

Traductores e intérpretes

Dpto. de Ingeniería de la Información y las Comunicaciones



Índice: secciones principales

- ① Qué son y por qué surgen los compiladores
- ② Evolución de los programas traductores
- ③ Traductores. Distintos tipos
- ④ Compiladores. Estructura, base gramatical y tipos
- ⑤ Intérpretes
- ⑥ Máquinas reales y abstractas
- ⑦ Compiladores interpretados
- ⑧ Compiladores portables

Qué son y por qué surgen los compiladores

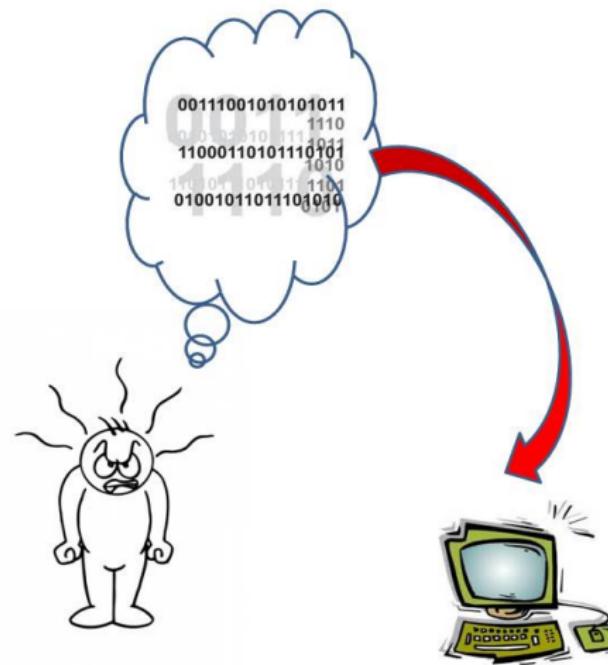
Inicialmente la programación se realizaba en lenguaje máquina...

```
00111001010101011  
010010101011110110  
11000110101110101  
10101  
100001010111110101  
01001011011101010
```



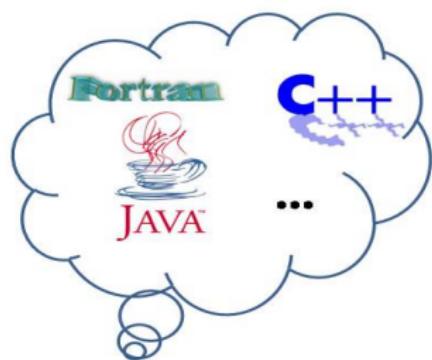
Qué son y por qué surgen los compiladores

...y eso la convertía en una tarea muy complicada.



Qué son y por qué surgen los compiladores

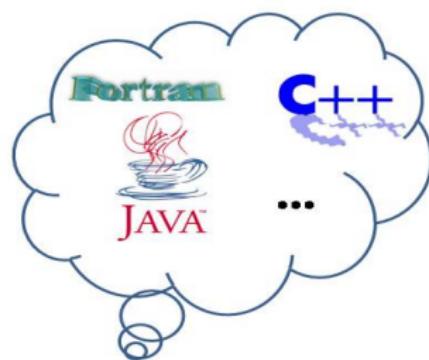
Con la aparición de los lenguajes de alto nivel, la programación de ordenadores se convierte en una tarea mucho más sencilla.



Esto provocó, además, que el mantenimiento de los programas fuera también más fácil.

Qué son y por qué surgen los compiladores

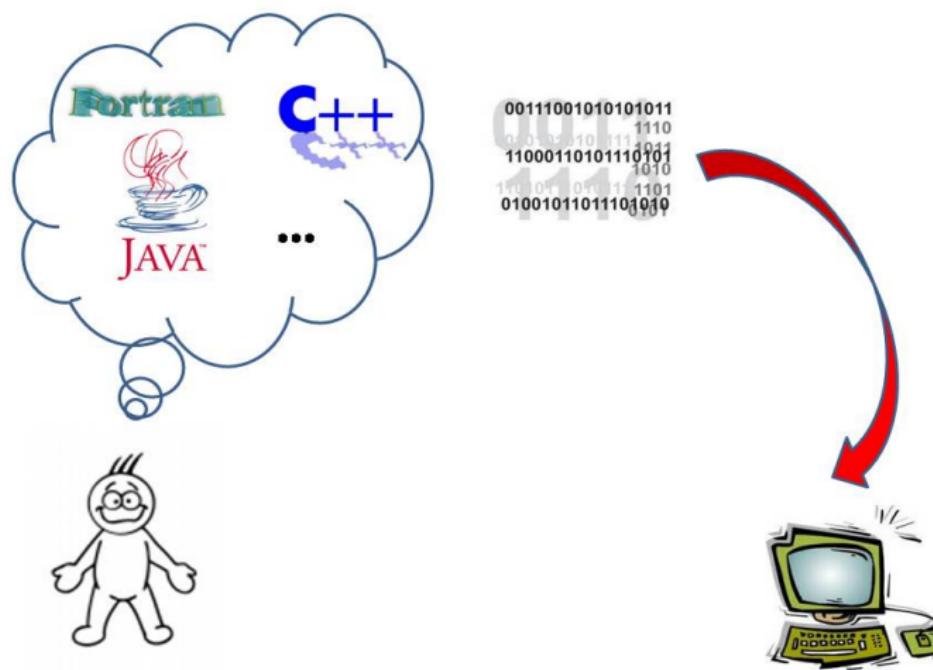
Con la aparición de los lenguajes de alto nivel, la programación de ordenadores se convierte en una tarea mucho más sencilla.



Esto provocó, además, que el mantenimiento de los programas fuera también más fácil.

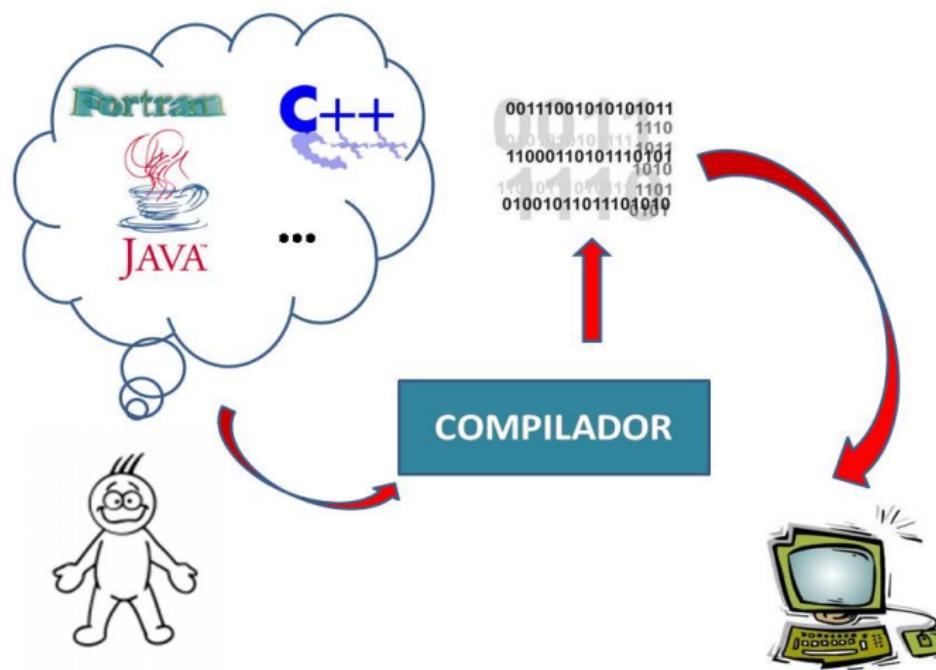
Qué son y por qué surgen los compiladores

Sin embargo, los ordenadores siguen entendiendo únicamente código máquina.



Qué son y por qué surgen los compiladores

Por esta razón, se necesita un programa que traduzca desde lenguaje de alto nivel a lenguaje máquina: **el compilador**.



Qué son y por qué surgen los compiladores

Un **compilador**, por tanto, traduce un programa escrito en lenguaje de alto nivel (lenguaje fuente) en un programa equivalente escrito en algún lenguaje de bajo nivel (lenguaje destino).

Una función importante del compilador es **informar de cualquier error** que detecte en el programa fuente durante el proceso de traducción.

Podría ejecutarse el programa destino si está escrito en l. máquina.



Qué son y por qué surgen los compiladores

Un **compilador**, por tanto, traduce un programa escrito en lenguaje de alto nivel (lenguaje fuente) en un programa equivalente escrito en algún lenguaje de bajo nivel (lenguaje destino).

Una función importante del compilador es **informar de cualquier error** que detecte en el programa fuente durante el proceso de traducción.

Podría ejecutarse el programa destino si está escrito en l. máquina.



Qué son y por qué surgen los compiladores

Otro tipo común de *procesador de lenguaje* es el **intérprete**, que no genera *programa destino* como una traducción, sino que da la apariencia de ejecutar directamente las operaciones especificadas en el *programa fuente* con las entradas proporcionadas por el usuario.



Qué son y por qué surgen los compiladores

Lo primero que podríamos plantearnos es **por qué es posible realizar un traductor para un lenguaje de programación y el hacerlo para un lenguaje natural exige aplicar técnicas de IA.**

Para contestar a esta pregunta, debemos plantearnos otras: **¿Qué surgió antes en un lenguaje natural, las reglas gramaticales o el propio lenguaje? ¿Y en un lenguaje formal?**

Y todo esto nos conduce a reflexionar acerca de la **base grammatical de los procesadores de lenguaje**, de la que hablaremos más adelante.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Lenguaje de alto nivel.
 - Se convierte en un lenguaje de máquina.
 - Permite la programación de cálculos numéricos.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Lenguaje de alto nivel.
 - Se convierte en el lenguaje más popular para la ciencia y la ingeniería.
 - Se convierte en el lenguaje más antiguo que sigue siendo usado.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la creación del primer compilador, para el que se necesitaron 16 años-persona y 25000 líneas de código.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la **aparición del primer compilador**, para el que se necesitaron 18 años-persona y 25000 líneas de código.
 - El lenguaje se iba diseñando e implementando al mismo tiempo.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la **aparición del primer compilador**, para el que se necesitaron 18 años-persona y 25000 líneas de código.
 - El lenguaje se iba diseñando e implementando al mismo tiempo.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la **aparición del primer compilador**, para el que se necesitaron 18 años-persona y 25000 líneas de código.
 - El lenguaje se iba diseñando e implementando al mismo tiempo.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la **aparición del primer compilador**, para el que se necesitaron 18 años-persona y 25000 líneas de código.
 - El lenguaje se iba diseñando e implementando al mismo tiempo.
- Le siguieron **Cobol** orientado a datos de negocios, y **Lisp** para computación simbólica.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

- Los primeros ordenadores aparecieron en la década de 1940 y se programaban en **lenguaje máquina**.
 - Programación lenta y propensa a errores.
 - Programas difíciles de comprender y mantener.
- A principios de la década de 1950 se desarrollaron los **lenguajes ensambladores**. Inicialmente cada instrucción correspondía a una instrucción máquina. Después se amplían con macros.
 - Programas aún dependientes de la máquina y difíciles de mantener.
- Aparece **FORTRAN** (FORmulae TRANslator) a final de los años 50 (John Backus en IBM).
 - Implicó la **aparición del primer compilador**, para el que se necesitaron 18 años-persona y 25000 líneas de código.
 - El lenguaje se iba diseñando e implementando al mismo tiempo.
- Le siguieron **Cobol** orientado a datos de negocios, y **Lisp** para computación simbólica.
- En las siguientes décadas aparecen numerosos lenguajes innovadores, que permiten una programación más natural y robusta.

Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:

**LENGUAJES DE 1^a
GENERACIÓN**

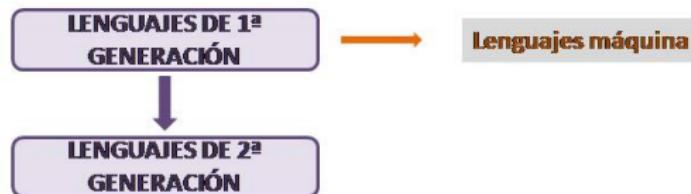
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



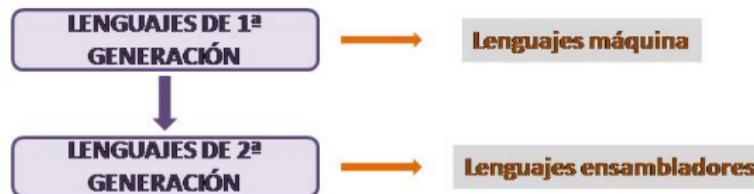
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



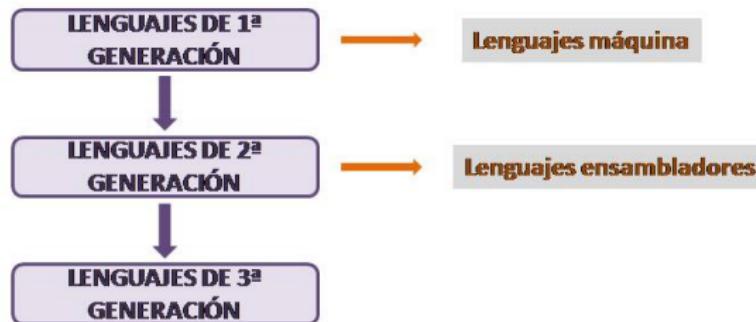
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



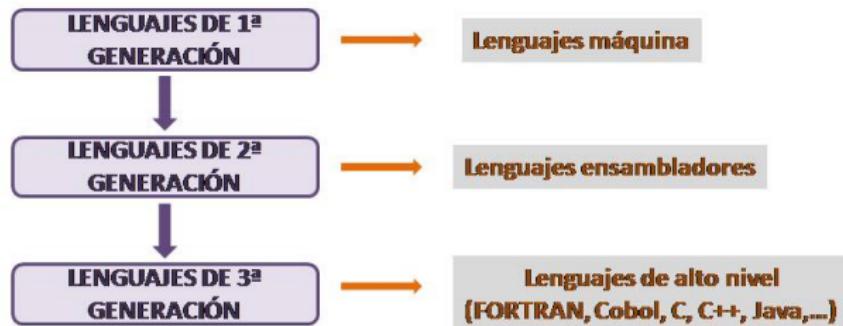
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



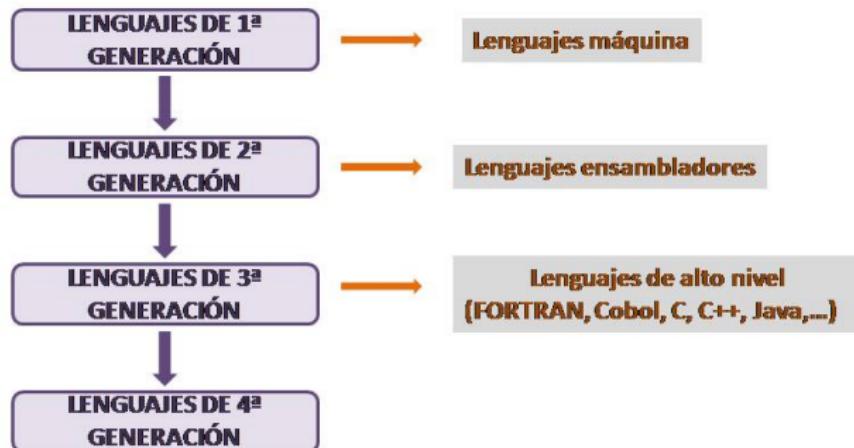
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



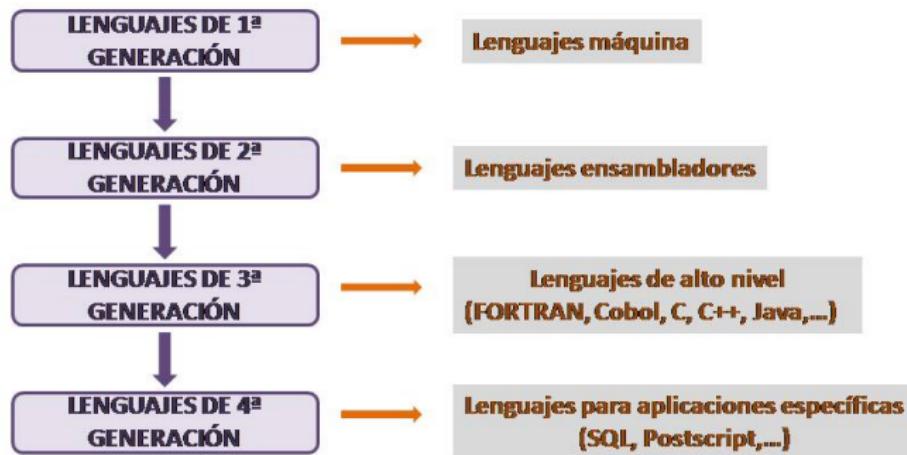
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



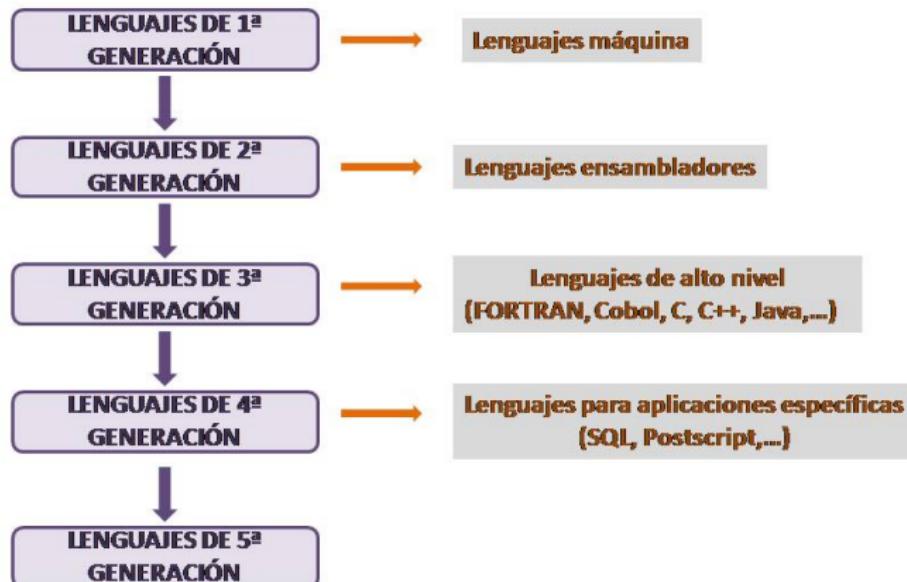
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



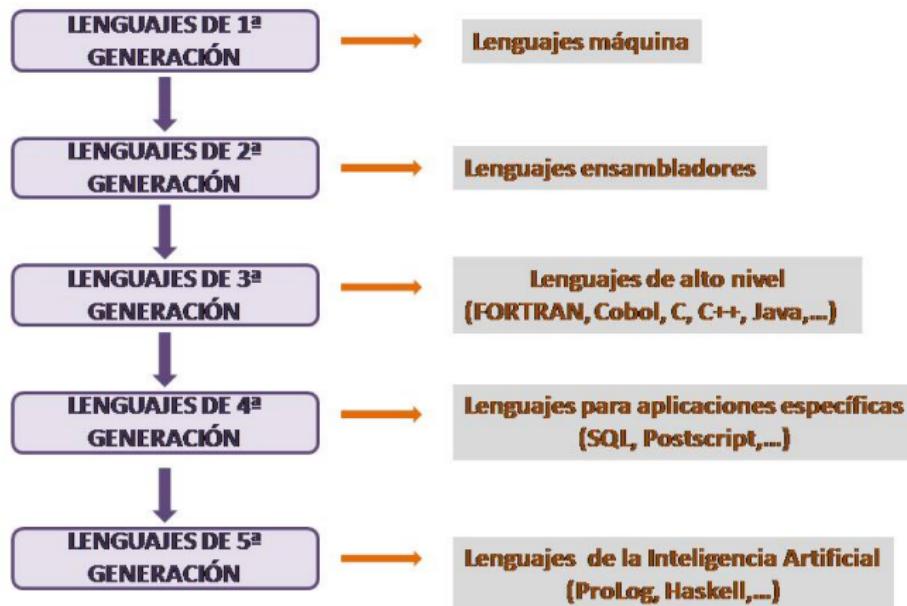
Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



Evolución de los programas traductores

Aunque no existe consenso absoluto acerca de la denominación de las **distintas generaciones de lenguajes**, una posible sería la siguiente:



Evolución de los programas traductores

La aparición de los lenguajes de alto nivel hizo necesaria la aparición de los compiladores.

La tarea de diseñar y desarrollar un compilador se ha simplificado enormemente desde entonces, gracias a:

- » El desarrollo de buenos lenguajes de implementación.
- » La aparición de entornos de programación y herramientas software adecuados.

Pero, sobre todo

el desarrollo de teoría que nos permite comprender el problema y las soluciones que se han ido desarrollando.

De hecho, *el estudio de los compiladores es también el estudio sobre cómo la teoría se encuentra con la práctica.*

Evolución de los programas traductores

La aparición de los lenguajes de alto nivel hizo necesaria la aparición de los compiladores.

La tarea de diseñar y desarrollar un compilador se ha simplificado enormemente desde entonces, gracias a:

- El desarrollo de buenos lenguajes de implementación.
- La aparición de entornos de programación y herramientas software adecuados.

Pero, sobre todo

- Al descubrimiento de técnicas sistemáticas, basadas en la teoría de lenguajes formales, para el manejo de muchas de las tareas que surgen en la compilación.

De hecho, *el estudio de los compiladores es también el estudio sobre cómo la teoría se encuentra con la práctica.*

Evolución de los programas traductores

La aparición de los lenguajes de alto nivel hizo necesaria la aparición de los compiladores.

La tarea de diseñar y desarrollar un compilador se ha simplificado enormemente desde entonces, gracias a:

- El desarrollo de buenos lenguajes de implementación.
- La aparición de entornos de programación y herramientas software adecuados.

Pero, sobre todo

- Al descubrimiento de técnicas sistemáticas, basadas en la **teoría de lenguajes formales**, para el manejo de muchas de las tareas que surgen en la compilación.

De hecho, *el estudio de los compiladores es también el estudio sobre cómo la teoría se encuentra con la práctica.*

Evolución de los programas traductores

La aparición de los lenguajes de alto nivel hizo necesaria la aparición de los compiladores.

La tarea de diseñar y desarrollar un compilador se ha simplificado enormemente desde entonces, gracias a:

- El desarrollo de buenos lenguajes de implementación.
- La aparición de entornos de programación y herramientas software adecuados.

Pero, sobre todo

- Al descubrimiento de técnicas sistemáticas, basadas en la **teoría de lenguajes formales**, para el manejo de muchas de las tareas que surgen en la compilación.

De hecho, *el estudio de los compiladores es también el estudio sobre cómo la teoría se encuentra con la práctica.*

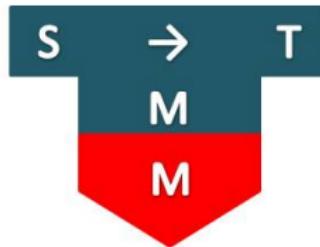
Traductores. Distintos tipos

Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



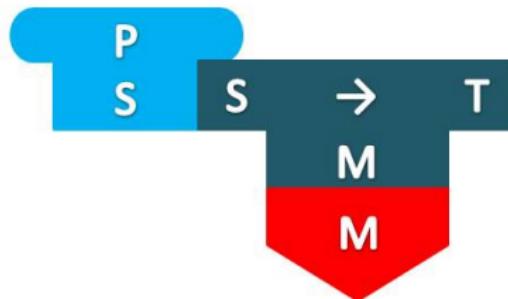
Traductores. Distintos tipos

Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



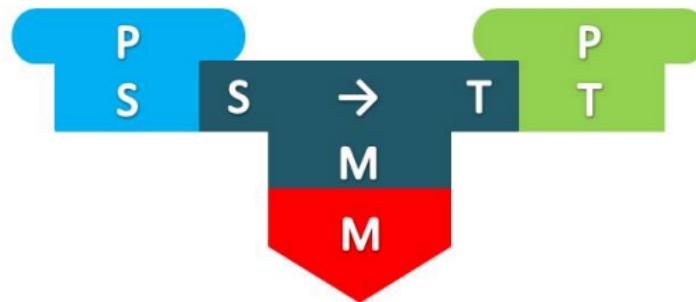
Traductores. Distintos tipos

Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



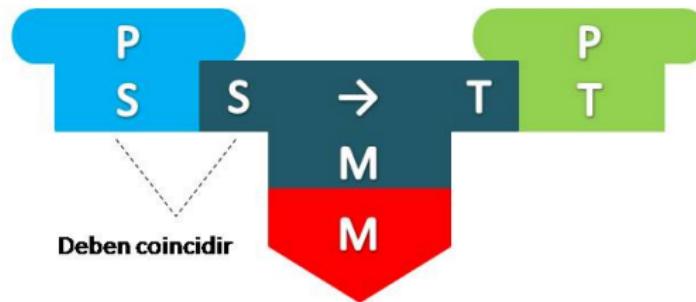
Traductores. Distintos tipos

Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



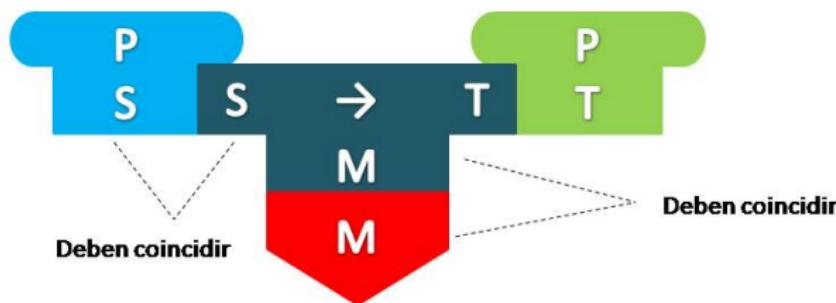
Traductores. Distintos tipos

Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



Traductores. Distintos tipos

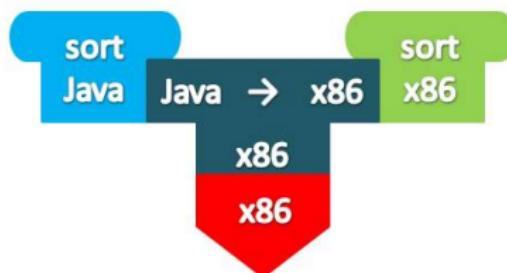
Generalizamos el concepto de compilador definiendo un **traductor** como un programa que acepta texto expresado en **lenguaje fuente** (S) y genera texto semánticamente equivalente expresado en **lenguaje destino** (T). El **lenguaje de implementación** (M) es aquel en el que está escrito el traductor, puesto que, al ser éste también un programa, debe ejecutarse sobre alguna máquina.



