UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

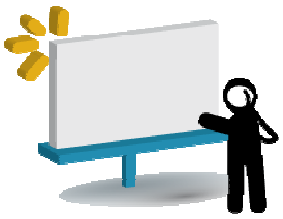
Centro de e-Learning

FUNDAMENTOS DE LA PROGRAMACIÓN

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

MÓDULO 1 - UNIDAD 1

CONCEPTOS INICIALES



Presentación:

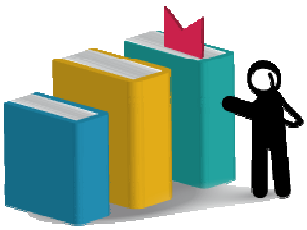
En esta primera Unidad nos enfocaremos en comprender cuales son los elementos esenciales de la informática, a modo de resumen, que servirá como base para adentrarnos oportunamente en los aspectos principales de la programación de software.

Inicialmente definiremos el concepto de software y su clasificación. Posteriormente veremos cuáles son los componentes necesarios para que ese software pueda funcionar. Pero antes de esto último daremos un marco a lo que se refiere la actividad de programar o desarrollar software en forma propiamente dicha.

Objetivos:

Que los participantes:

- Comprendan qué es el software, para qué sirve y para qué se utiliza
- Incorporen el concepto programación y sus principales variantes
- Comprendan cuáles son los componentes necesarios para programar
- Comprendan los conceptos esenciales de los lenguajes de programación en cuanto a sus características generales principales
- Conozcan las generaciones de los lenguajes de programación
- Conozcan los tipos de lenguajes de programación según la forma en que estos se ejecutan
- Conozcan los paradigmas en los cuales están basados los principales lenguajes de programación
- Comprendan como una computadora representa internamente la información



Bloques temáticos:

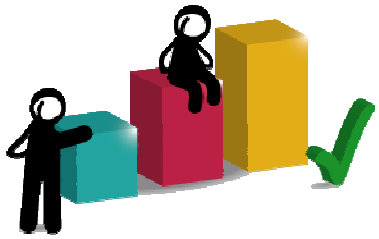
1. Objetivos del curso
2. ¿Qué es programar?
3. ¿Qué es software? ¿Qué se puede hacer?
4. ¿Qué se necesita para programar?
5. Partes componentes

5.1 Hardware

- 5.2 Sistema operativo
- 5.3 Software de base
- 5.4 Aplicación / programa
- 5.5 Lenguaje de programación
- 5.6 Compilador / intérprete / máquina virtual

6. Conceptos de lenguajes de programación
7. Generaciones

- 7.1 Lenguajes de primera generación
- 7.2 Lenguajes de segunda generación



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

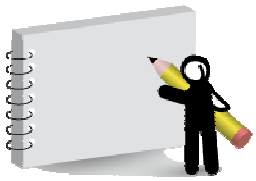
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

* El MEC es el modelo de E-learning colaborativo de nuestro Centro.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Objetivos del curso

La programación o desarrollo de software tiene una reciente, breve y muy rica historia, debido, en esencia, a la predisposición de la sociedad contemporánea hacia la tecnología y su consecuente masificación.

En la informática existen dos componentes principales: hardware y software, siendo este último el más heterogéneo y, a su vez, el menos estandarizado a nivel productivo. Más allá del propósito final que se le quiera dar a un componente de software o su aplicación, todos los desarrollos tienen como bases fundacionales la necesidad de modelar en forma abstracta una realidad dentro de un contexto determinado, a través de la utilización de soluciones surgidas del razonamiento lógico, llamadas algoritmos.

Los conceptos fundacionales son aplicables a los distintos lenguajes de programación, plataformas, entornos y metodologías de desarrollo de software. Teniendo claros los Fundamentos de la Programación, este curso presenta los aspectos esenciales y necesarios para sentar las bases del pensamiento resolutivo que permitirá a los alumnos iniciar el camino hacia las distintas técnicas de desarrollo de software, como por ejemplo, las especializaciones en los lenguajes Java/JEE, MS .NET, PHP y muchos otros lenguajes actuales.

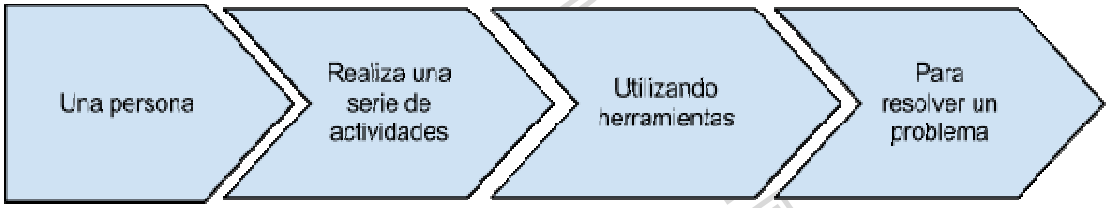
Para esto, durante el curso nos enfocaremos en comprender los principales paradigmas de programación a través del planteo del conceptos teóricos que serán llevados a la práctica en formato de pseudocódigo, lo que evitará que el alumno acote su visión resolutiva de los "problema programáticos" en una única solución, en un único lenguaje de programación. Por el contrario, el pseudocódigo presentado en este curso abrirá el camino a la especialización sobre cualquiera de las tecnologías antes mencionadas o incluso otras.

2. ¿Qué es programar?

Programar, desarrollar software (o simplemente "desarrollar", dentro de un contexto acorde), escribir aplicaciones o… "codear". Son todos términos que se utilizan indistintamente para describir la actividad que realiza un programador con relación a la creación de una pieza de software, sea un componente, módulo, servicio o aplicación, independiente o relacionada con otros componentes, módulos, servicios o aplicaciones.

Se trata ni más ni menos de aplicar una serie de técnicas para la resolución de problemas en base al razonamiento lógico y con distintas herramientas, para resolver un problema en particular, como puede ser la consulta de un cliente o de una historia clínica, o el funcionamiento de un robot o de los frenos de un vehículo.

Podemos decir entonces que (en forma muy resumida) programar consiste en:



Vamos por partes:

- **Una persona**: podemos tomarlo en el sentido literal de un individuo o un equipo de trabajo, con los conocimientos suficientes para realizar la actividad. Inicialmente debería interiorizarse del problema que deberá enfrentar, para poder pensar en una solución.

- **Realiza una serie de actividades**: plantea la solución al problema antes mencionado. Se utiliza el razonamiento lógico, que sirve para realizar un correcto planteo del problema para buscar una solución a posteriori.
- **Utilizando herramientas**: normalmente la actividad de programación en sí se realiza con herramientas especializadas y creadas para tales fines. Por ejemplo, para navegar en Internet usamos un browser o explorador, como Internet Explorer/Edge, Firefox o Chrome, entre otros. Para escribir un documento de texto usamos el MS Word o los Google Docs. De aquí se desprenden al menos 2 preguntas:
    - La primera sería: si tengo tantas opciones para hacer lo mismo, ¿cuál es la opción indicada? Lamentablemente, no hay una respuesta tajante: depende de:
        - a) Lo que se necesite hacer (por ejemplo, si necesitara escribir un documento que sea visto online por otras personas, utilizaría Google Docs ya que MS Word no lo permite).
        - Y por otro lado, y no menos importante: b) el gusto personal. En caso de similitud de prestaciones y/o costo-beneficio, me inclinaré a utilizar la herramienta con la cual me sienta más cómodo.
    - La pregunta consiguiente sería: ¿utilizo software para escribir software? Sí. Ya sea una compleja interfaz gráfica o un editor de textos en modo ASCII, todo el software es creado en base a otro.

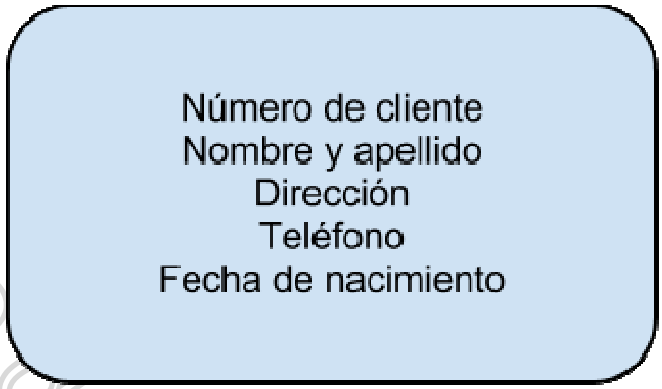
Para esto se utilizan las "IDE" ("Integrated Development Enviroment" o "Entorno de Desarrollo Integrado"). Más información acá:
https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

- **Para resolver un problema**: sin caer en el sentido negativo literal de la palabra "problema", entendemos que si estamos desarrollando un software nuevo es porque existe un requerimiento propio o de un tercero (cliente) de resolver una situación para la cual no se dispone de una aplicación preexistente que la resuelva (o simplemente tenemos ganas de codear y de reinventar la rueda, pero a nuestro modo). Una vez más: si necesito escribir un texto: ¿desarrollo un sistema que me permita realizar esta acción, o abro el MS Word y lo escribo? Nuevamente, deberemos tener en cuenta el contexto y las variables en juego: tal vez no tenga la licencia de MS Word y lo que necesito es escribir una breve carta sin ahondar en formato o funcionalidades complejas. En ese caso, quizá pueda programar un editor de textos sencillos que resuelva mi requerimiento de escribir una breve carta, sin formato, etc.

En la actualidad, nos encontramos con un numeroso grupo que pregona que la programación es un arte, en el sentido de que se lo relaciona con una actividad creativa y artesanal. Ambos conceptos son reales y concretos. Por otro lado, los detractores afirman que se trata de una actividad ingenieril, prácticamente una Ciencia, ya que se utilizan estándares y técnicas probadas y mejoradas con el tiempo y experiencia previa.

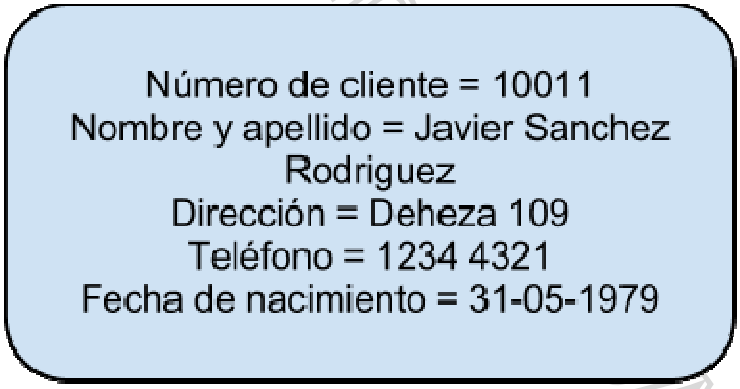
Estos también son conceptos reales y concretos, por lo que se presenta una discusión cuasi filosófica sobre el asunto, donde no parece haber una posición que pueda retractar definitivamente del todo a la otra.

Lo concreto es que la programación es una actividad que reúne todas las características antes nombradas con el fin de **modelar una determinada realidad dentro de un contexto particular.** ¿Por qué decimos que "modela la realidad"? Porque (nuevamente) debemos resolver un problema de nuestro entorno. Por ejemplo, si necesito consultar los datos de un cliente (en el caso de que cliente sea una persona física), me voy a encontrar con que un programa probablemente me muestre los siguientes datos (o más, o menos, dependiendo del requerimiento):

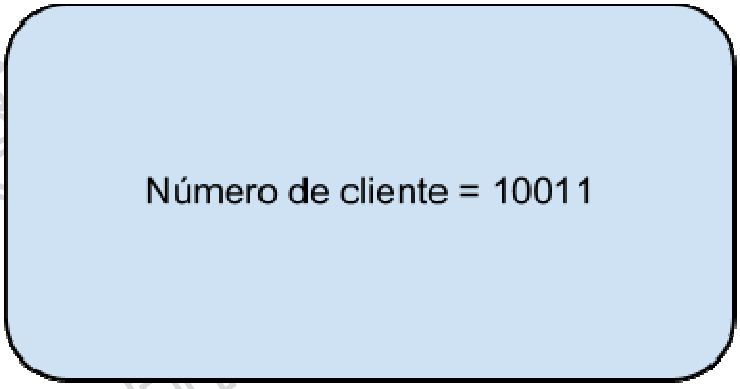


Tenemos a un cliente, que es una persona, sobre la cual contamos con ciertos datos personales. Podemos decir que estamos ensamblando una persona-cliente en base a **datos aislados, que combinados resultan en información**. Esta combinación de datos describe a una persona en particular.

Si les ponemos valores:



En caso de tener como único dato:



No tendríamos información concreta. Si unimos los datos resulta en información de contacto de una persona, en este caso, un cliente. Al ver estos datos unidos, ¿no podríamos decir que tenemos un ejemplo de una persona-cliente?, ¿un modelo abstracto basado en datos?

De esto se trata: ya sea una persona, una cuenta de un banco o una turbina de un avión, todas estas entidades son susceptibles a ser desglosadas en las partes que las componen y en el nivel de detalle que sea necesario. ¿Con qué fin? Mostrar datos de contacto, realizar una transacción online o controlar la velocidad del avión, a través de la manipulación de los datos individuales de cada una de estas entidades.

Arte o ingeniería, lo concreto es que la programación nos brinda la posibilidad de modelar la realidad con el fin de hacer algo con la misma.

ACTIVIDAD DE REFLEXIÓN:

La programación... ¿Ciencia/Ingeniería o Arte? ¿Qué definición le cabe mejor sobre su propia opinión? ¿Qué argumentos tiene en base a una o la otra postura?

Les dejo una cita de Robert C. Martin de su gran obra "Código Limpio" del año 2012:

"No desarrollo un programa de principio a fin en su forma final y, sobre todo, no espero que puedas crear programas limpios y elegantes a la primera. Si algo hemos aprendido en las dos últimas décadas es que la programación es un arte más que una ciencia. Para escribir código limpio, primero debe crear código imperfecto y después limpiarlo. No debería sorprenderle. Ya lo aprendimos en el colegio cuando los profesores (normalmente en vano) nos obligaban a crear borradores de nuestras redacciones. El proceso, nos decían, era escribir un primer borrador, después otro, y después otros muchos hasta lograr una versión definitiva. Para escribir redacciones limpias, el refinamiento debía ser continuado.

Muchos programadores noveles (como sucede con los alumnos) no siguen este consejo. Creen que el objetivo principal es que el programa funcione. Una vez que lo consiguen, pasan a la siguiente tarea, y conservan el estado funcional del programa, sea cual sea. Los programadores experimentados saben que esto es un suicidio profesional."

El autor tiene una visión crítica y una postura muy marcada. ¿Están de acuerdo? ¿Es realmente una forma de arte? ¿En el sentido amplio o con algún tipo de matiz?

Tómense un momento para pensarlo...

UNA POSIBLE INTERPRETACIÓN:

Esta discusión no tiene una única respuesta, por eso, justamente, se plantea. Muchas conclusiones suelen asociarse a apreciaciones sobre la "resolución de problemas", o que la programación es una actividad relacionada con el hecho de "crear" o "imaginar".

