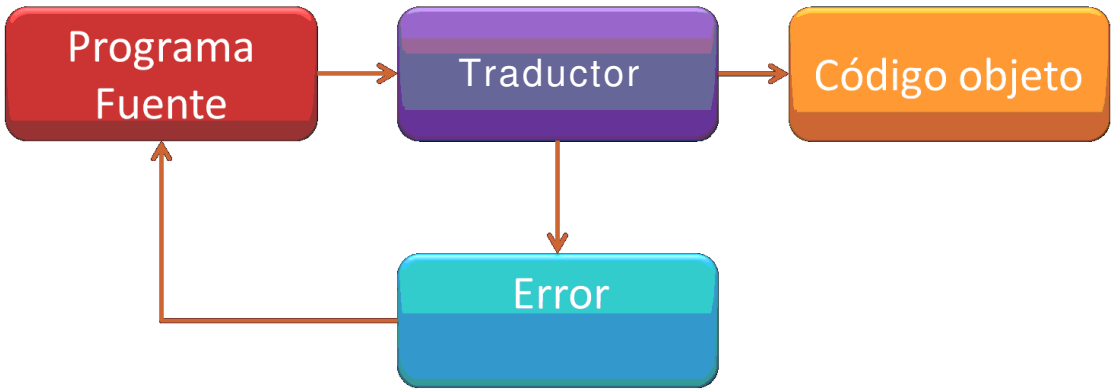
Tema 1. Algoritmos, programas y lenguajes de programación. (III)

1. TRADUCTORES.

Un programa escrito en un lenguaje de alto nivel necesita de un proceso de traducción para obtener un código máquina ejecutable por el ordenador.

Un **traductor** es un programa que toma como entrada un **programa fuente** y lo traduce o convierte obteniendo un **programa objeto**, produciendo, si es necesario, mensajes de error.



Tipos de traductores:
- Compiladores
- Intérpretes
- Máquina Virtual

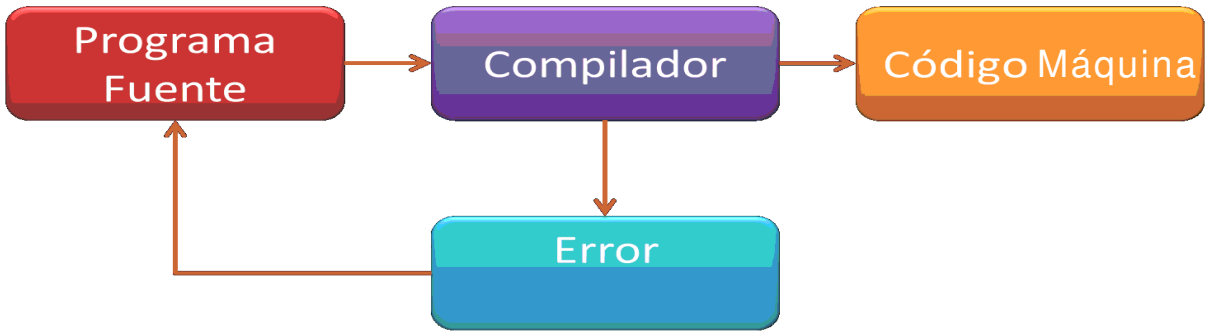
Esta división de los traductores hace que los lenguajes se clasifiquen en lenguajes compilados, lenguajes interpretados y lenguajes mixtos.