Traductores. Distintos tipos

Ejemplo de traducción...

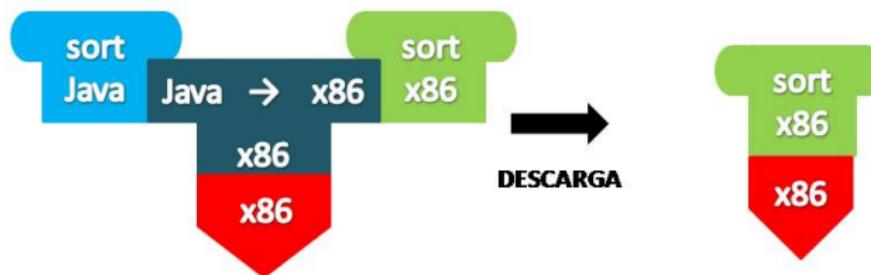
...del programa sort escrito en Java y su posterior ejecución.



Traductores. Distintos tipos

Ejemplo de traducción...

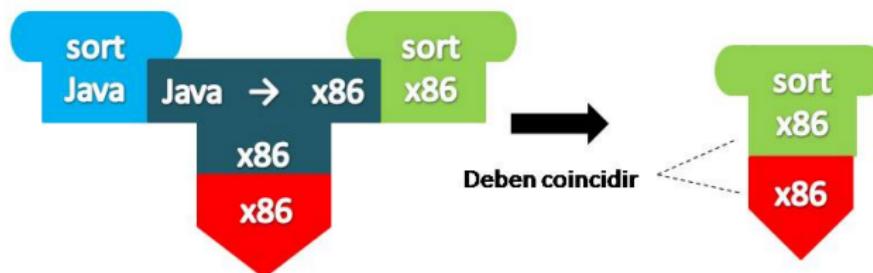
...del programa sort escrito en Java y su posterior ejecución.



Traductores. Distintos tipos

Ejemplo de traducción...

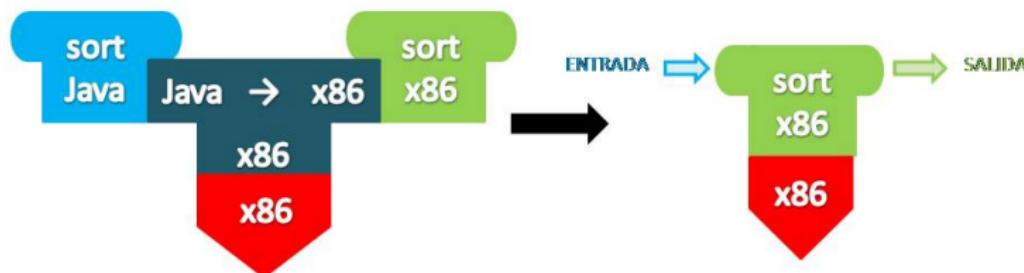
...del programa sort escrito en Java y su posterior ejecución.



Traductores. Distintos tipos

Ejemplo de traducción...

...del programa sort escrito en Java y su posterior ejecución.



Traductores. Distintos tipos

Reglas para el correcto comportamiento de un traductor:

- Puede ejecutarse sobre una máquina M si y sólo si está escrito en código máquina para M.
- El programa fuente debe escribirse en el lenguaje fuente S del traductor.
- El programa objeto estará escrito en el lenguaje destino T del traductor.
- El programa objeto debe ser semánticamente equivalente al programa fuente.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el lenguaje de implementación es el lenguaje fuente, el procesador de lenguaje puede usarse para procesarse a sí mismo.



Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el lenguaje de implementación es el lenguaje fuente, el procesador de lenguaje puede usarse para procesarse a sí mismo.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el **lenguaje de implementación es el lenguaje fuente**, el procesador de lenguaje puede usarse para procesarse a sí mismo.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el **lenguaje de implementación es el lenguaje fuente**, el procesador de lenguaje puede usarse para procesarse a sí mismo.

→ El primer compilador capaz de compilar su propio código fuente fue el creado para Lisp por Hart y Levin en el MIT en 1962.

→ Los compiladores más avanzados hoy en día generan código optimizado para la arquitectura de destino, pero siguen usando el lenguaje de implementación como lenguaje fuente.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el **lenguaje de implementación es el lenguaje fuente**, el procesador de lenguaje puede usarse para procesarse a sí mismo.

El primer compilador capaz de compilar su propio código fuente fue el creado para LISP por Hart y Levin en el MIT en 1962.

- Desde 1970 se ha convertido en una práctica común escribir el compilador en el mismo lenguaje que éste compila, aunque Pascal y C han sido alternativas muy usadas.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el **lenguaje de implementación es el lenguaje fuente**, el procesador de lenguaje puede usarse para procesarse a sí mismo.

- El primer compilador capaz de compilar su propio código fuente fue el creado para **Lisp** por Hart y Levin en el MIT en 1962.
- Desde 1970 se ha convertido en una práctica común escribir el compilador en el mismo lenguaje que éste compila, aunque Pascal y C han sido alternativas muy usadas.

Traductores. Distintos tipos

En la traducción de textos que son programas:

- **Programa fuente** Texto en lenguaje fuente.
- **Programa objeto** Texto en lenguaje destino.

Antes de la traducción hay que comprobar la sintaxis y la semántica del programa fuente (con informe de errores, en su caso), para generar un programa objeto semánticamente equivalente.

Arranque (bootstrapping) Cuando el **lenguaje de implementación es el lenguaje fuente**, el procesador de lenguaje puede usarse para procesarse a sí mismo.

- El primer compilador capaz de compilar su propio código fuente fue el creado para **Lisp** por Hart y Levin en el MIT en 1962.
- Desde 1970 se ha convertido en una práctica común escribir el compilador en el mismo lenguaje que éste compila, aunque Pascal y C han sido alternativas muy usadas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Dependiendo de sus *lenguajes fuente y destino*:

- **Compilador** Traduce desde un *lenguaje de alto nivel* a otro *lenguaje de bajo nivel*.
- **Ensamblador** Traduce *lenguaje ensamblador* en su correspondiente *código máquina*.
- **Traductor de alto nivel** Su fuente y destino son *lenguajes de alto nivel*.
- **Desensamblador** Traduce un *código máquina* en su correspondiente *lenguaje ensamblador*.
- **Descompilador** Traduce un *lenguaje de bajo nivel* en un *lenguaje de alto nivel*.
- **Preprocesador** Traduce desde una *forma extendida de un lenguaje de alto nivel* a la *forma estándar de ese mismo lenguaje*. Reúne el programa fuente, expande macros, etc.
- **Cargador y enlazador** Traducen desde *código máquina reubicable* a *código máquina absoluto*. El cargador usa las tablas de direcciones para convertir las posiciones de memoria en absolutas.

Traductores. Distintos tipos

Muchos de estos traductores trabajan usualmente de forma conjunta:



Traductores. Distintos tipos

Muchos de estos traductores trabajan usualmente de forma conjunta:



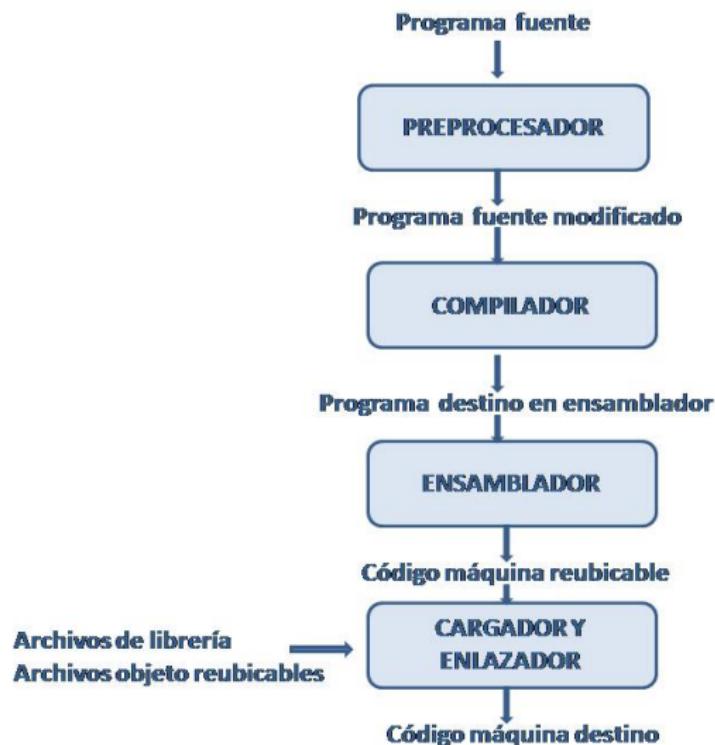
Traductores. Distintos tipos

Muchos de estos traductores trabajan usualmente de forma conjunta:



Traductores. Distintos tipos

Muchos de estos traductores trabajan usualmente de forma conjunta:



Compiladores

Como hemos dicho, los compiladores, como programas que son, pueden (y, de hecho, suelen) escribirse en lenguaje de alto nivel. Esto implica la necesidad de compilarse.

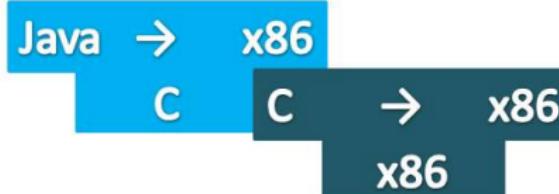
Ejemplo de compilación de un compilador de Java escrito en C:



Compiladores

Como hemos dicho, los compiladores, como programas que son, pueden (y, de hecho, suelen) escribirse en lenguaje de alto nivel. Esto implica la necesidad de compilarse.

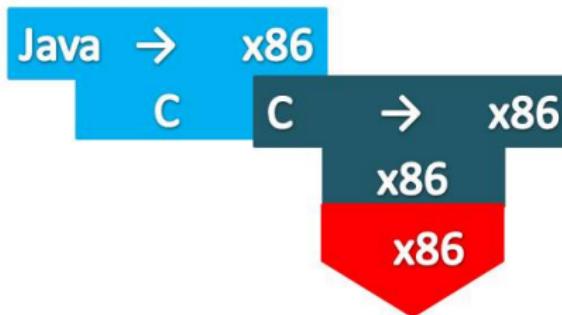
Ejemplo de compilación de un compilador de Java escrito en C:



Compiladores

Como hemos dicho, los compiladores, como programas que son, pueden (y, de hecho, suelen) escribirse en lenguaje de alto nivel. Esto implica la necesidad de compilarse.

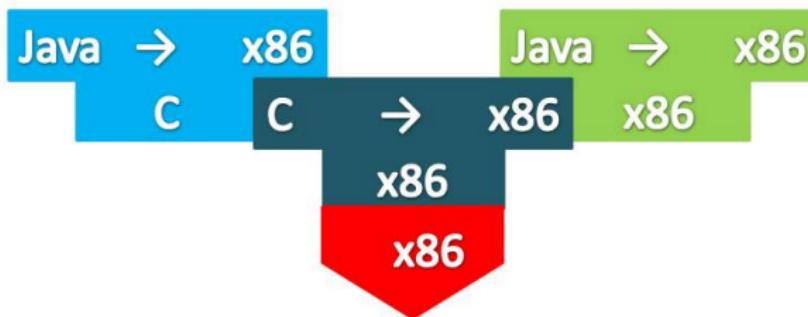
Ejemplo de compilación de un compilador de Java escrito en C:



Compiladores

Como hemos dicho, los compiladores, como programas que son, pueden (y, de hecho, suelen) escribirse en lenguaje de alto nivel. Esto implica la necesidad de compilarse.