Personalmente, creemos que se debe a que la programación es un oficio relativamente nuevo y con sus herramientas y paradigmas están en constante cambio. Eso hace que, como actividad, sea interpretada y reinterpretada sucesivamente. Y en ese juego es donde las opiniones personales (y, por qué no, las modas y tendencias) sostienen a veces que la programación es una ciencia (más bien un tipo de ingeniería) o un arte (incluso se sabe de programadores que hablan en términos de "belleza" del código fuente) o un mix más o menos equilibrado de ambas.

Y es dónde nos sentimos más cómodos, reconociendo una actividad que tiene un componente artesanal, creativo y visual que puede generar subjetividades en términos de agrado/desagrado, pero que también se encuentra enmarcado dentro de ciertas reglas y usos que la asemejarían a un proceso ingenieril.

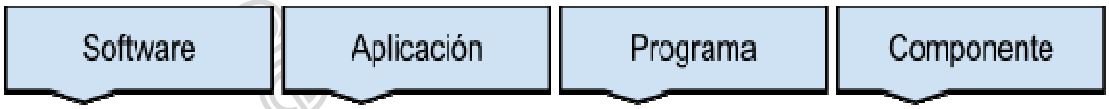
Como verán, las opiniones personales son las que mandan. Los extremos siempre están, claro. Algunos descartan completamente una opción por la otra. Pero también existen muchos términos medios.

Será cuestión que cada uno inicie este camino en el mundo de la programación y que pueda, fundamentado en la experiencia por venir, determinar qué definición le calza mejor.

Seguimos...

3. ¿Qué es software? ¿Qué nos permite hacer?

Cuando hablamos de software, no hablamos de nada literalmente “blando” (soft). De hecho, el software es físicamente, literalmente, intangible. En la actualidad, existe una serie de términos que se utilizan con un mismo fin, normalmente relacionados con la terminología de una tecnología o contexto en particular.



Ya sea un formulario de registración online o un controlador (driver) de un hardware que permite que un sonido sea emitido por una computadora, se trata de lo mismo: software, el "comodín" que utilizamos para definir todo lo intangible que compone la informática.

El software se encuentra en todos lados en la actualidad: en televisores, automóviles, teléfonos, electrodomésticos... no solamente en una PC. Puede estar incluso impreso en un circuito o componente electrónico, algo que se conoce como firmware, aunque el concepto haya evolucionado recientemente.

¿Por qué tenemos software en todo lo que nos rodea? Básicamente, porque la tecnología se ha masificado en forma considerable en las últimas décadas, por lo que surge la necesidad de cubrir los requerimientos de funcionalidad de una infinidad de aparatos. Como **la construcción de software es una actividad relativamente simple (dependiendo lo que se quiera hacer, por supuesto) y mucho más barata que los componentes físicos, se tiende a crear dispositivos con el menor costo posible**. Además de las obvias ventajas de mantenimiento: crear un circuito impreso con determinada lógica permitiría un funcionamiento extremadamente más rápido y a prueba de errores que software, aunque una vez impreso, no se podría modificar, agregarle una nueva funcionalidad o simplemente mejorarlo.

Tenemos en claro, entonces, que el software es creado para un determinado fin que no es posible (por el motivo que fuera) enfrentar o solucionar con un programa preexistente. Este aspecto debe ser evaluado antes de comenzar con el desarrollo, para descartar que exista tal solución previamente creada por un tercero. Este es un ejemplo de la complejidad inherente al mundo del desarrollo: no nos podemos limitar solamente a escribir código. Necesitamos tener en cuenta que existe una serie de actividades que se realizan antes y otras después de la programación. En unidades posteriores nos adentraremos en estas actividades que corresponden a lo que llamamos un "ciclo de vida" de desarrollo de software. Como ejemplo, basta decir por el momento que algunos postulados indican (sin ciencia cierta) que las actividades de programación dentro de un ciclo de vida de un desarrollo de software representan entre el 50% y el 60% del tiempo total; el resto corresponde a estas otras actividades, entre las que podemos incluir (entre otras) el hecho de investigar si nuestro problema planteado ya fue resuelto con anterioridad.

Estas actividades del ciclo de vida se encuentran normalmente enmarcadas dentro de lo que llamamos "proyectos". La definición tácita de **proyecto se refiere a la realización única de ciertos pasos o actividades con el fin de obtener un producto o servicio como resultado.** Se hace hincapié en el hecho de que las actividades son únicas e irrepetibles ya que no existen dos proyectos iguales. Si fuera así, estamos haciendo retrabajo o rehaciendo un producto o servicio preexistente, entrando en falta (por así decirlo) con respecto a lo apuntado el párrafo anterior.

A modo de resumen integrador, podemos decir que la programación es una parte dentro de un ciclo de vida de un proyecto de desarrollo de software, que tiene como objetivo (en nuestro caso) obtener un producto de software único, cuya finalidad es solucionar un problema en particular dentro de un contexto determinado.

4. ¿Qué se necesita para programar?

Como comentamos anteriormente, se requiere de una serie de conocimientos y herramientas que se combinan para obtener el resultado buscado.

Los conocimientos se refieren al hecho no solo de conocer CÓMO se programa, sino de ciertas habilidades de razonamiento lógico, tópico relacionado a la lógica proposicional. **El objetivo del uso de la lógica es realizar planteos sistemáticos que permitan en forma declarativa proponer una solución a un determinado problema, o al menos una parte de él.**

Para ello, en ocasiones nos podemos basar en soluciones probadas y pensadas anteriormente. Esto, a veces, nos facilita componer una solución compleja a través de planteos simples sucesivos y relacionados.

En otras ocasiones, nos encontraremos con soluciones, conocidas como **best practices ("mejores prácticas"). Estos son postulados que se difunden ya sea mediante estándares formales o de facto y que tienen una aceptación general**. Dichas convenciones normalmente plantean soluciones de más alto nivel, que van desde un algoritmo o porción reducida de código hasta grandes frameworks ("entornos de trabajo") que suelen ser software "empaquetado" que aporta soluciones relativamente genéricas.

Desde este punto de vista, podemos pasar de lo micro a lo macro: **la clave siempre es dividir un problema complejo en un conjunto de problemas simples.** De esta forma, atacando los más simples, se logra componer una solución compleja y completa. A través del uso de planteos lógicos, apoyándonos en conversiones que solucionan una parte o un conjunto de problemas simples.

Por otro lado, las herramientas, tienden a facilitar cada vez más el proceso de desarrollo, lo que tan bien explica la masificación de la profesión de la programación a nivel global. Con el correr de los años, las herramientas de desarrollo sumaron en complejidad, integración y uso. Inicialmente surgieron programas, editores de texto, con ciertas funcionalidades que facilitaban la programación. Posteriormente fueron ganando en prestaciones, a lo que se sumaron otras herramientas "satélites" que contribuían a simplificar las tareas. Finalmente, la tendencia en la última década apunta a integrar las actividades de programación propiamente dichas y todas aquellas tareas utilitarias que complementan el desarrollo.

En resumidas cuentas podemos decir que para programar es necesario combinar ciertos conocimientos técnicos y lógicos con las herramientas adecuadas (de hardware y software) para crear o modificar el producto buscado.

5. Partes componentes

Está claro que el software por sí mismo no cumple ninguna función, si no es complementándose con otros componentes que hacen “al todo informático”.

Estos componentes (por utilizar un término generalista) se pueden dividir en:

1. Hardware
2. Sistema operativo
3. Software de base
4. Aplicación/programa
5. Lenguaje de programación
6. Compilador/intérprete/máquina virtual

5.1 Hardware

Definición

Por hardware entendemos todos los elementos físicos de la informática, comúnmente denominados “computadora” y enmarcados dentro de la ciencia de la computación. Es importante diferenciar los términos PC (o computadora) de servidor. El primero se refiere a un tipo de máquina estándar o propietaria orientada al uso hogareño o profesional aunque tendiendo a soportar un uso no intensivo en cuanto a usuarios. Por más poderosa o moderna que resulte una PC genérica o de marca, o una Mac, su intención es prestar servicio a un único usuario, el operador.

El concepto de servidor tiene por sentido representar un tipo máquina que presta servicio a un número indefinido de usuarios, conectados en forma remota simultáneamente.

Actualmente nos encontramos utilizando lo que se denomina tecnología de 3ra generación (a pesar de que algunos autores mencionan que la actual es una 4ta generación, aunque los principios sean los mismos), la cual **basa su electrónica en circuitos integrados. Esto permite integrar cientos de transistores y otros componentes electrónicos en un único circuito integrado impreso en una placa de silicio**, reduciendo así su costo, consumo y tamaño, a la vez que se incrementa su capacidad, velocidad y durabilidad.

Tipos de hardware

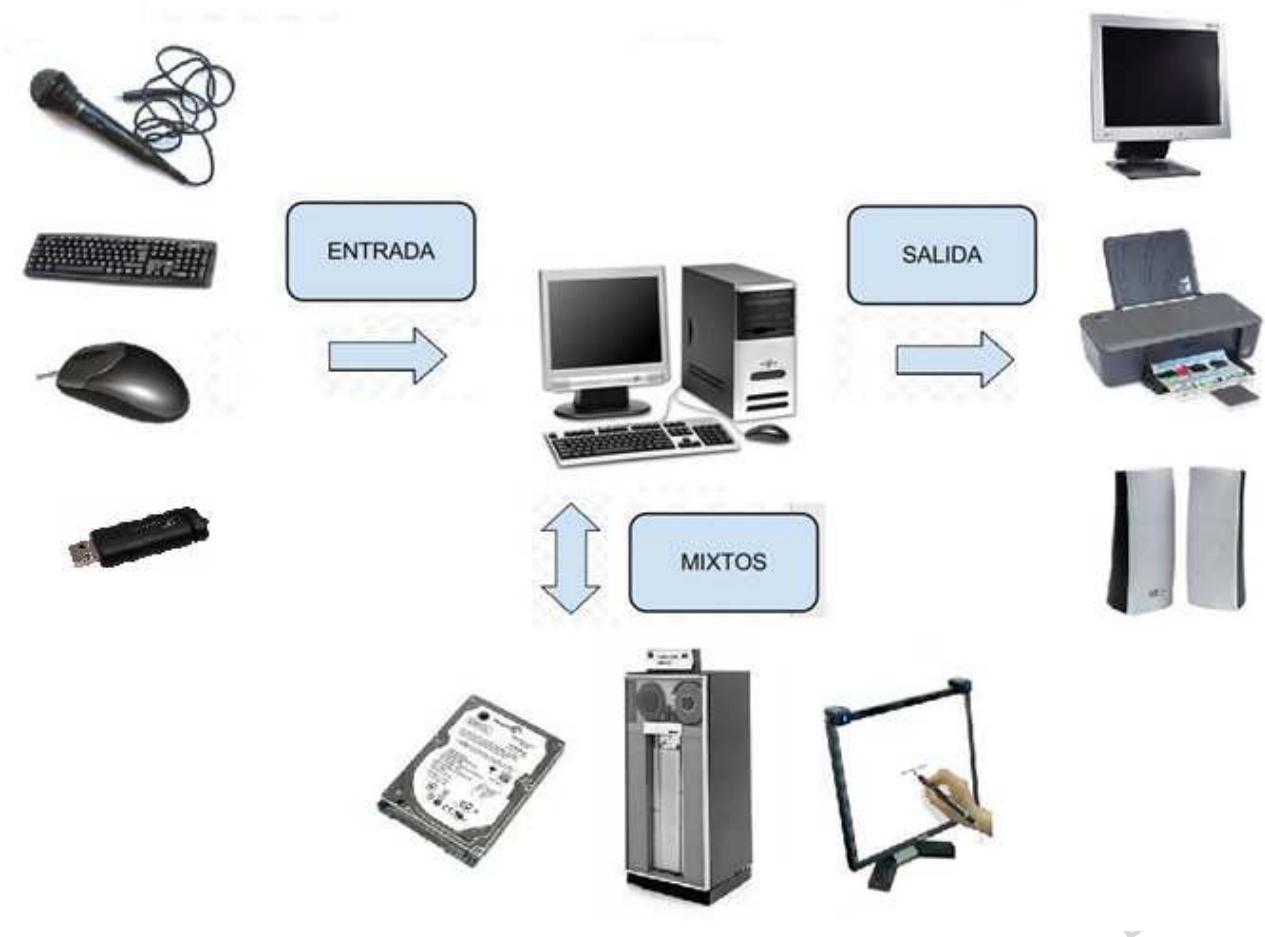
Los componentes principales son la CPU (Central Process Unit o Unidad central de procesamiento) y la memoria. Al primero se lo puede considerar como el "cerebro" de toda computadora. Es el encargado de interpretar y ejecutar todas las operaciones y procesamientos de datos. La función de la CPU la cumplen los microprocesadores, que son ni más ni menos que circuitos integrados. Dependiendo de su función y propósito, una máquina puede tener un procesador (PC) o incluso miles (en el caso de los grandes servidores) de microprocesadores.

Con respecto a la memoria, nos encontramos principalmente con la llamada RAM (Random Access Memory o Memoria de acceso aleatoria, en referencia a que los tiempos de acceso tanto para lectura como escritura son los mismos, a diferencia de las memorias secuenciales). Es el tipo de memoria utilizada en una computadora para el almacenamiento temporal y de trabajo. **En la RAM, se almacena transitoriamente la información, datos y programas que la CPU lee, procesa y ejecuta**. **La memoria RAM es conocida como "memoria principal" o "de trabajo" de la computadora**, en contraposición a las llamadas memorias auxiliares, de soporte, secundarias o de almacenamiento masivo (como discos duros, cintas magnéticas u otras memorias). Una característica de las memorias RAM es su estado de volatilidad, lo cual significa que pierden inmediatamente su contenido al interrumpir su alimentación eléctrica.

Por otro lado nos podemos encontrar con distintos tipos de periféricos, entendiendo por periférico a todo dispositivo o unidad que permita a la computadora comunicarse con el exterior, esto es, tanto ingresar como extraer información. Los periféricos son los que permiten realizar las operaciones conocidas como de entrada/salida. siendo posible agruparlos en periféricos de:

- **Entrada**: son aquellos que permiten el ingreso de información por parte de un usuario o sistema externo, ya sea en forma local o remota. Entre los periféricos de entrada se pueden mencionar: teclado, mouse, escáner, micrófono, cámara web, lectores ópticos de código de barras, lectora de CD/DVD/BluRay, etc.

