FILMINAS COMENTADAS
SINTÁXIS Y SEMÁNTICA DE LENGUAJES

INTRODUCCIÓN

El presente documento, toma como hilo conductor las filminas de clase e incluye notas y ejemplos que pretenden explicar sintéticamente los principales contenidos de la asignatura. Es una guía, que debe ser ampliada luego con la lectura bibliográfica, el material digital complementario propuesto, la resolución de trabajos prácticos y la indagación personal y autónoma por parte de los alumnos.

UNIDAD II

Leguajes y proceso de traducción

Introducción

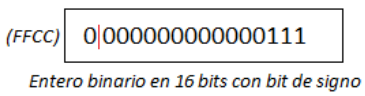
Mientras que en la primera unidad nos ocupamos de aprender conceptos vinculados al léxico y la sintaxis de los lenguajes de programación; en este segundo bloque profundizaremos en temas vinculados a su semántica. En este sentido, nos ocuparemos de abordar en detalle los aspectos que hacen al ***¿Cómo?*** se implementan los lenguajes de programación.

Sintéticamente, cuando hablamos de **semántica**, nos referimos a la información que determina cómo debe resolverse o traducirse al bajo nivel cada uno de los enunciados que pueden escribirse en el lenguaje de alto nivel.

A modo de ejemplo ilustrativo, supongamos que un programa en C/C++ incluye la sentencia

int x = 7;

Claramente esta sentencia es léxica y sintácticamente correcta. Luego, su semántica, será aquella información establecida en el diseño del lenguaje que permita a un compilador saber cómo se debe traducir esta instrucción, para obtener un conjunto de sentencias de bajo nivel que ejecuten: la reserva de una celda de memoria, la vinculen a una dirección y almacenen en ella el valor entero 7. Además, la información semántica deberá indicar también el formato de representación en binario de este valor entero y todos los detalles que refieran a cómo se debe manipular esta variable: qué tipo de operaciones pueden hacerse con este valor, su ámbito, su alcance, si el valor es constante o variable, cómo se gestiona la memoria que la aloja, etc.

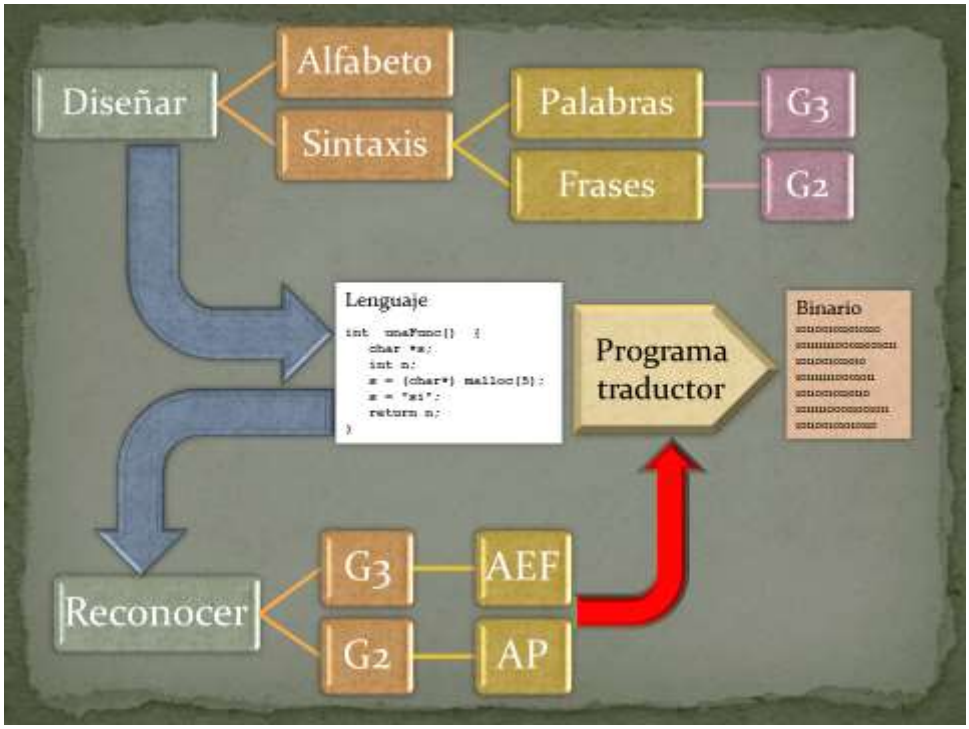


Entero binario en 16 bits con bit de signo

De este modo, abordaremos contenidos básicos de la teoría de compiladores e intérpretes, contenidos vinculados a la semántica de datos (es decir, cómo se resuelve el almacenamiento y la recuperación de datos ya sean simples o estructurados), la semántica del control de flujo, las estructuras de información, la implementación de subprogramas, la gestión de la memoria, etc.

Compiladores e Intérpretes

Para avanzar en este sentido, volvemos sobre el **problema de la traducción**, que hemos tomado como hilo o eje conductor de nuestros desarrollos teóricos.