Ejemplo de compilación de un compilador de Java escrito en C:



Fases y estructura de un compilador

Hasta ahora, hemos considerado al compilador como una caja que hace corresponder un *programa fuente* a un *programa destino* equivalente semánticamente. Si abrimos esta caja, observamos que consta de dos procesos: **análisis** y **síntesis**.

Análisis o front-end Determina la estructura y el significado del programa fuente creando una **representación intermedia** de él. Informa de los **errores de sintaxis** o de **semántica**. Recolecta información sobre el programa fuente en la **tabla de símbolos**, que, junto con la representación intermedia, es pasada a la fase de síntesis.

Síntesis o back-end Construye el programa destino a partir de la representación intermedia y de la información contenida en la tabla de símbolos.

Conceptualmente se dividen en **fases** que transforman el programa. En la práctica algunas fases desaparecen o se agrupan, sin necesidad de construir explícitamente las representaciones intermedias entre ellas.

Fases y estructura de un compilador

Hasta ahora, hemos considerado al compilador como una caja que hace corresponder un *programa fuente* a un *programa destino* equivalente semánticamente. Si abrimos esta caja, observamos que consta de dos procesos: **análisis** y **síntesis**.

Análisis o front-end Determina la estructura y el significado del programa fuente creando una **representación intermedia** de él. Informa de los **errores de sintaxis** o de **semántica**. Recolecta información sobre el programa fuente en la **tabla de símbolos**, que, junto con la representación intermedia, es pasada a la fase de síntesis.

Síntesis o back-end Construye el programa destino a partir de la representación intermedia y de la información contenida en la tabla de símbolos.

Conceptualmente se dividen en **fases** que transforman el programa. En la práctica algunas fases desaparecen o se agrupan, sin necesidad de construir explícitamente las representaciones intermedias entre ellas.

Fases y estructura de un compilador

Hasta ahora, hemos considerado al compilador como una caja que hace corresponder un *programa fuente* a un *programa destino* equivalente semánticamente. Si abrimos esta caja, observamos que consta de dos procesos: **análisis** y **síntesis**.

Análisis o front-end Determina la estructura y el significado del programa fuente creando una **representación intermedia** de él. Informa de los **errores de sintaxis** o de **semántica**. Recolecta información sobre el programa fuente en la **tabla de símbolos**, que, junto con la representación intermedia, es pasada a la fase de síntesis.

Síntesis o back-end Construye el programa destino a partir de la representación intermedia y de la información contenida en la tabla de símbolos.

Conceptualmente se dividen en **fases** que transforman el programa. En la práctica algunas fases desaparecen o se agrupan, sin necesidad de construir explícitamente las representaciones intermedias entre ellas.

Fases y estructura de un compilador



- Tareas
- Ventajas de dividir el análisis en AL y AS
- AL y **autómatas finitos**
- Herramientas para generar AL's

Fases y estructura de un compilador



- Tareas
- AS ascendente y descendente
- AS y **autómatas con pila**
- Herramientas para generar AS's

Fases y estructura de un compilador



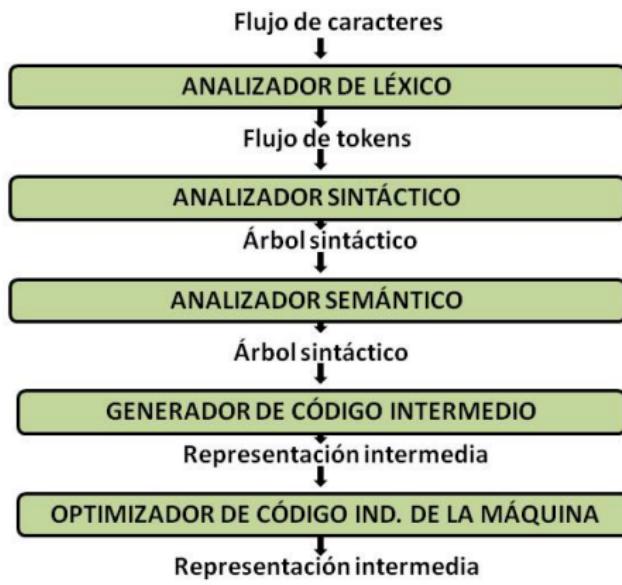
- Tareas
- Papel de la **tabla de símbolos**
- Métodos formales de descripción de la semántica

Fases y estructura de un compilador



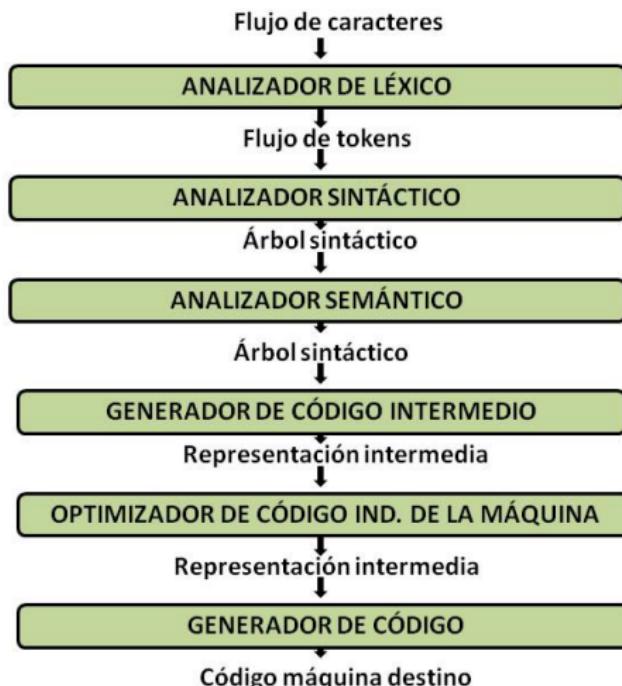
- Tareas
- Representaciones intermedias
- Propiedades que debe cumplir el código intermedio
- Distintas formas de código intermedio

Fases y estructura de un compilador



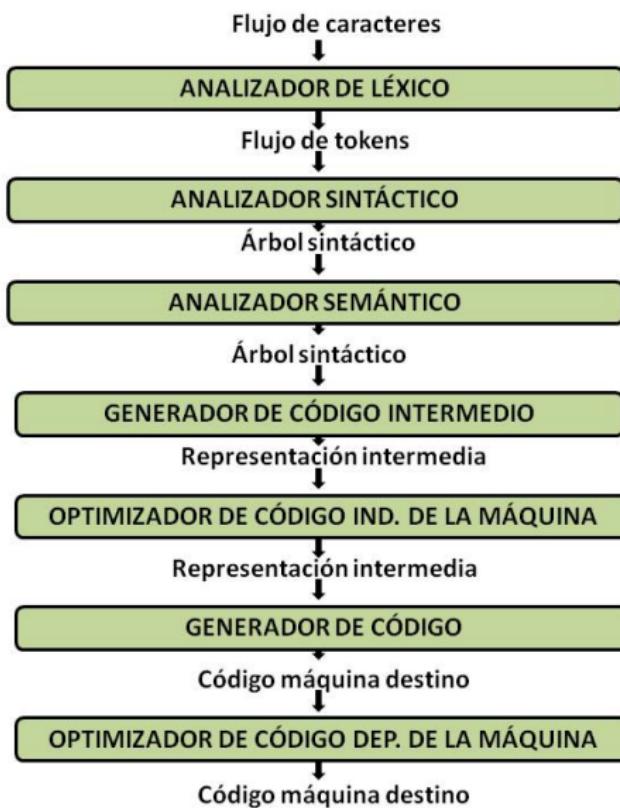
- Tareas
- Es dependiente del compilador
- Puede optimizarse el código en función del tamaño o la velocidad
- La generación de *código óptimo* es un problema **NP-Completo**
- La optimización aumenta el tiempo de compilación ⇒ a veces se puede desactivar
- Distintas técnicas de optimización

Fases y estructura de un compilador



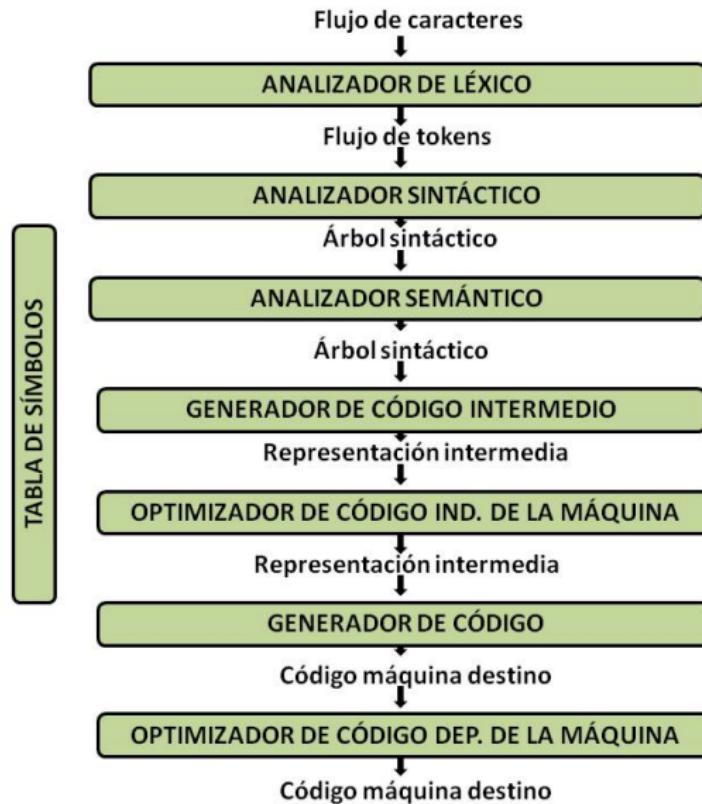
- Tareas
- Distintos métodos y posibilidades
- *Traducción dirigida por la sintaxis*

Fases y estructura de un compilador



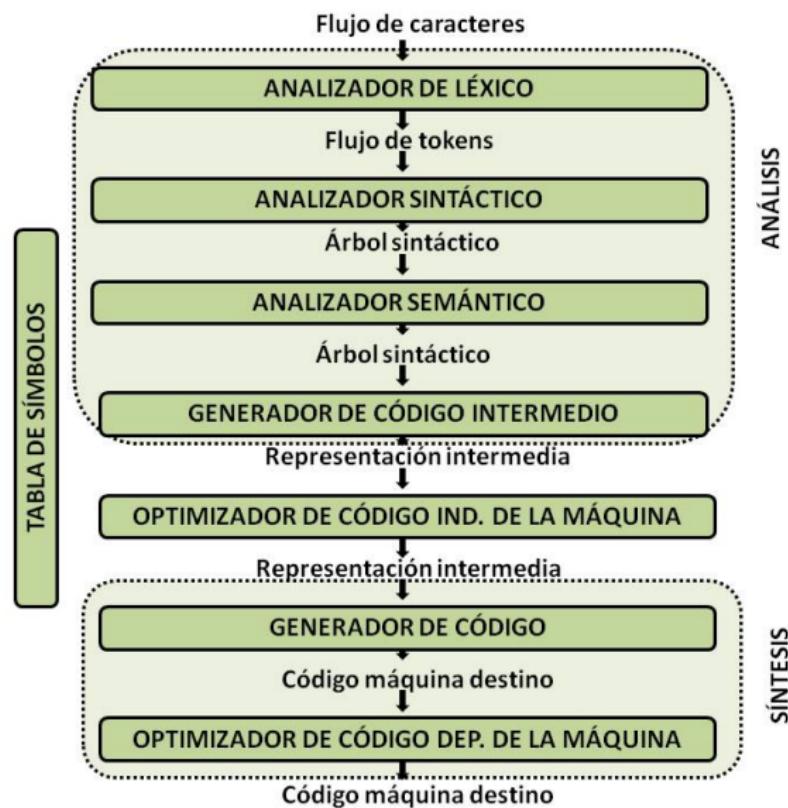
- Tareas
- Es dependiente del compilador
- Puede optimizarse el código en función del tamaño o la velocidad
- La generación de *código óptimo* es un problema **NP-Completo**
- La optimización aumenta el tiempo de compilación ⇒ a veces se puede desactivar
- Distintas técnicas de optimización