- **Salida**: son aquellos que permiten emitir o extraer la información resultante de las operaciones realizadas por la CPU. Los dispositivos de salida aportan el medio fundamental para exteriorizar y comunicar la información y datos procesados; ya sea al usuario o a un sistema externo local o remoto. Los dispositivos más comunes de este grupo son los monitores clásicos (los de pantalla táctil no entrarían es esta categoría), las impresoras, los parlantes, etc.
- **Mixtos**: son aquellos dispositivos que cumplen ambas funciones, tanto de entrada como de salida. Típicamente, se puede mencionar como periféricos mixtos o de Entrada/Salida a: discos rígidos, pendrives (aunque se trate también de un tipo de memoria), unidades de cinta magnética, lecto-grabadoras de CD/DVD/BluRay, monitores de pantalla táctil, etc.



5.2 Sistema operativo

Definición

Un sistema operativo (comúnmente conocido por sus siglas "SO") es un software que contiene un conjunto de programas o aplicaciones que **median entre el hardware y los programas que utiliza un usuario para realizar una actividad**, por lo que se puede decir que es una capa de abstracción entre el hardware y el usuario. También es el encargado de administrar los dispositivos del tipo hardware de una computadora o servidor, así como también tiene como función la carga y ejecución de aplicaciones.

Breve reseña histórica

En los años 50 surgen los primeros SO, orientados a facilitar la interacción entre las computadoras y los programadores, dado que estos por ese entonces debían interactuar directamente con el hardware, sin un controlador intermedio. Así surgen los primeros SO, con una funcionalidad que se limitaba a cargar los programas en memoria, gestionar el almacenamiento temporal y simplificar la ejecución de procesos.

En la década de 1960, con la aparición de los circuitos integrados, se logra un incremento exponencial en las prestaciones de las computadoras, obteniendo como principal ventaja la posibilidad de cargar en memoria más de un programa a vez, lo que se conoce como multiprogramación. Así surgen los sistemas operativos como Multics, OS/360 o Unix.

En los años 80, continuando con un mayor poder de procesamiento, surgen sistemas operativos como DOS, Mac OS y MS Windows, siendo estos dos últimos los que mayor evolución y difusión tuvieron, de la mano del GNU/Linux, concebido en la década de 1990.

Actualmente, el mayor desarrollo e innovación en el campo de los sistemas operativos está dado en los dispositivos móviles, con la constante evolución de Android, iOS, Windows Phone (en franca decadencia), entre otros.

5.3 Software de base

Definición

Es un concepto altamente relacionado al mundo del desarrollo de software y la tecnología de la información (IT, Information Technology). Hace referencia a un conjunto de aplicaciones que tienen como función soportar la ejecución de aplicaciones utilitarias.

Los tipos de software que comúnmente se engloban dentro del concepto del software de base, son:

- **Sistema Operativo**: como se mencionó anteriormente, su función es abstraer a los usuarios del funcionamiento interno a nivel hardware.
- **Base de datos**: es un repositorio de datos que, a través de un DBMS (Data Base Management System, o Sistema de gestión de base de datos) permite interactuar con ese repositorio mediante la modificación y el acceso a los datos. Los más difundidos en el mercado local son Oracle, MS SQL Server, MySQL/MariaDB, MongoBD, entre otros.
- **Controladores de dispositivos**: se refiere a todos los controladores a nivel de hardware para dispositivos no soportados en forma estándar por un determinado sistema operativo. Como ejemplo podríamos mencionar un controlador de un sistema de control meteorológico o de una maquinaria industrial.
- **Servidor web**: es un contenedor que permite la ejecución de aplicaciones desarrolladas para la web. Podemos citar como ejemplo el IIS (normalmente utilizado para desarrollos hechos en .NET), Tomcat (comúnmente utilizado para software desarrollado en Java) o Apache (normalmente para aplicaciones web desarrolladas en PHP), entre muchos otros.
- **Servidor de aplicaciones**: es un contenedor que permite la ejecución de aplicaciones desarrolladas bajo ciertos estándares específicos. Normalmente, un servidor de aplicaciones (o application server) se encuentra desarrollado para soportar un tipo de estándar de tecnología. Un ejemplo podría ser referenciar a los

servidores de aplicaciones que implementan los estándares de Java como ser JBoss, Weblogic o Websphere, entre otros.

5.4 Aplicación / programa

Definición

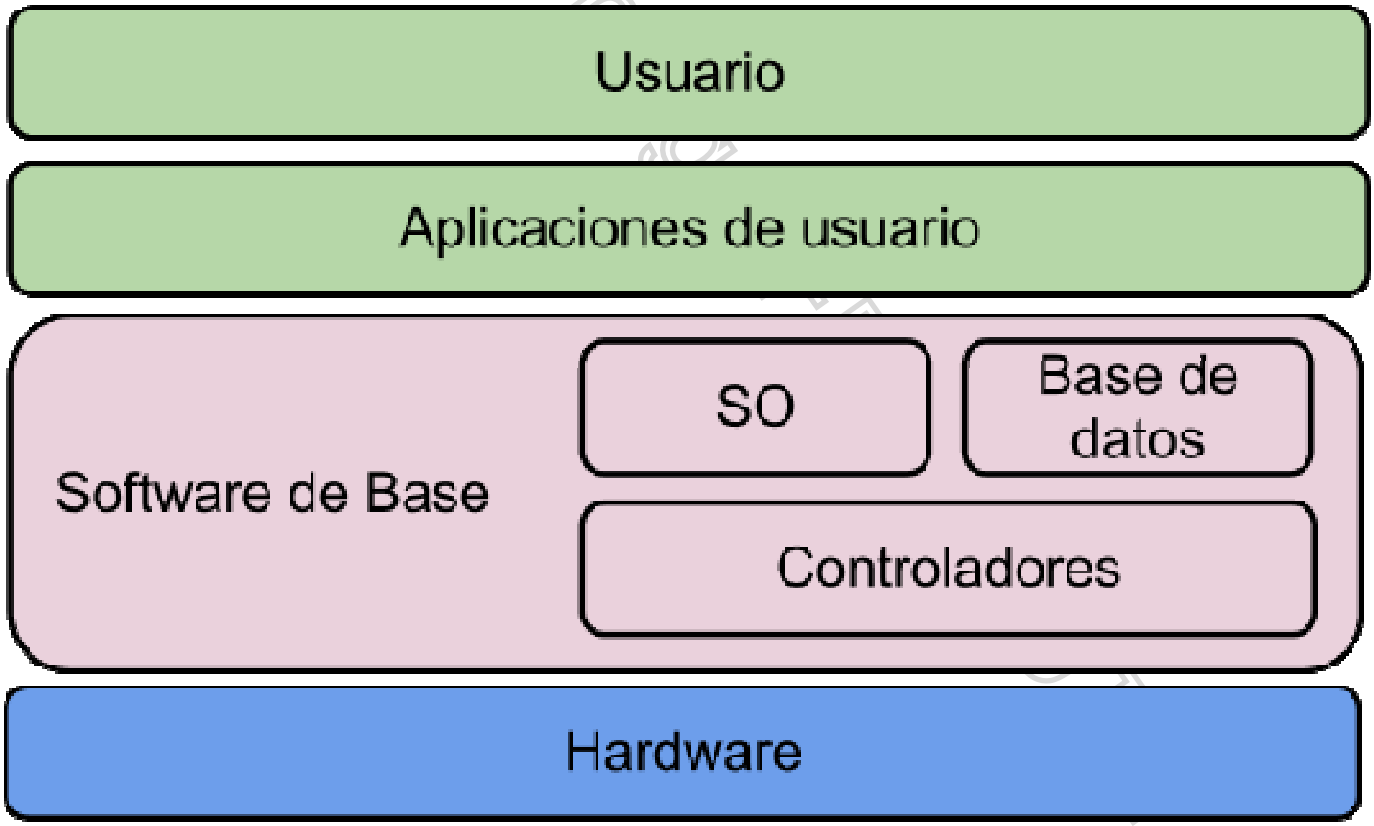
En este punto nos referimos a todos aquellos programas o aplicaciones que utiliza el usuario de una computadora o servidor para realizar una actividad en particular. Los podemos dividir en:

- **Utilitarios**: son todos aquellos programas que (normalmente) fueron desarrollados como una herramienta de soporte genérica, aunque enfocada en alguna actividad particular. Podemos citar planillas de cálculo (como Microsoft Excel), procesadores de texto (como Microsoft Word), diseño gráfico (como Adobe Photoshop), etc.
- **Navegadores**: también conocidos como browsers, permiten acceder a un sitio en la Word Wide Web (www), a través de una dirección web o URL (Uniform Resource Locator, o localizador uniforme de recursos). Entre los más populares se cuentan el Mozilla Firefox, el Chrome de Google, el Internet Explorer de Microsoft y Safari de Apple, además de otros menos conocidos, como Opera.
- **Software de gestión**: todas aquellas aplicaciones desarrolladas con el fin de realizar una actividad o conjunto de actividades de administración o gestión. Se trata normalmente de desarrollos realizados a medida y que suelen cumplir ciertos estándares según su concepción. Podemos nombrar sistemas como Tango o SAP, entre muchos otros.
- **Juegos**: aplicaciones realizadas a fines de entretenimiento, comerciales y/o pedagógicos. En la actualidad se encuentran muy difundidos los del tipo multiplayer o multijugador, que permiten compartir una sesión de juegos entre usuarios localizados en forma remota, sobre una misma plataforma.
- **Otros**: existen innumerables tipos adicionales de aplicativos difíciles de catalogar y agrupar, aunque realicen tareas muy particulares, como ser gestores de archivos

(como Total Commander), herramientas de interacción mediante texto ("chat") o imagen (como Hangout de Google o Skype de Microsoft), gestores de correo electrónico (como Microsoft Outlook), entre muchos otros.

De esta forma podemos ver esta división de componentes como "capas" específicas que median entre el usuario final, quién realiza un orden, y el hardware, quien efectivamente cumple con esa orden, y donde cada una de estas capas cumple con una determinada función, aunque es posible que algunas veces las responsabilidades se superpongan. Un ejemplo de esto podría ser alguna aplicación de usuario que realice en forma interna la administración de de sus datos, salteándose la capa de software de base que podría ser la responsable de hacerlo a través de una base de datos.

Este concepto de capas se grafica a continuación:



¿Se puede definir a la BIOS de una PC como Software de Base, dentro de la categoría de "Controladores"?

La BIOS es un software que está instalado en un chip. Es un tipo específico de software llamado firmware, que se actualiza como cualquier software. Es lo primero que se ejecuta cuando se enciende una máquina y su función es permitir bootear un sistema operativo (que sabemos que es un software de base).

Ahora bien, por definición, un software de base es aquel que controla los dispositivos de la computadora, como almacenamiento, memoria, procesos, etc..

Desde este punto de vista se puede decir que la BIOS corresponde a una instancia previa a la ejecución del software de base.

Por otro lado, los controladores o drivers, están una "capa más arriba": son parte o están relacionados con el sistema operativo y determinados dispositivos. El hecho de estar vinculados a un sistema operativo los pone en una instancia posterior y diferente a la BIOS.

En conclusión, se puede decir que la BIOS no es software de base ni está al mismo nivel que los controladores.

5.5 Lenguaje de programación

Definición

Si bien es un tópico que se abordará exhaustivamente más adelante en esta misma unidad, podemos decir que todos los programas fueron escritos en uno o más lenguajes de programación. **Los lenguajes de programación son un conjunto de símbolos y reglas sintácticas y semánticas que forman una estructura definida.** Aquellos cuya sintaxis se acerca más al lenguaje humano, reciben el nombre de lenguajes de “alto nivel”, los que se acercan más al ordenador son los de "bajo nivel".

El tipo de programación más habitual es aquella en la que se crean programas en un lenguaje de alto nivel (llamado "código fuente"), para después ser convertidos

("compilados") al lenguaje propio que es interpretado directamente por la máquina ("ejecutable").

Al conjunto de actividades, que tiene como parte central el hecho de escribir y/o modificar el código fuente es denominado "programación".

5.6 Compilador / intérprete / máquina virtual

Definición

Nuevamente, estos tópicos serán tratados en esta unidad con mayor profundidad, aunque vale aclarar algunos conceptos iniciales:

- **Compilador**: es un software que se usa para convertir los programas creados en un determinado lenguaje de programación al lenguaje interno de la computadora, también conocido como "código de máquina". En los compiladores, todo el programa original (fuente) se convierte a código de máquina en bloque, y el programa resultante (programa ejecutable) se puede utilizar en otro ordenador sin necesidad de recurrir otra vez al compilador, siempre y cuando se trate de la misma arquitecturas de computadoras y sistemas operativos o que sea compatibles. Por ejemplo, un programa compilador para Windows no va funcionar en un Linux y viceversa (diferencia de sistemas operativos), así como un programa compilador para una Macintosh no va a funcionar en una PC (diferencia de arquitectura de hardware).
- **Intérprete**: es un software con el cual el programa fuente se convierte a código máquina, línea por línea, en el momento en que se ejecuta, sin crear un archivo ejecutable. Por ello es necesario distribuir el programa fuente junto con el intérprete que es capaz de entenderlo.
- **Máquina virtual**: es un software que emula a una computadora y puede ejecutar programas como si fuera una máquina real. Esto permite una gran portabilidad del software desarrollado, aunque requiere de una máquina virtual para cada tipo de

máquina en particular. También es necesario en este caso distribuir la máquina virtual con el software desarrollado.

En este punto será importante diferenciar el concepto de "máquina virtual" relacionado a la programación y cómo un programa puede ser portable entre diferentes plataformas y el concepto de "máquina virtual" utilizado en la administración de la infraestructura tecnológica, también conocido como "virtualización de hardware".

- El primero se refiere a que un programa desarrollado en cierto lenguaje debe contar con una máquina virtual para que se adapte a los diferentes SO y/o hardware, con el objetivo de programar y compilar una sola vez y que ese programa pueda ser portable. Un ejemplo de esto sería Java.

- El segundo se refiere a cierto software que permite crear "máquinas dentro de máquinas". El efecto logrado es que, por ejemplo, en una máquina con Linux pueda ejecutarse, por ejemplo, 3 máquinas con Windows y 2 máquinas con diferentes versiones de Linux. Esto se logra gracias al concepto de "hipervisor" que no es más que una capa de software que controla a las diferentes máquinas virtuales y las abstrae del hardware.

Más información, acá: https://es.wikipedia.org/wiki/Virtualizaci%C3%B3n

¿Para qué sirve una máquina virtual y un servidor de aplicaciones?

Una máquina virtual (VM) en el contexto de un lenguaje de programación es un software que media (comunica) un programa hecho por un programador con un tipo particular de plataforma de hardware y/o sistema operativo, como ya se dijo. ¿Porqué? Básicamente para que el programador programe una vez y se independice de las particularidades de cada plataforma. Ya existen muchas y se van creando cada vez que es necesario nuevas VM para las nuevas plataformas que van surgiendo. Esto lo hacen los "dueños"

*(comunidades, grupos o empresas) de los lenguajes. Cada VM se desarrolla una vez y es el programador el que puede usar esa VM si lo requiere. Esto facilita lo que se denomina como "**portabilidad**", que implica que el programador programe una vez y pueda instalar el mismo software en diferentes dispositivos ("plataformas").*

Por ejemplo, el sistema de manejo de archivos y directorios en Windows es muy distinto al de Linux. Pero el programador que tal vez necesite usar esa funcionalidad no va a querer hacer dos programas diferentes, uno para Windows y otro para Linux. Lo hace una vez y se despreocupa del tema, porque va a poder usar una VM para Windows y otra para Linux y el código que programó va a ser el mismo en ambos casos.