Como se ha visto previamente, **diseñar** un lenguaje de programación implica definir su sintaxis y su semántica. Respecto de la sintaxis es necesario definir su alfabeto, su léxico (conjunto de palabras válidas) y las reglas de buena formación de sentencias o frases en el lenguaje.

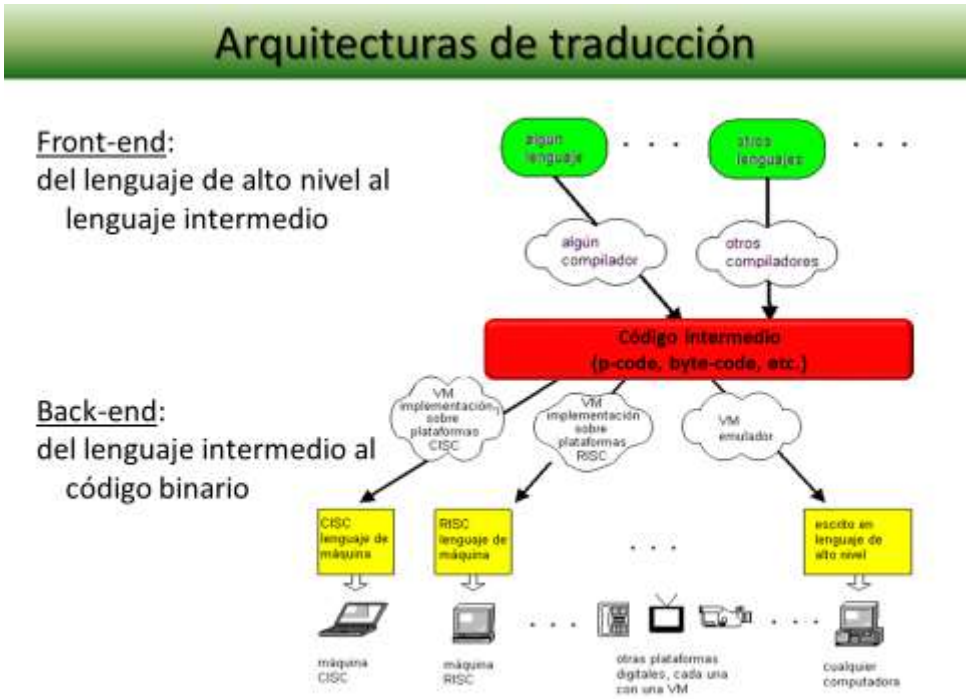
El léxico de los lenguajes de programación puede ser definido por Gramáticas Regulares (G3 o ER) y las reglas de buena formación de sentencias o frases pueden ser definidas mediante Gramáticas Independientes del Contexto (G2, BNF o EBNF).

Luego, para ejecutar un programa (escrito en un lenguaje de alto nivel) en un ordenador, se necesita previamente traducirlo a código binario. Esta tarea la realiza un **Traductor** que toma el código fuente como entrada y produce código binario en su salida.

La primera tarea que realiza el traductor, es analizar el código fuente para **reconocer** si las palabras y frases están correctamente escritas en el lenguaje de partida. Para ello

se emplea un conjunto de autómatas: de estados finitos para el léxico (AEF) y de pila para las sentencias, bloques y estructuras (AP). Estos autómatas son en definitiva algoritmos que integran el traductor en sus fases de reconocimiento y análisis de código.

A continuación se abordarán una serie de temas e ideas que permitirán ampliar el conocimiento de las etapas que involucra este proceso de traducción, y las diferentes formas en que puede implementarse un programa traductor, o más precisamente, las diferentes **arquitecturas de traducción**.



Si pensamos en el proceso de traducción, tal como lo hemos descripto hasta ahora, -traducir un programa de alto nivel a código binario que la computadora puede comprender- es claro que el programa ejecutable que obtendremos será altamente dependiente del hardware.

Bajo este modelo de traducción, debe existir un traductor para cada lenguaje y cada arquitectura de hardware (y su software de base). Así, para un mismo programa escrito por ejemplo en C/C++, necesitamos un traductor para arquitecturas CISC con Windows, otro para arquitecturas CISC con Linux y así para cada plataforma posible.

Pensemos por ejemplo, que sucedería si cambiamos de un procesador de 32bits a uno de 64bits? A bajo nivel, no es lo mismo representar los números enteros disponiendo de 32bits que de 64bits. Y las operaciones tampoco son las mismas. Un ejemplo cotidiano de esto son los programas que corren en Windows de 64bits, pero no podemos ejecutarlos en Windows de 32bits.

