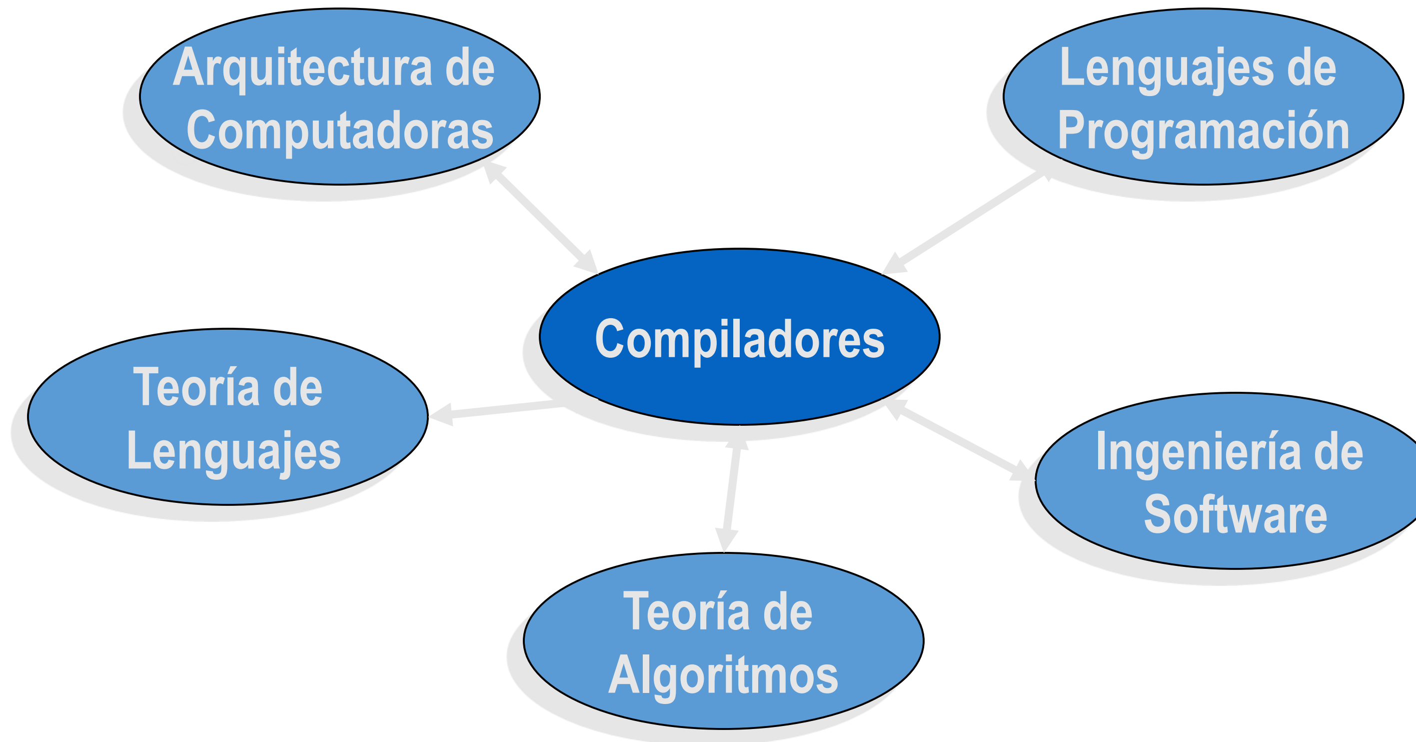
COMPILADORES

Conceptos relacionados



Con algunas técnicas básicas de escritura de compiladores se pueden construir traductores para una gran variedad de lenguajes y máquinas

Introducción

Historia de los compiladores

- 1946, se desarrolla el primer ordenador digital (lenguaje de máquina)
- 1950, John Backus dirige una investigación en IBM en un lenguaje algebraico
- 1954, se comienza a desarrollar FORTRAN
- 1957, FORTRAN se utiliza en la IBM modelo 704
- Surge el concepto traductor
- El primer compilador de FORTRAN tardó 18 años-persona en realizarse
- FORTRAN era dependiente de la máquina
- Paralelamente al desarrollo de FORTRAN en América, en Europa surge una corriente que pretende que los lenguajes fuesen independientes de la máquina, esta corriente estaba influida por los trabajos sobre GLC de Chomsky
- Surge un grupo Europeo encabezado por F.L. Bauer, en la que participó ACM y John Backus. De este grupo surge un informe que define un Lenguaje Algebraico Internacional, publicado en Zurich en 1958
- 1969, aparece Algol 60
- Junto con los lenguajes también la técnica de los compiladores avanza

Introducción

Historia de los compiladores

- 1958, Strong y otros proponen una solución al problema de que un compilador fuera portable, y esta era dividir al compilador en dos fases "front end" (analiza el programa fuente) y "back end" (genera código objeto para la máquina objeto).
- El puente de unión era un lenguaje intermedio denominado UNCOL (no funcionó)
- 1959, Rabin y Scott proponen el empleo de AFD y AFN para el reconocimiento lexicográfico de los lenguajes
- Aparece BNF (Backus-1960, Naur-1963, Knuth-1964) como una guía para el desarrollo del análisis sintáctico
- 1959, Sheridan describe un método de parsing de FORTRAN para introducir paréntesis en una expresión
- En los 60's se desarrollan diversos métodos de parsers ascendentes y descendentes

