Tema 1. Introducción a los sistemas de información

1.1 El Programa Informático

Definición de programa informático: “Conjunto de instrucciones que se ejecutan de manera secuencial con el objetivo de realizar una o varias tareas en un sistema”.

Todo programa informático es creado por un programador en un lenguaje determinado, que será compilado (o interpretado) y ejecutado por un sistema. Cuando un programa es invocado para ser ejecutado, el procesador ejecuta el código compilado del programa instrucción por instrucción.

1.2 Lenguajes de programación

Definición:

“Un lenguaje de programación es un conjunto de instrucciones, operadores y reglas de sintaxis y semánticas, que se ponen a disposición del programador para que éste pueda comunicarse con los dispositivos de hardware y software existentes”.

Un lenguaje de programación es, por tanto, un idioma artificial diseñado para expresar operaciones que pueden ser llevadas a cabo por máquinas como los ordenadores. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión, o como modo de comunicación humana. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Al proceso por el cual se escribe, se prueba, se depura, se compila y se mantiene el código fuente de un programa informático se le llama **programación**.

También la palabra programación se define como el proceso de creación de un programa informático, mediante la aplicación de procedimientos lógicos, a través de los siguientes pasos:

- El desarrollo lógico del programa para resolver un problema en particular.
- Escritura de la lógica del programa empleando un lenguaje de programación específico (codificación del programa).
- Ensamblaje o compilación del programa hasta convertirlo en lenguaje de máquina.
- Prueba y depuración del programa.

- Desarrollo de la documentación.

Existe un error común que trata por sinónimos los términos 'lenguaje de programación' y 'lenguaje informático'. Los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como por ejemplo HTML (lenguaje para el marcado de páginas web que no es propiamente un lenguaje de programación, sino un conjunto de instrucciones que permiten diseñar el contenido de los documentos).

Permite especificar de *manera precisa* sobre qué datos debe operar un ordenador, cómo deben ser almacenados o transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar *relativamente* próximo al lenguaje humano o natural.

Una característica relevante de los lenguajes de programación es precisamente que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos para realizar la construcción de un programa de forma colaborativa.

1.2.1 Historia

Código Fortran en una tarjeta perforada, mostrando el uso especializado de las columnas 1-5, 6 y 73-80.

Para que el ordenador entienda nuestras instrucciones debe usarse un lenguaje específico conocido como código máquina, el cual la máquina comprende fácilmente, pero que lo hace excesivamente complicado para las personas. De hecho sólo consiste en cadenas extensas de números 0 y 1.

Para facilitar el trabajo, los primeros operadores de ordenador decidieron hacer un traductor para reemplazar los 0 y 1 por palabras o abstracción de palabras y letras provenientes del inglés; éste se conoce como *lenguaje ensamblador*. Por ejemplo, para sumar se usa la letra A de la palabra inglesa *add* (sumar). El lenguaje ensamblador sigue la misma estructura del lenguaje máquina, pero las letras y palabras son más fáciles de recordar y entender que los números.

La necesidad de recordar secuencias de programación para las acciones usuales llevó a denominarlas con nombres fáciles de memorizar y asociar: `ADD` (sumar), `SUB` (restar), `MUL` (multiplicar), `CALL` (ejecutar subrutina), etc. A esta secuencia de posiciones se le denominó "instrucciones", y a este conjunto de instrucciones se le llamó lenguaje ensamblador.

Un ejemplo del clásico programa que muestra por pantalla “Hola Mundo” en ensamblador sería el siguiente:

```
.data  # lo dice el nombre ;-)

msg:    .string "Hola, Mundo!\n"       # la cadena tipica
        len = . - msg                  # y su longitud

.text   # es típico que la zona de código se llame así :-?

        .global main    # decimos donde empezamos, el "entry point"

main:
        movl    $len, %edx      # tercer argumento, longitud de la cadena
        movl    $msg, %ecx      # segundo argumento, puntero a la cadena
        movl    $1, %ebx        # primer argumento, handler (1 = STDOUT)
        movl    $4, %eax        # num. de llamada a sistema (4 = write)
        int     $0x80           # llamada al kernel

        movl    $0, %ebx        # primer argumento, código de salida
        movl    $1, %eax        # num. de llamada a sistema (1 = exit)
        int     $0x80           # llamada al kernel
```

archivo HolaMundo.s (ensamblador). Compilar con gcc

Posteriormente aparecieron diferentes lenguajes de programación, los cuales reciben su denominación porque tienen una estructura sintáctica similar a los lenguajes escritos por los humanos, denominados también lenguajes de alto nivel.

La primera programadora de computadora conocida fue Ada Lovelace, hija de Anabella Milbanke Byron y Lord Byron. Anabella introdujo en las matemáticas a Ada quien, después de conocer a Charles Babbage, tradujo y amplió una descripción de su máquina analítica. Incluso aunque Babbage nunca completó la construcción de cualquiera de sus máquinas, el trabajo que Ada realizó con éstas le hizo ganarse el título de primera programadora de ordenadores del mundo. El nombre del lenguaje de programación ADA fue escogido como homenaje a esta programadora.