2. COMPILADORES

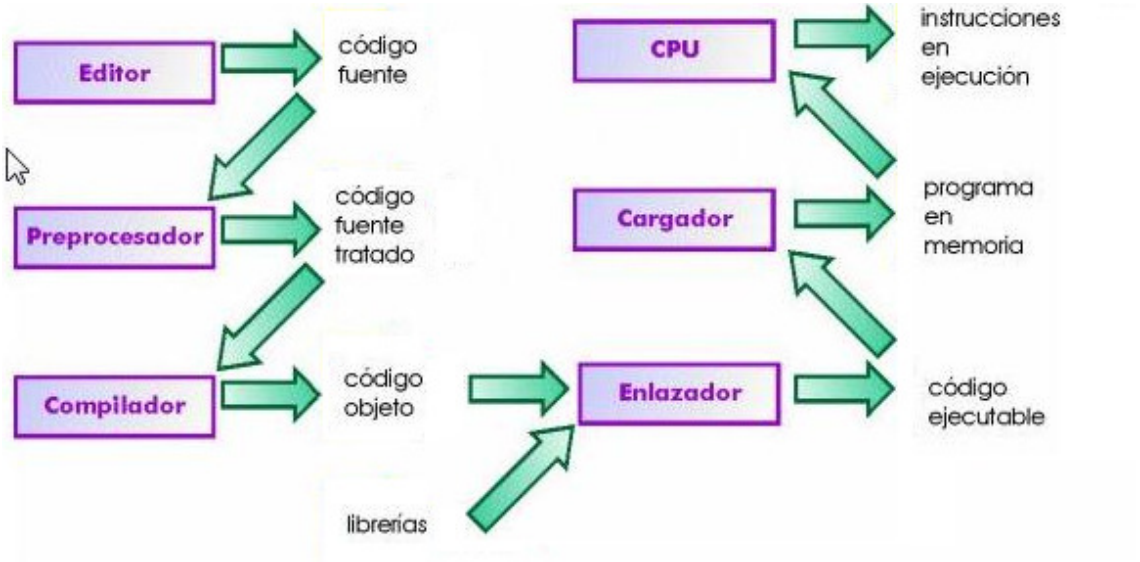
Un **compilador** es un programa que toma un programa fuente escrito en un lenguaje de alto nivel (C++, Pascal, etc.), lo **traduce completamente** y obtiene un programa objeto en lenguaje máquina.

La traducción del programa se efectúa de manera que cada instrucción escrita en lenguaje de alto nivel se transforma en una o varias instrucciones de lenguaje máquina.

Si durante el proceso de traducción se producen errores, el compilador informa de ellos y el programador deberá corregirlos, una vez que el código esté libre de errores se generará el programa traducido a código máquina.



El proceso de compilación suele llevar más operaciones asociadas (enlazador, cargador,..) para obtener el código ejecutable.



El **Preprocesador** modifica el programa fuente antes de la verdadera compilación. Hacen uso de macroinstrucciones y directivas. Por ejemplo, sustituye las directivas #include de C/C++ por el código correspondiente, de forma que el compilador se encuentra con el código ya incluido en el programa fuente.

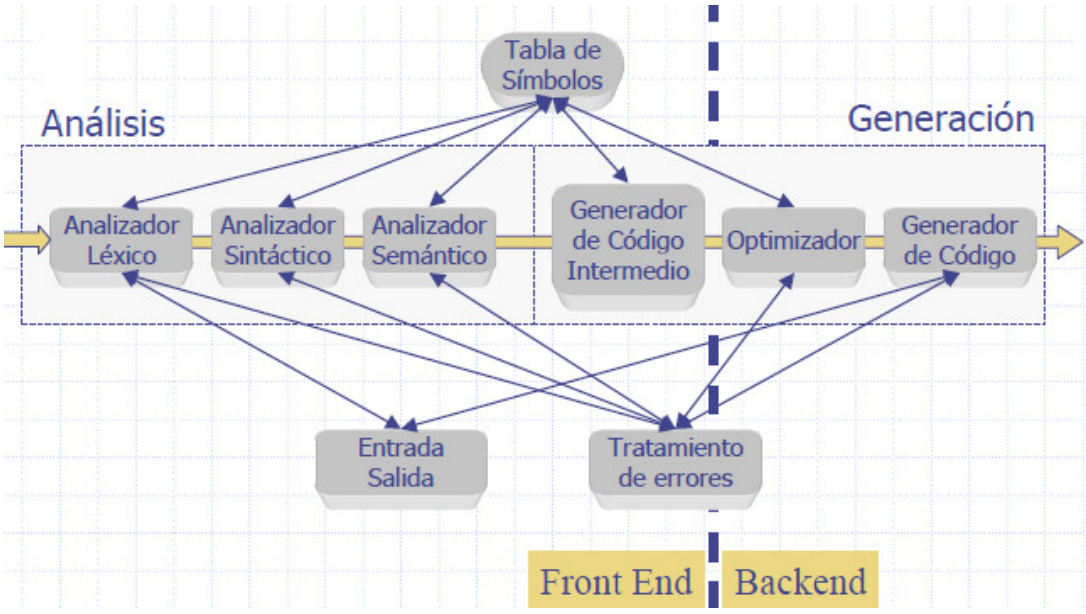
El **Enlazador** (linker) se encarga de incluir en el programa determinadas librerías que son necesarias para que el programa pueda realizar ciertas tareas como, por ejemplo, manejar los dispositivos de entrada/salida.

