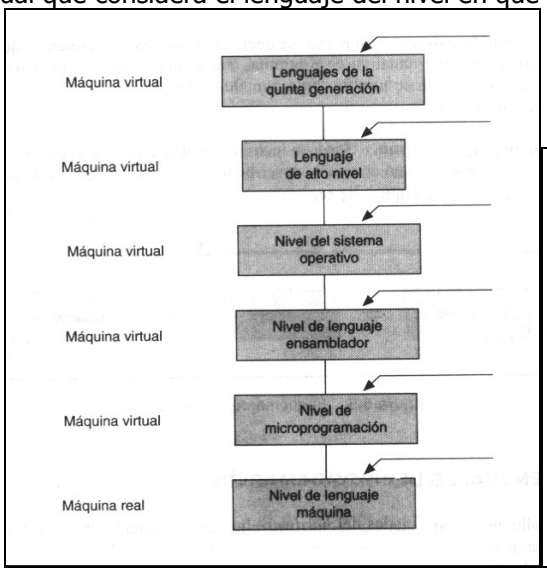
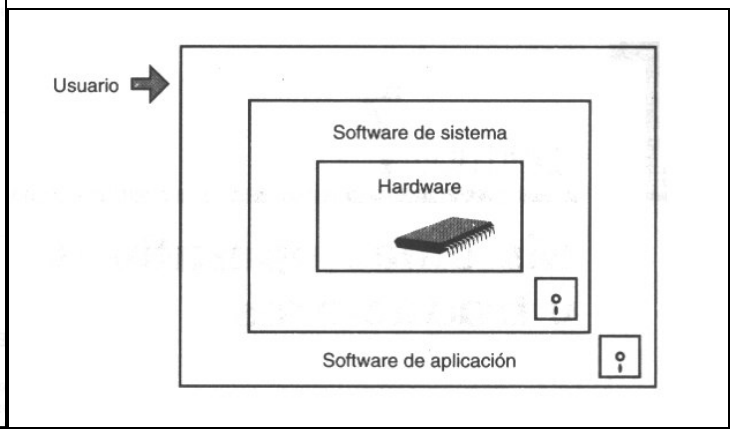
INTRODUCCION

Lenguajes.- Las relaciones humanas se llevan a cabo a través del lenguaje. Un lenguaje permite la expresion de ideas y razonamientos y sin el la comunicación sería imposible. Las computadoras solo aceptan y comprenden un lenguaje de bajo nivel, que consiste en largas secuencias de unos y ceros. Estas secuencias son ininteligibles para muchas personas, y además, son específicas para cada computadora, constituyendo el denominado lenguaje máquina. La programación de computadoras se realiza en los llamados lenguajes de programación que posibilitan la comunicación de órdenes a la computadora.
En informática también se utilizan otros lenguajes que no son de programación y que tienen otras aplicaciones, como pueden ser para describir formatos de texto, gráficos, de sonido, etc. En cualquier caso, todos los lenguajes no naturales son formales, surgen primero las reglas gramaticales y se ajustan con todo rigor a ellas.

Lenguajes de programación.- Un lenguaje de programación es una notación para escribir programas, a través de los cuales es posible la comunicación con el hardware, para dar así las ordenes adecuadas para la realización de procesos.
Un lenguaje está definido por una gramática.
Una gramática es un conjunto de reglas que se aplican a un alfabeto constituido por el conjunto de símbolos utilizados en ese lenguaje.
En el siguiente esquema se presentan distintos niveles de acceso al Hardware, teniendo en cuenta que por el único que se accede al hardware directamente es por el lenguaje máquina; por el resto se accede a una máquina virtual que considera el lenguaje del nivel en que se encuentra como su propio lenguaje máquina.