Fases y estructura de un compilador

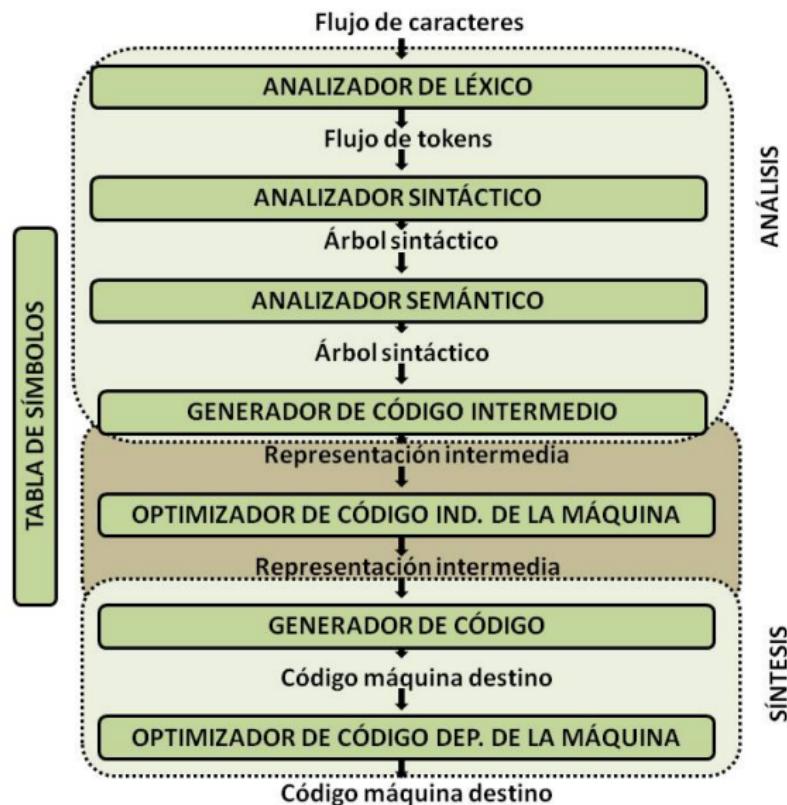


- Tareas
- Se trata de una estructura de datos con acceso rápido
- Almacena **identificadores** y reúne información sobre sus atributos. Ejemplo: Nombre, memoria asignada, tipo y ámbito para las *variables* o número y tipo de argumentos para los *procedimientos*.
- Generalmente es en el AS dónde se inicializa y se comienza a usar, actualizándose en el resto de las fases.

Fases y estructura de un compilador



Fases y estructura de un compilador



Fases y estructura de un compilador

Ejemplo de traducción de una sentencia

```
posición = inicial + velocidad * 60
```



ANALIZADOR DE LÉXICO



```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

Fases y estructura de un compilador

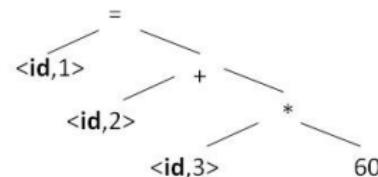
Ejemplo de traducción de una sentencia

```
posición = inicial + velocidad * 60
```

ANALIZADOR DE LÉXICO

```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

ANALIZADOR SINTÁCTICO



Fases y estructura de un compilador

Ejemplo de traducción de una sentencia

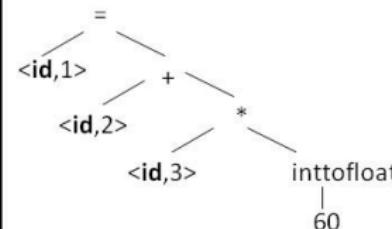
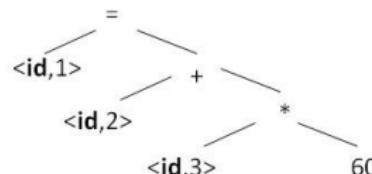
```
posición = inicial + velocidad * 60
```

ANALIZADOR DE LÉXICO

```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

ANALIZADOR SINTÁCTICO

ANALIZADOR SEMÁNTICO



Fases y estructura de un compilador

Ejemplo de traducción de una sentencia

```
posición = inicial + velocidad * 60
```

ANALIZADOR DE LÉXICO

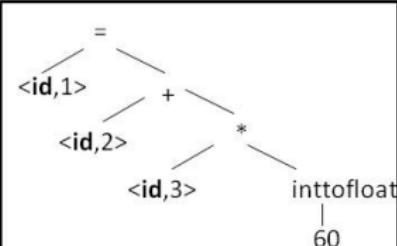
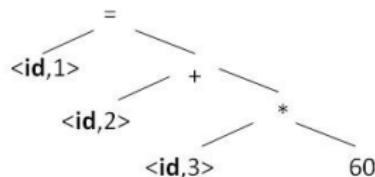
```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