Las VM se aplican (instalan y ejecutan) en el dispositivo que va a ejecutar el programa.

Por ejemplo: una PC y un celular. El hardware de ambos es MUY distinto, pero es posible ejecutar en ambos el mismo programa. ¿Cómo? Haciendo que su programa se ejecute sobre una VM u otra en cada dispositivo.

El concepto de servidor de aplicaciones es un poco más abstracto. Básicamente es un software que provee funcionalidades adicionales a un programa que las necesita. Y como no tiene mucho sentido "empaquetar" el programa con todas las cientos o miles de funcionalidades que tiene un servidor de aplicaciones, se dejan afuera para el programa use las que necesite. De esta forma se logran programas más pequeños (consumen menos recursos de hardware) y no se le agrega un exceso de funcionalidades adicionales que, tal vez, un programa en particular no va a necesitar.

Imaginémonos que vamos al colegio de nuevo. En la mochila podremos llevar algunas cosas, las que necesitamos. La mochila nuestro programa. Pero si necesitamos algo que no tenemos dentro de la mochila, vamos a la librería a comprarlo, ¿verdad? Entonces decimos que la librería es el servidor de aplicaciones. Ahora, imaginemos que llevamos todos los días a todos lados una librería completa. Imposible, ¿no? Además un sinsentido, porque no todos los días vamos a necesitar algo nuevo y jamás vamos a necesitar TODOS los productos de la librería al mismo tiempo... Entonces, separamos la mochila con nuestros insumos básicos y delegamos en la librería los adicionales. Ese sería la relación entre un programa (una aplicación) y un servidor de aplicaciones.

6. Conceptos de lenguajes de programación

Ya definimos a un lenguaje de programación **como un conjunto de reglas, símbolos y palabras especiales que son regidas por una sintaxis y son utilizadas por un programador que escribe una serie de instrucciones que, en su conjunto, constituyen un programa para, con él, darle solución a un problema determinado**.

Este lenguaje es el que permite la interacción entre el programador y la computadora, y logra que ésta realice paso a paso las tareas definidas.

Por ello decimos que un lenguaje de programación es el intermediario entre la máquina y el usuario, para que el último pueda resolver problemas a través de la computadora haciendo uso de un lenguaje que le traduce dicho programa a la computadora para la realización de dicho trabajo.

Existen numerosas tipologías y formas de clasificar los lenguajes de programación. Es importante conocer la evolución de los mismos, para entender su origen y tendencias actuales, a la vez que se los puede agrupar cronológicamente. De esta forma, también logramos apreciar claramente la evolución paralela entre el aumento del poder de procesamiento del hardware junto con la difusión de los lenguajes de programación, debido a las mayores posibilidades que ofrecen y cómo disminuyen la dificultad de uso dado el constante incremento del acceso a la información general.

7. Generaciones

En forma cronológica (pero no necesariamente en forma secuencial), se los puede agrupar en generaciones, como se detalla a continuación:

7.1 Lenguajes de primera generación

Están formados por los lenguajes de máquina, que son aquellos cuyas instrucciones son directamente interpretables y entendibles por la computadora, y no necesitan un artefacto que realice una traducción posterior para que la CPU pueda entender y ejecutar el programa.

Las instrucciones en lenguaje máquina se expresan en términos de la unidad de memoria más pequeña, el bit (dígito binario 0 o 1), que en forma combinada forman una secuencia de bits que especifican la operación y los espacios de memoria implicados en ella. Ejemplos de instrucciones en lenguaje máquina podrían ser:

```
1110 0000 1011 1001
1001 0001 1111 1110
```

Queda claro a simple vista que estas instrucciones serán difíciles (al menos) de entender para un programador, aunque resulten naturales para la computadora. Esto implica que se requiera de un gran esfuerzo (en tiempos y conocimientos) para escribir programas en lenguaje binario o de máquina, por lo que se surge la necesidad de contar con otro lenguaje para realizar la comunicación con la computadora, pero que sea más fácil de escribir y comprender para el programador.

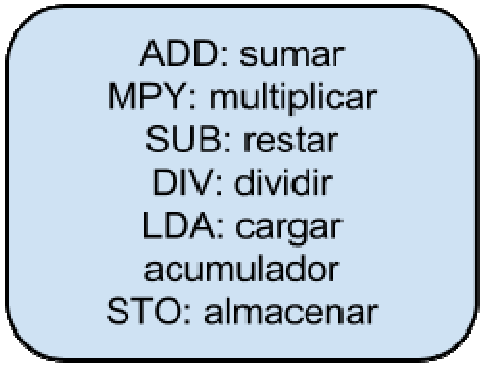
Para evitar la compleja actividad de escribir programas en lenguaje de máquina, se han diseñado otros lenguajes de programación que facilitan la escritura de los programas.

7.2 Lenguajes de segunda generación

La programación en lenguaje de máquina es difícil para el programador, por ello se necesitan lenguajes que simplifiquen el proceso de programación, de modo que el programador pueda concentrar sus esfuerzos en lograr escribir el mejor programa para resolver un problema en particular. Entonces, comienzan a surgir los lenguajes de bajo nivel.

Estos lenguajes están ligados a un tipo particular de máquina, es decir, dependen de un conjunto de instrucciones específicas de la computadora, por lo que no es posible ejecutar un programa de máquina en máquina.

Un ejemplo de lenguaje de bajo nivel es el lenguaje ensamblador (Assembler). En este lenguaje, las instrucciones se escriben en códigos alfabéticos conocidos como mnemotécnicos (abreviaturas de palabras inglesas o españolas). Así, por ejemplo, algunos mnemotécnicos típicos son:



Claramente, vemos la ventaja del uso de las palabras mnemotécnicas, ya que son mucho más fáciles de recordar que las secuencias de dígitos binarios compuestas por 0 y 1. Una instrucción típica en ensamblador puede ser:

```
ADD X, Y, Z
```

Esta instrucción implica que se deben sumar los números almacenados en las direcciones de memoria [x, y] y almacenar el resultado en la dirección de memoria z.

El programa ensamblador posteriormente traducirá la instrucción a código de máquina. Un ejemplo meramente ilustrativo podría ser:

```
ADD se puede traducir a 1110
x se puede traducir por 1001,
y 1010,
z por 1011
```

La instrucción en lenguaje de máquina sería:

```
1110 1001 1010 1011
```

Después de que un programa ha sido escrito en lenguaje ensamblador, se necesita otro programa (denominado justamente “ensamblador”) que lo traduzca a código de máquina.

7.3 Lenguajes de tercera generación

También conocidos como lenguajes de alto nivel son aquellos en los que las instrucciones o sentencias son escritas con palabras similares a los lenguajes humanos (normalmente derivadas del idioma inglés, como es el caso de QuickBASIC), lo que facilita la escritura y la comprensión por parte del programador. Otros ejemplos de lenguajes de tercera generación son: Ada, COBOL, FORTRAN, Modula-2 y Pascal.

Por ejemplo, la línea siguiente es una línea de un programa QuickBASIC:

```
IF (x=z) THEN PRINT "Esto es una prueba"
```

Que, en un lenguaje humano, quiere decir:

Si el contenido de x es igual al contenido de z,
entonces se muestra el mensaje "Esto es una prueba"

Esta porción de código se puede comprender fácilmente conociendo la traducción literal de las palabras inglesas IF (“si...”), THEN (“entonces”), PRINT (“escribir / imprimir”), sin necesidad de mucha explicación.

Una de las principales innovaciones en los lenguajes de alto nivel es que (en general) son transportables, lo que implica que un programa escrito en un lenguaje de este tipo se puede ejecutar con poca o ninguna modificación en diferentes tipos de computadoras. Esto se debe a que dichos lenguajes son independientes de la máquina, es decir, las sentencias del programa no dependen del diseño o hardware de una computadora específica.

7.4 Lenguajes de cuarta generación

Por lenguajes de cuarta generación nos referimos a ciertos entornos de programación formados por **un lenguaje de programación de alto nivel que se parece más al idioma inglés que a un lenguaje de tercera generación, por la sintaxis utilizada. Da una sensación de “conversación” entre programador y máquina**, en contraposición a la secuencia de comandos utilizados en los lenguajes de tercera generación. Algunos restringen a los lenguajes de cuarta generación a los lenguajes orientados a objetos, por una confusión de índole cronológica, ya que esta generación surgió en forma coetánea con la divulgación y propagación de la orientación a objetos.

Ejemplos de estos lenguajes son: NATURAL, PL-SQL, entre otros.

Algunas de las ventajas de estos lenguajes podrían ser que permiten escribir programas en menor tiempo, a la vez que se sufre menos agotamiento y se requiere de menos concentración, lo que conlleva a un aumento de la productividad por parte del programador. Por otro lado, cuando hay que realizar modificaciones o tareas de mantenimiento a los programas previamente elaborados, se aprecia una tendencia a que se simplifiquen estas tareas, debido a una más fácil comprensión y a un menor nivel de concentración requerido.

Por desventajas de estos lenguajes, podemos citar que (generalmente) son menos flexibles al tratarse de “paquetes cerrados” que restringen su uso, a la vez que generan dependencias con proveedores externos (más allá del tipo de licenciamiento que requieran), lo que conlleva la pérdida de autonomía, ya que es común que estos lenguajes utilicen componentes o librerías de terceros, de los que depende la evolución y mejora de los lenguajes.

Abundan los ejemplos de lenguajes prometedores que lograron un auge importante y que finalmente fueron discontinuados por motivos diversos (cierre de las empresas creadoras, fusiones entre compañías, etc.), generado en las empresas usuarias o usuarios finales elevados gastos iniciales y posteriores aún mayores dada la necesidad de migrar o modificar sus programas.

Cabe destacar que estos lenguajes (y en menor medida los de la tercera generación) son los más difundidos y utilizados en el ámbito laboral en la actualidad.

7.5 Lenguajes de quinta generación

Se conoce como lenguajes de quinta generación a aquellos que emplean la programación con restricciones para resolver problemas, en lugar de emplear algoritmos escritos por el programador. La mayoría de los lenguajes basados en restricciones, como los lenguajes de programación lógica y declarativa, son lenguajes de quinta generación. A diferencia de los lenguajes de programación de cuarta generación que son diseñados para construir programas específicos, los de quinta generación son diseñados para que la computadora resuelva un problema sin la necesidad de contar con una persona con conocimientos abundantes de programación. También se los suele asociar con lenguajes de inteligencia artificial, dada su aplicación específica (pero no exclusiva) en este ámbito.

Algunos ejemplos son Prolog, OPS5 y Mercury.

La programación basada en restricciones utiliza ecuaciones para resolver problemas particularmente complejos. Dista de la programación tradicional donde se dan instrucciones directas por otras (en forma de ecuaciones) para manejar estados compuestos por múltiples variables de valor desconocido. Una de las aplicaciones más comunes de este paradigma es en problemas referidos a combinatorias, como planificación de tareas y asignación de actividades en calendarios (scheduling).

ACTIVIDAD DE REFLEXIÓN:

Teniendo en cuenta lo analizado hasta ahora. ¿Cómo definiría, agruparía, caracterizaría los diferentes "niveles" de los lenguajes de programación?

Tómense un momento para pensarlo...

UNA POSIBLE RESPUESTA:

Sobre la cuestión del "nivel" de los lenguajes de programación, que NO está directamente relacionado con las generaciones (error en el que a veces caen algunos): la respuesta es simple: un lenguaje se dice que es de más alto nivel en la medida en que el programador se "olvida" sobre qué plataforma está trabajando y se independiza de restricciones externas como, por ejemplo, la operación para hacer una asignación explícita de memoria para guardar un resultado de una operación.

Algunos autores también hacen referencia a un "nivel medio" o "intermedio" y sobre lo que no coincidimos, aunque reconocemos que las argumentaciones a favor y en contra son igualmente válidas. Lo importante es saber que existe esta tercera opción, más allá de las valoraciones personales.

Seguimos...

8. Lenguajes compilados e interpretados

Generalmente, los lenguajes de programación se suelen dividir en tipos, dependiendo de la forma en que se ejecutan los programas resultantes o, dicho de otra manera, de la forma que tenemos de pasar del código fuente que escribimos a un programa ejecutable:

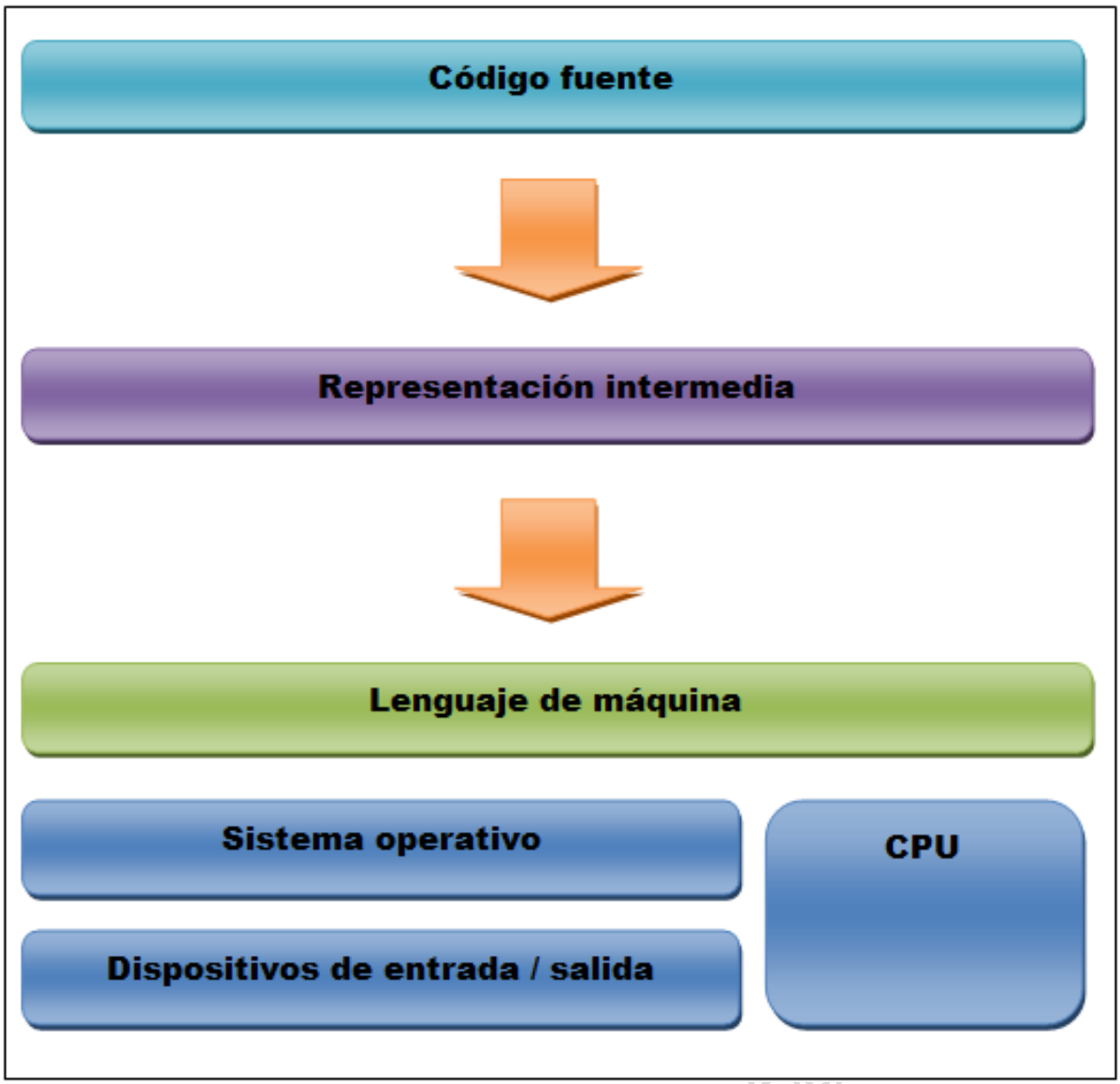
- El código fuente puede ser interpretado.