Clasificación de los lenguajes de programación.-

- De acuerdo a su proximidad al lenguaje máquina o al lenguaje natural (humano) se pueden clasificar en:
  - Bajo nivel (máquina)
  - Intermedios (ensambladores)
  - Alto nivel (evolucionados)

Generaciones de los lenguajes de programación.-

- Primera Generación.- Lenguajes máquina y ensambladores.
- Segunda Generación.- Primeros lenguajes de alto nivel imperativos (Fortran, Cobol)
- Tercera Generación.- Lenguajes de alto nivel imperativos (Algol, Pascal, PL/I)
- Cuarta Generación.- Orientados a aplicaciones y manejo de bases de datos (SQL)
- Quinta Generación.- Orientados a la inteligencia artificial y procesamiento de lenguaje natural (Lisp, Prolog).
- Generación orientada a objetos.- Con la proliferación de las interfaces gráficas de usuarios en los años 80
- Generación visual.- Exigencia de los usuarios de interfaces amigables en los años 90 (Visual Basic, Delphi)
- Generación internet.- Necesidad de manejar aplicaciones en diferentes plataformas dentro de internet (Java, XML, HTML, VRML, .NET)

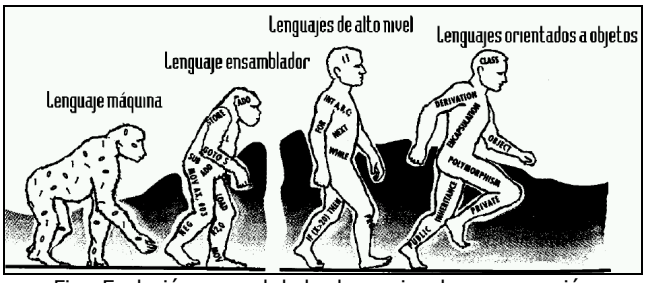


Fig.: Evolución general de los lenguajes de programación

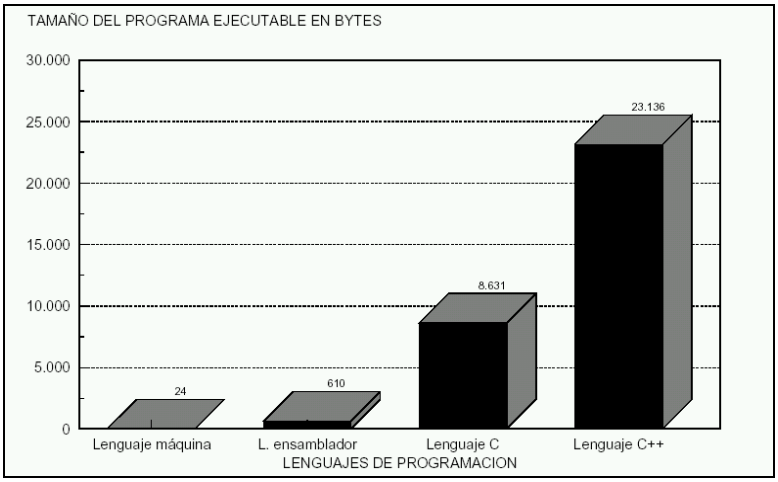
Lenguajes de Alto nivel.- Son aquellos que se encuentran en una capa superior a los ensambladores y lenguaje máquina.

Ventajas de los lenguajes de alto nivel.-

- Más fáciles de aprender, no se necesitan conocimientos hardware de la máquina, son prácticamente independientes de esta.
- Liberación de ocupaciones rutinarias con referencias a instrucciones simbólicas o numéricas, asignaciones de memoria, etc. El traductor se encarga de ello.
- No se necesita saber la forma en que se colocan los diferentes tipos de datos en la memoria.
- Ofrecen una gran variedad de estructuras de control que facilitan la programación estructurada.
- Se depuran más fácilmente. Tienen construcciones que reducen ciertos tipos de errores de programación.
- Mayor capacidad de creación de estructuras de datos complejas, tanto estáticas como dinámicas. Los Lenguajes Orientados a Objetos permiten la reutilización de código.
- Permiten un diseño modular del código, división del trabajo y mayor rendimiento en desarrollo de aplicaciones.