Introducción

Historia de los compiladores

- Floyd más adelante introduce la técnica de precedencia de operadores y uso de funciones de precedencia
- 1961, se usa por primera vez un parsing descendente recursivo
- En los 60’s se estudia el paso de parámetros por nombre, valor y referencia y se incluyen los procedimientos recursivos para Algol 60
- Se desarrolla la localización dinámica de datos
- 1968, se estudia y definen las GLC, los parsers predictivos y la eliminación de recursividad izquierda
- 1975, aparece LEX generador automático de analizadores léxicos a partir de expresiones regulares bajo UNIX
- A mitad de los 70’s Johnson crea YACC para UNIX (generador de analizadores sintácticos)
- Ahora un compilador de divide en varias varias
- El último lenguaje de programación de amplia aceptación es JAVA (es interpretado)

Introducción

Conceptos Básicos

Traductor. Cualquier programa que toma como entrada un texto escrito en un lenguaje llamado fuente y da como salida un programa equivalente en otro lenguaje, el lenguaje objeto.

Si el lenguaje fuente de un lenguaje de programación de alto nivel y el objeto un lenguaje de bajo nivel (ensamblador o código de máquina), al traductor se le denomina ***compilador***.

Ensamblador. Es un programa traductor cuyo lenguaje fuente es el lenguaje ensamblador.

Intérprete. Es un programa que no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel y la traduce al código equivalente y al mismo tiempo lo ejecuta.

En un principio debido a la escasez de memoria se utilizaban más los intérpretes, ahora se usan más los compiladores (a excepción de JAVA)

Introducción

Conceptos básicos

Ventajas de compilar vs a interpretar

- Se compila una vez, se ejecuta n veces
- En ciclos, la compilación genera código equivalente, interpretándolo se traduce tantas veces una línea como veces se repite el ciclo
- El compilador tiene un visión global del programa

Ventajas del intérprete vs el compilador

- Un intérprete necesita menos memoria que un compilador
- Permiten una mayor interactividad con el código en tiempo de desarrollo

Introducción

Programas que el compilador necesita para obtener un programa ejecutable

- Preprocesador
- Ligador
- Cargador
- Depurador
- Ensamblador

Tipos de compiladores

- De una pasada
- De múltiples pasadas
- De carga y ejecución
- De depuración

Introducción

- De optimización
- Ensamblador
- Compilador cruzado
- Compilador con montador
- Autocompilador
- Metacompilador
- Descompilador

Modelo de análisis y síntesis de la compilación

Partes en la que está dividida la compilación: *análisis y síntesis*

La *síntesis* es la que requiere las técnicas más especializadas.

Introducción

Herramientas de software que realizan algún tipo de análisis a los programas fuente que manipulan:

- Editores de estructuras
- Impresoras estéticas
- Verificadores estáticos
- Intérpretes

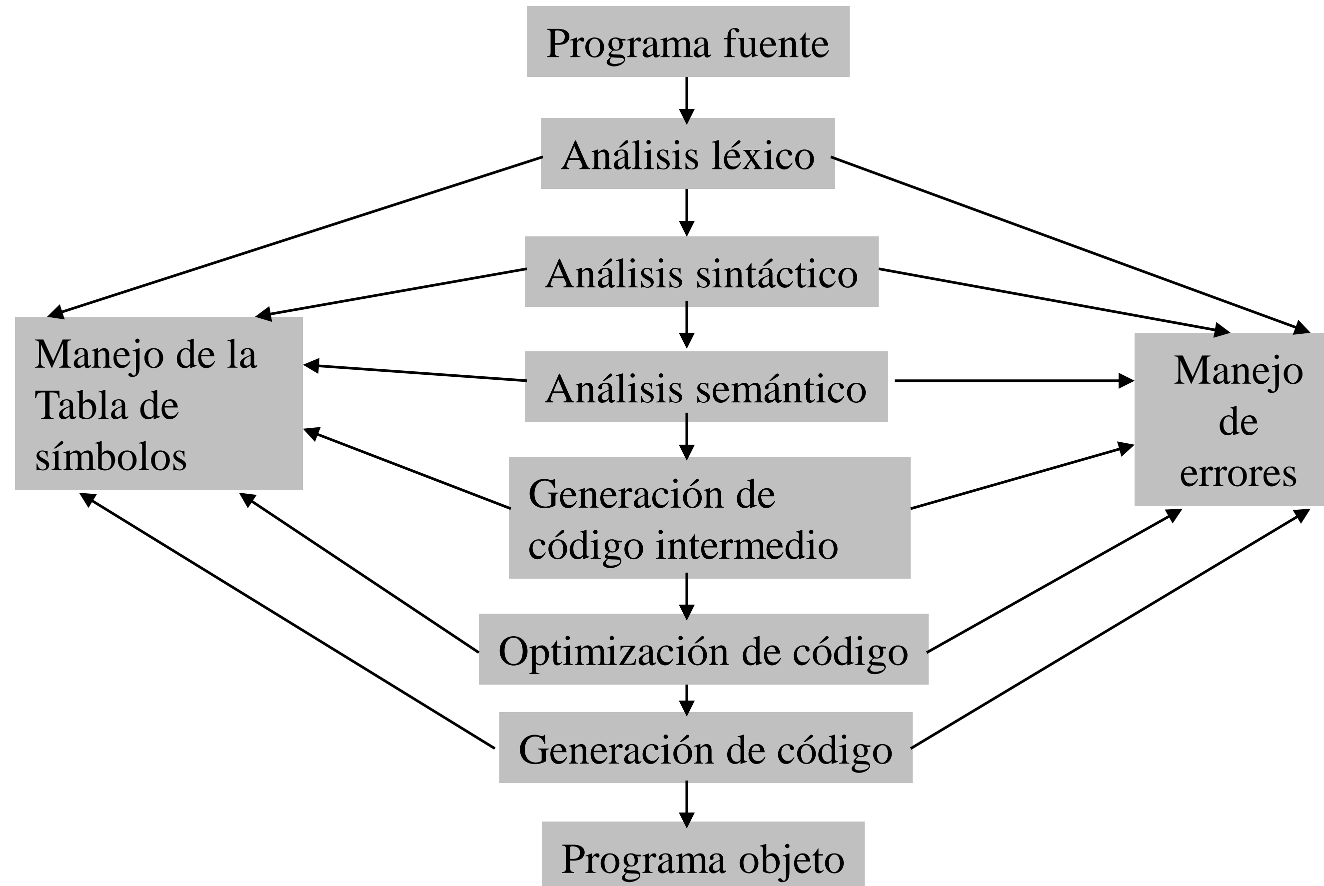
La tecnología de compiladores se aplica en otros lugares, en especial la parte de análisis de los siguientes ejemplos es parecida a la de un compilador convenvcional:

- Formadores de Texto . P/E T_EX
- Compiladores de circuitos de silicio
- Intérpretes de consultas

Introducción

Estructura de un compilador

Para la realización del proceso de traducción es necesario dividir el compilador en varias fases.



Introducción

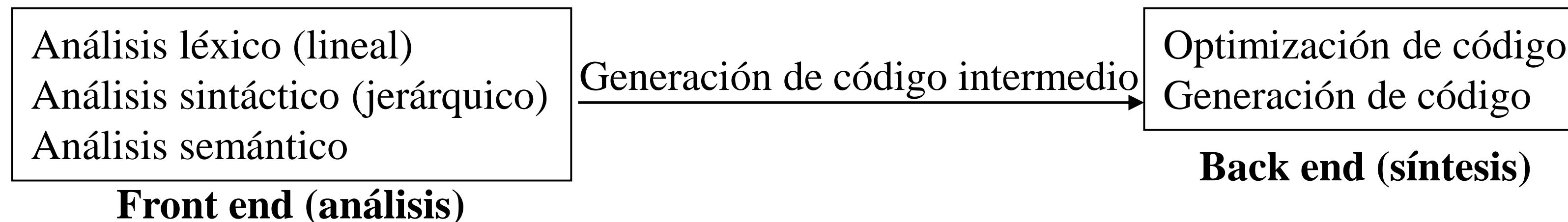
Antes

- Una computadora no tenía memoria suficiente
- Se tuvo que dividir al compilador en fases
- Cada fase leía un archivo y producía otro

Actualmente

- Se tiene memoria suficiente
- El tamaño del archivo ejecutable es relativamente pequeño
- Se han reducido el número de pasadas y el número de archivos que se tienen que leer y escribir

Las fases de un compilador se agrupan en dos partes o etapas:



Introducción

Front end

- Dependiente del lenguaje fuente
- Independiente de la máquina objeto para la que se va a generar código

Back end

- Independiente del lenguaje objeto
- Dependiente del lenguaje objeto

Análisis léxico

También llamado *exploración o scanner*. Lee caracteres uno a uno desde la entrada y va formando grupos de caracteres con alguna relación entre sí llamados tokens, los que serán la entrada para la siguiente etapa del compilador.

Tipos de tokens:

- Tiras específicas
- Tiras no específicas

Introducción

Componentes de un token

- Tipo
- Valor

Las tiras específicas solo tienen tipo.

Análisis sintáctico (parser)

Recibe como entrada los tokens que le pasa el analizados léxico y comprueba si estos van llegando en el orden correcto. Su salida "teórica" sería un árbol sintáctico.

Sus funciones son:

- Aceptar lo que es válido sintácticamente y rechazar lo que no lo es
- Hacer explícito el orden jerárquico que tienen los operadores en el lenguaje de que se trate
- Guiar el proceso de traducción (traducción dirigida por sintaxis)

Introducción

Un factor de división entre el análisis léxico y sintáctico es la recursión.

Análisis semántico

Es más difícil de formalizar que el sintáctico. Este trata de encontrar errores semánticos y reunir información sobre los tipos para la fase de generación de código, realizar verificación de tipos. En definitiva comprobará que el significado de lo que va leyendo es válido.

La salida teórica será un árbol semántico.