- El código fuente puede compilarse en un lenguaje de máquina.

- El código fuente puede ser procesado mediante alguna combinación de los dos métodos anteriores.

Las computadoras sólo son capaces de ejecutar lenguaje de máquina. El lenguaje de máquina es lo entiende la Unidad Central de Procesamiento (CPU) y tiene una estructura muy simple: por ejemplo, las instrucciones típicas son buscar un valor a la CPU, almacenar un valor desde la CPU en la memoria, sumar otro valor o comparar estos dos valores y, por ejemplo, si son iguales, saltar a otra instrucción.

El objetivo de cualquier implementación de un lenguaje de programación es traducir un programa fuente al lenguaje de máquina, que es una representación más simple, y para que, de esta manera, pueda ser ejecutado por la CPU. El proceso se puede ver en la siguiente imagen:

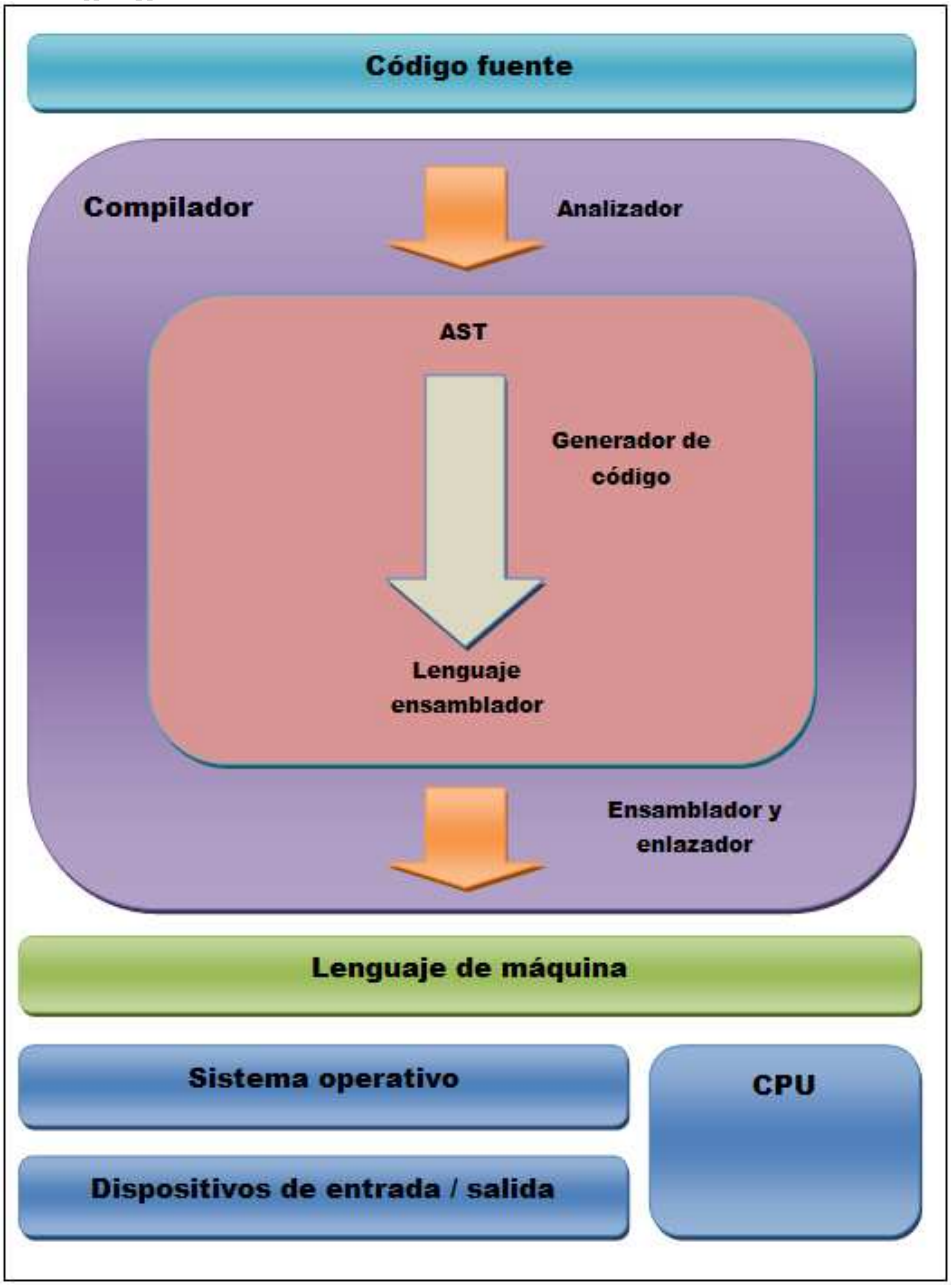


Todas las implementaciones de los diferentes lenguajes traducen un programa fuente a alguna representación intermedia antes de pasar a al lenguaje de máquina. La forma concreta en que se llevan a cabo estas dos traducciones varían significativamente de un lenguaje de programación a otro, pero afortunadamente la mayoría de las implementaciones se basan en una de las pocas metodologías conocidas. Veamos.

8.1 Lenguajes compilados

El método más directo para traducir el código fuente a lenguaje de máquina se llama compilación.

El proceso general se muestra a continuación:



Un compilador es un programa que internamente se compone de varias partes. El analizador ("parser") lee un programa fuente y lo traduce a una forma intermedia llamada árbol de sintaxis abstracta (AST, por las siglas en inglés). Un AST es una estructura de datos en forma de árbol que representa internamente el programa fuente. El generador de código luego toma el AST y se obtiene otra forma intermedia llamada programa de lenguaje ensamblador ("assembly language"). Este programa no es lenguaje de máquina, pero está mucho más cerca de serlo. Finalmente, un ensamblador ("assembler") y un enlazador ("linker") traducen un programa en lenguaje ensamblador a lenguaje máquina, haciendo que el programa esté listo para ejecutarse.

El proceso de compilación

Todo este proceso está encapsulado por una herramienta llamada compilador. En la mayoría de los casos, el ensamblador y el enlazador están separados del compilador, pero normalmente el compilador ejecuta el ensamblador y el enlazador automáticamente cuando se compila un programa, por lo que como programadores tendemos a pensar en un compilador que compila nuestros programas y no necesariamente pensamos en el proceso completo (¡por suerte!).

La programación en un lenguaje compilado es un proceso de tres pasos.

- Primero, se escribe un programa fuente (código fuente). Esto se conoce como "tiempo de programación" o "tiempo de codificación".

- Luego se compila el programa fuente, produciendo un programa ejecutable. Esto se conoce, también, como "tiempo de compilación".

- Finalmente se ejecuta el programa ejecutable. Esto se conoce como "tiempo de ejecución".

Cuando se finaliza la programación y la compilación, se obtiene un programa fuente y un programa ejecutable. Ambos son representaciones diferentes de lo mismo: uno en el lenguaje fuente y el otro en lenguaje máquina.

Si se realizan cambios en el programa fuente, este y el programa en lenguaje de máquina dejan de estar “sincronizados”. Por lo tanto hay que volver a compilar el código fuente para volver a tener esa “sincronización”, esas dos representaciones equivalentes, cada uno en su propio formato. ¿Por qué es esto importante? Porque siempre tenemos que tener en claro que el programa que vemos en ejecución es el mismo que tenemos en forma de código fuente. Esto es imprescindible, por ejemplo, para corregir un error en la ejecución de un programa: necesariamente vamos a tener que estar mirando lo mismo,

pero en sus diferentes representaciones (corregimos el error en el código fuente, volvemos a compilar y en ejecución, ese error, ya no debería aparecer).

El lenguaje de máquina es específico para una arquitectura de CPU y un sistema operativo en particular. Compilar un programa fuente en Linux significa que se ejecutará en la mayoría de las máquinas Linux con una CPU similar. Por lo tanto, no es posible tomar un ejecutable de Linux y ponerlo en una máquina con Windows y esperar que se ejecute, incluso si las dos computadoras tienen la misma CPU.

Los sistemas operativos Linux y Windows tienen cada uno su propio formato para programas ejecutables en lenguaje máquina. Además, los programas compilados utilizan diferentes servicios del sistema operativo para imprimir, leer entradas, obtener fecha y hora y realizar otras operaciones de entrada / salida (E/S). Estos servicios se invocan de manera diferente entre los sistemas operativos. Esto también difiere en Linux y Windows.

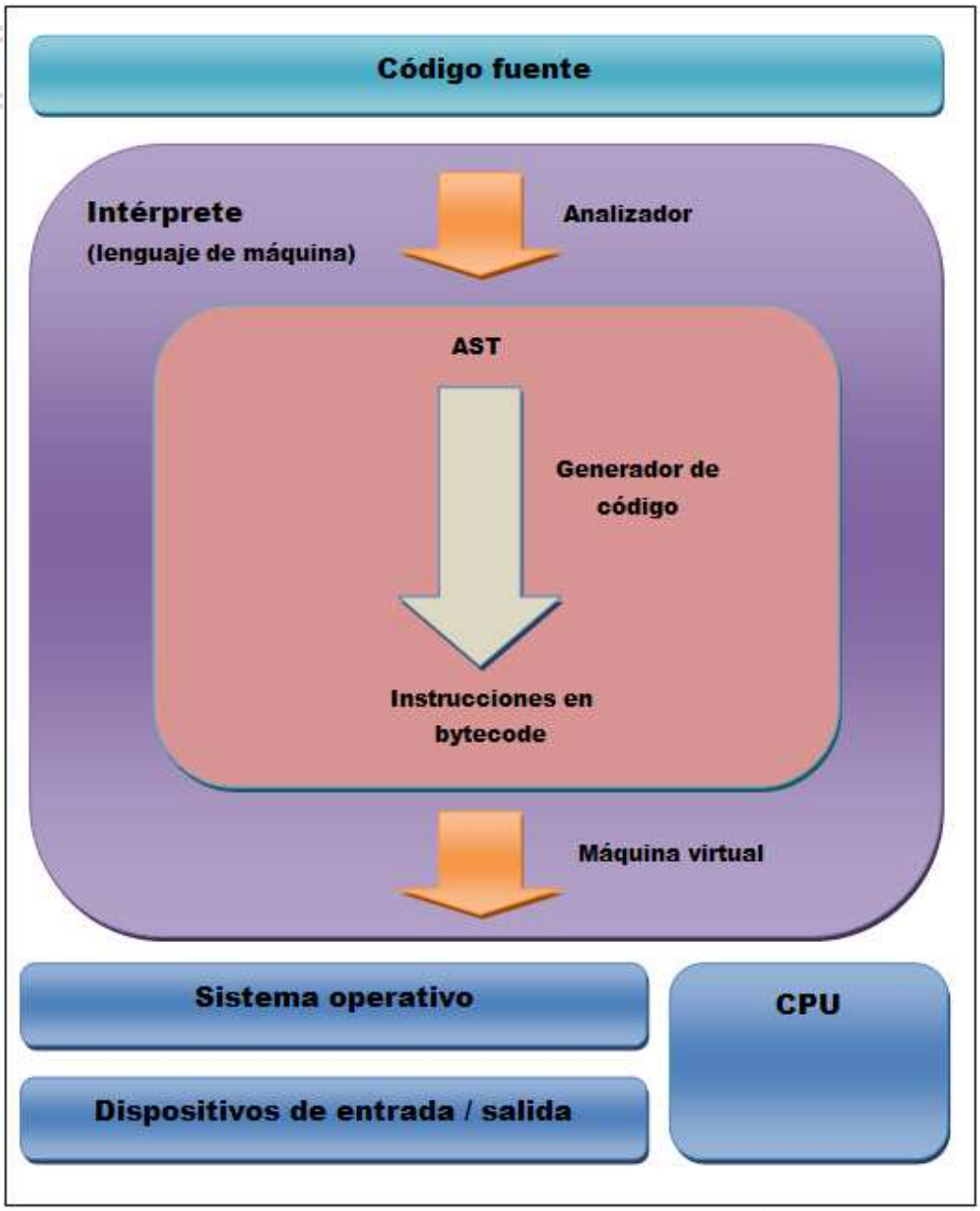
Los lenguajes como C++ van a “ocultarnos” estos detalles de implementación en el generador de código, pero el resultado final es que un programa compilado para un sistema operativo no funcionará en otro sistema operativo sin ser recompilado.

C, C ++, Pascal, Fortran, COBOL y muchos otros son típicamente lenguajes compilados.

8.2 Lenguajes interpretados

Un intérprete es un programa escrito en algún determinado lenguaje y compilado a lenguaje de máquina. El intérprete en sí es el programa en lenguaje de máquina. El propio intérprete está escrito para leer los programas fuente del lenguaje interpretado y así poder interpretarlos (ejecutarlos). Por ejemplo, Python es un lenguaje interpretado. Dicho intérprete está escrito en C y está compilado para una plataforma particular como Linux, Mac OS o Windows. Para ejecutar un programa en Python, hay que descargar e instalar el intérprete que sea compatible con el sistema operativo y CPU donde se va a ejecutar ese programa Python.

Cuando se ejecuta un programa fuente interpretado (como se puede ver en la imagen a continuación) en realidad se está ejecutando el intérprete. El programa no se está ejecutando porque nunca se traduce al lenguaje de máquina.