El **cargador** se encarga de ubicar el programa en memoria.

3. FASES DE UN COMPILADOR

El proceso de compilación se divide en dos fases:

- **Análisis**: analiza la entrada para comprobar la corrección del programa fuente y genera estructuras intermedias necesarias para comenzar la síntesis
- **Síntesis o Generación**: Construir el programa objeto a partir de las estructuras generadas por la fase de análisis.



Se llama **Front End** al conjunto de fases que dependen del lenguaje fuente y que son independientes de la máquina: Análisis léxico + Análisis sintáctico + Análisis semántico + Generador de código intermedio.

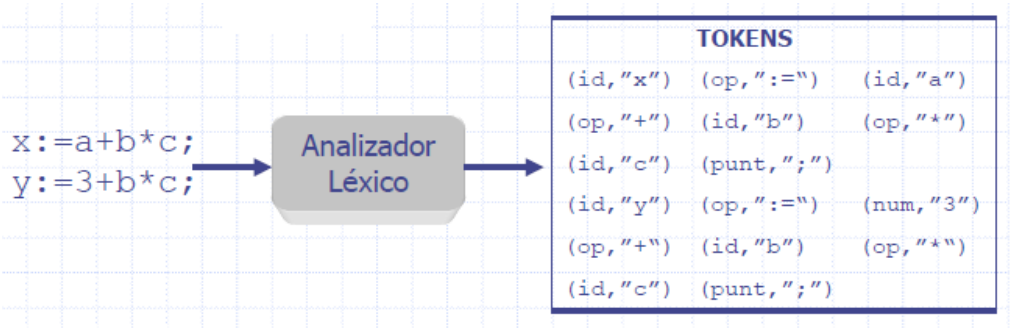
Se llama **Back End** al conjunto de fases que dependen de la máquina objeto donde se va a ejecutar el programa. (Optimizador de código + Generador de código)

3.1 FASES DE ANÁLISIS

Analizador Léxico (scanner)

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres del programa y elaborar como salida una secuencia de componentes léxicos básicos o tokens: literales, identificadores, (variables, constantes, nombres de métodos,...), palabras reservadas, signos de puntuación, etc. A cada componente le asocia la categoría a la que pertenece.

Por ejemplo:



Otras funciones que realiza:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información sobre donde se ha producido.
- Avisar de errores léxicos. Por ejemplo, si @ no pertenece al lenguaje, avisar de un error.

Analizador Sintáctico

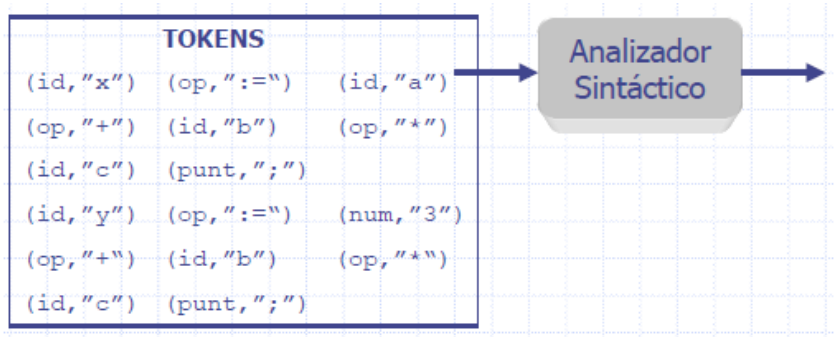
Comprueba que la estructura de los componentes básicos sea correcta según ciertas reglas gramaticales.

