Universidad Tecnológica Nacional
Facultad Regional Córdoba
Dpto de Ingeniería en Sistemas de Información
Cátedra Paradigmas de Programación

UNIDAD Nº 1

INTRODUCCIÓN A LOS LENGUAJES

Y

PARADIGMAS DE PROGRAMACIÓN

EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPILADO POR:

MGTR. ING. CORSO, CYNTHIA
ING. FRIAS PABLO
ESP. ING.GUZMAN, ANALIA
ESP. ING. LIGORRIA, LAURA
DR. ING. MARCISZACK, MARCELO
ING. ROMANI, GERMAN
ESP. ING. TYMOSCHUK, JORGE

ÚLTIMA REVISIÓN: MAYO 2021

ÍNDICE

Objetivos de la Unidad 4
Contenidos Abordados 4
1. Introducción y conceptos generales 4
1.1 Programas 5
1.2 Paradigmas 5
1.3 Lenguajes de Programación 5
1.4 Programación 6
1.4.1 Criterios para una buena programación 6
2. Paradigmas fundamentales 6
2.1 Clasificación y Evolución Histórica 7
2.2 Paradigma Imperativo 9
2.2.1 Paradigma Estructurado 9
2.3 Paradigma Orientado a Objetos 11
2.4 Paradigma Declarativo 12
2.4.1 Paradigma Funcional 12
2.4.2 Paradigma Lógico 13
3. Diferencias entre Lenguajes y Paradigmas de programación 14
4. Lenguajes de programación 15
4.1 Sintaxis, Semantica y Gramática 15
4.1.1 Sintaxis 15
4.1.2 Semántica 15
4.1.3 Gramática 16
4.2 Criterios de evaluación de los lenguajes de programación 16
4.2.1 Legibilidad 16
4.2.2 Codificación 17
4.2.3 Fiabilidad 17
4.2.4 Escalabilidad 17
4.2.5 Modularidad 18
4.2.6 Reutilización 18
4.2.7 Portabilidad 18

4.2.8 Nivel 18
4.2.9 Eficiencia 19
4.2.10 Modelado de datos 19
4.2.11 Modelado de procesos 19
4.2.12 Disponibilidad de compiladores y herramientas 19
4.2.13 Familiaridad 20
4.2.14 Costo 20
4.3 Reseña histórica y evolución de los lenguajes de programación 20
4.3.1 Evolución de los lenguajes de programación 20
4.4 Tipos de Lenguajes: híbridos y puros. 21
5. Bibliografía 22

OBJETIVOS DE LA UNIDAD

Que el alumno evidencie y valore las características constitutivas de cada uno de los lenguajes y paradigmas de programación y los asocie con los problemas tipos que son factibles de resolver por cada uno de estos.

CONTENIDOS ABORDADOS

Introducción y conceptos generales: Programas, paradigmas, lenguajes de programación y programación.

Paradigmas fundamentales: Clasificación y Evolución Histórica, Definición, lenguajes asociados, ventajas, limitaciones y áreas de aplicación de los paradigmas Imperativo, Estructurado, Orientado a Objetos, Declarativo, Funcional y Lógico.

Diferencia entre lenguaje y paradigma de programación.

Lenguajes de Programación: Conceptos, criterios de evaluación, reseña histórica y evolución y tipos de lenguajes: híbridos y puros.

1. INTRODUCCIÓN Y CONCEPTOS GENERALES

El diccionario de Oxford define a un paradigma en su sentido más amplio como "Un patrón o modelo, un ejemplo". En la aplicación científica, fue enunciado por Thomas Khun en el libro *La estructura de las revoluciones científicas,* publicado en el año 1962, refiriéndose a la visión del mundo que poseen los científicos en un determinado momento histórico. Khun resume el concepto de paradigma de la siguiente manera:

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

En términos más sencillos, podemos definir a un paradigma como:

- Un modelo o ejemplo a seguir por 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.

En cuanto a los lenguajes de programación, definimos como **Paradigma de Programación** a un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro. Un paradigma está constituido por los supuestos teóricos generales, las leyes y las técnicas para su aplicación que adoptan los miembros de una determinada comunidad científica.

1.1 PROGRAMAS

Un programa de computadoras, también llamado software, es un conjunto de códigos, instrucciones, declaraciones, proposiciones, etc. que describen, definen o caracterizan la realización de una acción en la computadora. Si bien en los inicios de la historia de la programación los programas dictaban instrucciones directamente a las máquinas, hoy los programas se diseñan según un paradigma de programación y se escriben usando algún lenguaje de programación asociado.

1.2 PARADIGMAS

Un **paradigma de programación** es un **modelo** básico de diseño e implementación de programas, que permite desarrollar software conforme a ciertos **principios** o **fundamentos específicos** que se aceptan como válidos. Otra definición lo expresa como un marco conceptual que determina los bloques básicos de construcción de software y los criterios para su uso y combinación. En otras palabras, 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.

Desde una mirada más amplia, 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.

En cada paradigma hay conceptos específicos que son diferentes, y a la vez, muchos elementos en común que son medulares de la programación. También existen conceptos que si bien reciben un mismo nombre para todos, tienen diferentes implicancias en cada paradigma.