Desventajas de los lenguajes de alto nivel.-

- Mayor tamaño de los ejecutables.
- Mayores tiempos de compilación y de ejecución.
- No se tiene acceso a ciertas zonas del hardware, sólo posible con los de bajo nivel.



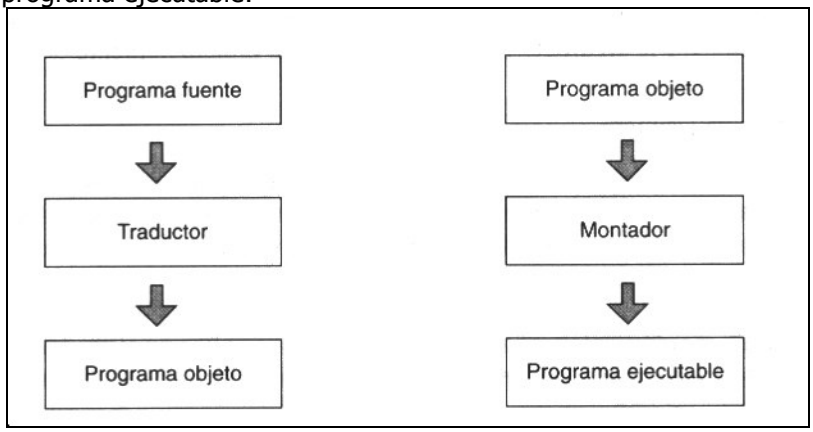
Otros Tipos de lenguajes.-

- De descripción de páginas: Postcript, True Page....
- De formatos gráficos no vectoriales: TIFF, GIF, PCX, BMP, JPEG....
- De formatos gráficos vectoriales: DXF, CGM....
- De formatos de bases de datos: DBF, DBT, MDX....
- De formatos de texto: RTF, ASCII, Word, WordPerfect....
- De formatos de archivos de sonido: WAV, MIDI, MP3....
- De formatos de archivos de vídeo: AVI, MOV,...
- De formatos de ayuda: HLP de Windows, HLP de Turbo Visión...
- De gestión electrónica de documentos e hipertexto: pdf de Acrobat, HTML, XML.....
- De multimedia o de autor: Toolbook, Director...

TRADUCTORES

Procesadores de Lenguaje.- Es el nombre genérico que reciben las aplicaciones informáticas en las que uno de los datos fundamentales de entrada es un lenguaje. Ejemplos de procesadores de lenguaje: Traductores, Ensambladores, Compiladores, Intérpretes, Cargadores, etc.

Traductores.- Son metaprogramas que toman como entrada un programa escrito en un lenguaje simbólico alejado de la máquina denominado **programa fuente** y proporcionan como salida otro programa equivalente escrito en un lenguaje comprensible por el hardware de la computadora denominado **programa objeto**. En algunos casos, un programa objeto necesita antes de su ejecución, alguna preparación adicional para incluir rutinas del propio lenguaje. Esta preparación la realiza un programa que complementa al traductor, denominado **montador o encadenador (Linker)** , produciéndose finalmente un programa listo para ser ejecutado que se denomina **Programa ejecutable**. En la siguiente figura se puede apreciar el proceso de traducción de un programa fuente a un programa ejecutable:



Los traductores pueden ser:

- **De Bajo Nivel**
  - **Ensambladores.-** Son programas traductores que transforman programas fuente escritos en lenguajes simbólicos de bajo nivel (denominados lenguajes ensambladores o assemblers), en programas objeto, escritos en lenguaje máquina y ejecutables directamente por el hardware de la computadora. La traducción del programa de usuario se efectúa de forma que cada instrucción en lenguaje fuente se transforma en una única instrucción en lenguaje objeto. Se puede decir que el lenguaje ensamblador es una simplificación simbólica del lenguaje máquina y el programa ensamblador es su traductor.
  - **Macroprocesadores.-** Caso especial de un traductor en el que se reemplazan macroinstrucciones no haciendo ningún tipo de análisis. Suelen ir incorporados en compiladores. Ejemplo: Bibliotecas de Macros.
- **De Alto Nivel**
  - **Intérpretes.-** Son programas traductores que transforman programas fuente escritos en lenguaje de alto nivel en programas objeto escritos en lenguaje máquina. En estos programas intérpretes la traducción se realiza de forma que después de transformar una instrucción del programa fuente en una o varias instrucciones en lenguaje máquina no esperan a traducir la siguiente instrucción, sino que inmediatamente la ejecutan.
  - **Compiladores.-** Son programas traductores encargados de transformar programas fuente escritos en lenguaje simbólico de alto nivel, en programas objeto escritos en lenguaje máquina. La traducción no suele ser directa, apareciendo un paso intermedio situado en un nivel similar al de ensamblador. Una característica fundamental de este tipo de traductores es que se realiza la traducción completa, y en caso de no existir errores se pasa a la creación del programa objeto. La traducción del programa fuente se efectúa, además, de forma que cada instrucción del programa fuente se transforma en una o mas instrucciones en el programa objeto.

TRADUCTORES DE BAJO NIVEL

ENSAMBLADORES

Cuando se desarrollaron las primeras computadoras no existían lenguajes de programación. Los programas se tenían que introducir directamente en la computadora en forma de lenguaje máquina, que estaba formado por códigos numéricos que representan instrucciones. La escritura y el mantenimiento de los programas en lenguaje máquina era extremadamente difícil; como respuesta a esta necesidad se desarrolló el lenguaje ensamblador.

El ensamblador usa etiquetas mnemotécnicas en vez de códigos numéricos, lo que hace más fácil tanto la escritura como el mantenimiento de programas. Aun así, la programación en lenguaje ensamblador es todavía una tarea tediosa y propensa a errores. Incluso para ejecutar simples tareas se necesitan muchas lineas de código. En los primeros días, cada tipo de computadora tenia su propio lenguaje ensamblador, lo que hacia muy difícil la transferencia de programas de una computadora a otra.

Los lenguajes de alto nivel, como COBOL y FORTRAN, fueron el siguiente paso después del ensamblador. Los programas se pueden escribir mas rápido puesto que una sentencia de un lenguaje de alto nivel hace el trabajo de muchas sentencias en lenguaje ensamblador. Otra ventaja de los lenguajes de alto nivel contra el lenguaje ensamblador es que sus programas pueden ser transferidos de una computadora a otra con solo realizar unas pocas modificaciones.

Así, a medida que paso el tiempo, el ensamblador se relegó a programas de propósito especial, mientras que las aplicaciones generales, se iban escribiendo en lenguajes de alto nivel.

El lenguaje Ensamblador es un programa especial que acepta como entrada un programa en lenguaje ensamblador y produce su equivalente en lenguaje máquina. Un lenguaje ensamblador utiliza palabras y frases para representar los códigos máquina del microprocesador.

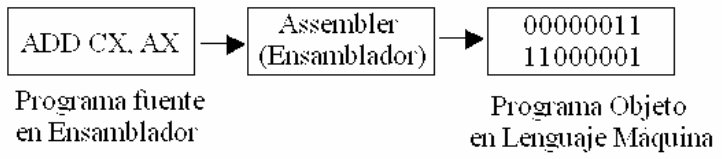


Figura: Función del Lenguaje Ensamblador

Anatomía de una instrucción en Lenguaje Ensamblador.-
Cada instrucción de ensamblador en un programa fuente se compone de un máximo de 4 campos, los cuales se muestran a continuación:

ETIQUETA	MNEMONICO	OPERANDOS	COMENTARIOS
Inicio:	MOV	CX,AX	;Pone valor de AX en CX

- El campo de **etiqueta** asigna un nombre simbólico a una instrucción de ensamblador. Si no se utiliza se puede dejar en blanco.
- El **Mnemónico** indica la instrucción (Operación) que se va a realizar.
- Los **operandos** indican en donde encontrar la información con la que trabajará la instrucción. Algunas instrucciones no utilizan operandos, en ese caso el espacio se deja vacío; en otras ocasiones se usa solo un operando. Otras veces se utilizan dos operandos los cuales se separan por una coma. Cuando hay dos operandos el primero es el destino y el segundo es el fuente.
- Los **comentarios** no se ejecutan, solo son ayuda valiosa para comprender el programa.