A finales de 1953, John Backus sometió una propuesta a sus superiores en IBM para desarrollar una alternativa más práctica al lenguaje ensamblador para programar el ordenador central IBM 704. El histórico equipo Fortran de Backus

consistió en los programadores Richard Goldberg, Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Lois Haibt y David Sayre.

El primer manual para el lenguaje Fortran apareció en octubre de 1956, con el primer compilador Fortran entregado en abril de 1957. Esto era un compilador optimizado, porque los clientes eran reacios a usar un lenguaje de alto nivel a menos que su compilador pudiera generar código cuyo rendimiento fuera comparable al de un código hecho a mano en lenguaje ensamblador.

En 1960, se creó COBOL, uno de los lenguajes más usados aún en 2011 en informática de gestión.

A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un método más eficiente para programarlas. Entonces, se crearon los lenguajes de alto nivel, como lo fue BASIC en las versiones introducidas en los microordenadores de la década de los 80. Mientras que una tarea tan sencilla como sumar dos números puede necesitar varias instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará una sola sentencia.

En el siguiente ejemplo, se muestra como sería la programación del clásico “Hola Mundo”, en lenguaje C:

```
#include <stdio.h>
int main() {
	printf("Hola, Mundo\n");
	return (0);
}
```

En lenguaje Java:

```
public class HolaMundo {
	public static void main (String[] args) {
		System.out.println("Hola, Mundo!");
	}
}
```

Archivo: HolaMundo.java

```
Main-Class: HolaMundo
```

Archivo: HolaMundo.manifest

Para compilar: $ javac HolaMundo.java **(resultado):** HolaMundo.class

Para generar el archivo .jar: $ jar cmf HolaMundo.manifest HolaMundo.jar HolaMundo.class

Para ejecutar el programa: $ java -jar HolaMundo.jar

Por último, en Fortran 95:

```
program hola
      implicit none
      print *, 'Hola, Mundo!'
end program hola
```

Archivo: hola.f

Para compilar: $ gfortran -ffree-form -o hola.o hola.f

1.2.2 Clasificación y características

La cantidad de lenguajes de programación es abrumadora, cada uno con unas características y objetivos determinados. Esto implica la necesidad de establecer unos criterios de clasificación.

Se pueden clasificar mediante una gran variedad de criterios (hasta once criterios válidos). Algunos de dichos criterios pueden ser redundantes, puesto que se encuentran incluídos dentro de otros, como el determinismo o el propósito.

Nos vamos a centrar en la clasificación de los lenguajes en base a tres criterios globales y reconocidos: *nivel de abstracción, forma de ejecución y paradigma*.

Según el Nivel de abstracción

Llamamos nivel de abstracción al modo en que los lenguajes se alejan del código máquina y se acercan cada vez más a un lenguaje similar al del ser humano. Dicho de otro modo, podría verse el nivel de abstracción como la cantidad de “capas” de ocultación de código máquina que hay entre el código que escribimos y el código que la máquina ejecutará finalmente.

- **Lenguajes de bajo nivel**
  - Primera generación: Solo hay un lenguaje de primera generación: el código máquina. Cadenas interminables de secuencias de 0s y 1s que forman operaciones que la máquina puede entender sin interpretación alguna.
- **Lenguajes de medio nivel**
  - Segunda generación: Tienen definidas unas instrucciones para realizar operaciones sencillas con datos simples o posiciones de memoria. El lenguaje clave de este tipo es el **ensamblador**.

- **Lenguajes de alto nivel**
  - Tercera generación: Son la mayoría de los lenguajes de programación que se utilizan hoy en día; son lenguajes de propósito general que permiten un alto grado de abstracción y una forma de programar mucho más intuitiva, donde algunas instrucciones parecen ser una traducción directa del lenguaje humano. Por ejemplo, nos podríamos encontrar con una línea de código como ésta:

    IF contador = 10 THEN stop

    Ejemplos de lenguajes de este tipo: C, C#, Java.
  - Cuarta generación: Son lenguajes creados con un propósito específico, lo cual permite reducir el número de líneas de código en comparación con un lenguaje de tercera generación. Por ejemplo, si tuviésemos que resolver una ecuación en un lenguaje de tercera generación, tendríamos que crear diversos y complejos métodos para resolverla, mientras que en un lenguaje de cuarta generación dedicado a este tipo de problemas ya posee esas rutinas incluidas en el propio lenguaje, bastando con invocar la instrucción necesaria. Ejemplos: NATURAL, SQL
  - Quinta generación: También llamados lenguajes naturales, pretenden abstraer más aún el lenguaje utilizando un lenguaje natural con una base de conocimientos que produce un sistema basado en el conocimiento. Pueden establecer el problema que hay que resolver y las premisas o condiciones que hay que reunir para que la máquina lo resuelva. Este tipo de lenguajes los podemos encontrar frecuentemente en inteligencia artificial y lógica. Ejemplos: Prolog, OPS5 y Mercury.