Estos conceptos, según la forma en que se presentan y articulan, constituyen las características que permiten definir cada paradigma (y también los lenguajes) y a la vez poder relacionarlos, ya sea por similitud u oposición. Hay conceptos originarios de ciertos paradigmas que se pueden aplicar en otros o ser combinados de múltiples formas. Son herramientas conceptuales cuya importancia es de primera magnitud para poder encarar con posibilidades de éxito proyectos de desarrollo de software de mediana o gran escala.

Esto significa que 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.

En otras palabras, un Paradigma de Programación es una colección de patrones conceptuales (estructuras o reglas) que juntos modelan el proceso de diseño y que determinan en última instancia la estructura de los programas realizados. En cambio, un Lenguaje de Programación se dice que pertenece a un determinado paradigma si recoge adecuadamente los patrones conceptuales definidos en el mismo.

1.3 LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es una técnica estándar de comunicación que permite expresar las instrucciones o declaraciones que pueden ser interpretadas en una computadora. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su gramática y el significado de sus elementos y expresiones. Se encuadra en un determinado paradigma, siendo la herramienta que permite expresar la solución a un problema.

1.4 PROGRAMACIÓN

La 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, además del dominio del lenguaje a utilizar, algoritmos especializados y lógica formal.

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 PARA UNA BUENA PROGRAMACIÓN

- Plantear **modelos cercanos a la realidad** que permitan abstraerse de las especificaciones computacionales y lograr una relación lo más fluida posibles entre el dominio de aplicación y el programa.
- Diseñar implementaciones de manera que puedan ser **extendidas y modificadas** con el mínimo impacto en el resto de su estructura, ante cambios en la realidad o nuevos requerimientos.
- Dar flexibilidad a las soluciones para que puedan ser **reutilizadas** en múltiples contextos, incluso diferentes a los que les dieron origen.
- Diseñar una **articulación funcional** adecuada de las diferentes entidades que conforman el sistema.
- Desarrollar un **código claro**, simple y compacto.
- Construir **soluciones genéricas** que permitan abstraerse de las particularidades propias de cada tipo de entidad de software y a la vez atender a la especificidad de cada una de ellas.
- **Focalización** de las funcionalidades y componentes del sistema para poder trabajar sobre eficiencia.

2. PARADIGMAS FUNDAMENTALES

En la actualidad, 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 los siguientes:

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

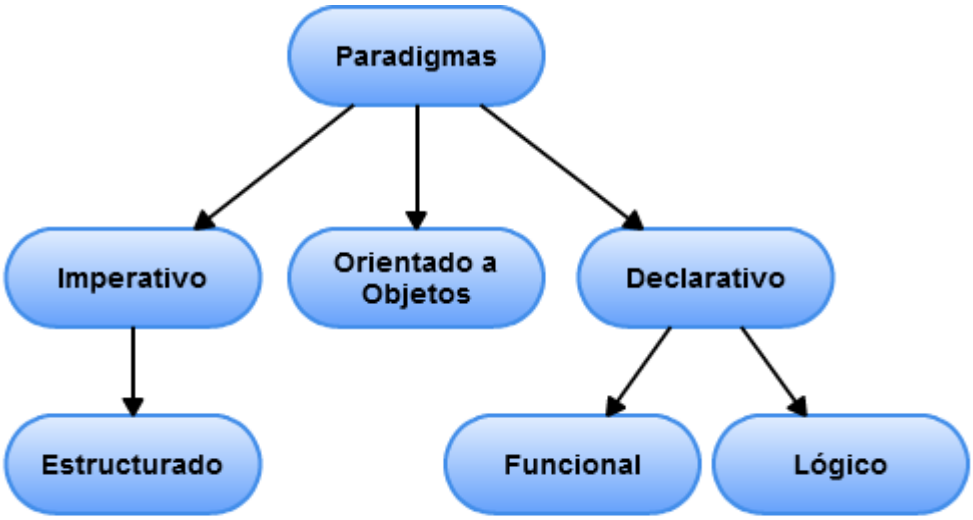


IMAGEN 1. PARADIGMAS FUNDAMENTALES

2.1 CLASIFICACIÓN Y EVOLUCIÓN HISTÓRICA

A los paradigmas se los puede clasificar conceptualmente de diversas maneras según los criterios que se prioricen. 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 orígenes imperativos

Los primeros lenguajes de programación se basaron sobre un conjunto de premisas de funcionamiento que luego, con el correr de los años, se fueron complejizando y ampliando, pero que comparten suficientes características en común para considerarlas parte de un mismo paradigma que se denomina **imperativo o procedural**, según el énfasis que se haga en un aspecto u otro**.** En un programa tienen un papel dominante los **procedimientos** compuestos por **sentencias imperativas**, es decir aquellas que indican realizar una determinada operación que modifica los datos guardados en memoria, mediante un algoritmo en el que se detalla la **secuencia de ejecución** mediante estructuras de control. Utiliza **variables y estructuras de datos**, vinculados a celdas de memoria**,** sobre las que se realizan **asignaciones destructivas**, provocando **efecto de lado**. Un programa realiza su tarea ejecutando una **secuencia de pasos** elementales regida por **instrucciones de control** expresamente indicadas en el programa que marcan el flujo de ejecución de operaciones para resolver un problema específico. Siendo muy variados y numerosos, sus lenguajes cuentan con herramientas que permiten organizar los programas en diversos módulos o procedimientos que se distribuyen las responsabilidades, generar unidades de software genéricas, utilizar tipos de datos simples y estructuras de datos, entre otras características. En otras palabras, un programa detalla una secuencia o algoritmo que indica "**cómo**" se va procesando paso a paso, la información necesaria.