ANALIZADOR SINTÁCTICO

ANALIZADOR SEMÁNTICO

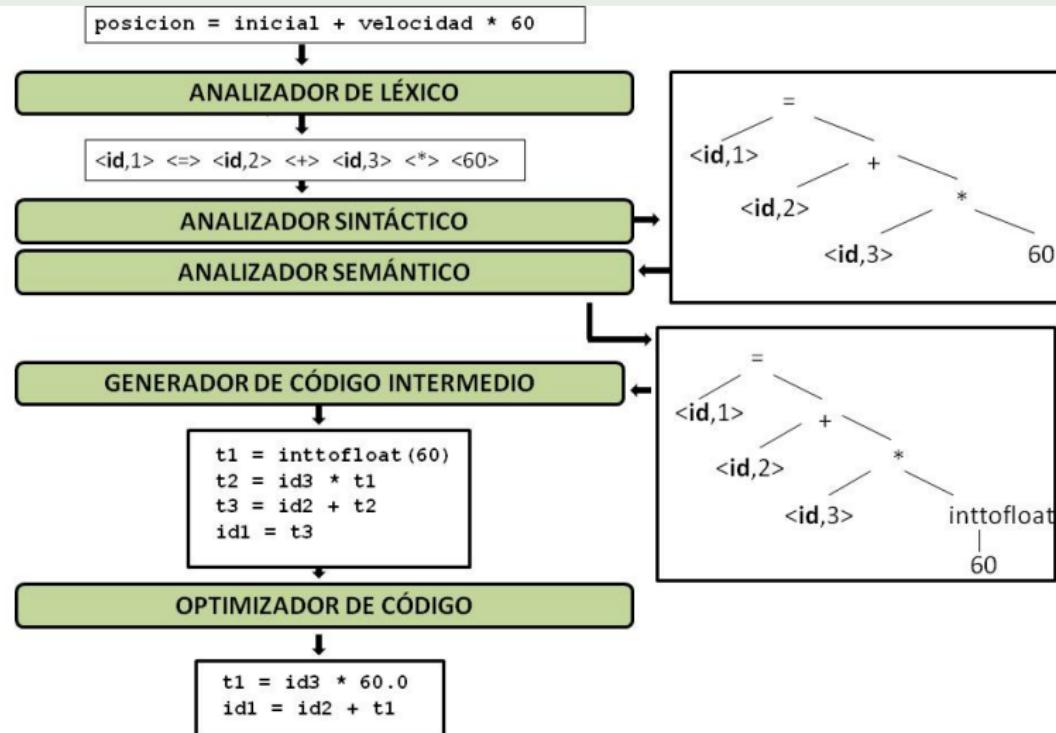
GENERADOR DE CÓDIGO INTERMEDIO

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



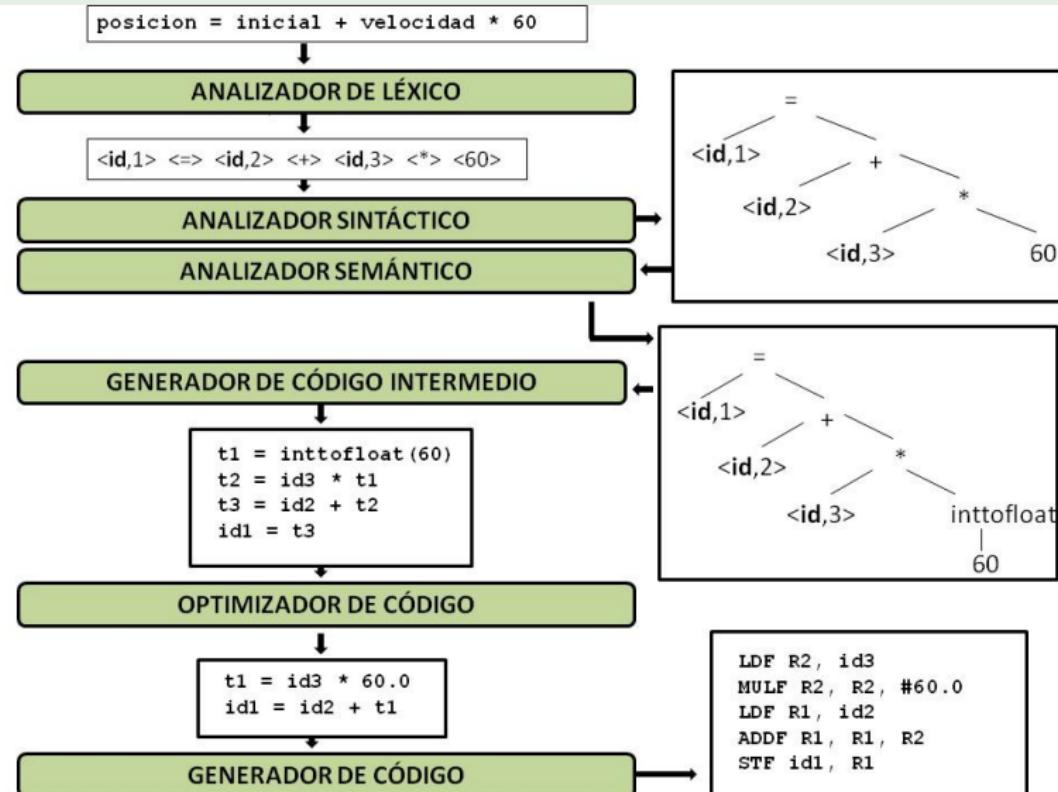
Fases y estructura de un compilador

Ejemplo de traducción de una sentencia



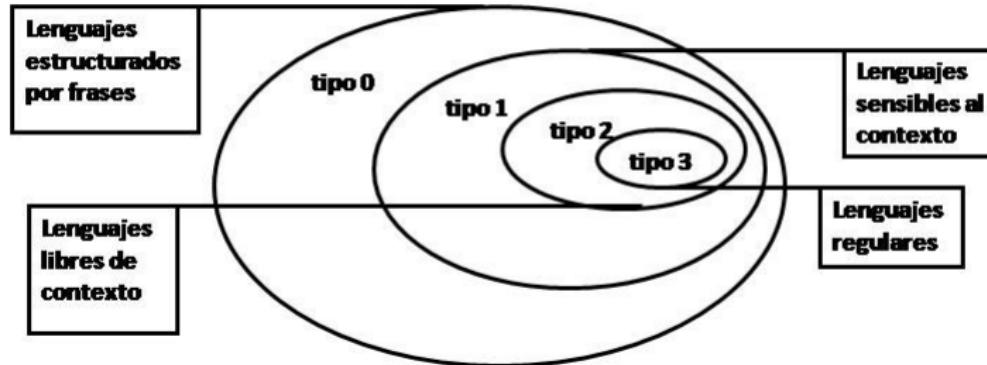
Fases y estructura de un compilador

Ejemplo de traducción de una sentencia



Base gramatical de los compiladores

Jerarquía de Chomsky

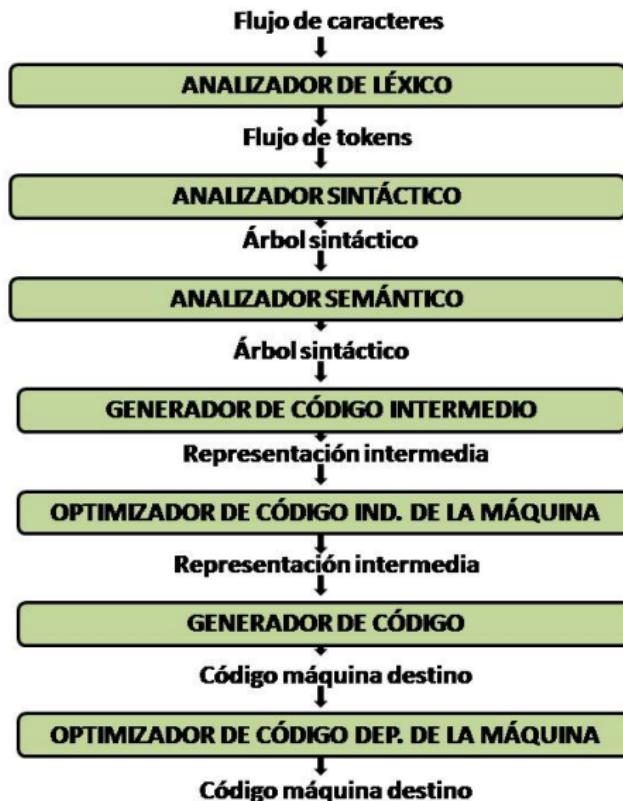


Base gramatical de los compiladores

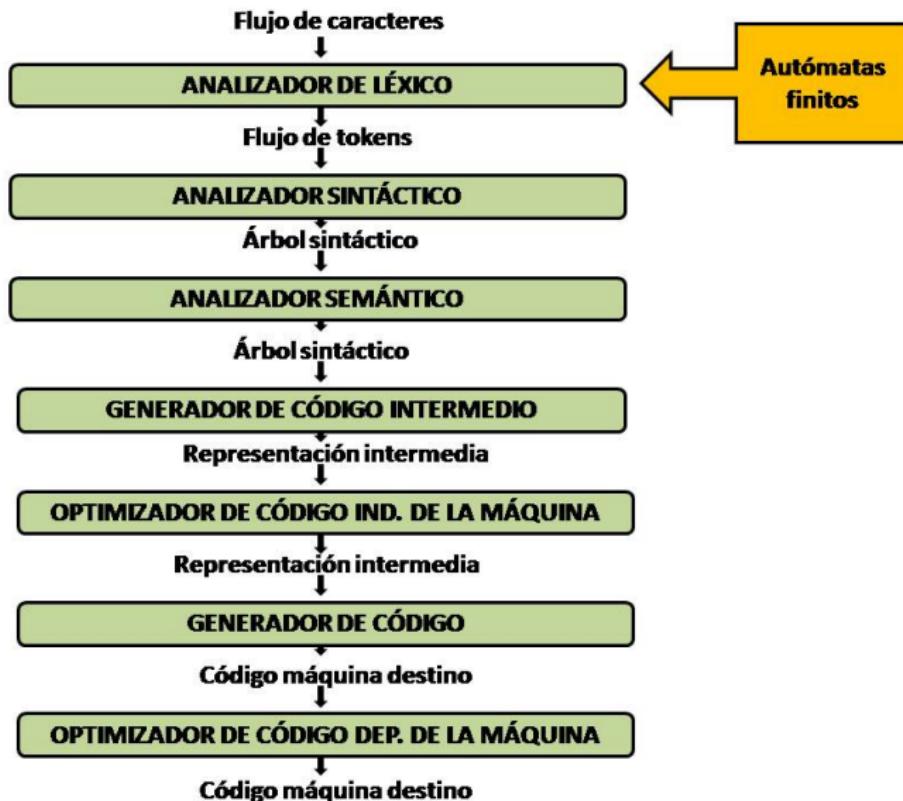
Jerarquía de Chomsky

Tipo de Gramática	Lenguaje que genera	Máquina Abstracta aceptadora	Aplicación
0 $\alpha \rightarrow \beta$	r.e. o de estructura de frase	Máquina de Turing	Computabilidad
1 $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$	Sensible al contexto	Autómata linealmente acotado	Tipo de los lenguajes de programación
2 $A \rightarrow \alpha$	Libres de contexto	Autómata de pila	Análisis sintáctico
3 $A \rightarrow aB$ $A \rightarrow a$ $A \rightarrow \lambda$	Regular	Autómata finito	Análisis léxico

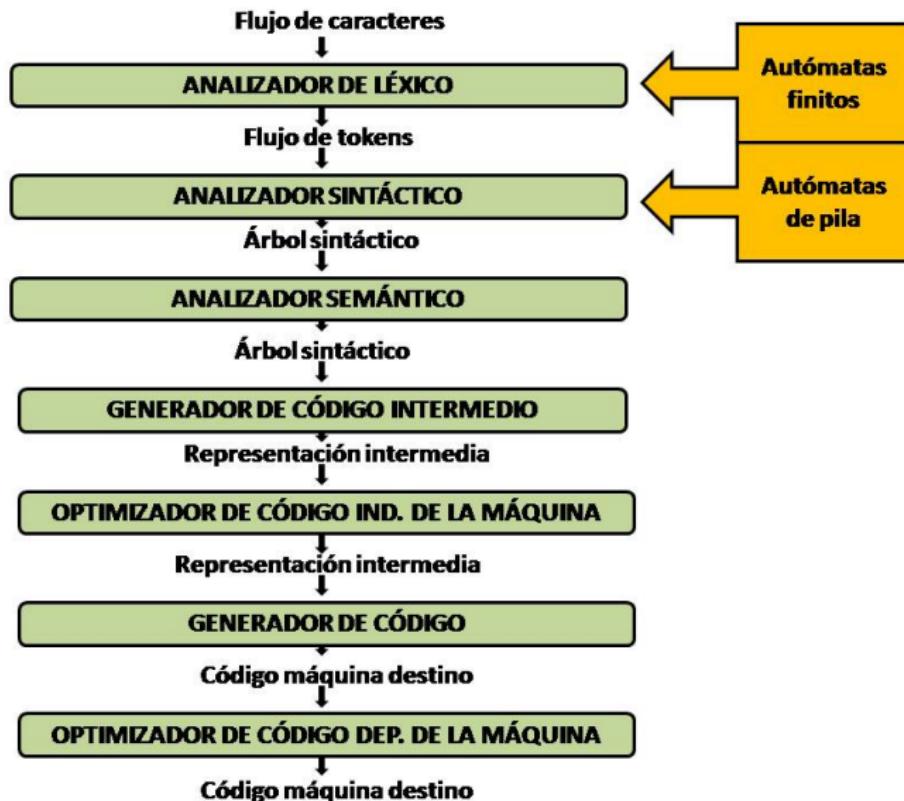
Base gramatical de los compiladores



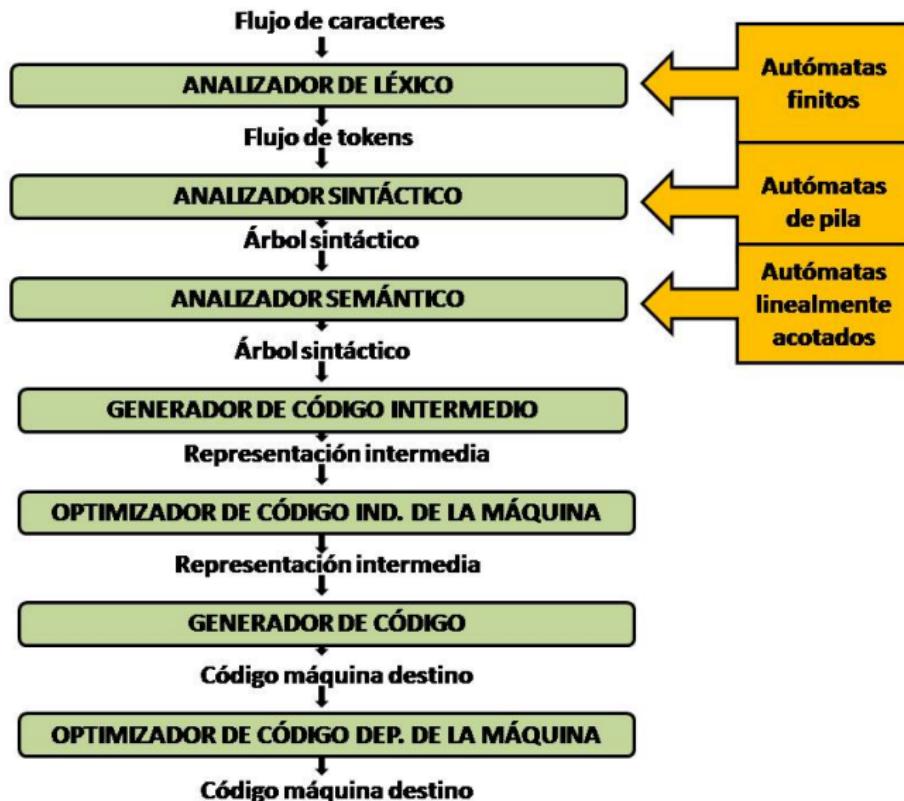
Base gramatical de los compiladores



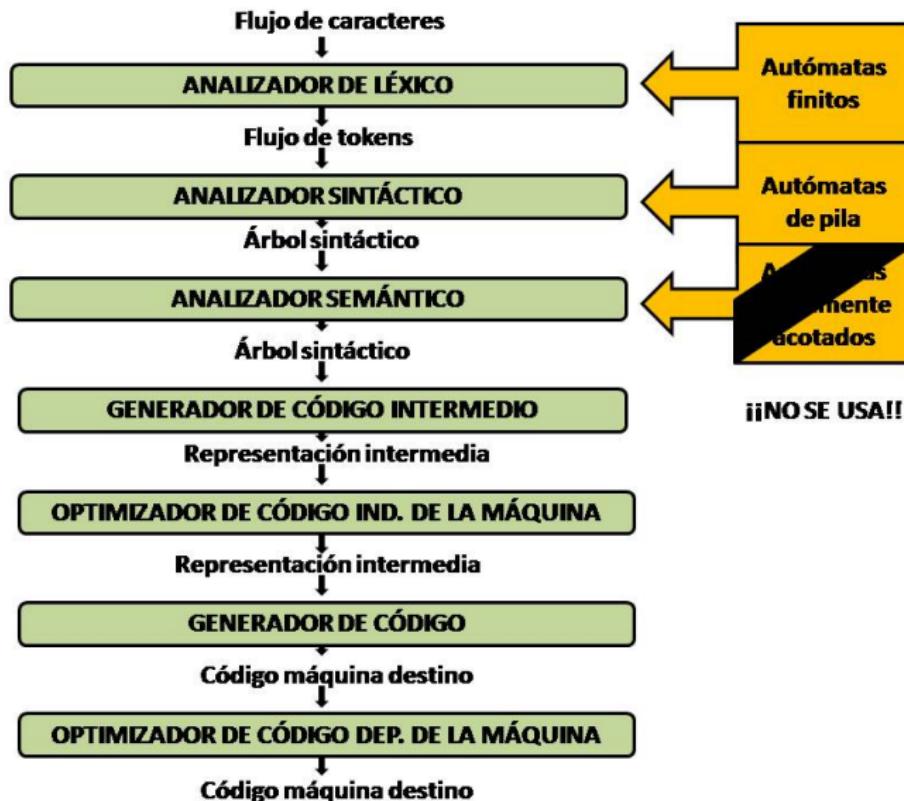
Base gramatical de los compiladores



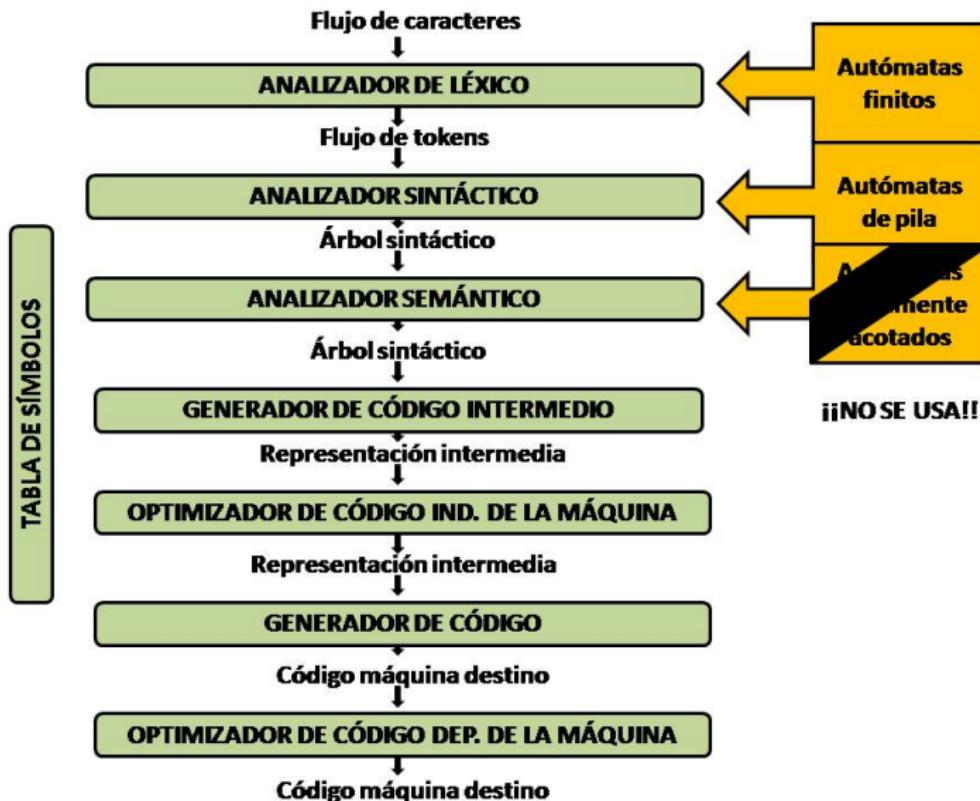
Base gramatical de los compiladores



Base gramatical de los compiladores



Base gramatical de los compiladores



Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (cross-compiler)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.
- **Compilador de una/varias pasada/s** El de **una pasada** genera código máquina a partir de una única lectura del código fuente. El de **múltiples pasadas** procesa varias veces el código fuente o alguna de las estructuras generadas en el proceso.
- **Compilador just-in-time** Traduce, de forma dinámica, *bytecode* a código máquina según se necesita. Suele formar parte de un intérprete.
- **Compilador optimizador** Modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).