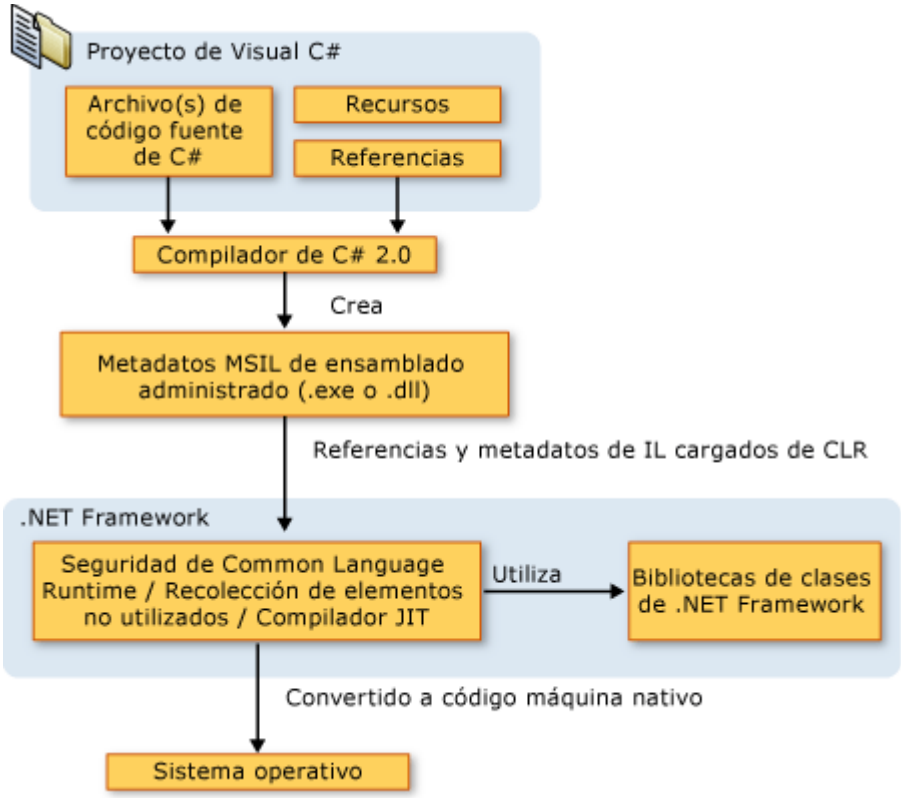
Según la forma de ejecución

Dependiendo de cómo un programa se ejecute dentro de un sistema, podríamos definir tres categorías de lenguajes:

- **Lenguajes compilados**: Un programa traductor (compilador) convierte el código fuente en código objeto y otro programa (linkador) unirá el código objeto del programa con el código objeto de las librerías necesarias para producir el programa ejecutable. Ejemplos: C, Cobol
- **Lenguajes interpretados:** Ejecutan las instrucciones directamente, sin que se ejecute código objeto. Para ello es necesario un programa intérprete

en el sistema operativo o en la propia máquina donde cada instrucción es interpretada y ejecutada de manera independiente y secuencial. La principal diferencia con el anterior es que se traducen a tiempo real solo las instrucciones que se utilicen en cada ejecución, en vez de interpretar todo el código, se vaya a utilizar o no. Ejemplos: HTML, JavaScript, PHP, Python

- **Lenguajes virtuales:** Tienen un funcionamiento muy similar al de los lenguajes compilados, pero, a diferencia de éstos, no es código objeto lo que genera el compilador, sino un *bytecode* que puede ser interpretado por cualquier arquitectura que tenga la máquina virtual que se encargará de interpretar el código bytecode generado para ejecutarlo en la máquina. Aunque de ejecución lenta (al igual que los interpretados), tiene la ventaja de poder ser multiplataforma y así un mismo código bytecode sería válido para cualquier máquina. Ejemplos: Java, C#



Según el Paradigma de programación

El paradigma de programación es un enfoque particular para la construcción de software, un estilo de programación que facilita la tarea o añade mayor funcionalidad al programa dependiendo del programa que haya que abordar. Todos los paradigmas de programación pertenecen a lenguajes de alto nivel, y es común que un lenguaje pueda usar más de un paradigma.