Los lenguajes de este paradigma que siguen vigentes en la actualidad, lo han logrado por su **simplicidad**, su **versatilidad** y **robustez**. Además, sus elementos característicos siguen siendo retomados por numerosos lenguajes de otros paradigmas, por lo que tiene una gran importancia dentro de las ciencias de la computación.

La ruptura declarativa

Posteriormente, surgieron diferentes corrientes que objetaron los fundamentos de la programación del momento y plantearon otras premisas desde las cuales desarrollar programas. En consecuencia, se crearon nuevos lenguajes de programación, que en la actualidad se los puede conceptualizar en el marco de los **paradigmas declarativos**, incluyendo al **paradigma lógico** y al **paradigma funcional**.

Lo que los diferenció de los otros lenguajes de la época, y que consiste hoy en su principal propiedad, es el poder lograr **desentenderse de los detalles de implementación de un programa** en cuanto al control de ejecución, las asignaciones de memoria y la secuencia de sentencias imperativas, o sea todo lo que implique "**cómo**" se resuelve el problema, y **concentrarse en la declaración o descripción de "qué" es la solución de un problema**, creando programas de un nivel más alto. Por lo tanto, su principal característica es la **declaratividad**, por la que sus programas especifican un conjunto de declaraciones, que pueden ser proposiciones, condiciones, restricciones, afirmaciones, o ecuaciones, que caracterizan al problema y **describen su solución**. A partir de esta información el sistema utiliza **mecanismos internos de control**, comúnmente llamado "motores", que evalúan y relacionan adecuadamente dichas especificaciones, de manera de obtener la solución. En general, las variables son usadas en expresiones, funciones o procedimientos, en los que se unifican con diferentes valores mediante el "encaje de patrones" (*pattern matching*) manteniendo la **transparencia referencial**. En ellas no se actualizan los estados de información ni se realizan asignaciones destructivas.

Que se denomine a algunos paradigmas como declarativos, no implica que no exista declaratividad en lo demás paradigmas, como tampoco que no haya ningún rasgo procesual o imperativo en estos. Significa que dichos conceptos son característicos, identificatorios y que se presentan con mayor claridad en los respectivos lenguajes. Tampoco quiere decir que sea esa la única diferencia. Simplemente es un modelo teórico para relacionar los grupos de paradigmas que tienen mayor afinidad conceptual entre ellos. La noción de **declaratividad** en mayor o menor medida está presente en cualquier paradigma, por lo que se pueden construir soluciones muy diferentes que la apliquen a su manera.

El mundo de objetos

Más recientemente, surgieron nuevos lenguajes de programación que, sin dejar de lado las premisas del paradigma imperativo, incorporaron nuevos elementos que fueron adquiriendo una importancia creciente y un impacto tal en la forma de desarrollar soluciones, que pueden conceptualizarse como otro paradigma: "**objetos"**.

Un programa hecho en un lenguaje orientado a objetos, si no es puro en su totalidad, incluye sentencias imperativas, requiere de la implementación de procedimientos en los que se indica la secuencia de pasos de un algoritmo y hay asignaciones destructivas con efecto de lado, pero son herramientas que se utilizan en un contexto caracterizado por la presencia de **objetos que interactúan entre sí** enviándose mensajes, que deja en un segundo plano a las otras propiedades. Es una perspectiva y una comprensión del sistema diferente, más integrales, más cercanas al dominio real del problema que a las características de la arquitectura de la computadora, que se traduce en un diseño y una programación con estilo propio.

Se fundamenta en concebir a un sistema como un conjunto de entidades que representan al mundo real, los **"objetos"**, que tienen **distribuida la funcionalidad e información** necesaria y que **cooperan entre sí** para el logro de un objetivo común.

Cuenta con una estructura de desarrollo modular basada en **objetos**, que son definidos a partir de **clases**, como implementación de tipos abstractos de datos. Utiliza el **encapsulamiento** como forma de abstracción que separa las interfaces de las implementaciones de la funcionalidad del sistema

(**métodos**) y oculta la información (**variables**) y un mecanismo de envío de **mensajes**, que posibilita la interacción entre los objetos y permite la **delegación** de responsabilidades de unos objetos a otros. Para realizar código genérico y extensible tiene **polimorfismo**, basado en el **enlace dinámico**, que permite que a entidades del programa interactuar indistintamente con otras y la **Herencia**, que permite que los objetos sean definidos como extensión o modificación de otros.

2.2 PARADIGMA IMPERATIVO

El paradigma imperativo describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar 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:

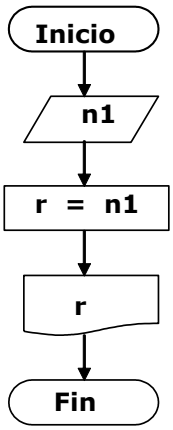
- Ejecutar un conjunto de instrucciones, encabezadas por una etiqueta (label)
- Ejecutar un rango de instrucciones definido por etiquetas desde/hasta.
- Derivar el flujo de control a determinada etiqueta. (Go to, go to .. depending).
- Realizar el uso intensivo de banderas para la detección de eventos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

Luego apareció el paradigma estructurado, que a continuación detallaremos.