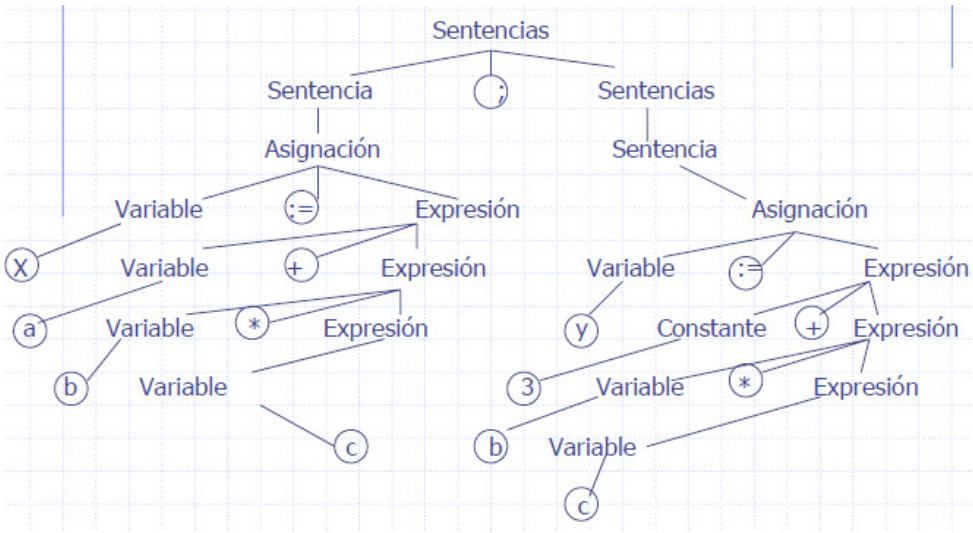
En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce.

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar código intermedio.
- Generar errores cuando se producen.





El analizador sintáctico necesita de una gramática que le proporcione la especificación sintáctica del lenguaje fuente.

Ejemplo de gramática simple:

```
Sentencias ::= Sentencia ";" Sentencias | Sentencia
Sentencia ::= Asignación | Condicional | Iterativa
Asignación ::= Variable ":=" Expresión
Condicional ::= "if" Condición "then" Sentencias "else"
   Sentencias
Iterativa ::= "while" Condición "do" Sentencias
Expresión ::= Variable-Número "+" Expresión |
              Variable-Número "*" Expresión |
              Variable-Número "-" Expresión |
              Variable-Número "/" Expresión |
              Variable-Número
Variable ::= [A-Za-z] [A-Za-z0-9]*
Variable-Número ::= Variable | Número
Número ::= [0-9]+
```

Analizador Semántico

Esta fase revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza el árbol sintáctico del análisis anterior para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Además se comprueba el rango de valores, existencia de variables, etc.