- **Imperativo**: Describe la programación como una secuencia de instrucciones que cambian el estado del programa, indicando cómo realizar una tarea. Los programas que usan un lenguaje imperativo especifican un *algoritmo*, usan *declaraciones, expresiones y sentencias*. Una **declaración** asocia un nombre de variable con un tipo de dato, por ejemplo:

  ```
  var x: integer;
  ```

  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 then haz_algo();
  ```

- **Declarativo**: Especifica o declara un conjunto de premisas y condiciones para indicar qué es lo que hay que hacer y no necesariamente cómo hay que hacerlo.

  Dos amplias categorías de lenguajes declarativos son los **lenguajes funcionales** y los **lenguajes lógicos**.

  Los lenguajes **funcionales** no permiten asignaciones de variables no locales, así, se hacen más fácil, por ejemplo, programas como funciones matemáticas. El principio detrás de los lenguajes lógicos es definir el problema que se quiere resolver (el objetivo) y dejar los detalles de la solución al sistema. El objetivo es definido dando una lista de subobjetivos. Cada subobjetivo también se define dando una lista de sus subobjetivos, etc. Si al tratar de buscar una solución, una ruta de subobjetivos falla, entonces tal subobjetivo se descarta y de forma sistemática se prueba otra ruta.

  La forma en la cual se programa puede ser por medio de texto o de forma visual. En la **programación visual** los elementos son manipulados gráficamente en vez de especificarse por medio de texto.

  Los lenguajes **lógicos** definen un conjunto de reglas lógicas para ser interpretadas mediante inferencias lógicas. Permite responder preguntas planteadas al sistema para resolver problemas.

- **Procedimental**: El programa se divide en partes más pequeñas, llamadas *procedimientos y funciones*, que pueden comunicarse entre sí. Permite reutilizar el código ya hecho y solventa el problema de la *programación espaguetti*.

- **Orientado a objetos:** Encapsula el estado y las operaciones en objetos, creando una estructura de clases y objetos que emula un modelo del mundo real, donde los objetos realizan acciones e interactúan con otros objetos. Permite la herencia e implementación de otras clases, pudiendo establecer tipos para los objetos y dejando el código más parecido al mundo real.

1.2.3 Elementos

Todos los lenguajes de programación tienen algunos elementos de formación primitivos para la descripción de los datos y de los procesos o transformaciones aplicadas a estos datos (tal como la suma de dos números o la selección de un elemento que forma parte de una colección). Estos elementos primitivos son definidos por reglas sintácticas y semánticas que describen su estructura y significado respectivamente.

1.2.2.1 Sintaxis

A la forma visible de un lenguaje de programación se le conoce como **sintaxis**. La mayoría de los lenguajes de programación son puramente textuales, es decir, utilizan secuencias de texto que incluyen palabras, números y puntuación, de manera similar a los lenguajes naturales escritos. Por otra parte, hay algunos lenguajes de programación que son más gráficos en su naturaleza, utilizando relaciones visuales entre símbolos para especificar un programa.

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> (' % nodename
      for  in :namechildren
         print '%s' % name,
```

> Con frecuencia se resaltan los elementos de la sintaxis con colores diferentes para facilitar su lectura. Este ejemplo está escrito en Python.

La sintaxis de un lenguaje de programación describe las combinaciones posibles de los símbolos que forman un programa sintácticamente correcto. El significado que se le da a una combinación de símbolos es manejado por su **semántica** (ya sea formal o como parte del código duro de la referencia de implementación).

La sintaxis de los lenguajes de programación es definida generalmente utilizando una combinación de expresiones regulares (para la estructura léxica) y la **Notación de Backus-Naur** (para la estructura gramática). Este es un ejemplo de una gramática simple, tomada de Lisp:

```
expresión ::= átomo  | lista
átomo     ::= número | símbolo
número    ::= [+-]?['0'-'9']+
símbolo   ::= ['A'-'Z' 'a'-'z'].*
lista     ::= '(' expresión* ')'
```

Con esta gramática se especifica lo siguiente:

- una *expresión* puede ser un *átomo* o una *lista*;
- un *átomo* puede ser un *número* o un *símbolo*;
- un *número* es una secuencia continua de uno o más dígitos decimales, precedido opcionalmente por un signo más o un signo menos;
- un *símbolo* es una letra seguida de cero o más caracteres (excluyendo espacios); y
- una *lista* es un par de paréntesis que abren y cierran, con cero o más expresiones en medio.

Algunos ejemplos de secuencias bien formadas de acuerdo a esta gramática:

`'12345'`, `'()'`, `'(a b c232 (1))'`

No todos los programas sintácticamente correctos son semánticamente correctos. Muchos programas sintácticamente correctos tienen inconsistencias con las reglas del lenguaje; y pueden (dependiendo de la especificación del lenguaje y la solidez de la implementación) resultar en un error de traducción o ejecución. En algunos casos, tales programas pueden exhibir un comportamiento indefinido. Además, incluso cuando un programa está bien definido dentro de un lenguaje,

todavía puede tener un significado que no es el que la persona que lo escribió estaba tratando de construir.

Usando el lenguaje natural, por ejemplo, puede no ser posible asignarle significado a una oración gramaticalmente válida o la oración puede ser falsa:

- "Las ideas verdes y descoloridas duermen furiosamente" es una oración bien formada gramaticalmente pero no tiene significado comúnmente aceptado.
- "Juan es un soltero casado" también está bien formada gramaticalmente pero expresa un significado que no puede ser verdadero.