Las instrucciones de un lenguaje ensamblador se pueden catalogar en:
- Instrucciones aritméticas (ADD, SUB, MUL, IMUL, DIV, IDIV)
- Instrucciones lógicas (AND, OR, NOT, XOR)
- Instrucciones de Transferencia de datos (MOV, XCHG, PUSH, POP)
- Instrucciones de bifurcación o salto en el programa (JMP)
- Instrucciones de llamada y vuelta de subrutina (CALL, RET)

MACROPROCESADORES

El programador de lenguaje ensamblador frecuentemente encuentra necesario repetir algunos bloques de código muchas veces en el curso de un programa. El bloque puede consistir en código para salvar o conjuntos de intercambio de registros, por ejemplo, código para adaptar enlaces o desempeñar unas operaciones aritméticas en serie. En esta situación el programador encontrará útil el uso de macroinstrucciones.

Las macro instrucciones (frecuentemente llamadas Macros) son abreviaturas de una sola línea para grupos de instrucciones. Al emplear un macro, el programador define esencialmente unas "instrucciones" únicas para representar un bloque de código.

Ventajas de utilizar macros en los programas:

1. Se suprimen y evitan errores
2. Se hace mas corta la programación
3. Facilitan la estandarización

Las macro instrucciones están consideradas generalmente como una extensión del lenguaje ensamblador básico, y el macro procesador está observado como una extensión del algoritmo de ensamblador básico.

Anatomía de una Macro.-
En su forma más simple, un macro es una abreviatura para una secuencia de operaciones. Una facilidad del macro nos permite incluir un nombre a una determinada secuencia y a utilizar este nombre en su lugar.
Una macro instrucción está generalmente formada de la siguiente manera:

Inicio de Defínición	MACRO
Nombre del Macro	[ ]
Cuerpo del Macro	……
	……
	…….
Fin de la Definición	MEND

La palabra **"MACRO"** es la primera línea de la definición e identifica la siguiente línea como el nombre de la instrucción macro. Después de la linea de nombre, se encuentra la secuencia de instrucciones que se abrevian.
La definición está terminada por una linea **MEND** (macro end).

Una vez que el macro se ha definido, puede usarse el nombre del macro como una instrucción mnemónica del lenguaje ensamblador.

Una macro puede contener argumentos o parámetros en las llamadas, de la misma manera que ocurre en los "Procedimientos" y "Funciones" de varios lenguajes de programación de alto nivel. Una macro también puede contener llamadas a otras macros.

Como ya se ha explicado, un macro permite al programador definir una abreviación para una parte de su programa. El macroprocesador trata de identificar las partes del programa definidos por la abreviación como una "Macrodefinición" y graba la definición.

El macroprocesador sustituye la definición de todas las ocurrencias de la abreviación (Macrollamada) en el programa. El macroprocesador además provee al programador con muchas herramientas y escencialmente le permite definir su propio lenguaje personal de "Alto nivel".