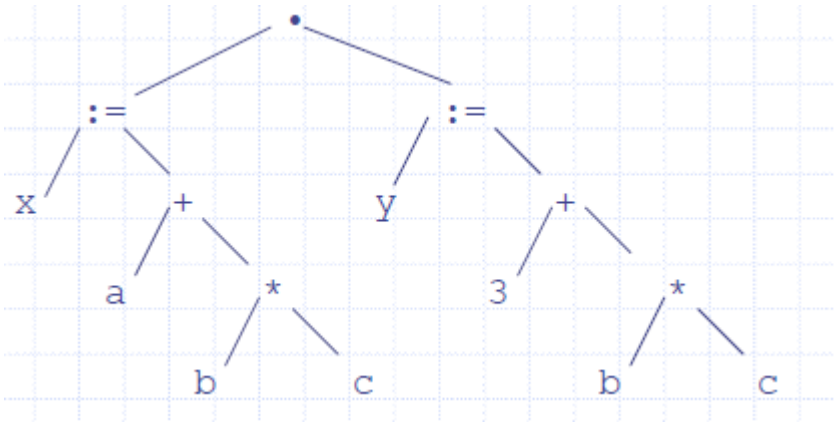


Tabla de símbolos

Durante el análisis de un programa, el compilador recopila una gran cantidad de información que es necesario tener organizada y fácilmente accesible (para hacer las comprobaciones semánticas y para generar el código); la principal estructura de datos donde el compilador almacena la información que obtiene del programa analizado se denomina tabla de símbolos.

También se la llama tabla de nombres o tabla de identificadores y tiene dos funciones principales:

- Efectuar chequeos semánticos.

- Generación de código.

Permanece sólo en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración.

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc. También sirve para guardar información referente a los tipos creados por el usuario, tipos enumerados y, en general, a cualquier identificador creado por el usuario.

3.2 FASES DE SÍNTESIS

Generador de Código Intermedio

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia del programa fuente independiente de la máquina.

Se puede considerar esta representación intermedia como un programa para una máquina abstracta.

Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir al programa objeto.

Aunque un programa fuente se puede traducir directamente al lenguaje objeto, una ventaja de utilizar una forma intermedia independiente de la máquina es que se facilita la portabilidad del programa.

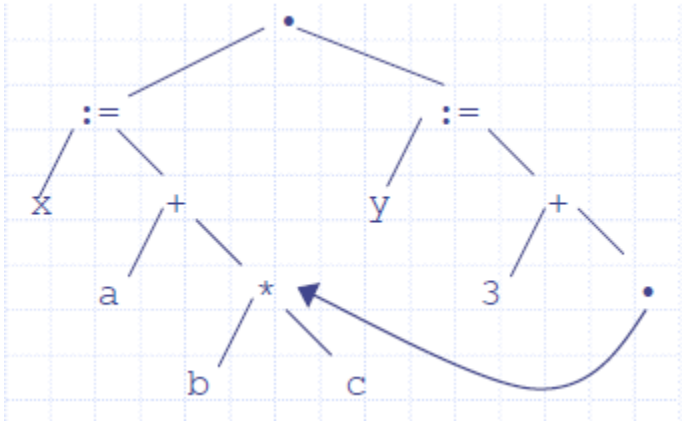
Se puede crear un compilador para una máquina distinta uniendo una etapa final (Back End) para la nueva máquina a una etapa inicial (Front End) ya existente.

El uso de código intermedio reduce la complejidad del desarrollo de compiladores ya que front ends y back ends comparten un código intermedio común.

Optimizador

La fase de optimización de código, trata de mejorar el código intermedio, de modo que resulte un código de máquina más eficiente posible.

En algunos casos también se realiza una optimización del código intermedio. Algunas optimizaciones que se pueden realizar son la evaluación de expresiones constantes, el uso de ciertas propiedades de los operadores, tales como la asociativa, conmutativa, y distributiva, así como la reducción de expresiones comunes.



Generación de código máquina

La fase degeneración de código máquina tiene por objeto traducir la secuencia de instrucciones de código intermedio en una colección de instrucciones ejecutables dependiente de la máquina donde se ejecuta.

4. DEFINICIONES RELACIONADAS CON COMPILADORES

Compilador cruzado: Compilador que traduce un lenguaje fuente a objeto, el objeto es para un ordenador distinto del que compila.

Compilador de una o varias pasadas: Una "Pasada" es cada recorrido total de todo el código fuente. En cada pasada se realiza alguna misión específica.

Compilador incremental: Una vez encontrados y corregidos los errores, solo se compilan estos.