2.2.1 PARADIGMA ESTRUCTURADO

Algoritmo + Estructura de Datos = programa

Describe la programación en términos del estado del programa y sentencias que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea.
La implementación de hardware de la mayoría de computadores es imperativa; prácticamente todo el hardware de los computadores está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que el hardware de los computadores implementa el paradigma de las Máquinas de Turing. Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo del computador.
Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma. Las recetas y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; cada paso es una instrucción, y el mundo físico guarda el estado.




```
Inicio
 Leer n1
 Procesar r = n1 + n1
 Mostrar r
Fin
```

Lenguajes asociados: Fortran, C, Pascal.

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.

Limitaciones:

- La complejidad de muchos sistemas actuales hace que sea complicado definir la distribución del código en módulos. Este paradigma posibilita modularizar la codificación, pero no define cual es el criterio a seguir. (Como si lo hace el paradigma Programación orientada a Objetos)
- No se adapta a todos los tipos de problemas.

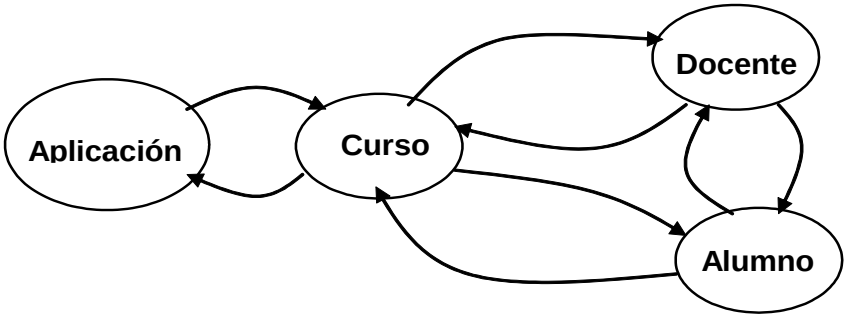
Aplicaciones:

- Sistemas de bajo nivel
- Sistemas de tiempo real
- Programación de micro controladores
- Máquinas de estado
- Desarrollo de video juegos de consola
- Aplicaciones que deben manejar recursos limitados

2.3 PARADIGMA ORIENTADO A OBJETOS

Objetos + Mensajes = Programa

Este paradigma está basado en la idea de encapsular estado y operaciones en objetos. Pero cada objeto no estará aislado de los demás. Los objetos colaboran entre sí para lograr un objetivo en común. Esta colaboración se realiza a través de mensajes. Llamaremos objeto a toda entidad que posee un estado y que puede realizar actividades. Llamaremos mensajes al medio por el cual se comunican los objetos.
Algunos lenguajes orientados a objetos son: Simula, Smalltalk, C++, Java, Visual Basic .NET, etc.
Su principal ventaja es ser un paradigma muy adecuado al diseño de aplicaciones relacionadas con todo lo que sea gestión.



Conjuntos de objetos que colaboran entre si

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

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.

Limitaciones:

- La implementación de algunas de los mecanismos soportados por el paradigma orientado a objetos como la herencia, hace que los programas sean más extensos lo que provoca como consecuencia que su ejecución sea más lenta en algunos casos.
- La curva de aprendizaje cuando se conoce otras formas de programación suele ser más lenta.

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

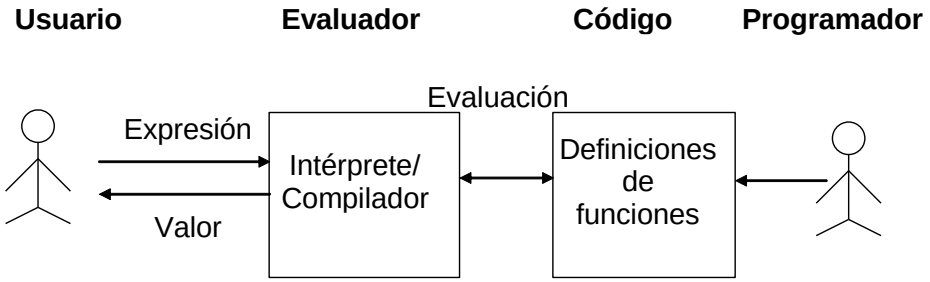
2.4 PARADIGMA DECLARATIVO

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 es obtenida 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

Este paradigma concibe a la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos (como lo hace el paradigma imperativo). Permite resolver ciertos problemas de forma elegante y los lenguajes puramente funcionales evitan los efectos secundarios comunes en otro tipo de programaciones.



Lenguajes asociados: Lisp, Haskell, ML

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.