El siguiente fragmento en el lenguaje C es sintácticamente correcto, pero ejecuta una operación que no está definida semánticamente (dado que `p` es un puntero nulo, las operaciones `p->real` y `p->im` no tienen ningún significado):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

Si la declaración de tipo de la primera línea fuera omitida, el programa dispararía un error de compilación, pues la variable "p" no estaría definida. Pero el programa sería sintácticamente correcto todavía, dado que las declaraciones de tipo proveen información semántica solamente.

Lenguajes tipados versus lenguajes no tipados

Se dice que un lenguaje tiene *tipos* si la especificación de cada operación define tipos de datos para los cuales la operación es aplicable, con la implicación de que no es aplicable a otros tipos. Por ejemplo, "`este texto entre comillas`" es una cadena.

En la mayoría de los lenguajes de programación, dividir un número por una cadena no tiene ningún significado. Por tanto, la mayoría de los lenguajes de programación modernos rechazaran cualquier intento de ejecutar dicha operación por parte de algún programa. En algunos lenguajes, estas operaciones sin significado son detectadas cuando el programa es compilado (validación de tipos "estática") y son rechazadas por el compilador, mientras en otros son detectadas cuando el programa es ejecutado (validación de tipos "dinámica") y se genera una excepción en tiempo de ejecución.

Un caso especial de lenguajes de tipo son los lenguajes de *tipo sencillo*. Estos son con frecuencia lenguajes de marcado o de scripts, como REXX o SGML, y

solamente cuentan con un tipo de datos; comúnmente cadenas de caracteres que luego son usadas tanto para datos numéricos como simbólicos.

En contraste, un lenguaje *sin tipos*, como la mayoría de los lenguajes ensambladores, permiten que cualquier operación se aplique a cualquier dato, que por lo general se consideran secuencias de bits de varias longitudes. Lenguajes de alto nivel *sin datos* incluyen BCPL y algunas variedades de Forth.

En la práctica, aunque pocos lenguajes son considerados con tipo desde el punto de vista de la teoría de tipos (es decir, que verifican o rechazan *todas* las operaciones), la mayoría de los lenguajes modernos ofrecen algún grado de manejo de tipos. Si bien muchos lenguajes de producción proveen medios para saltarse el sistema de tipos.

Tipos estáticos versus tipos dinámicos

En lenguajes con *tipos estáticos* se determina el tipo de todas las expresiones antes de la ejecución del programa (típicamente al compilar). Por ejemplo, 1 y (2+2) son expresiones enteras; no pueden ser pasadas a una función que espera una cadena, ni pueden guardarse en una variable que está definida como fecha.

Los lenguajes con tipos estáticos pueden manejar tipos *explícitos* o tipos *inferidos*. En el primer caso, el programador debe escribir los tipos en determinadas posiciones textuales. En el segundo caso, el compilador *infiere* los tipos de las expresiones y las declaraciones de acuerdo al contexto. La mayoría de los lenguajes populares con tipos estáticos, tales como C++, C# y Java, manejan tipos explícitos. Inferencia total de los tipos suele asociarse con lenguajes menos populares, tales como Haskell y ML. Sin embargo, muchos lenguajes de tipos explícitos permiten inferencias parciales de tipo; tanto Java y C#, por ejemplo, infieren tipos en un número limitado de casos.

Los lenguajes con tipos dinámicos determinan la validez de los tipos involucrados en las operaciones durante la ejecución del programa. En otras palabras, los tipos están asociados con *valores en ejecución* en lugar de *expresiones textuales*. Como en el caso de lenguajes con tipos inferidos, los lenguajes con tipos dinámicos no requieren que el programador escriba los tipos de las expresiones. Entre otras cosas, esto permite que una misma variable se pueda asociar con valores de tipos distintos en diferentes momentos de la ejecución de un programa. Sin embargo, los errores de tipo no pueden ser detectados automáticamente hasta que se ejecuta el código, dificultando la depuración de los programas. Ruby, Lisp, JavaScript y Python son lenguajes con tipos dinámicos.

Tipos débiles y tipos fuertes

Los lenguajes *débilmente tipados* permiten que un valor de un tipo pueda ser tratado como de otro tipo, por ejemplo una cadena puede ser operada como un número. Esto puede ser útil a veces, pero también puede permitir ciertos tipos de fallas que no pueden ser detectadas durante la compilación o a veces ni siquiera durante la ejecución.

Los lenguajes *fuertemente tipados* evitan que pase lo anterior. Cualquier intento de llevar a cabo una operación sobre el tipo equivocado dispara un error. A los lenguajes con tipos fuertes se les suele llamar *de tipos seguros*.

Lenguajes con tipos débiles como Perl y JavaScript permiten un gran número de conversiones de tipo implícitas. Por ejemplo en JavaScript la expresión `2 * x` convierte implícitamente `x` a un número, y esta conversión es exitosa inclusive cuando `x` es `null`, `undefined`, un `Array` o una cadena de letras. Estas conversiones implícitas son útiles con frecuencia, pero también pueden ocultar errores de programación.