Ejemplo:

int i,j,k;
char s[10];
*s=i+j*k;*

Análisis Léxico: Devuelve secuencia de tokens.

tipo id coma id coma id coma puntoycoma
tipo id cora entero corc puntoycoma
id asignación id suma id multiplicación id puntoycoma

Análisis sintáctico: El orden de los tokens es válido

Análisis semántico: Tipo de variables asignadas incorrecta

Introducción

Generación de código intermedio

Transforma un árbol sintáctico (semántico) en un representación en un lenguaje intermedio, que suele ser código sencillo que después se convertirá en código de máquina.

Propiedades de la representación intermedia:

- Debe ser fácil de producir
- Debe ser fácil de traducir al programa objeto

Un formato de código intermedio es el **código de tres direcciones**.

Forma: *A:= B op C*, donde *A,B,C* son operandos y *op* es un operador binario

Se permiten condicionales simples simples y saltos.

P/E

```
        while (a >0) and (b<(a*4-5)) do a:=b*a-10;
L1: if (a>0) goto L2
      goto L3
L2: t1:=a*4
      t2:=t1-5
      if (b < t2) goto L4
   goto L3
L4: t1:=b*a
      t2:=t1-10
      a:=t2
      goto L1
L3: …….
```

Introducción

Optimización de código

Trata de conseguir que el programa objeto sea más rápido en la ejecución y que necesite menos memoria a la hora de ejecutarse.(No todos los compiladores llevan a cabo esta etapa)

Posibles optimizaciones locales:

•Cuando hay dos saltos seguidos se puede quedar uno solo

P/E El ejemplo anterior quedaría así:

```
L1: if (a<=0) goto L3
    t1:=a*4
    t2:=t1-5
    if (b >= t2) goto L3
    t1:=b*a
    t2:=t1-10
    a:=t2
    goto L1
L3: …….
```

•Eliminar expresiones comunes en favor de una sola expresión

```
a:=b+c+d   Quedaría   t1:=b+c
b:=b+c+e              a:=t1+d
                      b:=t1+e
```

Introducción

•Optimización de bucles y lazos (sacar expresiones invariantes)

P/E

```
Repeat                  Se saca x:=3
   x:=3;
   y:=y-x*2;
until y<0;
```

Generación de código

Es la fase final de un compilador y consiste en generar código relocalizable o en ensamblador

Tabla de símbolos

En esta estructura se almacena información como: variables, etiquetas, tipos, etc.

Los accesos a la tabla deben ser lo más rápido posibles

Manejo de errores

Es una de las misiones más importantes del compilador. Se utiliza más en el análisis pero los errores pueden darse en cualquier fase. El manejo de errores es una tarea difícil por dos motivos:

Introducción

1. A veces algunos errores ocultan otros
2. Un error puede provocar una avalancha de errores que se solucionan con el primero

Criterios a seguir a la hora de manejar errores

1. Pararse al detectar el primer error (conveniente para un compilador interactivo)
2. Detectar todos los errores de una pasada (conveniente para un compilador de línea)

SOFTWARE Y SU EVOLUCIÓN

Clasificación de los Lenguajes de Programación

Naturaleza del Lenguaje

- Lenguaje de bajo nivel
- Lenguaje de nivel medio
- Lenguaje de alto nivel

Lenguaje de programación que el ordenador puede entender a la hora de ejecutar programas, lo que aumenta su velocidad de ejecución, pues no necesita un intérprete que traduzca cada línea de instrucciones.

1 1 0 1 1 0 1 0 0 0 1 1 0 1 1 1 0
1 1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 0
0 1 0 1 0 0 0 1 1 0 0 1 1 1 0 0 0
1 0 1 1 1 0 1 1 1 0 1 0 0 1 0 1 1
1 0 0 1 0 1 0 1 1 0 1 1 0 1 0 0 0
1 1 1 0 1 1 1 0 0 0 1 0 1 0 0 1 1
0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1 0

SOFTWARE Y SU EVOLUCIÓN

Clasificación de los Lenguajes de Programación

Naturaleza del Lenguaje

- Lenguaje de bajo nivel
- Lenguaje de nivel medio
- Lenguaje de alto nivel

Ensamblador Lenguaje de programación que está a un paso del lenguaje de máquina. El ensamblador traduce cada sentencia del lenguaje ensamblador a una instrucción de máquina.
Macroensamblador Lenguaje ensamblador que utiliza macros para su utilización (Ver Macro).