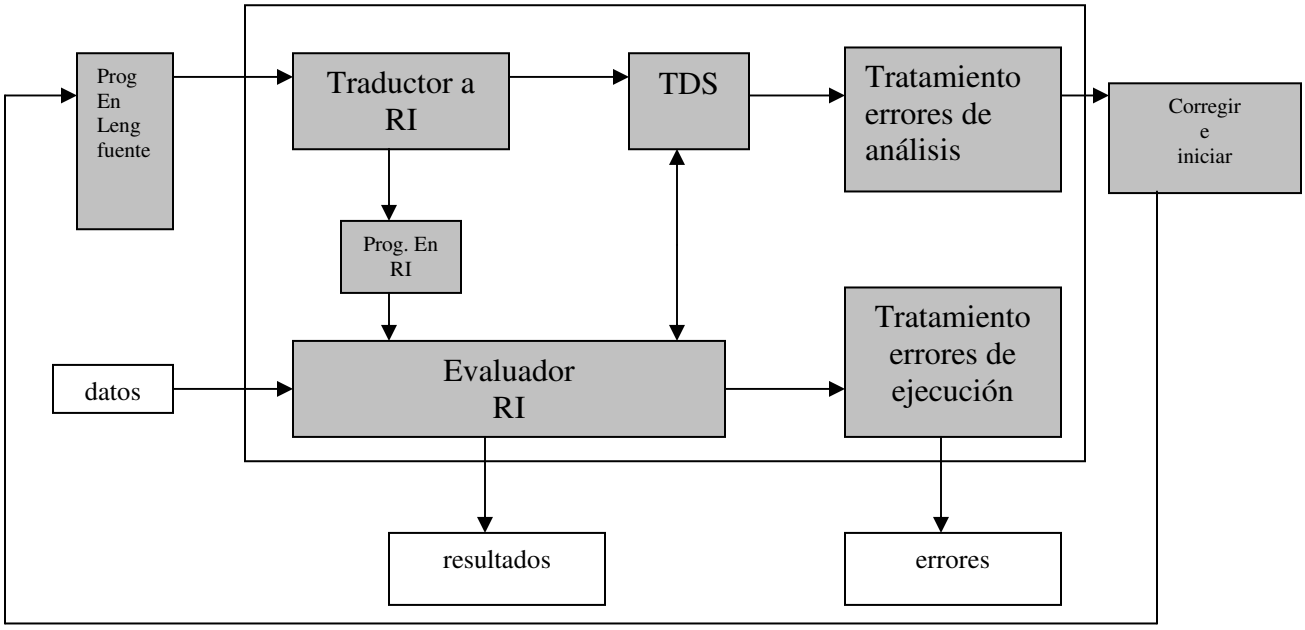
TRADUCTORES DE ALTO NIVEL

INTERPRETES

Un intérprete es un programa que analiza y ejecuta simultáneamente un programa escrito en un lenguaje fuente, es decir, no producen código objeto, siendo su ejecución simultánea a la del código fuente. Cualquier intérprete tiene dos entradas: un programa en lenguaje fuente y los datos de entrada. A partir de estas entradas, tras un proceso de interpretación va produciendo unos resultados.
Un compilador, sin embargo, tras un análisis de todo el programa fuente, transforma éste a un programa equivalente en código objeto (fase de compilación) y en un segundo paso el programa en código objeto junto con los datos de entrada generan un resultado (fase de ejecución)
Un compilador sería equivalente a un traductor de un libro, mientras que un intérprete lo sería a un traductor simultáneo.

Anatomía de un intérprete.- Actualmente en la construcción de la mayoría de los intérpretes se utiliza una representación interna del lenguaje fuente a analizar, organizándose en los módulos:

- *Traductor a representación interna (RI)* : Toma el programa fuente, lo analiza y lo transforma a la representación interna. Esta representación suele ser árboles sintácticos, pero si las características del lenguaje lo permiten, pueden utilizarse estructuras de pila para una mayor eficiencia.
- *Tabla de símbolos (TDS).-* Tabla donde se almacena información relativa a los símbolos que aparecen: etiquetas para las instrucciones de salto, lo relativo a identificadores, etc.
- *Evaluador de representación interna*: Partiendo de la representación interna y de los datos de entrada, se llevan a cabo las acciones indicadas para obtener los resultados.
- *Tratamiento de errores*: Durante el proceso de evaluación pueden aparecer diversos errores como desbordamiento de la pila, divisiones por cero, etc.



Dependiendo de la complejidad del código a analizar, el intérprete puede contener módulos similares a los de un compilador tradicional: análisis léxico, sintáctico y semántico. Durante la evaluación, el intérprete interactúa con los recursos del sistema como la memoria, los discos, etc.

A la hora de evaluar la representación interna existen dos métodos:

- Interpretación iterativa.
- Interpretación recursiva.