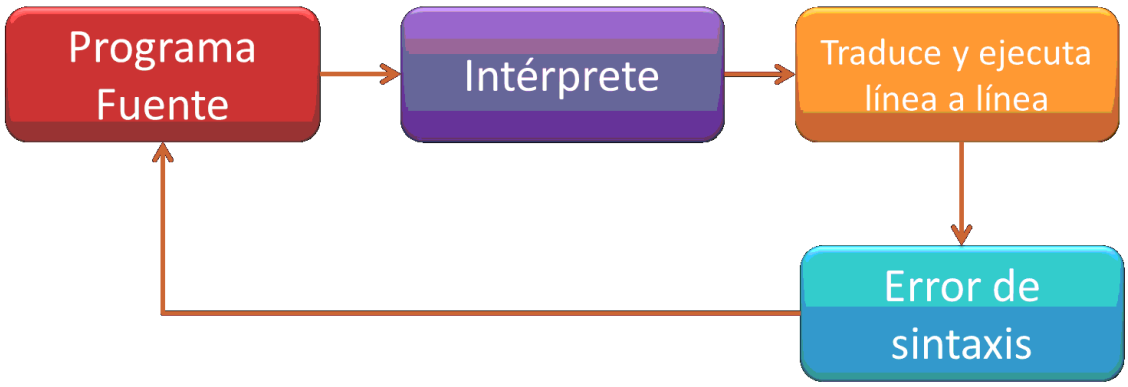
Autocompilador: Compilador escrito en el propio lenguaje que compila

Metacompilador: Programa que recibe un lenguaje y genera un compilador para ese lenguaje.

Decompilador: Programa que recibe como entrada código máquina y lo traduce a un lenguaje de alto nivel.

5. INTÉRPRETES

Un **intérprete** es un programa que simultáneamente traduce y ejecuta cada instrucción de un programa fuente.



La principal diferencia con los lenguajes compilados, es que una vez traducida cada instrucción, no se guarda en memoria, sino que se ejecuta y a continuación se elimina.

La principal ventaja que presentan los lenguajes interpretados frente a los compilados es que los programas ocupan menos memoria ya que no se guarda una versión completa del programa traducido en memoria.

También presentan algunas desventajas:

- Los programas interpretados son más lentos, ya que han de ser traducidos mientras que se ejecutan.
- El código máquina producido es peor ya que no se optimiza al valorar una sola línea cada vez. El código optimizado permite estudiar varias líneas a la vez para producir el mejor código máquina posible, por ello no es posible mediante el uso de intérpretes.
- Todos los errores son errores en tiempo de ejecución, no se pueden detectar antes de lanzar el programa. Esto hace que la depuración de los errores sea más compleja.
- Para poder ejecutar el programa es necesario tener el intérprete ejecutándose en la máquina, cosa que no es necesaria en el caso de los compiladores ya que en este caso bastará con tener una copia del programa ya traducido a código máquina para que el procesador sea capaz de ejecutarlo.

6. FASES DE UN INTÉRPRETE

La fase de análisis coincide con la fase de análisis de un compilador. A continuación se genera directamente el código máquina de la instrucción que se está tratando.

7. TIPOS DE INTÉRPRETES

En función de la estructura interna del intérprete se pueden distinguir:

- Intérpretes puros
- Intérpretes avanzados
- Intérpretes incrementales

Intérpretes puros

Analizan y ejecutan sentencia a sentencia todo el programa fuente. Siguen el modelo de interpretación iterativa y, por tanto, se utilizan principalmente para lenguajes sencillos.

Los intérpretes puros se han venido utilizando desde la primera generación de ordenadores al permitir la ejecución de largos programas en ordenadores de memoria reducida, ya que sólo

debían contener en memoria el intérprete y la sentencia a analizar y ejecutar en cada momento.

El principal problema de este tipo de intérpretes es que si a mitad del programa fuente se producen errores, se debe de volver a comenzar el proceso.