Limitaciones

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

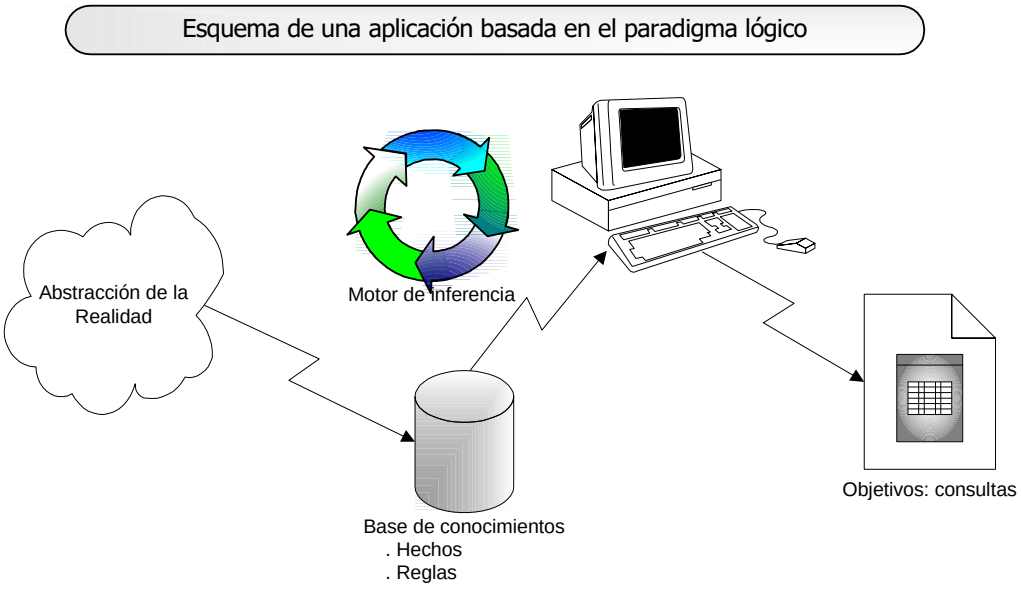
Aplicaciones:

- Programas de Inteligencia Artificial
- Sistemas distribuidos de control (NCS) tolerante a fallos de software (por ejemplo, control de tráfico aéreo, mensajería instantánea, servicios basados en Web)
- Aplicaciones académicas de gran complejidad matemática.
- Telecomunicaciones y telefonía

2.4.2 PARADIGMA LÓGICO

Lógica + Control = programa

Se basa 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. Ej.: prolog.



Lenguajes asociados: Lisp, Prolog, ML

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.

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. Diferencias entre Lenguajes y Paradigmas de Programación

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 procedimental (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)

Declaratividad y expresividad

Declaratividad y expresividad son conceptos diferentes. Decimos que una solución es más expresiva que otra si resuelve un problema en forma más clara y menos compleja (y por consiguiente es más fácil de entender). Por otro lado una solución es más declarativa que otra en tanto se tiene menor grado de

información sobre los detalles algorítmicos de cómo se resuelve y donde la idea de secuencia pierde relevancia.

Como la expresividad depende de algunos criterios que son subjetivos no siempre una solución declarativa es más clara que otra no declarativa.

4. LENGUAJES DE PROGRAMACIÓN

Para obtener una solución utilizando un determinado paradigma se necesita de un Lenguaje de Programación.
Pero con demasiada frecuencia, los lenguajes son seleccionados por razones equivocadas tales como: el fanatismo ("... es genial"), los prejuicios ("... es una basura"), la inercia ("... Es demasiado problema para aprender"), el miedo al cambio ("... es lo que mejor sabemos hacer, con todos sus defectos"), la moda ("... es lo que todos están utilizando ahora "), las presiones comerciales (" ... con el apoyo de MM. "), conformismo (" nadie fue despedido por la elección de ... "). Tales influencias sociales y emocionales son a menudo decisivas y reflejan un triste estado del software.
A continuación se detallan puntos a tener en cuenta en la correcta selección de un lenguaje de programación.

4.1 SINTAXIS, SEMANTICA Y GRAMÁTICA

4.1.1 SINTAXIS

La sintaxis es el conjunto de reglas que gobiernan la construcción o formación de sentencias (instrucciones) válidas en un lenguaje. La sintaxis de un lenguaje de programación es el aspecto que ofrece el programa. Proporcionar las reglas de sintaxis para un lenguaje de programación significa decir cómo se escriben los enunciados, declaraciones y otras construcciones del lenguaje.

Ejemplo: **Sintaxis -> Forma**

En el lenguaje Java, por ejemplo, la sintaxis de la sentencia if en Java es:

```
if(condicion)
     {
   // sentencias
     }
```

4.1.2 SEMÁNTICA

La semántica es el conjunto de reglas que proporcionan el significado de una sentencia o instrucción del lenguaje. La semántica de un lenguaje de programación es el significado que se da a las diversas construcciones sintácticas.

Ejemplo: **Semántica-> Significado**

En el lenguaje Java, la semántica de la sentencia if, vista anteriormente, significa que al ejecutarse esta sentencia, se evaluará la condición entre paréntesis y si el valor obtenido es verdadero se ejecutará la/s sentencia/s dentro del cuerpo de la misma.

4.1.3 GRAMÁTICA

La definición formal de la sintaxis de un lenguaje de programación se conoce como gramática, en analogía con la terminología común para los lenguajes naturales. Una gramática se compone de un conjunto de reglas (llamadas producciones) que especifican las series de caracteres (o elementos léxicos) que forman programas en el lenguaje que se está definiendo. Una gramática formal es simplemente una gramática que se especifica usando una notación definida de manera estricta.
Cada lenguaje posee su propia gramática con su correspondiente sintaxis y semántica.

Sintaxis	Es el conjunto de reglas que gobiernan la construcción o formación de sentencias válidas en un lenguaje	Forma:
		`if <condición> then`
		`<sentencia>`

Semántica	Es el conjunto de reglas que proporcionan el significado de una sentencia del lenguaje	Significado:
		`Si el valor de <condición> es verdadero, se ejecutará`
		`<sentencia>`

Gramática	Es la definición formal de la sintaxis de un lenguaje de programación. Cada lenguaje posee su propia gramática con su correspondiente sintaxis y semática.	