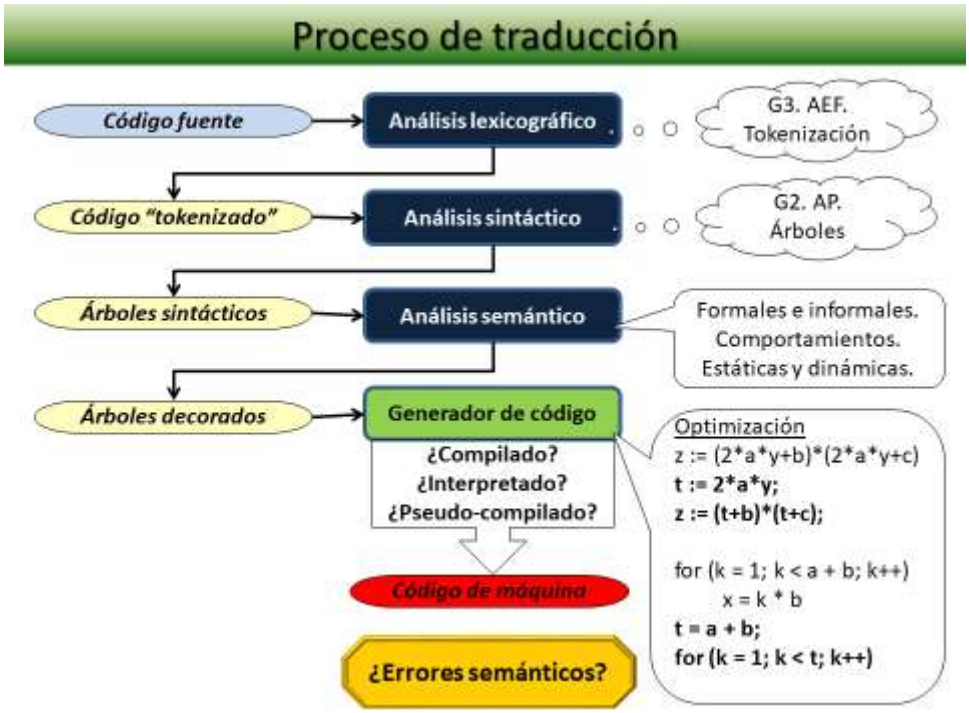
Si bien muchos traductores trabajan de este modo, las arquitecturas modernas introdujeron la idea de que la traducción no debe pensarse exclusivamente desde el lenguaje de alto nivel al código binario en un único proceso. Es factible, pensar en traducir nuestros programas a un código intermedio, que no es cercano al lenguaje natural, pero tampoco depende estrictamente de una arquitectura de hardware específica.

Así, podemos pensar en una primera capa de traducción (**front-end**) que convierta nuestros programas fuentes a un **Código intermedio**, más cercano al código de máquina pero sin resolver aún aquellos aspectos estrictamente vinculados al hardware.

Este código intermedio, puede luego ser interpretado o comprendido por una máquina virtual **VM – Virtual Machine**; típicamente un programa (aunque también pueden ser implementadas por hardware) encargado de traducir este código intermedio a lenguaje binario para una arquitectura de hardware particular (**back-end**).

Bajo este enfoque, el esfuerzo de traducción en el front-end, se hace una única vez para cada programa fuente, independientemente de la arquitectura de hardware donde vaya a ejecutarse el programa luego. La traducción en el back-end será responsabilidad de la VM.

Independientemente de la arquitectura de traducción que se emplee, el proceso de traducción comprende ciertas fases que deben llevarse a cabo.



Para el traductor, el programa fuente se presenta inicialmente como una serie larga y no diferenciada de símbolos (una serie de caracteres). Desde luego, un programador que ve un programa así lo estructura rápidamente en enunciados y bloques de enunciados: declaraciones, asignaciones, estructuras, funciones, procedimientos, etc. Para el compilador, nada de esto es manifiesto. Durante la traducción, se debe construir laboriosamente, carácter por carácter, un análisis de la estructura del programa.

Repasando un poco, las etapas de reconocimiento de código fuente son:

1. Análisis Léxico (Scanner)

Esta fase inicial de la traducción, se ocupa de agrupar esta serie secuencial de caracteres en sus constituyentes elementales: identificadores, espacios en blanco, palabras clave o reservadas, delimitadores, comentarios, etc.

Estos constituyentes elementales se llaman elementos o componentes léxicos ***(*Lexemas*)*** y a su categorización en una categoría de elementos específicos se les llama **Tokens**.

Token

Un token es la unidad de programa más pequeña a la que puede asociarse un significado.

Dado que todo componente léxico en un lenguaje, puede ser descripto por una ***G3***, o lo que es equivalente, dado que los tokens se especifican usando expresiones regulares, la implementación de un revisor o analizador léxico (**Scanner**) dentro de un compilador se hace a través de los Autómatas de Estado Finito (AEF) que vimos anteriormente.

Error de Tipo Léxico

Observar la expresión

Ese ombre cuenta.

(Aquí estamos frente a un problema léxico y no sintáctico)

Igual sucede ante la declaración.

int ¿12dc = 3;

2. Análisis Sintáctico (Parser)

En esta segunda etapa de análisis, se identifican las **estructuras sintácticas del programa**: enunciados, declaraciones, expresiones, etc.; y se verifica que sean válidas.

Dado que las estructuras sintácticas en un lenguaje pueden ser descriptas por ***G2*** (Gramáticas Independientes del Contexto), la implementación de un analizador sintáctico (o **Parser**) dentro de un compilador se hace a través de los Autómatas de Pila (AP) o analizadores recursivos como los anteriormente estudiados.