SOFTWARE Y SU EVOLUCIÓN

Clasificación de los Lenguajes de Programación

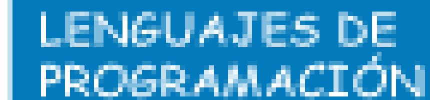
Naturaleza del Lenguaje

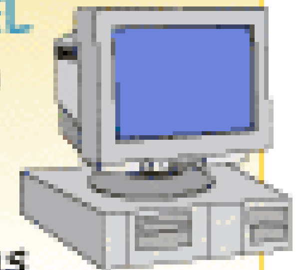
- Lenguaje de bajo nivel
- Lenguaje de nivel medio
- Lenguaje de alto nivel

Lenguaje de programación en el que las instrucciones enviadas para que el ordenador ejecute ciertas órdenes son similares al lenguaje humano. Dado que el ordenador no es capaz de reconocer estas órdenes, es necesario el uso de un intérprete que traduzca el lenguaje de alto nivel a un lenguaje de bajo nivel que el sistema pueda entender

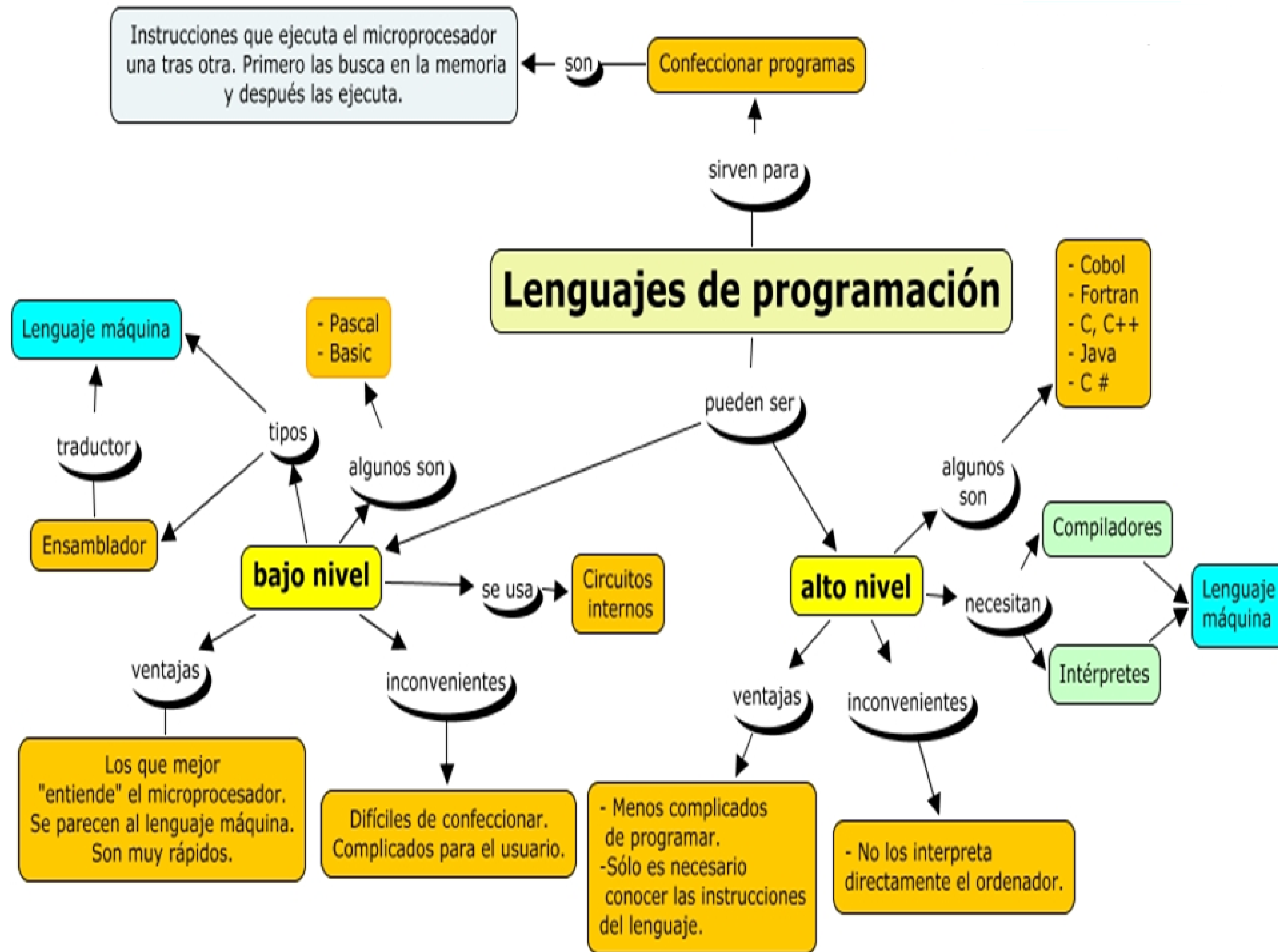
Principales lenguajes de alto nivel

- *C++*
- *C#*
- *COBOL*
- *Fortran*
- *Java*
- *Pascal*
- *Perl*
- *PHP*
- *SQL*
- *Python*