Intérpretes avanzados

Los intérpretes avanzados incorporan un paso previo de análisis de todo el programa fuente. Generando posteriormente un lenguaje intermedio que es ejecutado por ellos mismos.

De esta forma en caso de errores sintácticos no pasan de la fase de análisis.

Se utilizan para lenguajes más avanzados que los intérpretes puros, ya que permiten realizar un análisis más detallado del programa fuente (comprobación de tipos, optimización de instrucciones, etc.)

Intérpretes incrementales

Existen ciertos lenguajes que, por sus características, no se pueden compilar directamente. La razón es que pueden manejar objetos o funciones que no son conocidos en tiempo de compilación, ya que se crean dinámicamente en tiempo en ejecución. Entre estos lenguajes, pueden considerarse Smalltalk, Lisp o Prolog.

Con el propósito de obtener una mayor eficiencia que en la interpretación simple, se diseñan compiladores incrementales. La idea es compilar aquellas partes estáticas del programa en lenguaje fuente, marcando como dinámicas las que no puedan compilarse.

Posteriormente, en tiempo de ejecución, el sistema podrá compilar algunas partes dinámicas o recompilar partes dinámicas que hayan sido modificadas.

Estos sistemas no producen un código objeto independiente, sino que acompañan el sistema que permite compilar módulos en tiempo de ejecución (*run time system*) al código objeto generado.

Normalmente, los compiladores incrementales se utilizan en sistemas interactivos donde conviven módulos compilados con módulos modificables

8. MÁQUINA VIRTUAL

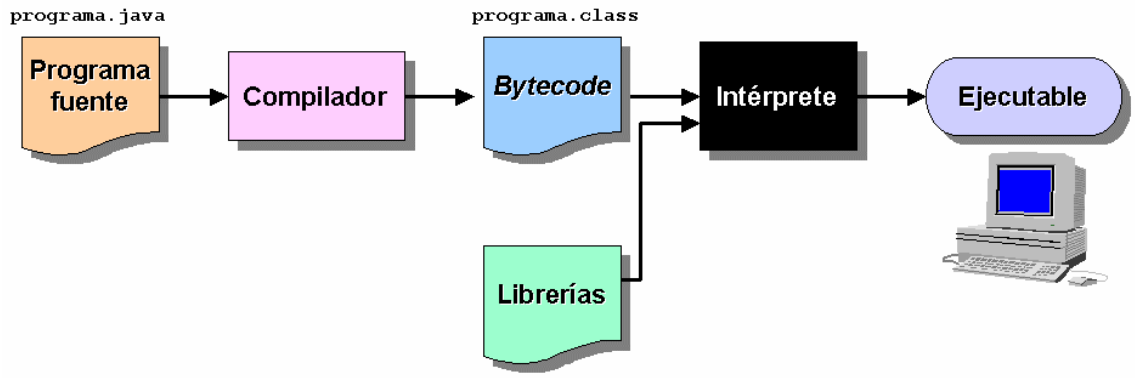
En los últimos años se está utilizando una solución intermedia entre compiladores e intérpretes. La solución consiste en que se define una máquina teórica o **máquina virtual** con su lenguaje máquina ficticio.

El código fuente se compila a ese lenguaje máquina virtual. El lenguaje máquina virtual se traduce mediante un intérprete al código máquina real.