Las características de *estáticos* y *fuertes* son ahora generalmente consideradas conceptos ortogonales, pero su trato en diferentes textos varia. Algunos utilizan el término *de tipos fuertes* para referirse a *tipos fuertemente estáticos* o, para aumentar la confusión, simplemente como equivalencia de *tipos estáticos*. De tal manera que C ha sido llamado tanto lenguaje de tipos fuertes como lenguaje de tipos estáticos débiles.

1.2.3 Implementación

```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();           // display the fra
}
```

Código fuente de un programa escrito en el lenguaje de programación Java.

La implementación de un lenguaje es la que provee una manera de que se ejecute un programa para una determinada combinación de software y hardware.

Existen básicamente dos maneras de implementar un lenguaje: compilación e interpretación.

- **Compilación**: es el proceso que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz interpretar. Los programas traductores que pueden realizar esta operación se llaman compiladores. Éstos, como los programas ensambladores avanzados, pueden generar muchas líneas de código de máquina por cada proposición del programa fuente.

- **Interpretación**: En lugar de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante la compilación para utilizarlo en una ejecución futura, el programador sólo carga el programa fuente en la computadora junto con los datos que se van a procesar. A continuación, un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el procesamiento de los datos. El código objeto no se graba para utilizarlo posteriormente. La siguiente vez que se utilice una instrucción, se la deberá interpretar otra vez y traducir a lenguaje máquina. Por ejemplo, durante el procesamiento repetitivo de los pasos de un ciclo o bucle, cada instrucción del bucle tendrá que volver a ser interpretada en cada ejecución repetida del ciclo, lo cual hace que el programa sea más lento en tiempo de ejecución (porque se va revisando el código en tiempo de ejecución) pero más rápido en tiempo de diseño (porque no se tiene que estar compilando a cada momento el código completo). El intérprete elimina la necesidad de realizar una compilación después de cada modificación del programa cuando se quiere agregar funciones o corregir errores; pero es obvio que un programa objeto compilado con antelación deberá ejecutarse con mucha mayor rapidez que uno que se debe interpretar a cada paso durante una ejecución del código.

La mayoría de lenguajes de alto nivel permiten la programación multipropósito, sin embargo, muchos de ellos fueron diseñados para permitir programación dedicada, como lo fue el **Pascal** con las matemáticas en su comienzo. También se han implementado lenguajes educativos infantiles como Logo que trabaja mediante una serie de simples instrucciones. En el ámbito de infraestructura de Internet, cabe destacar a Perl con un poderoso sistema de procesamiento de texto y una enorme colección de módulos.

1.2.4 Partes de un compilador

La construcción de un compilador involucra la división del proceso en una serie de fases que variará con su complejidad. Generalmente estas fases se agrupan en dos tareas: el análisis del programa fuente y la síntesis del programa objeto.

- ***Análisis***: Se trata de la comprobación de la corrección del programa fuente, e incluye las fases:
  - Análisis Léxico (que consiste en la descomposición del programa fuente en componentes léxicos, llamados tokens),
  - Análisis Sintáctico (agrupación de los componentes léxicos en frases gramaticales) y
  - Análisis Semántico (comprobación de la validez semántica de las sentencias aceptadas en la fase de Análisis Sintáctico).

- ***Síntesis***: Su objetivo es la generación de la salida expresada en el lenguaje objeto y suele estar formado por una o varias combinaciones de fases de Generación de Código (normalmente se trata de código intermedio o de código objeto) y de Optimización de Código (en las que se busca obtener un código lo más eficiente posible).

1.2.5 Tipos de compiladores

Esta clasificación de los tipos de compiladores no es excluyente, por lo que puede haber compiladores que se adscriban a varias categorías:

- **Compiladores cruzados**: generan código para un sistema distinto del que están funcionando.
- **Compiladores optimizadores**: realizan cambios en el código para mejorar su eficiencia, pero manteniendo la funcionalidad del programa original.
- **Compiladores de una sola pasada**: generan el código máquina a partir de una única lectura del código fuente.
- **Compiladores de varias pasadas**: necesitan leer el código fuente varias veces antes de poder producir el código máquina.
- **Compiladores JIT** (*Just In Time*): forman parte de un intérprete y compilan partes del código según se necesitan.

1.2.6 Técnica

Para escribir programas que proporcionen los mejores resultados, cabe tener en cuenta una serie de detalles.