4.2 CRITERIOS DE EVALUACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

4.2.1 LEGIBILIDAD

Facilidad de Lectura/Comprensión de los programas:
El software hay que mantenerlo (corregir errores, introducir cambios o mejoras,...), por lo tanto, el lenguaje debe facilitar la comprensión de los programas una vez escritos.
¿El lenguaje ayuda u obstaculiza la buena práctica de programación? Un lenguaje que aplica la sintaxis críptica, identificadores muy cortos, las declaraciones por defecto, y la ausencia de información de tipo hace que sea difícil de escribir código legible.

Simplicidad del lenguaje:
Es el número de componentes básicos y criterios de combinación para generar las estructuras de control y de datos del lenguaje.

Estructuras de control:
Permiten seguir el flujo del programa.

Tipos de datos y estructuras:
Definen los valores posibles de los datos y su administración.

Consideraciones sintácticas:
Identificadores, palabras reservadas.

4.2.2 CODIFICACIÓN

Facilidad de Escritura/Codificación de los programas:
Debe ser "fácil" utilizar el lenguaje para desarrollar programas que se ajusten al tipo de problemas para el que está orientado.

Simplicidad del lenguaje:
Pocas estructuras permiten codificar sistemas complejos.

Soporte para la abstracción de algoritmos y datos:
Ejemplo: subrutinas (funciones)

Expresividad del lenguaje:
Ejemplo: Un bucle for se puede construir con un bucle while, pero la estructura for es más expresiva en muchos casos.

4.2.3 FIABILIDAD

Un programa es fiable si realiza sus especificaciones en cualquier condición

¿Está el lenguaje diseñado de tal manera que los errores de programación pueden ser detectados y eliminados lo más rápidamente posible? Los errores detectados por controles en tiempo de compilación garantizan su ausencia al ejecutar el programa, sería lo ideal.
Los errores detectados por controles en tiempo de ejecución minimizan problemas, como perdida o corrupción de datos.

Comprobación de tipos:
Un lenguaje de programación específica que operaciones son válidas para cada tipo de dato. Esto nos permite detectar en forma temprana errores: Por ejemplo: Acceso incorrecto a memoria, mal uso de estructuras etc.
Es un recurso importante para evitar errores de ejecución.
Una comprobación estricta incrementa la robustez del programa.

Manejo de excepciones:
Capacidad del programa para interceptar errores de ejecución y tomar medidas adecuadas.

4.2.4 ESCALABILIDAD

La escalabilidad es una propiedad que indica su habilidad para reaccionar y adaptarse sin perder calidad, o bien para hacerse más grande sin perder calidad en los servicios ofrecidos. En el ámbito de los lenguajes de programación debería centrarse en los siguientes aspectos:

¿El lenguaje soporta el desarrollo ordenado de programas a gran escala? El lenguaje debe permitir que los programas se construyan a partir de unidades de compilación que han sido codificadas y probadas por separado, tal vez por diferentes programadores. La compilación separada es una necesidad práctica, dado que las inconsistencias de tipo son menos comunes dentro de una unidad de compilación (escrito

por un programador) que entre las unidades de compilación (tal vez escrita por diferentes programadores), y estos últimos no son detectados por la compilación independiente.

4.2.5 MODULARIDAD

El lenguaje soporta la descomposición de programas dentro de unidades de programas adecuadas, de tal manera que podamos distinguir claramente entre lo que una unidad de programa hace y cómo se codificaran. Esta separación de las preocupaciones es una herramienta intelectual esencial para la gestión del desarrollo de programas de gran envergadura. Conceptos pertinentes aquí son los procedimientos, paquetes, tipos abstractos y clases.

4.2.6 REUTILIZACIÓN

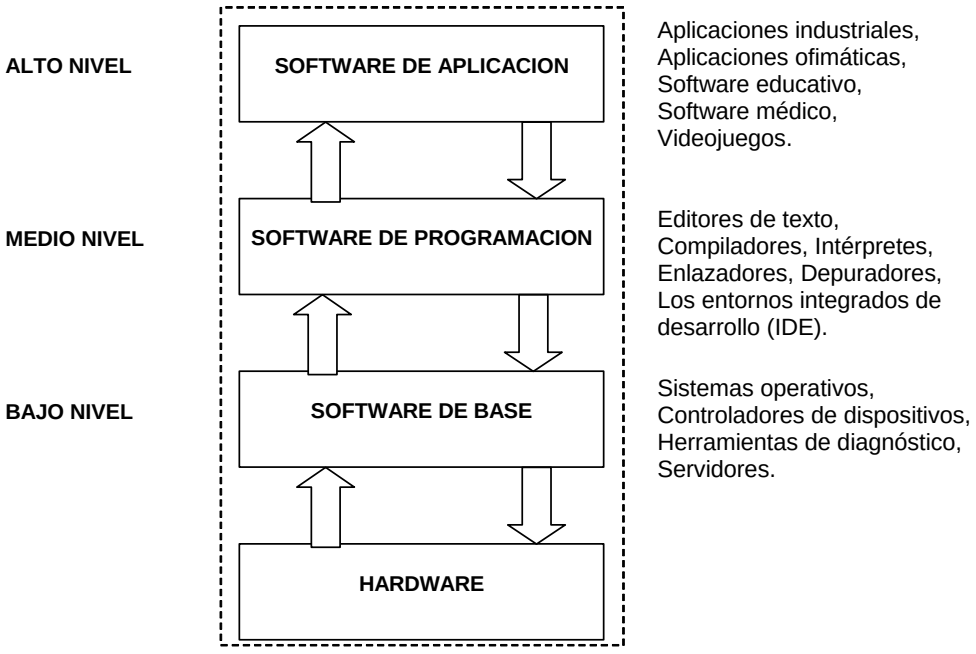
¿El lenguaje soporta la reutilización efectiva de las unidades de programa? Si es así, el proyecto puede optimizar sus tiempos de culminación por reutilización, probando y probando las unidades de programa, también se pueden desarrollar nuevas unidades de programas teniendo en cuenta su reutilización futura. Conceptos relevantes aquí son paquetes, tipos, clases abstractas y particularmente las unidades genéricas.