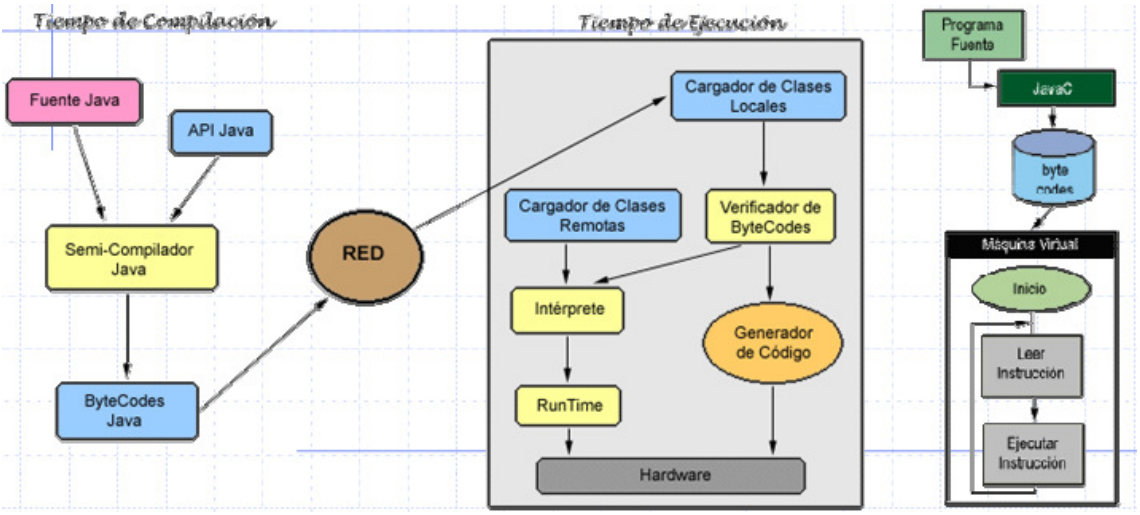
Esta es la solución adoptada por Java. Un programa escrito en JAVA se traduce en dos fases:

- En una primera fase el programa es **compilado**, generando un fichero en lo que se llama **bytecode**. El bytecode es un pseudolenguaje universal que sólo es entendido por la denominada **Máquina Virtual Java** (VM). La VM no es ninguna máquina real, sino una abstracción.
- En una segunda fase, el bytecode es **interpretado** por la VM y traducido al código máquina nativo de cada máquina particular.

Por eso Java es multi-plataforma, ya que existe un intérprete para cada máquina diferente. Por tanto, la compilación se produce una vez y la interpretación cada vez que el programa se ejecuta.



Esta característica es la que hace tan popular al lenguaje Java en el entorno de Internet: un servidor Web puede enviar junto con el código HTML de una página un archivo en bytecode a cualquier ordenador de la red, y éste es el que se encarga de traducirlo a su propio código nativo.



Ventajas de este sistema:

Se compila la aplicación una única vez y los ejecutables en bytecode obtenidos son válidos para cualquier plataforma.

La máquina virtual Java es capaz de detectar y notificar gran cantidad de errores durante la ejecución de la aplicación (como accesos a elementos fuera de un vector)

Los ejecutables son pequeños porque las librerías de clases vienen proporcionadas junto a la JVM en el JRE de la plataforma concreta

Inconvenientes:

Velocidad. Evidentemente la interpretación o incluso compilación just-in-time del bytecode produce aplicaciones más lentas que en el caso de la ejecución directa de un código máquina.

La generalidad del bytecode tiene como inconveniente que no se aprovecha totalmente la potencia de la máquina y del sistema operativo.

La misma estrategia ha seguido más recientemente Microsoft con su arquitectura **.Net** en que los programas fuente se compilan al MSIL (Microsoft Intermediate Language) para luego, ese código MSIL, ser interpretado al lenguaje máquina real.

9. COMPILADORES JUST-IN-TIME

Actualmente las máquinas virtuales modernas realizan una compilación JIT (Just In Time) en donde el bytecode no es interpretado sino que se compila directamente a código máquina en tiempo de ejecución de acuerdo con la arquitectura física en la que se ejecuta la máquina virtual.

Esto permite no tener que volver a traducir instrucciones ya procesadas, con lo que se pueden conseguir velocidades de ejecución similares al C. En la práctica las máquinas virtuales suelen utilizar técnicas mixtas de interpretación/compilación JIT normalmente según la frecuencia de paso por un bytecode concreto.

Ejercicio 1: Compiladores vs Intérpretes

Ejercicio 2: Máquina Virtual de Java vs Microsoft .NET