Error de Tipo Sintáctico

int C = 3;	*(declaración e inicialización en C sintácticamente correcta)*
X = Y + Z;	*(expresión sintácticamente correcta en javascript)*
X-Y= Z;	*(expresión sintácticamente incorrecta en C).*

La sintaxis de un lenguaje suministra información imprescindible para la traducción del programa fuente al programa objeto. El producto del análisis sintáctico de un enunciado, es el árbol de análisis sintáctico de ese enunciado. Así el resultado de la etapa de análisis sintáctico son los árboles de sintaxis concreta, correspondientes a ese programa.

3. Análisis Semántico

Volviendo sobre el ejemplo

Ese hombre cuenta

vemos que su significado es ambiguo.

Este ejemplo, nos permite considerar que se necesita algo más que sólo un conjunto de estructuras sintácticas para describir completamente un lenguaje.

El traductor, necesita poder interpretar ***el significado*** de los enunciados, declaraciones, expresiones, etc. Este significado, lo aporta la etapa de Análisis Semántico.

Ejemplo

dada la expresión

*4 + b * c*

una regla semántica es aquella que establece la precedencia de operadores, la cual en este caso, permite determinar que el producto se resuelve antes que la suma. Esto tiene que ver con el significado de la expresión (no es información que esté en su sintaxis), y determina su traducción posterior en una serie de macroinstrucciones más simples a bajo nivel, que resuelven los cálculos en orden.

Otras reglas semánticas son aquellas que verifican la correcta declaración y utilización de variables, las reglas de alcance y de ámbito o scope que determinan qué símbolos son accesibles desde un punto en el programa, las reglas de control de flujo de ejecución, las reglas de gestión de memoria, la implementación de subprogramas, etc.

Obviamente, no todas las reglas semánticas pueden ser chequeadas en tiempo de compilación. Aquellas que si pueden ser verificadas en tiempo de compilación son referidas como **Semántica Estática** del lenguaje. Aquellas reglas que deben ser chequeadas en tiempo de ejecución del programa son referidas como **Semántica Dinámica** del lenguaje.

El proceso de análisis sintáctico alterna con el análisis semántico. Primero, el analizador sintáctico identifica una serie de elementos léxicos que forman una unidad sintáctica, como una expresión, un enunciado, llamada de subprograma o declaración. Se llama entonces a un analizador semántico que procese esta unidad, para ir generando entonces la traducción de código, desde el código fuente al objeto.

El analizador semántico, es el puente entre las partes de **Análisis** (Reconocimiento de Código Fuente) y **Síntesis** (Generación de Código Objeto) de la traducción.

En la etapa de Análisis Semántico, ocurren también otras tareas complementarias importantes como el mantenimiento de la tabla de símbolos, la mayor parte de la detección de errores, la expansión de macros, etc. que veremos luego en detalle.

Por lo común, el analizador sintáctico y el semántico se comunican usando una pila. El analizador sintáctico introduce en la pila los diversos elementos de la unidad sintáctica hallada, y el analizador semántico los recupera y los procesa.

Las etapas finales de la traducción se ocupan de la construcción del programa objeto a partir de las salidas que produce el analizador semántico.

4. Generación de Código

El código objeto es ya una versión del programa en código de máquina, pero aún no es ejecutable porque falta su enlace con librerías dinámicas u otros bloques de código. Recién cuando todo el programa se enlaza y se carga en memoria estará disponible para su ejecución.

Optimización

La optimización, tal como se muestra en el ejemplo de la filmina, mejora el código de forma tal que no se traduzcan instrucciones repetidas, se sustituyan constantes literales por sus valores, etc.

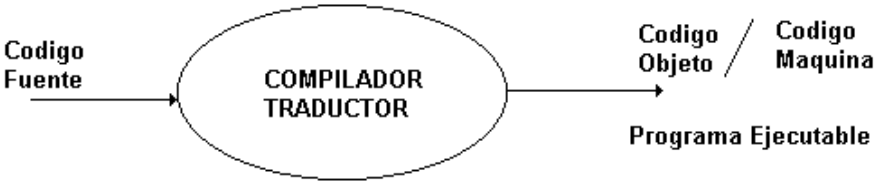
Muchas veces la calidad de un traductor se determina por la calidad de su proceso optimizador de código. Como veremos luego, algunos traductores no incorporan este proceso.

Las filminas que siguen ejemplifican los distintos tipos de traductores que podemos encontrar. A saber:

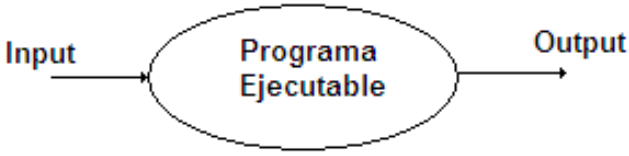
- ✓ Compiladores
- ✓ Intérpretes
- ✓ Pseudocompiladores o Intérpretes Parciales