Ejemplo de compilación cruzada:



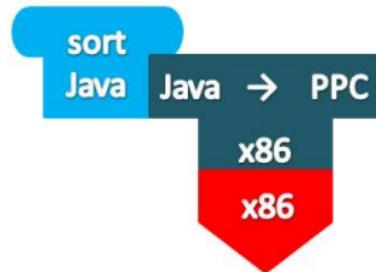
sort
Java

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).

Ejemplo de compilación cruzada:

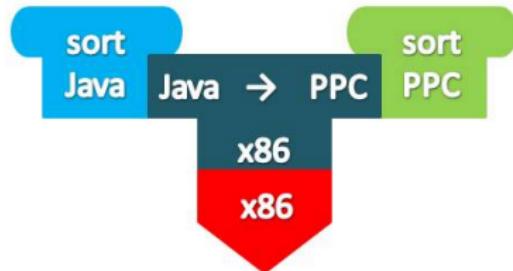


Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).

Ejemplo de compilación cruzada:

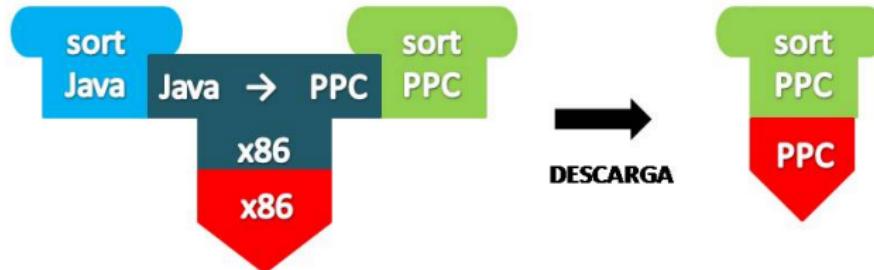


Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (cross-compiler)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).

Ejemplo de compilación cruzada:



Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (cross-compiler)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.
- **Compilador de una/varias pasada/s** El de **una pasada** genera código máquina a partir de una única lectura del código fuente. El de **múltiples pasadas** procesa varias veces el código fuente o alguna de las estructuras generadas en el proceso.
- **Compilador just-in-time** Traduce, de forma dinámica, *bytecode* a código máquina según se necesita. Suele formar parte de un intérprete.
- **Compilador optimizador** Modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.

Ejemplo de compilador de dos etapas de Java a x86:



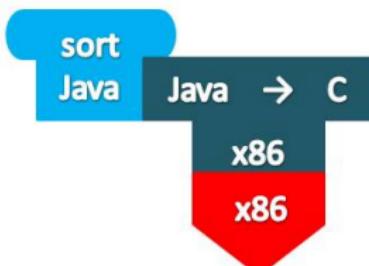
sort
Java

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.

Ejemplo de compilador de dos etapas de Java a x86:

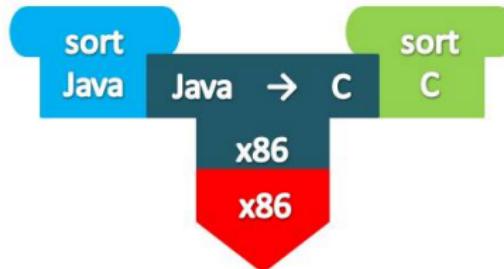


Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.

Ejemplo de compilador de dos etapas de Java a x86:

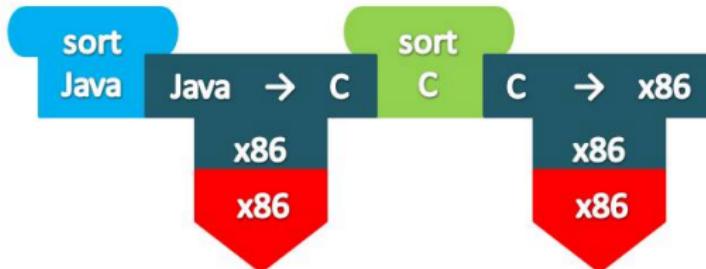


Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.

Ejemplo de compilador de dos etapas de Java a x86:

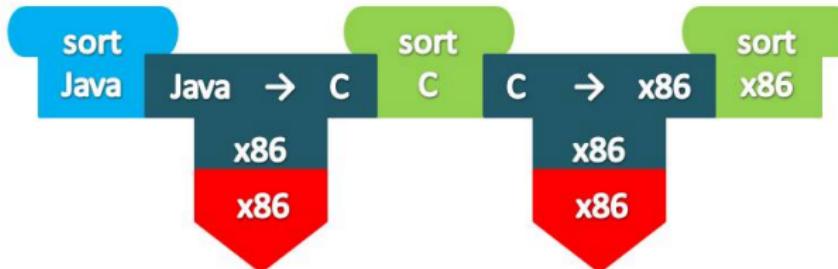


Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.

Ejemplo de compilador de dos etapas de Java a x86:



Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (cross-compiler)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.
- **Compilador de una/varias pasada/s** El de **una pasada** genera código máquina a partir de una única lectura del código fuente. El de **múltiples pasadas** procesa varias veces el código fuente o alguna de las estructuras generadas en el proceso.
- **Compilador just-in-time** Traduce, de forma dinámica, *bytecode* a código máquina según se necesita. Suele formar parte de un intérprete.
- **Compilador optimizador** Modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (cross-compiler)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.
- **Compilador de una/varias pasada/s** El de **una pasada** genera código máquina a partir de una única lectura del código fuente. El de **múltiples pasadas** procesa varias veces el código fuente o alguna de las estructuras generadas en el proceso.
- **Compilador just-in-time** Traduce, de forma dinámica, *bytecode* a código máquina según se necesita. Suele formar parte de un intérprete.
- **Compilador optimizador** Modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

Distintos tipos de compiladores

Dependiendo de algunos aspectos del proceso de traducción:

- **Compilador cruzado (*cross-compiler*)** Se ejecuta sobre una máquina (**máquina huesped**) pero genera código para una máquina distinta (**máquina destino**).
- **Compilador de varias etapas** Composición de n traductores, envolviendo n-1 lenguajes intermedios.
- **Compilador de una/varias pasada/s** El de **una pasada** genera código máquina a partir de una única lectura del código fuente. El de **múltiples pasadas** procesa varias veces el código fuente o alguna de las estructuras generadas en el proceso.
- **Compilador *just-in-time*** Traduce, de forma dinámica, *bytecode* a código máquina según se necesita. Suele formar parte de un intérprete.
- **Compilador optimizador** Modifica el código intermedio o el objeto para mejorar su eficiencia, manteniendo la funcionalidad del programa original.

Intérpretes

Un **intérprete** es un programa que acepta un programa fuente escrito en un determinado **lenguaje fuente**, y ejecuta el programa inmediatamente, **sin producir código objeto**.



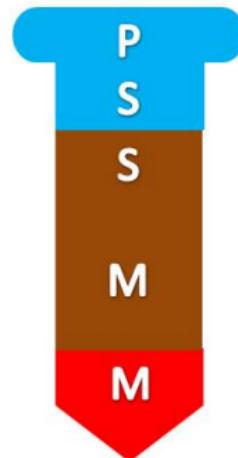
Intérpretes

Un **intérprete** es un programa que acepta un programa fuente escrito en un determinado **lenguaje fuente**, y ejecuta el programa inmediatamente, **sin producir código objeto**.



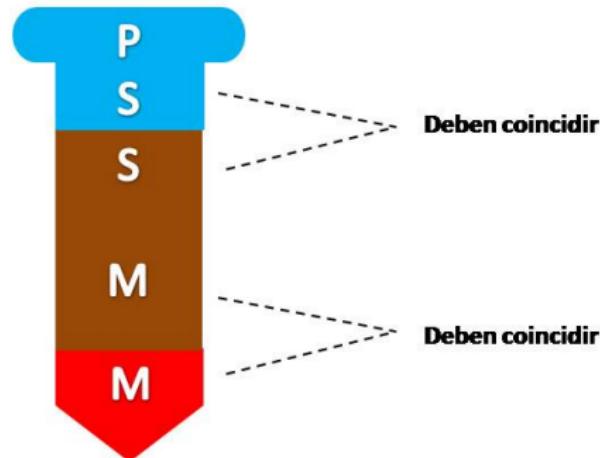
Intérpretes

Un **intérprete** es un programa que acepta un programa fuente escrito en un determinado **lenguaje fuente**, y ejecuta el programa inmediatamente, **sin producir código objeto**.



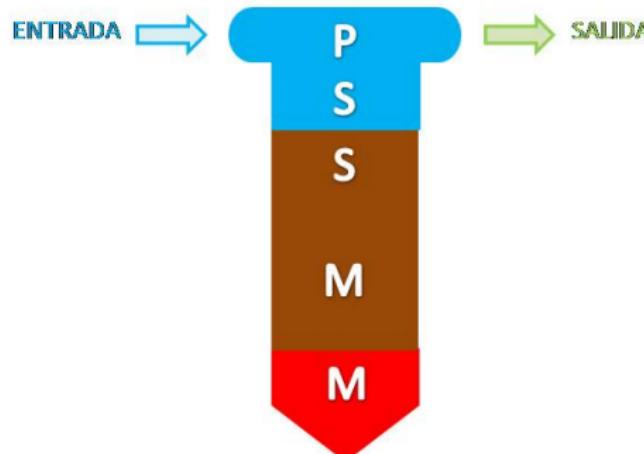
Intérpretes

Un **intérprete** es un programa que acepta un programa fuente escrito en un determinado **lenguaje fuente**, y ejecuta el programa inmediatamente, **sin producir código objeto**.



Intérpretes

Un **intérprete** es un programa que acepta un programa fuente escrito en un determinado **lenguaje fuente**, y ejecuta el programa inmediatamente, **sin producir código objeto**.



Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

El programa se va a ejecutar en modo de producción.

El código fuente es difícil de leer y mantener.

El código fuente es difícil de optimizar.

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

• El programa se va a ejecutar en modo de producción.

• Se espera que las instrucciones se ejecuten frecuentemente.

• El lenguaje de programación es difícil de aprender.

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

- El programa se va a ejecutar en modo de producción.
- Se espera que las instrucciones se ejecuten frecuentemente.
- Las instrucciones tienen formatos complicados.

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

- El programa se va a ejecutar en modo de producción.
- Se espera que las instrucciones se ejecuten frecuentemente.
- Las instrucciones tienen formatos complicados.

Intérpretes

Cuándo es adecuada la interpretación:

- Programador trabajando en forma interactiva.
- El programa se va a utilizar sólo una vez.
- Se espera que cada instrucción se ejecute una sola vez.
- Instrucciones con formato simple.

La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina.

Cuándo la interpretación no es interesante:

- El programa se va a ejecutar en modo de producción.
- Se espera que las instrucciones se ejecuten frecuentemente.
- Las instrucciones tienen formatos complicados.

Intérpretes

Algunos intérpretes conocidos:

- ① Intérprete Caml
- ② Intérprete Lisp
- ③ Intérprete de comandos de Unix (shell)
- ④ Intérprete SQL
- ⑤ Intérprete de código intermedio

Máquinas reales y abstractas

A veces interesa escribir **intérpretes para lenguajes de bajo nivel**.

Ejemplo:

Problema: Testar el diseño de la arquitectura e instrucciones de una máquina (Ultima) para modificarlo antes de plasmarlo en hardware.

Solución: Escribir un intérprete para el código máquina de Ultima, por ejemplo, en C y traducirlo en algún código máquina, por ejemplo M.



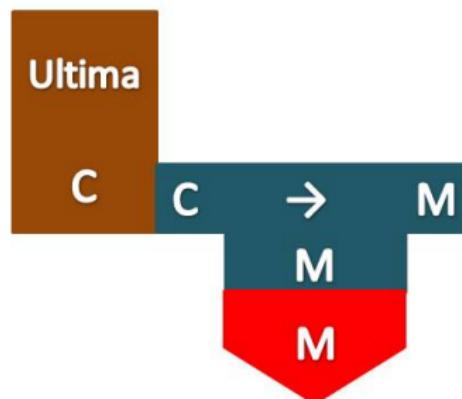
Máquinas reales y abstractas

A veces interesa escribir **intérpretes para lenguajes de bajo nivel**.

Ejemplo:

Problema: Testar el diseño de la arquitectura e instrucciones de una máquina (Ultima) para modificarlo antes de plasmarlo en hardware.