4.2.7 PORTABILIDAD

¿El lenguaje ayuda u obstaculiza la escritura de código portable? En otras palabras, ¿El código puede ser movido desde una plataforma para una plataforma diferente, sin grandes cambios?

4.2.8 NIVEL

¿Tiende el lenguaje a fomentar a que los programadores piensen en términos de alto nivel de abstracciones orientado a la aplicación? ¿O fuerza a los programadores a pensar todo el tiempo en detalles de bajo nivel, tales como bits y punteros? El código de bajo nivel es notoriamente propenso a errores, especialmente cuando se trata de punteros. Es, sin embargo, necesario en algunas partes de algunas aplicaciones.



4.2.9 EFICIENCIA

La eficiencia en términos generales se puede definir como la capacidad para realizar o cumplir adecuadamente una función.

En el contexto de los lenguajes de programación se dice que no existe un lenguaje de programación mejor que los demás, sino que hay que escoger el lenguaje más apropiado para cada tarea ("use the right tool for the job"). Uno de los factores que influyen en la eficiencia son el tiempo de ejecución y los recursos consumidos por un programa (memoria, acceso a la red, etc.) y dependen sobre todo del algoritmo utilizado. Por otro lado el mismo algoritmo puede comportarse de manera muy diferente dependiendo el lenguaje de programación utilizado para su implementación.

La eficiencia del lenguaje puede depender de parámetros cómo cuál es la plataforma sobre la que se ejecuta, la versión de las librerías, la versión del compilador/intérprete/máquina virtual y los parámetros de optimización que se le suministran. Por esta razón la eficiencia de un lenguaje de programación puede variar con el tiempo.

Hay que tener en cuenta que un lenguaje puede ser muy eficiente para un tipo de problemas en concreto y ser muy ineficiente para otros. Por ejemplo, Perl está pensado para el tratamiento de cadenas de caracteres y Fortran para el cálculo matemático.

4.2.10 MODELADO DE DATOS

¿Proporciona el lenguaje los tipos, y operaciones conexas, que son adecuadas para la representación de las entidades en el área de aplicación pertinente? Ejemplos de ello son los registros y archivos en el procesamiento de datos, los números reales y matrices; las cadenas en el procesamiento de textos; listas, árboles, y las asignaciones de los traductores y de las colas en el funcionamiento de sistemas y simuladores. Si el lenguaje en sí mismo carece de los tipos necesarios, permite a los programadores definir nuevos tipos y operaciones que modelen las entidades de manera precisa en el área de aplicación, los conceptos relacionados aquí son tipos abstractos y clases.

4.2.11 MODELADO DE PROCESOS

¿El lenguaje proporciona las estructuras de control que son adecuadas para modelar el comportamiento de las entidades en el área de aplicación pertinente? En particular, nosotros necesitamos de la concurrencia para modelar los procesos simultáneos en los simuladores, sistemas operativos y controladores en tiempo real del proceso.

4.2.12 DISPONIBILIDAD DE COMPILADORES Y HERRAMIENTAS

¿Son de buena calidad, los compiladores disponibles para los lenguajes? Un compilador de buena calidad aplica la sintaxis del lenguaje y las reglas de tipo, genera código objeto correcto y eficiente, genera en tiempo de ejecución los controles (al menos como una opción) para atrapar a los errores que no pueden ser detectados en tiempo de compilación, e informa de todos los errores de forma clara y precisa. Además, dispone de un ambiente de desarrollo integrado (IDE) de buena calidad para el lenguaje? Un IDE mejora la productividad mediante la combinación de un editor de programas, compilador, enlazador, depurador, y relaciona herramientas dentro de un sistema integrado simple.

4.2.13 FAMILIARIDAD

¿Están los programadores familiarizados con el lenguaje? Si no, es una formación de calidad disponible, y la inversión en formación se justifica en futuros proyectos.

4.2.14 COSTO

Depende de sus características referente a:

- Costo de aprendizaje
- Costo de codificación
- Costo de compilación
- Costo de ejecución
- Costo de uso del lenguaje (compiladores, entornos, ...)
- Costo de fiabilidad (aplicaciones críticas)
- Costo de mantenimiento de los programas

4.3 RESEÑA HISTÓRICA Y EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Tras el desarrollo de las primeras computadoras surgió la necesidad de programarlas para que realizaran las tareas deseadas.