Compilador Nativo

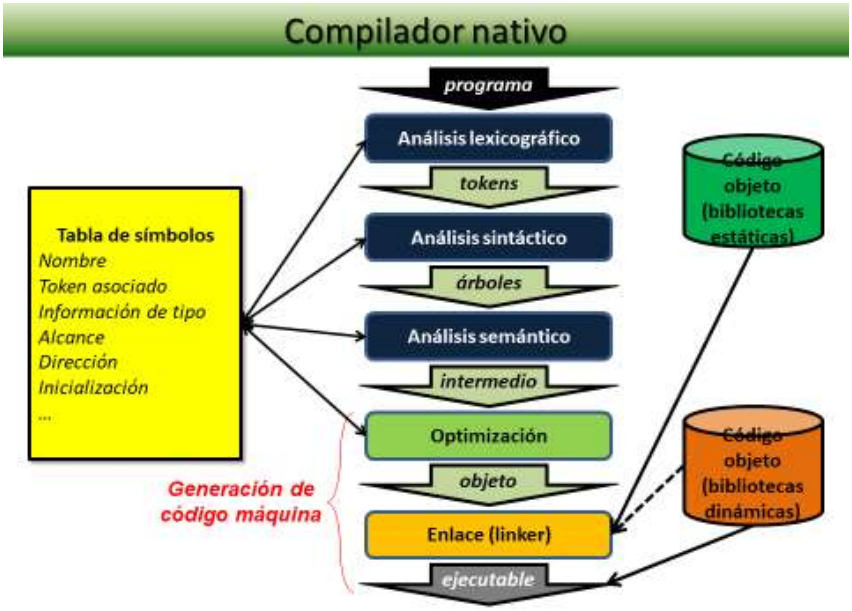
Un compilador nativo es un traductor que realiza todas las fases de traducción en un único proceso, tomando como entrada el código fuente de un programa y produciendo como salida un programa ejecutable en código binario.



El tiempo que se necesita para traducir un lenguaje de alto nivel a lenguaje objeto se denomina ***tiempo de compilación.***



El tiempo que tarda en ejecutarse un programa objeto se denomina ***tiempo de ejecución***.



La información proporcionada en las declaraciones del programa, son incorporadas durante el proceso de traducción a la **tabla de símbolos*.***

Ejemplo

Para el código en C/C++

```
int main(char *argv[]){
        int X1;
        …
}
void una() {
        int Z;
        …
}
```

Tabla de símbolos

Identificador	***Tipo***	***Dir. Memoria***	***Scope***	***variable***
X1	*int*	*FFA1*	*main*	*true*
Z	*int*	*FF00*	*una*	*True*
main	*int*	*CC88*	*global*	*código estático*
Una	*Void*	*CC03*	*Global*	*código estático*

Luego el analizador semántico se vale de esta información para producir un código de máquina que haga una manipulación correcta de los símbolos declarados.

Usando la tabla de símbolos, el analizador semántico verifica el cumplimiento de una variedad de reglas de buena formación de los programas que no pueden ser expresadas en las estructuras sintácticas definidas por gramáticas independientes del contexto y los árboles de sintaxis, tales como:

- Que cada identificador sea declarado antes de ser usado.
- Que los identificadores no sean utilizados en ámbitos inadecuados.
- La verificación de tipos en las expresiones y pasaje de argumentos a funciones
- Y otros que estudiaremos más adelante con detalle.

Enlace

En la actualidad, los programadores no escriben de principio a fin todo el código fuente que constituye luego el programa ejecutable. En general, se utilizan librerías de código, previamente escrito y compilado que resultan convenientes para una necesidad específica. Pensemos por ejemplo en los usos que hacemos en nuestros programas en C, respecto de librerías que manejan strings o cadenas, aquellas especializadas en el manejo de fechas, librerías para el manejo de la IU (interface de usuario) como botones, menues, etc.

Estas librerías, deben en algún momento vincularse (enlazarse) a nuestro código objeto para componer el programa ejecutable final. Dependiendo del **momento** en que estas librerías o bibliotecas se vinculan al programa, diremos que son estáticas o dinámicas.

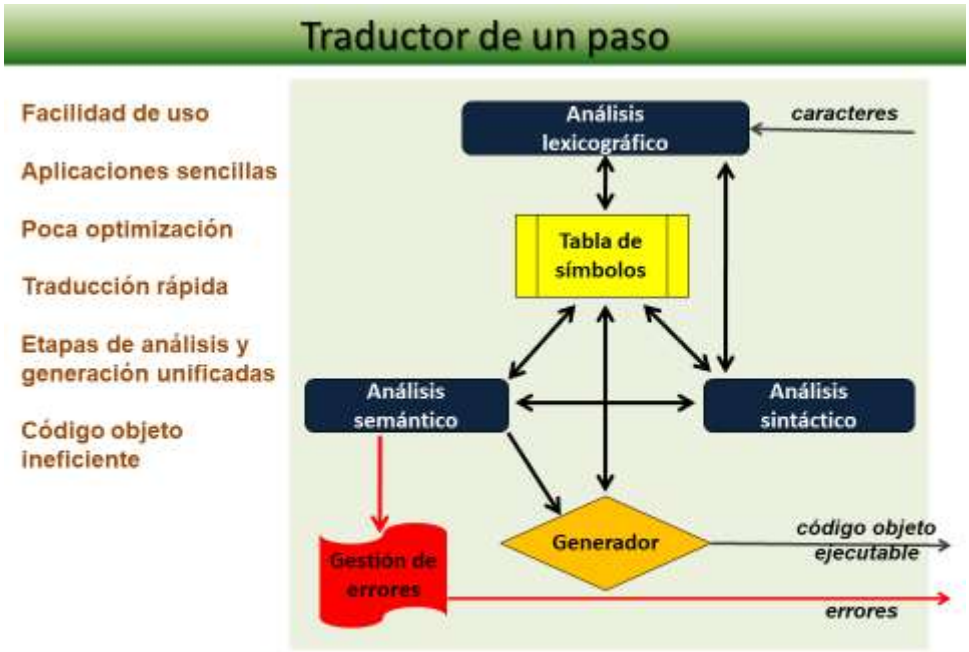
- ✓ Una biblioteca es estática cuando se vincula al código en tiempo de compilación (tiempo de traducción del programa).
- ✓ Una biblioteca es dinámica cuando se vincula al código en tiempo de ejecución. Ejemplo de estas últimas son las DLL (dynamic-link library) que habitualmente encontramos en los entornos Windows.