- ***Corrección***. Un programa es correcto si hace lo que debe hacer tal y como se estableció en las fases previas a su desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué

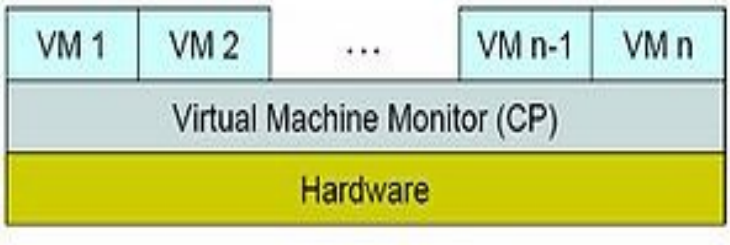
debe hacer el programa antes de desarrollarlo y, una vez acabado, compararlo con lo que realmente hace.

- ***Claridad***. Es muy importante que el programa sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Al elaborar un programa se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo en la edición; de esta forma se ve facilitado el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo cual la claridad es aún más necesaria para que otros programadores puedan continuar el trabajo fácilmente. Algunos programadores llegan incluso a utilizar Arte ASCII para delimitar secciones de código. Otros, por diversión o para impedir un análisis cómodo a otros programadores, recurren al uso de código ofuscado.

- ***Eficiencia***. Se trata de que el programa, además de realizar aquello para lo que fue creado (es decir, que sea correcto), lo haga gestionando de la mejor forma posible los recursos que utiliza. Normalmente, al hablar de eficiencia de un programa, se suele hacer referencia al tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita, pero hay otros recursos que también pueden ser de consideración al obtener la eficiencia de un programa, dependiendo de su naturaleza (espacio en disco que utiliza, tráfico de red que genera, etc.).

- ***Portabilidad***. Un programa es portable cuando tiene la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquella en la que se elaboró. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha desarrollado para sistemas GNU/Linux ejecutarse también en la familia de sistemas operativos Windows. Esto permite que el programa pueda llegar a más usuarios más fácilmente.

1.3 Máquinas virtuales



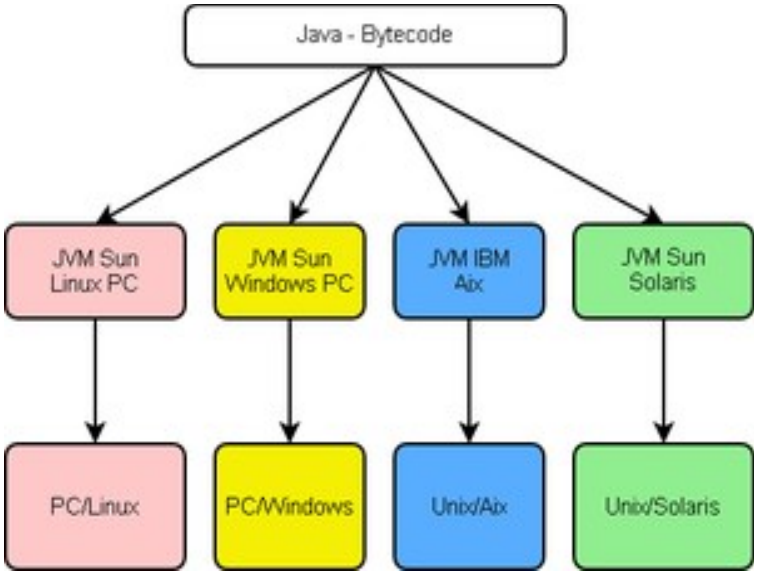
Arquitectura típica de una máquina virtual.

En informática una **máquina virtual** es un software que emula a un ordenador y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física". La acepción del término actualmente incluye a máquinas virtuales que no tienen ninguna equivalencia directa con ningún hardware real.

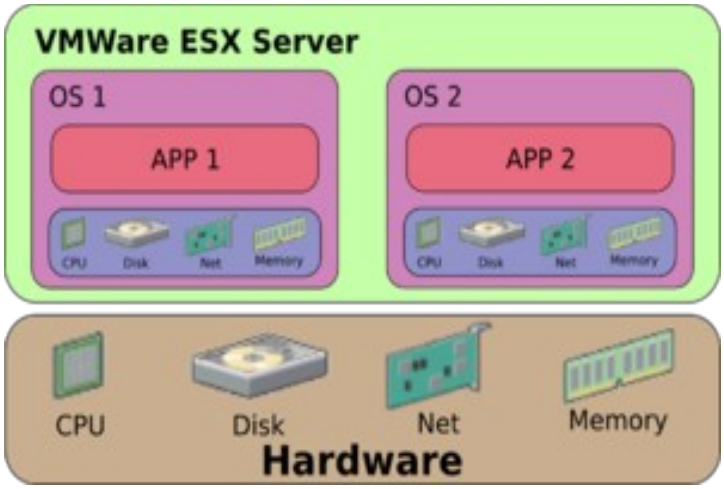
Una característica esencial de las máquinas virtuales es que los procesos que ejecutan están **limitados por los recursos y abstracciones proporcionados por ellas**. Estos procesos no pueden escaparse de esta "computadora virtual".