El proceso de interpretación

La programación en un lenguaje interpretado es un proceso consistente, básicamente, en dos pasos:

1. Primero se escribe un programa fuente.

2. Luego ejecuta el programa fuente ejecutando el intérprete.

Cada vez que se ejecuta el programa, este se traduce a un AST por un componente específico del intérprete llamado analizador (“parser”, en inglés). Puede haber un paso adicional que traduzca el AST a alguna representación de nivel inferior, a menudo llamada bytecode. Luego, una parte del intérprete, llamada “máquina virtual”, ejecuta las instrucciones del bytecode.

No todos los intérpretes traducen el AST a bytecode. A veces, el intérprete interpreta (ejecuta) directamente el AST, pero en general se considera más conveniente traducir el AST del programa fuente a una representación más simple antes de ejecutarlo.

Eliminar el paso de compilación tiene algunas implicaciones.

Por el hecho de tener un paso menos en el proceso de desarrollo, es recomendable ejecutar el código con más frecuencia durante la programación. Esta es una buena práctica y puede acortar el ciclo de desarrollo al hacer pruebas (testing, "testeos") del programa con más frecuencia.

En segundo lugar, debido a que ya no tenemos una versión ejecutable del código, no es necesario administrar dos versiones: solo tenemos un programa (el código fuente).

Finalmente, debido a que el código fuente no depende de la plataforma, generalmente puede moverse fácilmente entre plataformas: el intérprete “aísla” el programa de las dependencias de la plataforma, lo que permite obtener un atributo importante de todo software que es la portabilidad.

Desde ya, los programas fuente (el propio código fuente) en los lenguajes compilados son generalmente independientes de la plataforma. Pero, deben ser recompilados para mover el programa ejecutable de una plataforma a otra. El intérprete en sí no es independiente de la plataforma. Debe haber una versión de un intérprete para cada combinación de plataforma / lenguaje. Por lo tanto, vamos a tener un intérprete de Python para Linux, otro para Windows y otro para Mac OS.

Hay muchos lenguajes interpretados: Python, PHP, Javascript, Ruby, Standard ML, lenguajes de script Unix como Bash y Csh, Prolog y Lisp. La portabilidad de los lenguajes interpretados los ha hecho muy populares entre los programadores, especialmente por permitir que un mismo código fuente pueda ejecutarse en múltiples plataformas.

La desventaja de un lenguaje interpretado está en la velocidad de ejecución. Los programas interpretados generalmente se ejecutan más lentamente que los compilados. En un programa compilado, el análisis y la generación de código ocurren una vez cuando se compila el programa. Al ejecutar un programa interpretado, el análisis y la generación

de código ocurren cada vez que se ejecuta el programa, agregando de esta manera tareas adicionales (memoria y procesamiento) al programa ejecutado.

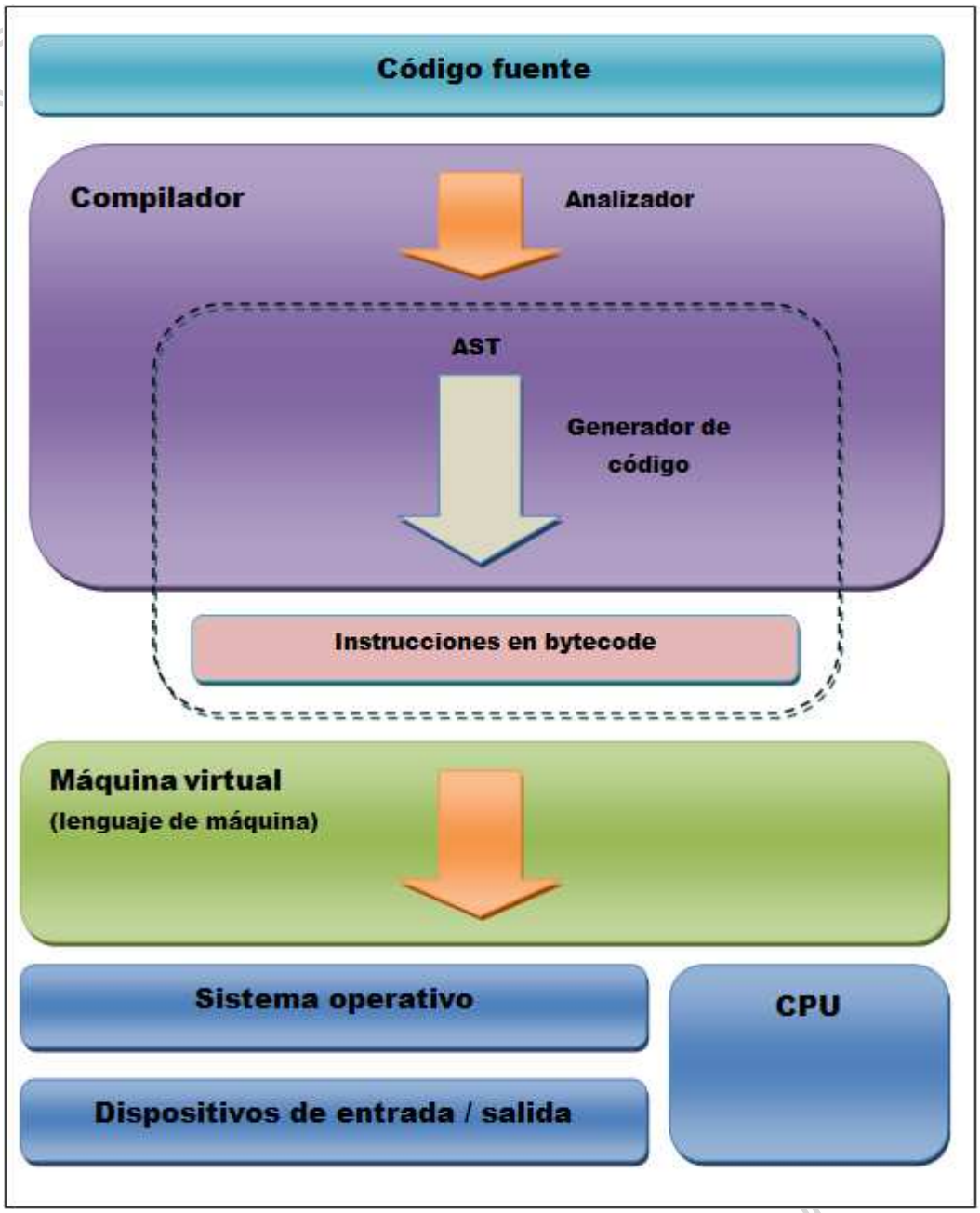
8.3 Máquinas virtuales

Las ventajas de la interpretación sobre la compilación son bastante significativas. Una de las mayores ventajas es la portabilidad de los programas. Es bueno saber de antemano que el código programado se podrá ejecutar en Linux, Microsoft Windows, Mac OS u otro sistema operativo.

Implementación de máquina virtual

Una máquina virtual es un programa que proporciona una "capa de aislamiento" del hardware y del sistema operativo de una máquina, al tiempo que proporciona una implementación consistente de un conjunto de instrucciones de bajo nivel, comúnmente llamadas bytecode.

En la imagen que se muestra a continuación se puede cómo una máquina virtual se encuentra "por encima" (figurativamente) del sistema operativo / CPU para actuar como este "aislante".



No hay una especificación única para las instrucciones de bytecode. Son específicos de la máquina virtual que se define. Por ejemplo, Python tiene una máquina virtual dentro del intérprete. Prolog tiene un intérprete que utiliza una máquina virtual como parte de su implementación. Algunos lenguajes, como Java, han llevado esta idea un paso más allá.

Java tiene una máquina virtual que ejecuta instrucciones de bytecodes al igual que Python. Pero los creadores de Java separaron la máquina virtual del compilador. En lugar de almacenar las instrucciones de bytecode internamente en un intérprete, el compilador de Java, llamado “javac”, compila el código fuente a un archivo de bytecode. Este archivo no contiene lenguaje de máquina, por lo que no se puede ejecutar directamente en el hardware. Es un archivo de bytecodes de Java que es interpretado (y ejecutado) por la máquina virtual Java, llamada “java” y es uno de los programas que están incluidos en el SDK (Software Development Kit) de Java. Los archivos de bytecode de Java se pueden reconocer ya que tiene la extensión .class.

En resumen, el código fuente se compila y se obtiene un programa en bytecode que, para ser ejecutado, debe ser interpretado por la máquina virtual. Al agregar este paso intermedio, el intérprete puede ser más pequeño y más rápido que los intérpretes tradicionales.

Los lenguajes que se incluyen en esta categoría de máquina virtual incluyen Java, Python, C#, Visual Basic.NET, JScript y otros lenguajes de plataforma .NET. Python cuenta con un intérprete interactivo, así como la capacidad de compilar y ejecutar programas de bytecodes de bajo nivel. Los archivos de bytecode de Python se nombran con una extensión .pyc. En el caso de Python, la máquina virtual no es un programa separado, sino que está integrado en el intérprete de Python.

Los entornos de programación Java y .NET no incluyen intérpretes interactivos. La única forma de ejecutar programas con estas plataformas es compilar el programa y luego ejecutar el programa compilado (el bytecode) utilizando la máquina virtual. Los programas escritos para la plataforma .NET se ejecutan bajo Windows y, en algunos casos, Linux. Microsoft presentó algunas de las especificaciones de .NET a la ISO (www.iso.org) para permitir que otras compañías de software desarrollen soporte para .NET en otras plataformas.

La plataforma Java se ha implementado y lanzado en todas las plataformas principales. De hecho, en noviembre de 2006, Sun, la compañía que creó Java, anunció que lanzaría la máquina virtual Java y el software relacionado bajo la Licencia Pública GNU para alentar el desarrollo del lenguaje y de herramientas relacionadas. Desde entonces, Oracle ha adquirido los derechos de Java.

Las implementaciones del lenguaje Java y de los lenguajes de .NET mantienen la compatibilidad con versiones anteriores de sus máquinas virtuales. Esto significa que un programa compilado para una versión anterior de Java o .NET continuará ejecutándose en implementaciones más recientes de la máquina virtual del lenguaje. Por el contrario, la máquina virtual de Python no mantiene la compatibilidad con versiones anteriores. Un

archivo .pyc compilado para una versión de Python no se ejecutará en una versión más reciente de Python.

Mantener la compatibilidad con versiones anteriores de la máquina virtual significa que los programadores pueden distribuir sus aplicaciones Java o .NET sin liberar su código fuente. Esto permite que las aplicaciones .NET y Java se puedan distribuir manteniendo la privacidad del código fuente, dado que la propiedad intelectual es un activo importante de las empresas (y las personas), la capacidad de distribuir programas en forma binaria (los bytecodes) es importante.

El desarrollo de máquinas virtuales hizo que la administración de la memoria y la portabilidad de los programas fueran mucho más fáciles en lenguajes como Java y los diversos lenguajes .NET, al tiempo que proporcionaba un medio para que los programadores distribuyeran programas en formato binario para que el código fuente pudiera mantenerse privado.

9. Sintaxis

Una vez que se aprende a programar en un determinado lenguaje de programación, aprender un lenguaje de programación similar no es tan difícil. Lograr escribir en el nuevo lenguaje requiere mirar ejemplos o leer documentación para conocer sus detalles. En otras palabras, es necesario conocer la mecánica de armar un programa en el nuevo lenguaje. ¿Están los punto y coma en los lugares correctos? ¿Utiliza “begin” ... “end” o usa llaves (“{“ y “}”) para delimitar los bloques de código? Aprender estos detalles se denomina aprender la SINTAXIS del lenguaje. La sintaxis se refiere a las palabras y símbolos de un lenguaje y a cómo escribir los símbolos en un orden correcto, válido y significativo.

SEMÁNTICA es el término que se usa cuando se quiere entender el significado de lo que está escrito. La semántica de un programa se refiere a lo que el programa hará cuando se ejecute. Informalmente es mucho más fácil decir qué hace un programa que describir la estructura sintáctica del programa. Sin embargo, la sintaxis es mucho más fácil de describir formalmente que la semántica. En cualquier caso, cuando estén aprendiendo un nuevo lenguaje, se debe aprender algo sobre la sintaxis y la semántica de dicho lenguaje de programación.

9.1 Terminología

Una vez más, la sintaxis de un lenguaje de programación determina si los programas están bien formados o gramaticalmente correctos. La semántica describe cómo o si dichos programas se ejecutarán.

- La sintaxis es cómo se ven los programas

- La semántica es cómo funcionan los programas

Muchas preguntas que podríamos hacer sobre un programa se relacionan con la sintaxis del lenguaje o con su semántica, pero no siempre está claro qué preguntas pertenecen a la sintaxis y cuáles pertenecen a la semántica. Algunas preguntas pueden referirse a problemas semánticos que pueden determinarse estáticamente, es decir, antes de ejecutar el programa (lo que se conoce como “tiempo de programación”). Otros problemas semánticos pueden ser problemas dinámicos, lo que significa que solo se pueden determinar en tiempo de ejecución (los errores o “bugs”). Muchas veces es difícil de observar la diferencia entre problemas semánticos estáticos y problemas sintácticos.

El código:

```
a = b + c;
```

es la sintaxis correcta en muchos idiomas. ¿Pero es una declaración correcta en C++, por ejemplo?

1) ¿"b" y "c" tienen valores?

2) ¿Se han declarado "b" y "c" de un tipo que permita realizar la operación "+"? ¿O los valores de "b" y "c" admiten la operación "+"?

3) ¿Es una asignación compatible con el resultado de la expresión "b + c"?

4) ¿La declaración de asignación (guardar el resultado de la operación en "c") tiene la forma adecuada?

Hay muchas preguntas que deben responderse sobre esta declaración de asignación y algunas de ellas pueden responderse antes que otras. Cuando se compila el código fuente de C++, este se traduce al lenguaje de máquina. Las preguntas 2 y 3 son cuestiones que pueden responderse cuando se compila el programa C++. Sin embargo, la respuesta a la primera pregunta no se conocerá hasta que el programa se ejecute. Las respuestas a las preguntas 2 y 3 se pueden responder en tiempo de compilación y se denominan problemas semánticos estáticos. La respuesta a la pregunta 1 es un problema dinámico y probablemente no sea determinable hasta el tiempo de ejecución. En algunas circunstancias, la respuesta a la pregunta 1 también podría ser un problema semántico estático. La pregunta 4 es definitivamente un problema sintáctico.

A diferencia de los problemas semánticos dinámicos, la sintaxis correcta de un programa es estática y determinable. Dicho de otra manera, determinar un programa sintácticamente válido se puede hacer sin ejecutar el programa. La sintaxis de un lenguaje de programación se especifica mediante una gramática.