Los programas compilados (por ejemplo aquellos escritos en lenguaje C/C++), son los más rápidos (en términos de tiempo de ejecución) y óptimos con respecto a los interpretados o pseudo-compilados; pues los compiladores producen mejor calidad del código objeto. No obstante, como antes mencionamos, el problema es que se necesita un compilador nativo para cada tipo de arquitectura de hardware.

El modelo de compilador nativo presentado, separa el proceso de traducción en distintas etapas, dando idea de secuencialidad. Sin embargo en muchos casos estas etapas se ejecutan simultáneamente intercambiando información de forma colaborativa para su desarrollo. Así, dependiendo de cómo se produce la secuencia de ejecución entre los distintos módulos o procesos del traductor, existe en la práctica, un amplio espectro de estrategias de implementación de los traductores. Los traductores de un paso que presentamos a continuación, son otro ejemplo de compilador.

Traductor de un paso

En los traductores de un paso, las fases de análisis y generación de código se realizan a la vez.



Se debe observar que en una sola pasada sobre el código fuente, se obtiene el código objeto. No hay proceso de optimización (lo cual muchas veces requiere varias pasadas de análisis sobre el código fuente).

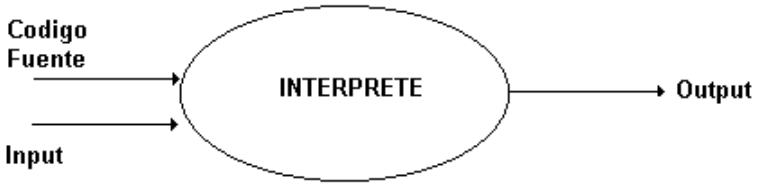
Como lo muestra la filmina, en los traductores de un paso, el proceso está dirigido por el analizador sintáctico, que constituye el centro del proceso de traducción o módulo principal, y el resto de los módulos cooperan a su alrededor.

Este tipo de traductores tienen como ventaja la velocidad de traducción (traducen muy rápido porque no optimizan) y son muy útiles cuando el objetivo es detectar y corregir errores en el código; pero como desventaja, al no optimizar el código, generan ejecutables lentos. Los traductores de un paso son los más rápidos para traducir.

Este tipo de compiladores son generalmente utilizados con fines académicos o en etapas tempranas del desarrollo, donde la velocidad de ejecución del programa obtenido no importa mucho. Los compiladores profesionales, en general no suelen de ser de un paso, sino que son de varios pasos. Se suele cuidar mucho la fase de optimización de código.

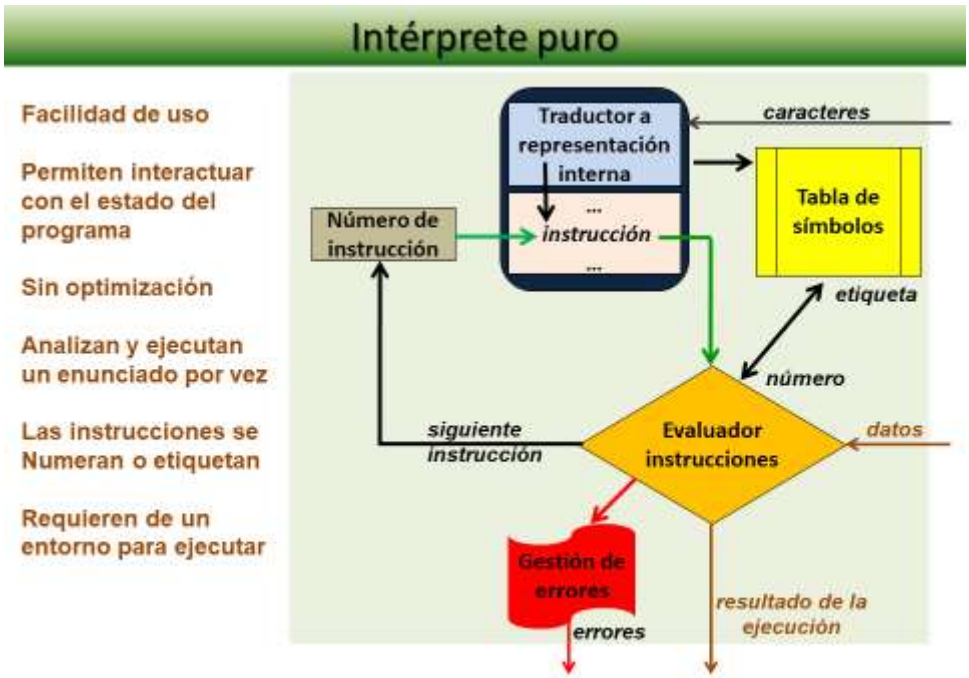
Intérprete puro

Los intérpretes son programas traductores que analizan y ejecutan simultáneamente el programa fuente, es decir no diferencian entre tiempo de traducción y ejecución.