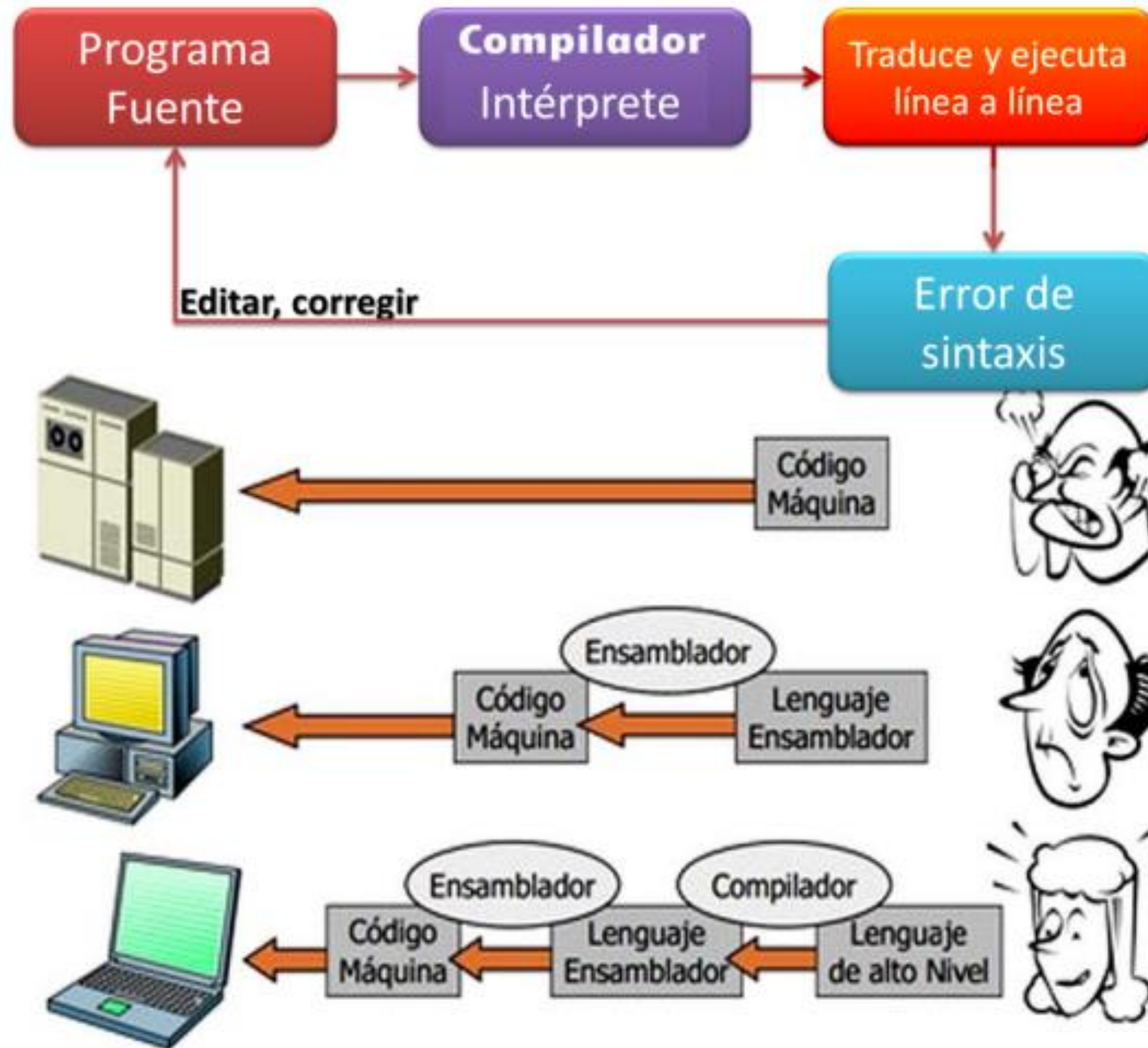




COMPILADORES

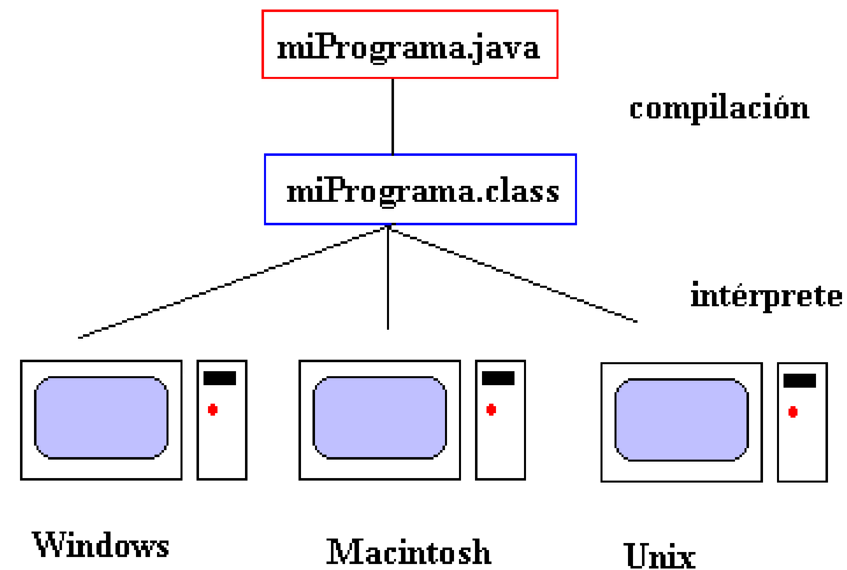


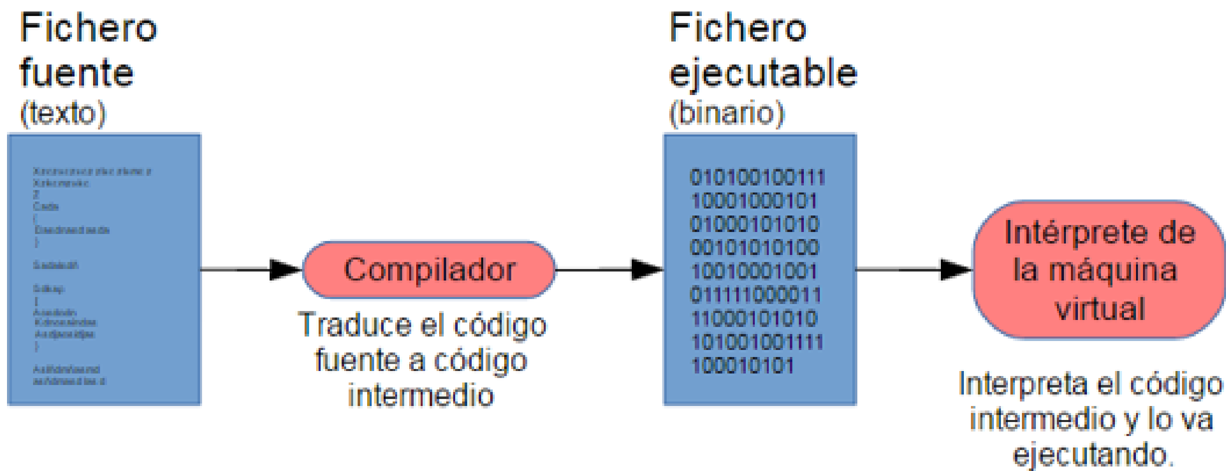
COMPILADORES



COMPILADORES

COMPILADORES





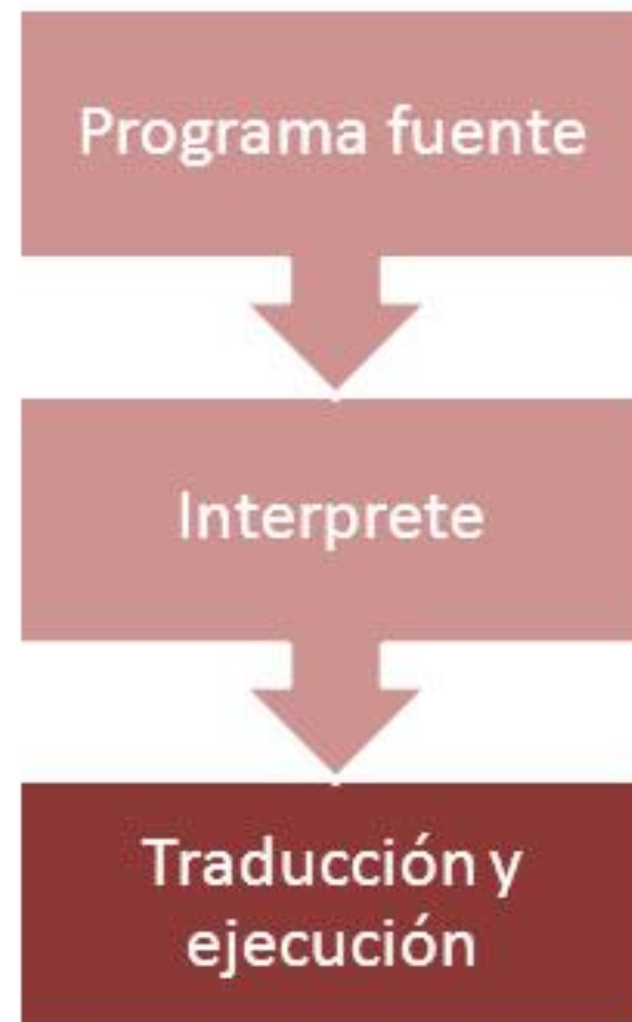
COMPILADORES

Interprete	Compilador
Es un programa que lee línea a línea un programa escrito en un lenguaje; en lenguaje fuente y lo va traduciendo a un código intermedio, para ejecutarlo.	Es un programa que lee totalmente un programa escrito en un lenguaje; el lenguaje fuente, y lo traduce a un programa equivalente a otro lenguaje, lenguaje objeto.
Un intérprete traduce el programa cuando lo lee, convirtiendo el código del programa directamente en acciones.	Un programa que ha sido compilado puede correr por si solo, pues en el proceso de compilación se lo transformo en otro lenguaje (lenguaje máquina).
La ventaja del intérprete es que dado cualquier programa se puede interpretarlo en cualquier plataforma (sistema operativo).	El archivo generado por el compilador solo funciona en la plataforma en donde se lo ha creado.
No genera un ejecutable	Un archivo compilado puede ser distribuido fácilmente conociendo la plataforma, mientras que un archivo interpretado no funciona si no se tiene el intérprete.
El proceso de traducción se realiza en cada ejecución	Hablando de la velocidad de ejecución un archivo compilado es de 10 a 20 veces más rápido que un archivo interpretado.
La ejecución es más lenta, ya que para cada línea del programa es necesario realizar la traducción	Genera un ejecutable
No hay ejecutable, así que si existe un intérprete para una plataforma concreta, el programa se podrá ejecutar en ambas. Típicamente, los programas interpretados son mucho más portables que los compilados, ya que suelen existir intérpretes del mismo lenguaje en distintas plataformas. Los programas que se van a interpretar no suelen ser muy dependientes de su plataforma de destino, siendo más portables.	El proceso de traducción se realiza una sola vez