Interpretación iterativa.- Apropiada para lenguajes sencillos, donde se analiza y ejecuta cada expresión de forma directa, como podrían ser los códigos de máquinas abstractas o lenguajes de sentencias simples.

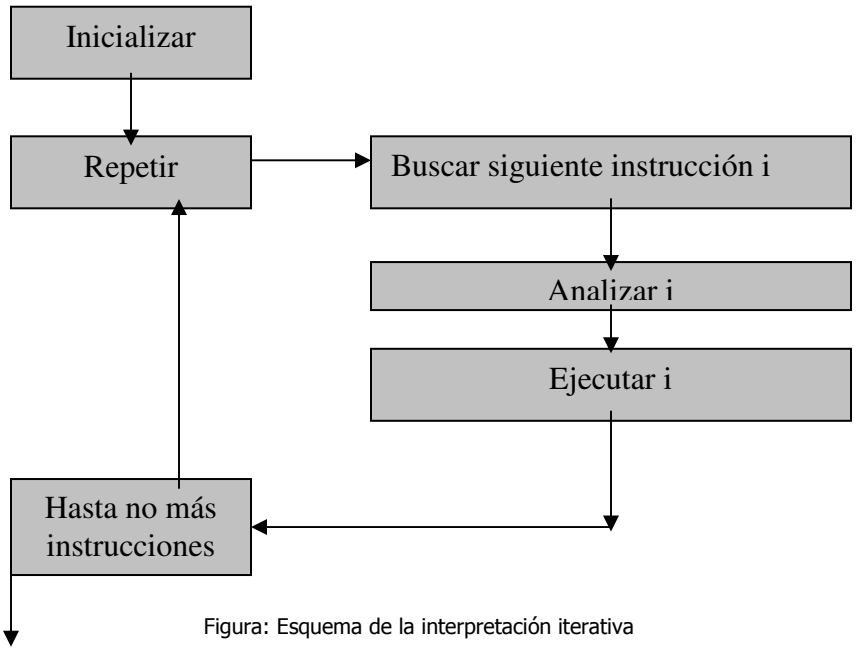


Figura: Esquema de la interpretación iterativa

Cada instrucción se busca en la memoria o en el disco, o en algunos casos es introducida directamente el usuario. La instrucción es analizada en sus componentes y ejecutada. Normalmente el lenguaje fuente contiene varios tipos de instrucciones, de forma que la ejecución se descompone en varios casos, uno por cada tipo de instrucción.

Interpretación recursiva.- En un intérprete recursivo, las sentencias pueden estar compuestas de otras sentencias y la ejecución de una sentencia puede lanzar la ejecución de otras en forma recursiva. Éstos no son adecuados para aplicaciones prácticas a causa de su ineficiencia y se usan tan sólo como prototipo ejecutable del lenguaje.

Ventajas del uso de intérpretes.- En general, el uso de compiladores permite construir programas más eficientes que los correspondientes interpretados. Ello es debido a que durante la ejecución no es necesario realizar procesos de análisis complicados. Además un compilador es capaz de detectar errores y optimizar el código generado.

El intérprete realiza el análisis y la ejecución a la vez, lo que imposibilita las optimizaciones, por lo que suelen ser menos eficientes que los compilados.

Sin embargo, los nuevos avances informáticos aumentan la velocidad de procesamiento y la capacidad de memoria de los procesadores. Ahora la eficiencia es un problema menos grave y en ocasiones se prefieren un sistema que permita un desarrollo rápido.

Se podrían enumerar las siguientes ventajas al utilizar intérpretes:

- Los intérpretes son más sencillos de implantar.
- Proporcionan mayor flexibilidad que permiten modificar y ampliar las características del lenguaje fuente.
- No es necesario contener en memoria todo el código fuente, lo que permite su utilización en sistemas de poca memoria o entornos de red, en los que se puede obtener el código fuente a medida que se necesita.
- Aumenta la portabilidad del lenguaje. Sólo es necesario que el intérprete funcione en otra máquina.
- Puesto que no hay etapas intermedias de compilación, los sistemas interpretados facilitan el desarrollo rápido de prototipos., potencian el uso de sistemas interactivos y facilitan las tareas de depuración.