Se pueden clasificar desde el punto de vista de su estructura en varios tipos: intérpretes puros, intérpretes avanzados o normales, e intérpretes incrementales.

Los ***intérpretes puros*** son los que analizan una sentencia y la ejecutan, y así sucesivamente todo el programa fuente. Fueron los intérpretes desarrollados en la primera generación de ordenadores, pues permitían la ejecución de largos programas con ordenadores de memoria muy reducida, ya que sólo debían contener en memoria el intérprete y la sentencia a analizar y ejecutar. 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.



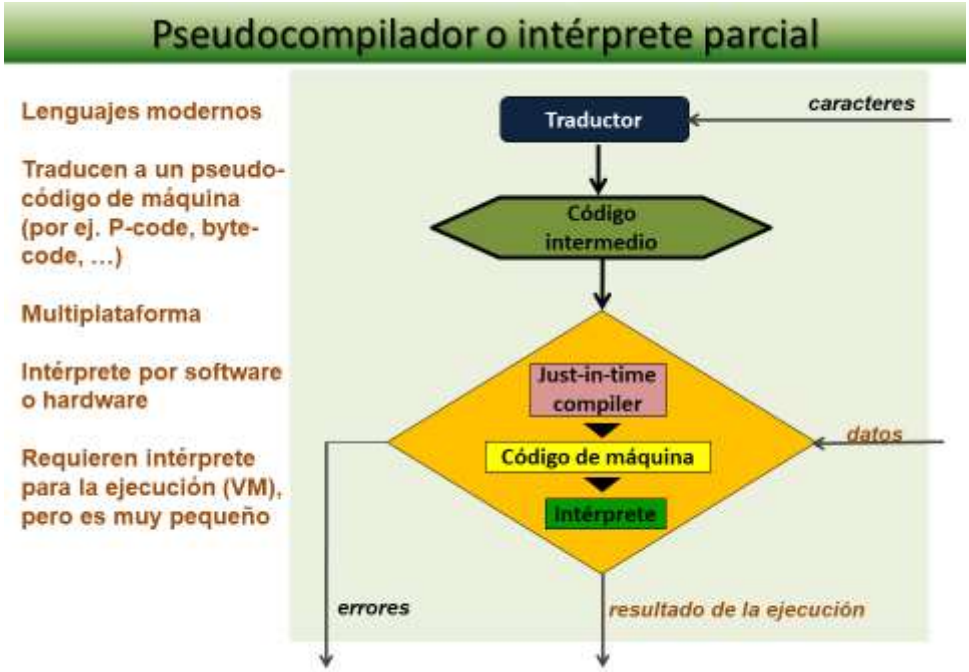
En la filmina se representa el esquema general de un intérprete puro, donde se puede observar que el lenguaje fuente se traduce a una **representación interna** (texto o binaria) que puede ser almacenada en memoria o en disco. Esta representación interna tiene todas las instrucciones numeradas o colocadas consecutivamente en estructuras de tamaño fijo (por ejemplo un array o posiciones consecutivas de memoria, o un fichero binario de estructuras de tamaño fijo). Mientras se realiza este paso se puede construir la tabla de etiquetas, una tablas que contiene las etiquetas y su posición en el programa fuente (las etiquetas se utilizan tanto en las instrucciones de salto como en las llamadas a procedimientos y funciones). Una vez que este proceso ha finalizado, comienza la ejecución por la primera instrucción del código, que se envía al evaluador de instrucciones, éste la ejecuta (recibiendo datos si es necesario o enviando un mensaje de error). El evaluador de instrucciones también determina la instrucción siguiente a ejecutar, en algunos casos previa consulta a la tabla de etiquetas. En el caso de que no haya saltos (GOTO) o llamadas a procedimientos o funciones se ejecuta la siguiente instrucción a la instrucción en curso.

El evaluador de instrucciones puede utilizar dos métodos de evaluación. El método clásico es la evaluación **voraz** o **ansiosa**, donde se evalúan las expresiones completamente. Otro método es la evaluación **perezosa**, evaluándose sólo la parte necesaria de la expresión (el resto no se evalúa).

Ejemplos de lenguajes interpretados son HTML, JavaScript, Ruby, Python, entre muchos otros. Siempre requieren de un entorno intérprete para poder ejecutar. En el caso de HTML o JavaScript el intérprete está en incluido en el browser o navegador de Internet.