10. Historia de los lenguajes de programación

A mediados del siglo XIX, Charles Babbage concibió las ideas que resultarían en las bases fundamentales de los lenguajes de programación modernos. Babbage, era un inventor ingles y profesor de matemática en la Universidad de Cambridge, que a principios del siglo XIX sentó las bases de muchas de las teorías en las que se basan las computadoras actuales. Describió lo que él denominaba la máquina analítica, pero por falta de elementos tecnológicos de la época, no pudo construirse hasta mediados del siglo XX. Contó con la colaboración de Ada Lovelace, a quien se considera la primera programadora de la historia, ya que realizó programas para aquélla futura máquina de Babbage, utilizando tarjetas perforadas. Las tarjetas perforadas son cartones de medidas y contornos determinados que, al estar **perforadas o no** en un determinado lugar de la misma, representa un código binario, emulando los **dígitos binarios 1 o 0**. Estas tarjetas se utilizaban para ingresar información e instrucciones a una computadora hacia mediados del siglo pasado.

Como su máquina analítica nunca llegó a construirse, los programas de Ada consecuentemente tampoco llegaron a ejecutarse, pero sí se los toma como el punto de partida de la programación, sobre todo si observamos que, en cuanto se empezó a programar, los programadores continuaron utilizando las técnicas diseñadas por Charles Babbage y Ada Lovelace de tarjetas perforadas.

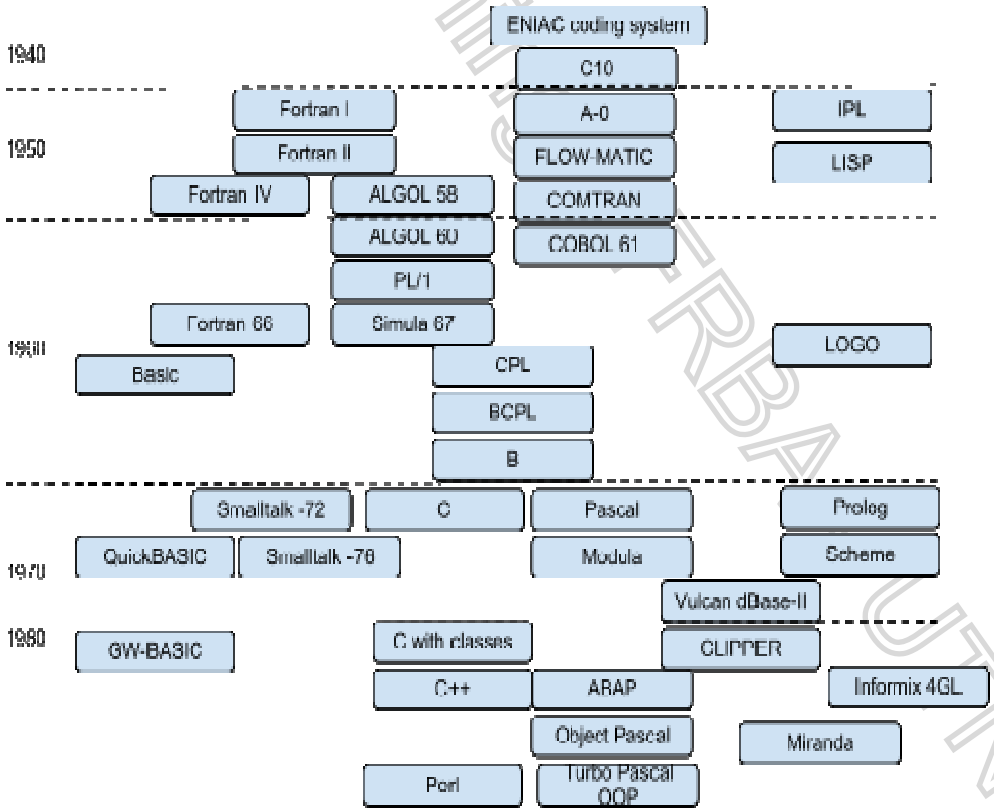
Alrededor de 1820, el gobierno de Inglaterra lo instó a crear una máquina de diferencias: un dispositivo mecánico para efectuar sumas repetidas. Posteriormente, abandonando la máquina de diferencias, Babbage se dedicó al proyecto de su máquina analítica para que se pudiera programar con tarjetas perforadas, gracias a la idea inicial de Charles Jacquard, un fabricante de tejidos francés que había creado un telar que podía reproducir patrones de tejidos, leyendo la información codificada en patrones de agujeros perforados en tarjetas de papel rígido. Entonces Babbage diseñó una máquina que se pudiera programar a través de tarjetas perforadas para realizar cualquier cálculo básico con una precisión de hasta 20 dígitos. Lamentablemente, la tecnología de su época no bastaba para llevar a cabo sus ideas.

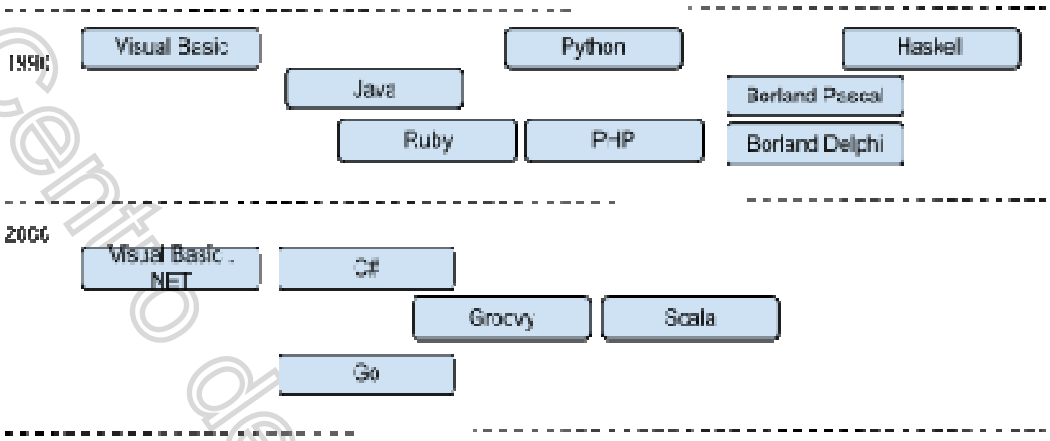
En su diseño, la máquina analítica contaba de cinco componentes o unidades básicas:

1. Unidad de entrada: para ingresar datos e instrucciones.
2. Memoria: para almacenar datos y resultados temporales intermedios.
3. Unidad de control: para coordinar la secuencia de ejecución de las instrucciones.
4. Unidad Aritmético-Lógica: efectúa las operaciones propiamente dichas.
5. Unidad de salida: encargada de comunicar los resultados al exterior.

Por haber sentado las bases de las máquinas modernas, a Charles Babbage se lo conoce como el "padre de la informática", aunque no haya podido ver completa en aquella época la construcción del computador que había soñado y diseñado, dado que faltaba algo fundamental: la electrónica.

Posteriormente, con el avance tecnológico como aliado natural, la programación se fue expandiendo y especializando, a través del surgimiento de lenguajes de programación en forma corriente.





11. Dígitos y representaciones

11.1 Bit

Venimos nombrando oportunamente los conceptos de lenguaje de máquina, código binario, bit, entre otros, para referirnos a lo que figuradamente representamos como 1 o 0, aunque se puede representar de varias formas distintas, siempre en forma metafórica: vaso lleno o vacío, encendido o apagado, blanco o negro, y un sin fin de sinónimos.

El término bit es derivado de la conjunción de los términos inglés **bi**nary digi**t** (dígito binario, en español).

Combinando estos 2 dígitos, se pueden obtener diferentes representaciones. A continuación, podemos ver cómo representar los primeros 16 números binarios:

Decimal:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binario:	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Cada vez que se coloca una cifra binaria más, se duplican las posibilidades. Esto se debe a que se toman todas las posiciones anteriores y se las hace corresponder con un "0" y un "1". Por lo que si se tiene 5 cosas, el total sería 32, con 6 cosas, sería 64, etc.

Por ejemplo, el mayor número decimal que se puede obtener usando 5 cifras binarias es 31.

Porque 11111 es binario es el 31 decimal.

11111 es el mayor número binario que se puede formar combinando 5 bits (las "5 cosas")

Del 0 al 11111 se tienen 32 valores posibles.

Usando 6 cifras binarias (las "6 cosas") el mayor número en binario que se puede obtener es el 111111.

Que es el 63 en decimal.

Entonces desde el 0 al 111111, se obtienen del 0 a 63 en decimal, lo que son 64 valores posibles.

11.2 Byte

Entendemos por byte a una secuencia consecutiva de un número fijo de bits, también conocido como “palabra”. Comúnmente, la utilización de 8 bits, representa 1 byte, aunque existen casos como la serie CDC 6000 de servidores científicos en los que las palabras están compuestas por 6 bits. En otro caso, el PDP-10 utilizaba instrucciones de ensamblado de 12 bits para componer bytes. Los bytes de 6, 7 ó 9 bits se han utilizado en algunas computadoras, por ejemplo, los ordenadores del UNIVAC 1100/2200 series (ahora Unisys) direccionaban los campos de datos de también 6 bits.

Además de los bits y bytes, se cuenta con un conjunto de representaciones utilizadas comúnmente:

Nombre	Abreviatura	Factor binario
bytes	B	$2^0 = 1$
kilo	K	$2^{10} = 1024$
mega	M	$2^{20} = 1\ 048\ 576$
giga	G	$2^{30} = 1\ 073\ 741\ 824$
tera	T	$2^{40} = 1\ 099\ 511\ 627\ 776$
peta	P	$2^{50} = 1\ 125\ 899\ 906\ 842\ 624$
exa	E	$2^{60} = 1\ 152\ 921\ 504\ 606\ 846\ 976$

zetta	Z	$2^{70} = 1\ 180\ 591\ 620\ 717\ 411\ 303\ 424$
yotta	Y	$2^{80} = 1\ 208\ 925\ 819\ 614\ 629\ 174\ 706\ 176$

11.3 Hexadecimal

Finalmente, es importante notar la relación especial entre el sistema binario y el hexadecimal.

Existen 16 dígitos hexadecimales, y sabemos que 4 cifras binarias dan 16 valores posibles. La relación exacta entre ellos sería:

Binario:	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

De esta forma podemos apreciar que la utilización de un único dígito hexadecimal en contraposición con los 4 dígitos binarios requeridos resta complejidad de uso.

Para dar un ejemplo más claro, podemos citar la siguiente comparación:

"100110110100" = "9B4"

El sistema hexadecimal es particularmente utilizado en la actualidad para representar direcciones de memoria que, de otra forma, en binario, sería extremadamente complejo de representar y comprender.

12. Paradigmas

Los lenguajes de programación se pueden clasificar por el paradigma que utilizan, así como antes se diferenciaron por su tipo de ejecución o la generación a la que pertenecen.

Los principales paradigmas son:

12.1 Imperativos

Imperativos o por procedimientos, también se los suele agrupar dentro de los lenguajes regidos por las prácticas definidas como “estructuradas”, a las cuales nos referiremos en detalle más adelante.

Los programas que usan un lenguaje imperativo especifican un algoritmo a través del uso de declaraciones, expresiones y sentencias.

Una declaración asigna un nombre de variable con un tipo de dato, por ejemplo: **"var x: integer;"** que significa que en una porción de memoria a la que se accede haciendo referencia a “x” se va a almacenar un dato numérico del tipo entero.

Una expresión contiene un valor, por ejemplo: 2 + 2 contiene el valor 4.

Finalmente, una sentencia debe asignar una expresión a una variable o usar el valor de una variable para alterar el flujo de un programa, por ejemplo: **"x := 2 + 2; if x == 4 thenhaz_algo();"**. De esta forma, decimos que a la porción de memoria llamada “x” se le asigna el valor 2 + 2 = 4. Posteriormente, se consulta si el contenido de “x” es realmente “4”, y en caso positivo se ejecuta otra instrucción.

Son ejemplos de lenguajes imperativos: C, Fortran, Pascal, BASIC, ASP, entre otros.

12.2 Declarativos

En este caso, los lenguajes regidos por el paradigma declarativo deben especificar los parámetros que el programa debe dar como salida o resultado, aunque se desentiende de cualquier detalle de implementación.

Existen dos subtipos de lenguajes declarativos: los funcionales y los lógicos. Los primeros están más bien orientados a la construcción de programas como funciones matemáticas. En cuanto a los lógicos, se los tiende a utilizar para definir el problema que se quiere resolver (planteo) y dejar los detalles de la solución a cuenta del sistema. **El planteo es definido dando una lista de sub-objetivos, donde cada uno se define dando una lista de sus sub-objetivos, etc. Si al tratar de buscar una solución, un camino de sub-objetivos falla, entonces esa ruta se descarta y sistemáticamente se prueba con otro camino.**

La forma en la cual se programa puede ser por medio de la escritura de código fuente o de forma visual, a través de una interfaz gráfica que permite "arrastrar y soltar" distintos elementos, que, en su conjunto, representan cada uno de estos caminos antes mencionados.

12.3 Orientados a objetos

Los lenguajes de Programación Orientada a Objetos o POO (derivado del inglés Object Oriented Programming - OOP) se basan en el uso de entidades llamadas objetos y en sus interacciones.

Los objetos son entidades que tienen:

- **Estado**: valores internos que pueden tomar, llamados atributos.
- **Comportamiento**: representaciones de funcionalidades que pueden realizar, llamadas métodos.
- **Identidad**: forma que tiene un objeto de diferenciarse de otro.

Las bases sobre las cuales se sienta la POO son:

- **Herencia**: posibilidad de crear jerarquías de objetos.
- **Polimorfismo**: el hecho de realizar operaciones distintas bajo un mismo nombre.
- **Abstracción**: permitir resolver distintos problemas a través de un único modelado de la realidad.

- **Encapsulamiento**: posibilidad de unificar todos los aspectos del modelado de una solución en particular en una única entidad.

Todos estos términos serán desarrollados próximamente en forma más amplia.

Sobre los lenguajes que podemos decir que cumplen con el paradigma orientado a objetos suele haber cierto nivel de discusión. Lo concreto es que uno de los pocos lenguajes creados a tal fin y que cumplen completamente con el paradigma es Smalltalk, aunque existen otros ampliamente difundidos que, o bien se basaron en otros lenguajes a los que se incorporaron las características de la POO o bien fueron creados como lenguajes orientados a objetos, aunque con algunas características híbridas, como ser C++, Java, Objetive-C, Eiffel y muchos otros.

12.4 Funcionales

Compuesto por lenguajes que también se encuentran englobados dentro del paradigma declarativo. Por ejemplo, Scheme, Haskell, ML o Lisp.

12.5 Lógicos

Sus lenguajes también se encuentran dentro del tipo declarativo, como por ejemplo: Prolog.

ANEXO - Preguntas Frecuentes

1) Pregunta: "¿Qué es servidor de aplicaciones? Es un servidor para crear aplicaciones a clientes o es algo más?"

Respuesta: Un servidor de aplicaciones es un software ("A") que permite la ejecución de otro software ("B"), donde "A" contiene funcionalidades que necesita "B". Ejemplos de servidores de aplicaciones son Jboss para Java, por ejemplo.