Uno de los usos domésticos más extendidos de las máquinas virtuales es **ejecutar sistemas operativos** para "probarlos". De esta forma podemos ejecutar un sistema operativo que queramos probar (GNU/Linux, por ejemplo) desde nuestro sistema operativo habitual (Mac OS X por ejemplo) sin necesidad de instalarlo directamente en nuestro ordenador y sin miedo a que se desconfigure el sistema operativo primario, también llamado anfitrión.

1.3.1 Tipos de máquinas virtuales



Funcionamiento de la máquina virtual de Java, una de las máquinas virtuales de proceso más populares.



Funcionamiento de VMWare, una de las máquinas virtuales de sistema más populares.

Las máquinas virtuales se pueden clasificar en dos grandes categorías según su funcionalidad y su grado de equivalencia a una verdadera máquina.

- **Máquinas virtuales de sistema** (en inglés System Virtual Machine)
- **Máquinas virtuales de proceso** (en inglés Process Virtual Machine)

1.3.1.1 Máquinas virtuales de sistema

Las máquinas virtuales de sistema, también llamadas **máquinas virtuales de hardware**, permiten a la máquina física subyacente multiplicarse entre varias

máquinas virtuales, cada una ejecutando su propio sistema operativo. A la capa de software que permite la virtualización se la llama **monitor de máquina virtual** o **"hypervisor"**. Un monitor de máquina virtual puede ejecutarse o bien directamente sobre el hardware o bien sobre un sistema operativo (**"host operating system"**).

Aplicaciones de las máquinas virtuales de sistema

- Varios **sistemas operativos distintos** pueden **coexistir** sobre el mismo ordenador, en sólido aislamiento el uno del otro, por ejemplo para probar un sistema operativo nuevo sin necesidad de instalarlo directamente.
- La máquina virtual puede proporcionar una **arquitectura de instrucciones** (ISA) que sea algo distinta de la verdadera máquina. Es decir, podemos simular hardware.
- Varias máquinas virtuales (cada una con su propio sistema operativo llamado sistema operativo "invitado" o "guest"), pueden ser utilizadas para **consolidar servidores**. Esto permite que servicios que normalmente se tengan que ejecutar en computadoras distintas para evitar interferencias, se puedan ejecutar en la misma máquina de manera completamente aislada y compartiendo los recursos de una única computadora. La consolidación de servidores a menudo contribuye a reducir el coste total de las instalaciones necesarias para mantener los servicios, dado que permiten ahorrar en hardware.
- La virtualización es una excelente opción hoy día, ya que las máquinas actuales (portátiles, pcs de sobremesa, servidores) en la mayoría de los casos están siendo "infrautilizados" (gran capacidad de disco duro, memoria RAM, etc.), llegando a un uso de entre 30% a 60% de su capacidad. Al virtualizar, la necesidad de nuevas máquinas en una ya existente permite un ahorro considerable de los costos asociados (energía, mantenimiento, espacio, etc).

1.3.1.2 Máquinas virtuales de proceso

Una máquina virtual de proceso, a veces llamada **"máquina virtual de aplicación"**, se ejecuta como un proceso normal dentro de un sistema operativo y soporta un solo proceso. La máquina se inicia automáticamente cuando se lanza el proceso que se desea ejecutar y se detiene para cuando éste finaliza. Su objetivo es el de proporcionar un entorno de ejecución **independiente de la plataforma de hardware y del sistema operativo**,

que oculte los detalles de la plataforma subyacente y permita que un programa se ejecute siempre de la misma forma sobre cualquier plataforma.

El ejemplo más conocido actualmente de este tipo de máquina virtual es la máquina virtual de Java. Otra máquina virtual muy conocida es la del entorno .Net de Microsoft que se llama "Common Language Runtime".

Inconvenientes de las máquinas virtuales

Uno de los inconvenientes de las máquinas virtuales es que agregan gran complejidad al sistema en tiempo de ejecución. Esto tiene como efecto la ralentización del sistema, es decir, el programa no alcanzará la misma velocidad de ejecución que si se instalase directamente en el sistema operativo "anfitrión" (host) o directamente sobre la plataforma de hardware. Sin embargo, a menudo la flexibilidad que ofrecen compensa esta pérdida de eficiencia.

Emulación de un sistema no nativo

Las máquinas virtuales también pueden actuar como emuladores de hardware, permitiendo que aplicaciones y sistemas operativos concebidos para otras arquitecturas de procesador se puedan ejecutar sobre un hardware que en teoría no soportan.

Algunas máquinas virtuales emulan hardware que sólo existe como una especificación.

Por ejemplo:

- La máquina virtual P-Code que permitía a los programadores de Pascal crear aplicaciones que se ejecutasen sobre cualquier computadora con esta máquina virtual correctamente instalada.
- La máquina virtual de Java.
- La máquina virtual del entorno .NET.
- Open Firmware

Esta técnica permite que cualquier ordenador pueda ejecutar software escrito para la máquina virtual. Sólo la máquina virtual en sí misma debe ser portada a cada una de las plataformas de hardware.