Los lenguajes más primitivos fueron los denominados lenguajes máquina. Como el hardware se desarrollaba antes que el software, estos lenguajes se basaban en el hardware, con lo que cada máquina tenía su propio lenguaje y por ello la programación era un trabajo costoso, válido sólo para esa máquina en concreto.

El primer avance fue el desarrollo de las primeras herramientas automáticas generadoras de código fuente. Pero con el permanente desarrollo de las computadoras, y el aumento de complejidad de las tareas, surgieron a partir de los años 50 los primeros lenguajes de programación de alto nivel.

4.3.1 EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

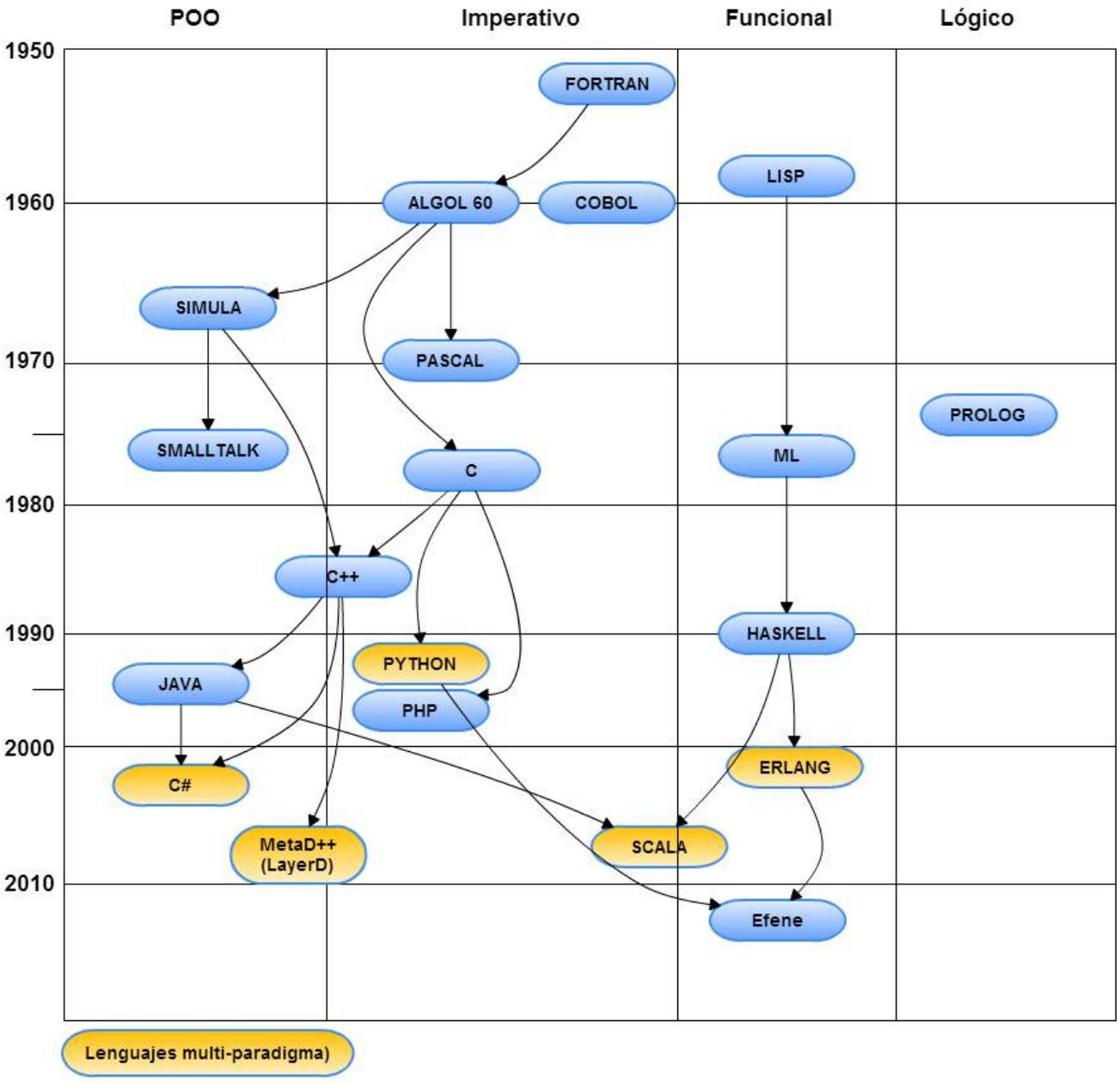


IMAGEN 2. RESEÑA HISTÓRICA Y EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

4.4 TIPOS DE LENGUAJES: HÍBRIDOS Y PUROS.

Cuando se crea un lenguaje de programación, se pueden seguir e implementar todas las características definidas en un paradigma, o incorporar además, características adicionales desde otros paradigmas.
Los lenguajes de programación puros son los que mantienen las características propias del paradigma, esto significa que son estrictos en su implementación. En cambio, un lenguaje de programación hibrido agrega características propias de otro paradigma distinto. Por ejemplo:

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.
De ahí la clasificación en:

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

5. BIBLIOGRAFÍA

- David A. Watt, William Findlay - 2004 - Programming Language Desing Concepts
- Robert Sebesta - 2011 - Concepts of Programming Languages
- Ghezzi, Carlo - Mehdi Jazayeri - 1996 - Programming Languages Concepts - 3a Edición
- Spigariol, Lucas – 2008 – Apunte de Paradigmas de Programación – UTN, FRBA