capacidad para representar conocimiento, razonar sobre este conocimiento y llegar a conclusiones lógicas.

Un Programa en Prolog

Prolog es un programa conformado por un conjunto de hechos y reglas que representan el problema que se pretende resolver. Ante una determinada pregunta sobre el problema, el Prolog utilizará estos hechos y reglas para intentar demostrar la veracidad o falsedad de la pregunta que se le ha planteado.

Elementos de un Programa en Prolog



Principio de Resolución o Regla de Inferencia

Es un algoritmo que, a partir de la negación de la pregunta y los hechos y reglas del programa, intenta llegar al absurdo para demostrar que la pregunta es cierta. Prolog utiliza este principio.

La Unificación

La unificación en Prolog es el proceso mediante el cual el lenguaje intenta hacer que dos términos sean iguales al encontrar una asignación de valores a las variables involucradas. Este mecanismo es fundamental para la ejecución de programas en Prolog y **se basa en las siguientes normas:**

1. Una variable siempre unifica con un término, quedando ésta ligada a dicho término.
2. Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que se unifican se ligan a dicho término.
3. Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.
4. Si algún término no unifica, ninguna variable queda ligada.

Búsqueda de soluciones

Una llamada a un predicado se llama objetivo (consulta) y puede tener éxito o fallar, indicando si el predicado es cierto o falso para los argumentos dados.

Características de las Secuencias de Objetivos

- Ejecución secuencial: Los objetivos se ejecutan de izquierda a derecha.
- Fallo de objetivos: Si un objetivo falla, los siguientes no se ejecutan y toda la conjunción falla.
- Éxito de objetivos: Si un objetivo tiene éxito, sus variables quedan ligadas y afectan al resto de objetivos.
- Conjunción exitosa: Si todos los objetivos tienen éxito, la conjunción tiene éxito y mantiene las ligaduras.

Proceso de Búsqueda:

- Unificación: Determina un subconjunto de cláusulas que pueden ejecutar el objetivo.
- Backtracking: Si un objetivo falla, Prolog retrocede y prueba otras opciones.

Backtracking

El backtracking es una técnica que recorre sistemáticamente todos los caminos posibles para encontrar soluciones. Si un camino no lleva a una solución, retrocede al paso anterior para buscar un nuevo camino.

Características Generales

- **Secuencia de decisiones:** Cada solución proviene de una serie de decisiones que pueden deshacerse.
- **Optimización:** Se busca satisfacer u optimizar una función objetivo.
- **Árbol de expansión:** Las etapas del algoritmo se representan como un árbol implícito, no construido físicamente, con cada nivel representando una decisión.

Funcionamiento

- **Puntos de elección:** Prolog identifica de antemano posibles soluciones alternativas y las almacena en una pila.
- **Ejecución de consultas:** Prolog elige el primer punto de elección y ejecuta la consulta.
- **Éxito y fallo:** Si el objetivo tiene éxito, continúa con el siguiente. Si falla, retrocede (backtracking), deshaciendo ligaduras y probando nuevas soluciones desde los puntos de elección.
- **Repetición:** El proceso se repite mientras haya puntos de elección y consultas pendientes. El programa finaliza cuando no quedan puntos de elección ni consultas por ejecutar.

El backtracking asegura que Prolog explore todas las posibilidades para encontrar una solución

Predicado de Corte

El predicado de corte es un predicado predefinido, representado por “ ! ”, que siempre se cumple inicialmente y falla durante el backtracking, impidiendo el retroceso. El corte elimina los puntos de elección del predicado que lo contiene, comprometiendo el resultado del objetivo al éxito o fallo de los objetivos que siguen. Es como si Prolog "olvidara" que el objetivo puede tener varias soluciones.

Beneficios de usar predicado de corte

- **Optimización:** Evita explorar puntos de elección inútiles, podando el árbol de búsqueda de posibles soluciones.
- **Legibilidad y Comprensión:** Facilita la legibilidad y comprensión del algoritmo programado.
- **Implementación de Algoritmos Diferentes:** Permite implementar diferentes algoritmos según la combinación de argumentos de entrada.
- **Limitación de Soluciones:** Asegura que un predicado tenga solo una solución al ejecutar un corte después de encontrar la primera.

Predicado de Fallo

El predicado de fallo (*fail*) es un predicado predefinido sin argumentos que siempre falla, desencadenando el proceso de retroceso (backtracking) para buscar nuevas soluciones. Prolog normalmente se detiene al encontrar una solución, pero “ *fail* ” fuerza a que continúe explorando el árbol de búsqueda hasta agotar todas las posibles soluciones.

Recursividad

Una definición recursiva se basa en definir relaciones en términos de ellas mismas. Si en el lado derecho de una cláusula aparece el mismo predicado que en el lado izquierdo, la cláusula tiene un llamado recursivo, es decir, "se llama a sí misma".

En una definición recursiva, es necesario considerar dos casos:

1. Caso Básico: El punto donde se detiene el proceso.
2. Caso Recursivo: Se descompone el caso actual en uno más simple, suponiendo que este último ya está resuelto.

Representar información y relaciones en Prolog

→ Términos

Los términos son los componentes básicos que incluyen constantes, variables, y estructuras.

→ Lógica de Predicados

Se usa para expresar relaciones entre objetos mediante predicados, que son funciones que devuelven verdadero o falso.

→ Fórmulas atómicas

Representan afirmaciones simples o relaciones entre entidades del dominio del problema. Tienen la forma $p(t_1, \dots, t_n)$, donde p es un predicado y t_i son términos. Son la base de las expresiones lógicas en prolog.

→ Orden de Precedencia de los Conectores Lógicos

Determina cómo se evalúan las expresiones lógicas.

→ Semántica de las Fórmulas Lógicas

La semántica de las fórmulas lógicas se refiere al significado de las expresiones lógicas y cómo se evalúan en términos de verdad.

→ Inferencia Lógica

Es el proceso de derivar nuevas verdades a partir de hechos y reglas conocidas.

→ Cláusulas Definidas

En Prolog son reglas que tienen la forma de implicaciones, donde la cabeza es una fórmula atómica y el cuerpo es una conjunción de fórmulas atómicas.