COMPILADORES

Interprete	Compilador
Los lenguajes interpretados no suelen ser muy dependientes de la plataforma de destino, pero en contrapartida suelen ser menos flexibles y potentes que los compilados.	La ejecución es muy rápida debido a que el programa ya ha sido traducido a código máquina
El código fuente es necesario en cada ejecución, así que no puede permanecer en secreto	El ejecutable va dirigido a una plataforma concreta (una CPU, un sistema operativo, y quizá alguna otra consideración), siendo prácticamente imposible portarlo a otra. En ocasiones, si existe un compilador para otra plataforma, se puede recompilar el programa, aunque normalmente esto plantea serias dificultades. Los programas que se van a compilar suelen estar muy ligados a la plataforma de destino.
Los errores sintácticos se detectan durante la ejecución, ya que traducción y ejecución se van haciendo simultáneamente. Algún error sintáctico podría quedar enmascarado, si para una ejecución concreta no es necesario traducir la línea que lo contiene. (Algunos intérpretes son capaces de evitar esto)	Los lenguajes compilados suelen proporcionar al programador mecanismos más potentes y flexibles, a costa de una mayor ligazón a la plataforma.
Un programa interpretado con un comportamiento torpe normalmente puede ser interrumpido sin dificultad, ya que su ejecución está bajo el control del intérprete, y no sólo del sistema operativo.	Una vez compilado el programa, el código fuente no es necesario para ejecutarlo, así que puede permanecer en secreto si se desea.
	Los errores sintácticos se detectan durante la compilación. Si el fuente contiene errores sintácticos, el compilador no producirá un ejecutable.
	Un programa compilado puede, por error, afectar seriamente a la estabilidad de la plataforma, comprometiendo la ejecución de los otros procesos, por ejemplo, acaparando la CPU, la memoria o algún otro recurso, siendo a veces complicado para el sistema operativo interrumpir su ejecución.

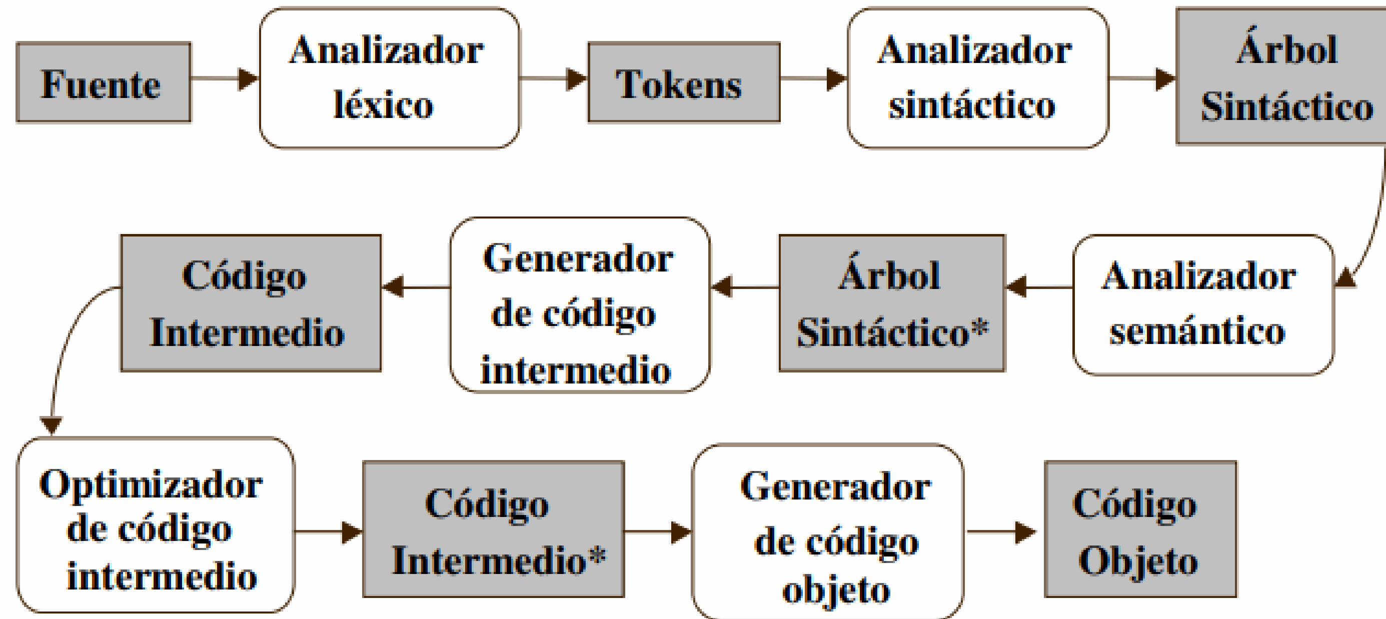
COMPILADORES



Compilador

Intérprete

DETALLE DE LA ESTRUCTURA



Cada fase genera la entrada de la siguiente