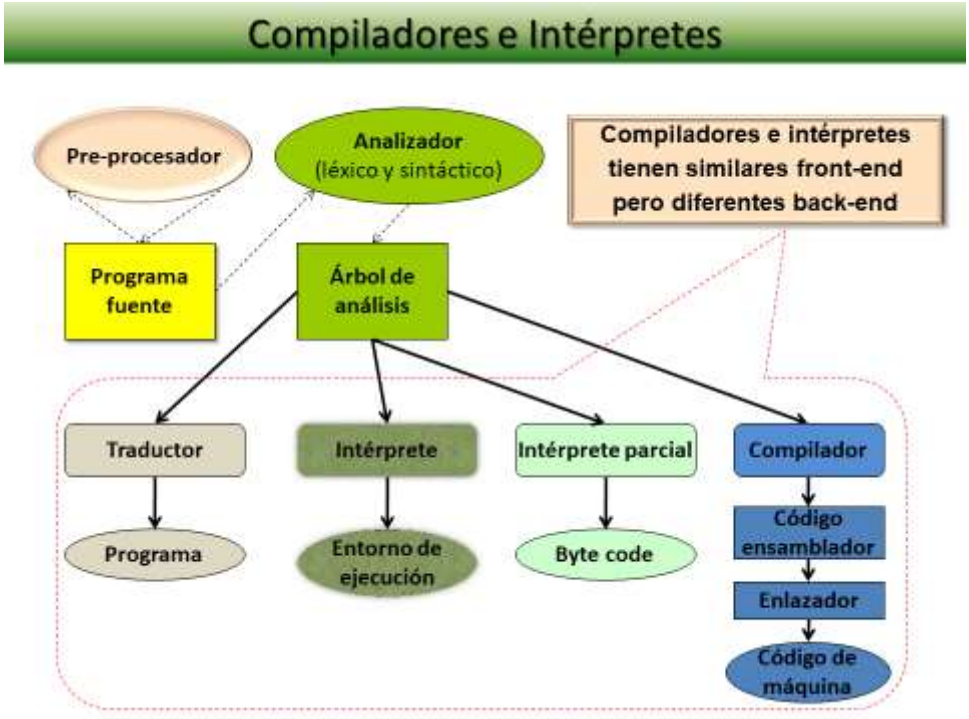
Pseudocompilador o Intérprete parcial

Los intérpretes parciales incorporan un paso previo de análisis de todo el programa fuente. Generando posteriormente un lenguaje intermedio. Este código intermedio es luego interpretado por una VM-máquina virtual que lo ejecuta. De esta forma en caso de errores sintácticos no pasan de la fase de análisis.



Un ejemplo de intérprete avanzado es el que utiliza el lenguaje Java. Así un programa en lenguaje java (archivo con extensión *.java*) se pseudo-compila y produce uno o varios ficheros con la extensión *.class.* Estos ficheros están en un formato binario denominado *byte-code* independiente de plataforma, que se interpreta posteriormente por la JVM o JRE (Java Virtual Machin o Java Runtime Enviroment). Esto permite que el *byte-code* se ejecute en cualquier sistema operativo que disponga de un intérprete de este entorno de ejecución.

Sea cual fuera la arquitectura de traducción adoptada por los diseñadores de un lenguaje, en todas deben llevarse a cabo de alguna forma las etapas del proceso de traducción estudiado. La siguiente filmina ilustra el hecho de que compiladores e intérpretes tienen similares front-ends pero difieren en la forma de resolver el back-end.



En general, los programas compilados son de 10 a 50 veces más rápidos que los interpretados, ya que en general, las decisiones que son tomadas en tiempo de compilación no vuelven a tomarse en tiempo de ejecución.

Los programas interpretados o pseudo-compilados son mucho más **portables** entre plataformas, ya que es más fácil escribir intérpretes que compiladores complejos. Esto se convirtió con la aparición de Internet en una característica relevante. Ejemplo: Javascript, PHP, HTML, Java, Kotlin, etc.

Los lenguajes interpretados permiten también mayor **flexibilidad** (por ejemplo declaración dinámica de tipos) y mejor diagnóstico (mensajes de error) que los compilados, ya que dado que el código se traduce directamente, el intérprete puede incluir un excelente depurador de código.

De hecho, existen algunas características de lenguajes que no pueden ser implementadas sin interpretación, tales como por ejemplo en Lisp o Prolog que el mismo programa puede ser modificado en tiempo de ejecución.

Aunque la diferencia conceptual entre compilación e interpretación es clara, muchos lenguajes incluyen una mixtura de ambos. El problema, entonces es establecer en qué medida, un lenguaje cae en la categoría de compilado y cuándo se puede decir que es interpretado.

En relación a lo anterior, debemos considerar que no necesariamente el proceso de compilación traduce programas en código fuente de alto nivel a código de máquina. Hay

niveles intermedios de traducción que también son considerados compilación. Un caso de esto es por ejemplo un motor de SQL que traduce sentencias SQL en operaciones primitivas sobre archivos. En líneas generales, nosotros entenderemos que hay compilación cuando la traducción implica un completo análisis semántico sobre el significado de los enunciados.