Formalmente, un servidor de aplicaciones es un software que proporciona un entorno en el que se pueden ejecutar aplicaciones, sin importar cuáles sean o qué hagan. Está dedicado a la ejecución eficiente de procedimientos (programas, rutinas, scripts) para apoyar la construcción de aplicaciones.

2) P: ¿Cuál fue el software primigenio? y ¿Cómo se empezó? De acuerdo al tipo de programación habrá un software inicial? ¿Hay algo que tengan en común todos?

R: Se puede considerar al lenguaje binario como el primer lenguaje, pero lo que tienen en común todos los programas es el lenguajes de máquina que se puede interpretar en forma de dígitos binarios o con notación hexadecimal (tema que veremos en la próxima unidad).

3) P: ¿Los lenguajes de programación de alto nivel siempre terminan traducidos en lenguajes de bajo nivel? Es decir, ¿se podría pensar como fichas de dominó, donde el programador tira la ficha más cercana y amena, y que luego eso va traducir la fichas más lejanas y complejas (pero más cercanas al hardware)?

R: Si bien la analogía no es exacta, se pueden pensar que la última ficha en caer sí es siempre la misma, pero es como una especie de camino de fichas de dominó en forma de "Y": se desde las los extremos superiores (de un lado el alto nivel y de otro el bajo nivel) y siempre terminan ambos caminos en el mismo lugar.

4) P: Al crear una capa de abstracción a través del hipervisor, la maquina virtual:

¿Crea una capa de abstracción para cada MV distinta o estas se encuentran dentro de la misma capa?

Si se ejecutan, por ejemplo, 3 versiones diferentes de Windows en una máquina ¿Cada versión seria una maquina virtual diferente?

R: Depende de la implementación en cada caso, pero en líneas generales el hipervisor forma parte de esa capa de abstracción en la cual se ejecutan las máquinas virtuales.

Sobre el caso de las 3 versiones, si la idea es ejecutar 3 sistemas operativos en un hardware al mismo tiempo, solo es posible hacerlo virtualizando y cada uno de los SO en ejecución será una MV distinta.

5) P: En "Dígitos y representaciones", en las tablas, ¿cómo se relacionan las diferentes representaciones?

R: La primera tabla muestra un ejemplo de cómo cada dígito de la numeración decimal (del 0 al 15), que es la que todos conocemos, se traduce al lenguaje binario que es el lenguaje que utiliza la máquina. Por ejemplo, al escribir el número 6 en un lenguaje de programación de alto nivel la traducción al lenguaje máquina para que lo pueda interpretar es 110, esto sería pasarlo a un lenguaje de más bajo nivel para que la máquina entienda las instrucciones.

El segundo cuadro compara estos dos sistemas de numeración con el hexadecimal, que facilita la escritura del lenguaje binario pudiendo reemplazar un código binario de 4 dígitos por ejemplo por una letra. Su uso principal es hacer más legibles las numeraciones binarias (que son muy extensas).

Sobre cómo convertir números decimales a binarios, hay que dividir el número por dos hasta que llegues al resultado, que se conforma por el último cociente y el resto de cada división de atrás hacia adelante.

Por ejemplo, el "11" en decimal es equivalente al "1011" binario. ¿Cómo llegamos de uno a otro? Así:

11	2		
1	5	2	
	1	2	2
		0	1

Esto se lee:

- **11 dividido 2, es 5 y el resto 1.**
- **5 dividido 2, es 2 y el resto 1.**
- **2 dividido 2, es 1 y el resto 0.**

Tomamos el último resultado y luego los restos anteriores: 1, 0, 1, 1 = 1011

Y para volver al número decimal se hace:

$(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) =$

(1 * 8) + (0 * 4) + (1 * 2) + (1 * 1) =

8 + 0 + 2 + 1 = 11

Esto es apenas un anticipo, porque se verá con más detalle en la próxima unidad.

6) P: ¿Los lenguajes de quinta generación se ubican dentro de los de alto nivel?

R: Es posible, pero no es necesariamente así, no hay una correlación directa entre niveles y generaciones. Son dos formas independientes de categorización. Y no es correcto generalizar en este caso, más allá de coincidencias puntuales.

7) P: ¿El Bytecode se puede tomar como un nivel intermedio que es interpretado por la maquina virtual? En otras palabras, ¿es como una traducción para la maquina virtual que corresponda?

R: Se puede decir que el bytecode es un "paso previo" al lenguaje de máquina. Todavía le falta una instancia más de "transformación".

8) P: ¿Qué es conceptualmente una asignación? Por ejemplo, ¿cómo se interpreta la expresión "a = b + c"? Es posible expresarlo como "a : b + c"

R: El este caso el valor que toma "a" va a ser el resultante de la suma de los valores que contienen "b" y "c". Posteriormente, cuando se hace referencia a "a", se va a estar haciendo referencia al resultado de la operación. SIEMPRE los valores que reciben un valor van a la izquierda del "=" y el valor nuevo u operación van a la derecha del "=".

Dándole números al ejemplo:

Teniendo:

a = b + c

Para:

b = 1

c = 2

Obtenemos:

a = 3

Y no, las operaciones de asignación se hacen únicamente usando "=", y no el ":".

9) P: ¿Qué es el entorno de desarrollo integrado o IDE? Wikipedia dice que es una aplicación informática que busca facilitarle al programador el desarrollo de software, y además dice que consta de diferentes herramientas como un editor de código fuente,

autocompletado inteligente de código y un depurador. Si la herramienta de autocompletado básicamente escribe código, ¿para qué se toma la molestia el programador de desarrollar software?

R: Una cosa es que complete una palabra que se empieza a escribir (como lo hacen muchas aplicaciones actuales) y otra cosa muy distinta es que la IDE adivine qué tiene en la mente el programador: eso no ocurre. De la misma manera que contextualmente aplicaciones como Gmail o Whatsapp puede sugerir correcciones o palabras, ninguna aplicación va a poder escribir un mensaje por sí misma. Solo ayuda a teclear más rápido o mejorar la escritura del usuario.

Llegará ese día en que el software escriba software, más temprano que tarde, pero por ahora no es así: es el programador el que escribe el código, no la IDE.

10) P: Con respecto a la virtualización, ¿un ejemplo de esta puede ser Remote Desktop de Google o Programas como Team Viewer ?

R: No, esos son ejemplos de aplicaciones de gestión remota de equipos. Ejemplos de virtualización de hardware son KVM, Virtualbox, Hyper-V Server, RHV, etc.

11) ¿Cuál es la diferencia entre intérprete y compilador?

R: Los lenguajes compilados toman el código fuente y generan un programa en código de máquina (ejecutable o para máquina virtual), lo que se conoce como "programa objeto". Los lenguajes interpretados toman el código fuente y lo compilan línea a línea y ejecutan cada línea, sin generar código objeto.

Pero en ambos casos hay que separar el "tiempo de programación" (que es cuando se escribe el código) del "tiempo de compilación" (que es cuando el código fuente se envía a procesar). Ambos "tiempos" son determinados por el programador en forma explícita. Un ciclo común puede ser, por decir algo figurativo, "programo una hora, veo si compila (*), si no compila corrijo y pruebo de nuevo hasta que funcione, si compila, sigo programando y repito el proceso en forma iterativa hasta terminar".

Y por (*) nos referimos en forma indistinta a compiladores o intérpretes.

12) P: Varios autores cuando hablan de lenguaje de programación agregaban un nivel medio de programación: ¿esto es únicamente una cuestión de autores o es un nivel el curos no contempla?

R: Son diferentes interpretaciones de diferentes autores. No se rechaza la existencia del nivel medio, pero se lo incorpora en el curso ya que muchas veces parece ser una categoría utilizada únicamente para el lenguaje C, ya que este lenguaje combina los aspectos de ambos niveles, como un set de instrucciones interpretables a simple vista (alto nivel) y características de control y uso de recursos de SO y hardware propios de los lenguajes de bajo nivel.

13) P: Cuando hablamos de que al compilar un código fuente este se traduce a un código máquina que es específico de una (o algunas similares) CPU y sistema operativo ¿es por ello que cuando vamos a descargar un software o aplicación de usuario, se nos brinda la opción de descargar para Windows, MacOS u otro? Es decir, ¿viene a ser el mismo código fuente compilado de diferentes formas?

R: Exacto: cuando te se descarga un instalador y te la da opción de sistemas operativos y sus versiones es porque son compilaciones distintas.

14) P: Cuando se menciona que los lenguajes de quinta generación están enfocado a la resolución de problemas a través de la programación con restricciones, ¿a qué tipo de restricciones está haciendo referencia?

R: Las "restricciones" son ecuaciones matemáticas, básicamente. No se cumple con el esquema del algoritmo tradicional donde se tienen una serie de pasos que se van resolviendo sucesivamente, sino que se aplican ciertos valores o supuestos en forma sucesiva en diferentes ecuaciones para obtener un resultado determinado.

Esto se usa, por ejemplo, en ciertas ramas de la inteligencia artificial, como el procesamiento de lenguaje natural.

15) P: ¿Las instrucciones bytecode y lenguaje ensamblador tienen alguna similitud? ¿O son lo mismo?

R: No son lo mismo, y pueden tener similitudes a nivel conceptual (no son "amigables" para la lectura de una persona, por ejemplo). O, también, podría pensarse que son "instancias intermedias" entre el código fuente y el programa ejecutable. Pero no mucho más. Y estas similitudes ya son bastante forzadas...

16) P: ¿Los IDE, necesariamente cuentan con un compilador o IDE y compilador son independientes?.

R: IDE y compilador son programas independientes. Pero la "I" de IDE es de "integrado", por lo que su función es dar cierta utilidad propia, desde ya, pero también vincularse con otros programas. Esta vinculación, esta "integración" permite, por ejemplo, compilar el código fuente que se está desarrollando desde el mismo IDE.

¿Cómo sería un proceso de programación sin IDE, por ejemplo, en Java?

- Se abre un editor de textos

- Se guarda el archivo con la extensión .java

- Se escribe el código fuente

- Se guardan los cambios

- Se cierra (o no) el archivo

- Desde la línea de comandos del sistema operativo

- Se escribe "javac NombreDeArchivo.java" y enter

- Esto genera el bytecode de Java

- Para ejecutar ese programa, desde la línea de comandos se escribe "java NombreDeArchivo" y enter

¿Cómo es esto mismo con un IDE?

- Se abre el IDE

- Se crea un archivo .java

- Se escribe el código fuente

- Normalmente hay un botón tipo "play", se le hace clic

- Se compila y ejecuta

Sin tener que usar la línea de comandos, sin tipear (salvo escribir el código fuente, claro) y con el mouse de acá para allá.

17) P: ¿Podemos decir que los niveles de los lenguajes de programación son la estructura del lenguaje en sí misma (instancia previa a la ejecución). ¿Y que cuanto más alto es el nivel del lenguaje, su sintaxis se acerca más al lenguaje humano?

R: Se puede decir que sí, aunque en lugar de "son la estructura del lenguaje en sí misma" se debería decir que "son la forma en la que un lenguaje de programación se estructura". Ya que de esta forma suena más concreto y preciso.

Y si: cuanto más alto el nivel, más se parece a una "conversación" (¡con muchas comillas!) entre una máquina y un programador.

18) P: ¿A qué tipo de hardware (principal o periférico) pertenece la memoria de almacenamiento masivo y porqué un disco rígido se considera periférico y no principal?

R: La memoria de almacenamiento masivo y un disco duro se pueden calificar como periféricos ya que almacenan información que no es utilizada todo el tiempo para que funciones una computadora. Se les dice memorias secundarias, siendo la memoria principal la memoria RAM. Por ejemplo, el sistema operativo como información está en un disco rígido (o un USB, etc.), pero cuando se enciende la computadora, en la memoria principal se ejecuta el sistema operativo almacenado que permite utilizar la computadora al usuario.

19) P: ¿El paradigma de cada lenguaje de programación es algo que ya está determinado desde el momento de su concepción o es algo que se genera por la comunidad de programadores?

R: Esto es, a veces, una línea muy delgada. EN GENERAL el lenguaje se crea para que califique como tal o cual paradigma: ya viene "de fábrica".

Ejemplos: con C nunca se podría hacer programación orientada a objetos (POO) porque no maneja los conceptos de clases, objetos, herencia, etc. que son esenciales del paradigma. Con Smalltalk no se podría hacer otra cosa que POO, porque es un lenguaje puro OO. Lisp o ML son lenguajes funcionales, porque fueron creados así.

Ahora bien: Java, que es un lenguaje OO híbrido permite cierta forma de programación estructurada o programación funcional por sobre POO.

Scala es un lenguajes que ya se creó pensando en que sea multiparadigma.

Entonces?

Entonces, hay un poco de todo. Hay lenguajes que se cierran a un único paradigma en particular, lenguajes que se crean para un paradigma pero se puede usar para otros y lenguajes que permiten varios paradigmas. ¿De qué depende esto? Cuánto más PURISTA de un paradigma sea un lenguaje, menos se podrá mutar a otros paradigmas.

Cuánto menos purista, también, más le permite a los programadores "experimentar" cosas nuevas. Como el extraño caso de la "moda" actual de la programación funcional en Java.

20) P: ¿Qué libros puedo consultar para complementar esta unidad u otros temas del curso?

R: Si tienen tiempo, no lean libros de programación aún (*): usen ese tiempo para aprender o mejorar tu inglés.

(*) Básicamente hay 2 tipos de libros de programación: los "manuales" que enseñan tal o cual lenguaje o tecnología, y los libros que hablan sobre el "trabajo del programador".

Estos últimos sí valen la pena, pero si los leen ahora, se vas a perder de muchas cosas que no van a entender, por eso la recomendación de esperar (al menos) hasta terminar este curso. De este tipo de libros recomiendo "The pragmatic Programmer". Es un hermoso libro de programación, que todo programador debería

leer al inicio de su carrera y releerlo cada 5 o 6 años, porque siempre se le encuentran cosas nuevas.

Otro libro del estilo es "Código limpio".

Además de estos, cualquier libro del maestro Joyanes Aguilar va a ser útil, sobre todo los introductorios y los que tratan sobre algoritmos. Pero sus contenidos (salvando las distancias) no van a diferir mucho de veamos en este curso que se complementa con foros, videos y actividades.

De los manuales... hay infinidad. Los de O'really suelen ser muy buenos.

Lo que vimos

- Qué es el software, para qué sirve y para qué se utiliza
- Concepto programación y sus principales variantes
- Componentes necesarios para programar
- Conceptos esenciales de los lenguajes de programación en cuanto a sus características generales principales
- Generaciones de los lenguajes de programación
- Tipos de lenguajes de programación según la forma en que estos se ejecutan
- Paradigmas en los cuales están basados los principales lenguajes de programación
- Cómo una computadora representa internamente la información

Lo que viene:

- Lógica Matemática
- Proposiciones y Operaciones lógicas
- Algebra
- Reglas de Inferencia y el *modus ponendo ponens*
- Sistemas de numeración
- Representaciones de números en distintas bases
- Matrices