Solución: Escribir un intérprete para el código máquina de Ultima, por ejemplo, en C y traducirlo en algún código máquina, por ejemplo M.



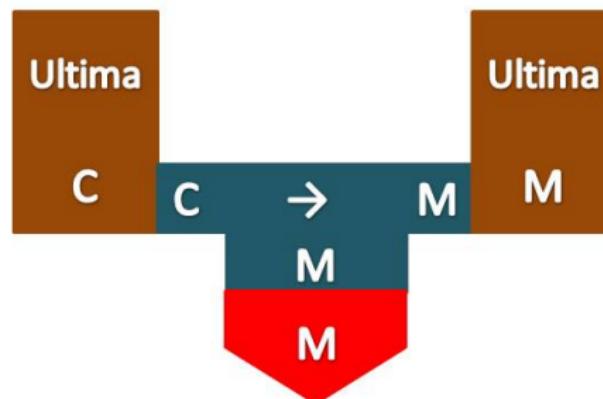
Máquinas reales y abstractas

A veces interesa escribir **intérpretes para lenguajes de bajo nivel**.

Ejemplo:

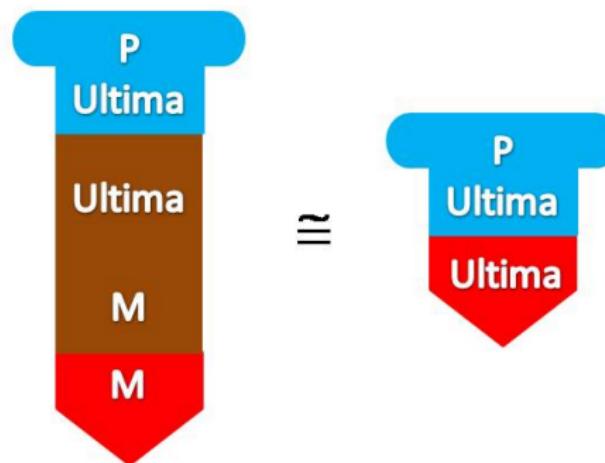
Problema: Testar el diseño de la arquitectura e instrucciones de una máquina (Ultima) para modificarlo antes de plasmarlo en hardware.

Solución: Escribir un intérprete para el código máquina de Ultima, por ejemplo, en C y traducirlo en algún código máquina, por ejemplo M.



Máquinas reales y abstractas

En todos los aspectos, salvo en la velocidad, el efecto de correr un programa escrito en código máquina de Ultima sobre este intérprete es el mismo que si el programa se ejecutara sobre Ultima:

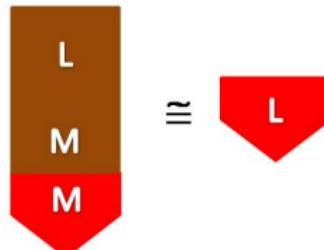


Máquinas reales y abstractas

Este tipo de intérprete se llama **emulador**, y sólo difiere de la máquina en cuanto a la velocidad.

Una **máquina real** puede ser vista como un **intérprete implementado en hardware**. Un **intérprete** puede verse como una **máquina implementada en software** (**máquina abstracta**).

Máquina real y **máquina abstracta** son funcionalmente equivalentes si ambas implementan el mismo lenguaje:



No existe una diferencia fundamental entre **código máquina** y **lenguaje de bajo nivel**: un **código máquina** es un lenguaje para el cual existe un intérprete hardware.

Compiladores interpretados

Un **compilador interpretado** es la combinación de *compilador* e *intérprete*, reuniendo algunas de las ventajas de cada uno de ellos. Se traduce el programa fuente en un **lenguaje intermedio**, que cumple:

- Nivel intermedio entre el lenguaje fuente y el código máquina.
- Instrucciones con formato simple.
- Traducción desde el lenguaje fuente al lenguaje intermedio fácil y rápida.

Combina la rapidez de la compilación con una velocidad tolerable en la ejecución.

Compiladores interpretados

Ejemplo: Código de la Máquina Virtual de Java (JVM-code)

- Se trata de un lenguaje intermedio orientado a Java.
- Java Development Kit (**JDK**) consiste en un traductor de Java a JVM-code y un intérprete de JVM-code, los cuales se ejecutan sobre alguna máquina M.
- Dado un programa P escrito en Java, primero se traduce a JVM-code, y a continuación el programa objeto JVM-code es interpretado.

Compiladores interpretados

Ejemplo: Código de la Máquina Virtual de Java (JVM-code)

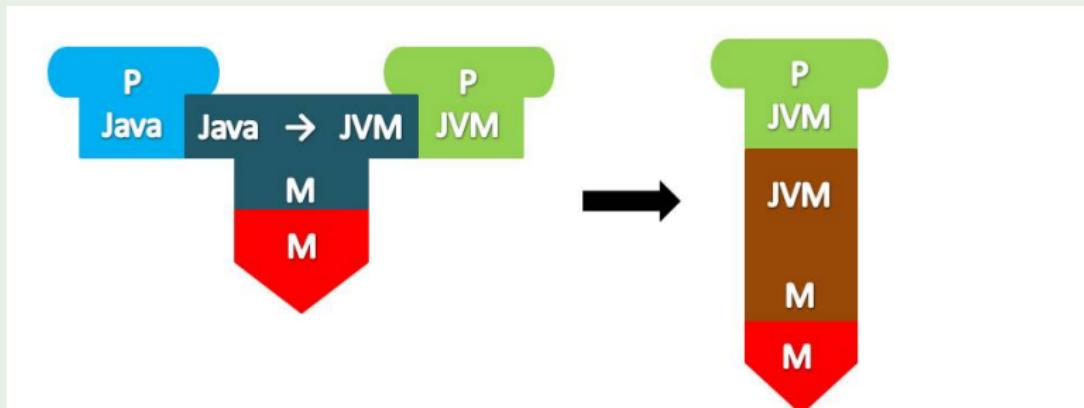
- Se trata de un lenguaje intermedio orientado a Java.
- Java Development Kit (**JDK**) consiste en un traductor de Java a JVM-code y un intérprete de JVM-code, los cuales se ejecutan sobre alguna máquina M.



Compiladores interpretados

Ejemplo: Código de la Máquina Virtual de Java (JVM-code)

- Se trata de un lenguaje intermedio orientado a Java.
- Java Development Kit (**JDK**) consiste en un traductor de Java a JVM-code y un intérprete de JVM-code, los cuales se ejecutan sobre alguna máquina M.
- Dado un programa P escrito en Java, primero se traduce a JVM-code, y a continuación el programa objeto JVM-code es interpretado.



Compiladores portables

Programa portable Puede ser (compilado y) ejecutado en cualquier máquina, sin cambios.

Medición de portabilidad Proporción de código que no se cambia cuando el programa se mueve entre máquinas diferentes.

- Influye el lenguaje de escritura: ensamblador (0 %) vs. lenguaje de alto nivel (95 %-100 %).
- Característica añadida en procesadores de lenguajes: generan código máquina y esto hace que la dependencia sea inevitable (50 %).
- Un compilador que genera código intermedio es potencialmente más portable que otro que genera código máquina.

Compiladores portables

Programa portable Puede ser (compilado y) ejecutado en cualquier máquina, sin cambios.

Medición de portabilidad Proporción de código que no se cambia cuando el programa se mueve entre máquinas diferentes.

- Influye el lenguaje de escritura: ensamblador (0 %) vs. lenguaje de alto nivel (95 %-100 %).
- Característica añadida en procesadores de lenguajes: generan código máquina y esto hace que la dependencia sea inevitable (50 %).
- Un compilador que genera código intermedio es potencialmente más portable que otro que genera código máquina.

Compiladores portables

Programa portable Puede ser (compilado y) ejecutado en cualquier máquina, sin cambios.

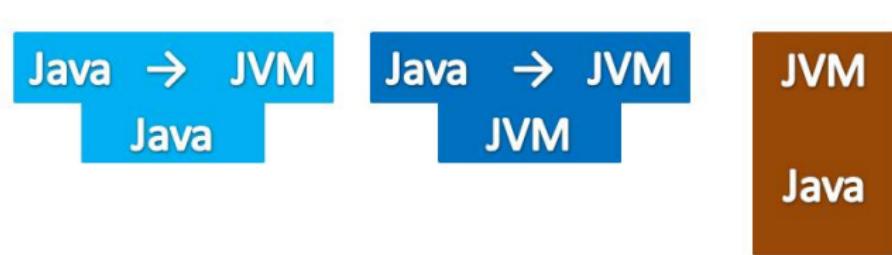
Medición de portabilidad Proporción de código que no se cambia cuando el programa se mueve entre máquinas diferentes.

- Influye el lenguaje de escritura: ensamblador (0 %) vs. lenguaje de alto nivel (95 %-100 %).
- Característica añadida en procesadores de lenguajes: generan código máquina y esto hace que la dependencia sea inevitable (50 %).
- Un compilador que genera código intermedio es potencialmente más portable que otro que genera código máquina.

Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

→ Constaría de dos traductores y un intérprete.



Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

- Constaría de dos traductores y un intérprete.
- **¿Cómo funcionaría?**

- ➊ Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.
- ➋ Se consigue un compilador interpretado, pero escrito en JVM-code, por tanto tiene que ejecutarse sobre un intérprete de JVM-code.

Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

→ Constaría de dos traductores y un intérprete.

→ **¿Cómo funcionaría?**

- ➊ Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.



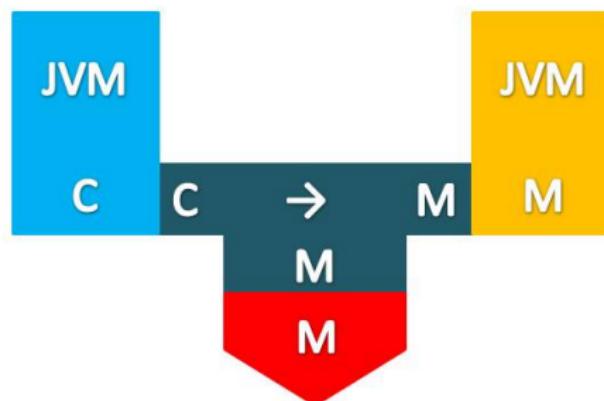
Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

→ Constaría de dos traductores y un intérprete.

→ **¿Cómo funcionaría?**

- ➊ Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.



Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

→ Constaría de dos traductores y un intérprete.

→ **¿Cómo funcionaría?**

- ➊ Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.
- ➋ Se consigue un compilador interpretado, pero escrito en JVM-code, por tanto tiene que ejecutarse sobre un intérprete de JVM-code.



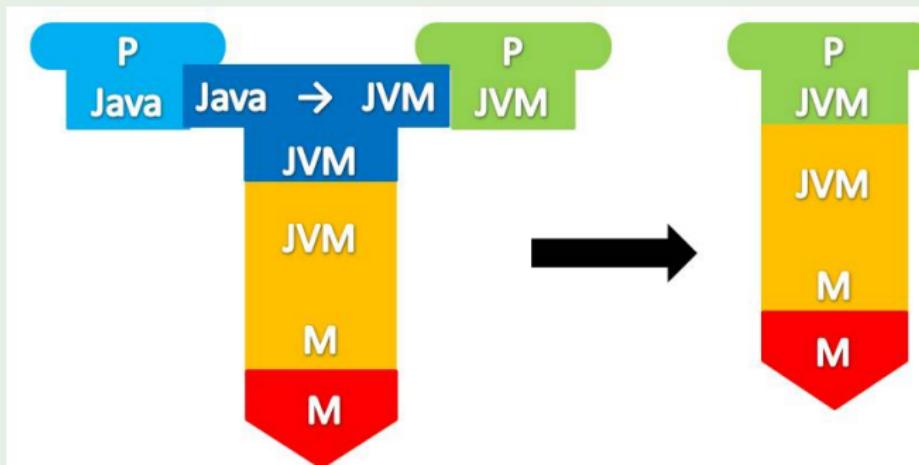
Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

→ Constaría de dos traductores y un intérprete.

→ ¿Cómo funcionaría?

- ➊ Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.
- ➋ Se consigue un compilador interpretado, pero escrito en JVM-code, por tanto tiene que ejecutarse sobre un intérprete de JVM-code.



Compiladores portables

Ejemplo: 'Kit' de un compilador Java portable

- Constaría de dos traductores y un intérprete.
- **¿Cómo funcionaría?**
 - ① Si tenemos un sistema que se ejecuta sobre M y un compilador (p. e. para C), escribimos el intérprete de JVM en C y lo compilamos.
 - ② Se consigue un compilador interpretado, pero escrito en JVM-code, por tanto tiene que ejecutarse sobre un intérprete de JVM-code.

El intérprete de JVM-code es mucho menor y más sencillo que el compilador (95 % portable).