Aplicaciones de los sistemas basados en intérpretes

- ***Intérpretes de comandos***: los sistemas operativos utilizan estos intérpretes.

- ***Lenguajes de "Script":*** diseñados para que sirvan de enlaces entre diferentes sistemas o aplicaciones (Perl, JavaScript,etc.)

- ***Entornos de programación*** **:** Hay lenguajes que impiden su compilación o cuya compilación no es efectiva. Suelen disponer de un complejo entorno de desarrollo interactivo con facilidades para la depuración (Lisp, Visual Basic,etc.)

- ***Lenguajes de propósito específico*** **:** Lenguajes de consultas de bases de datos, robótica, simulación, etc.

- ***Sistemas en tiempo real*** **:** Entornos que permiten modificar el código de una aplicación en tiempo de ejecución de manera interactiva.

- ***Intérprete de código intermedio*** : Hay una tendencia actual al diseño de compiladores con la generación de código intermedio para una máquina abstracta, como los bytecode de Java. La generación de código objeto se hace a partir del código intermedio interpretándolo en una máquina concreta. Para ello se define una máquina virtual que contenga las instrucciones definidas para el lenguaje intermedio, permitiendo una mayor portabilidad. La Máquina Virtual de Java es simulada en la mayoría de los visualizadores / navegadores WEB.

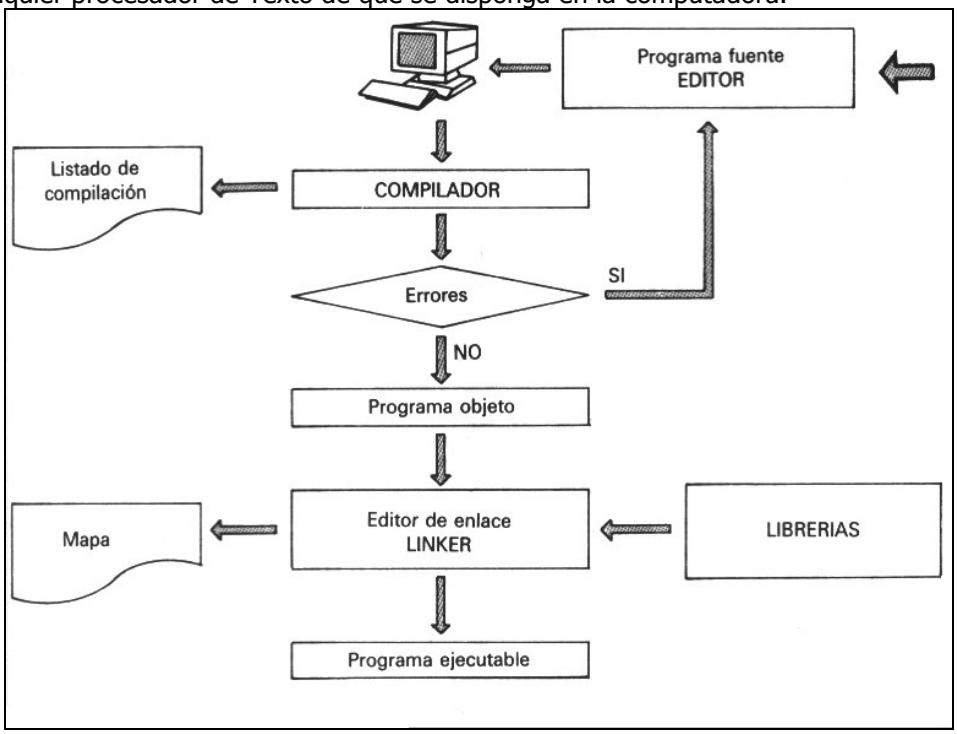
COMPILADORES

En la disciplina de Procesadores de lenguaje, los compiladores ocupan una posición privilegiada, dado que son las herramientas mas utilizadas por los profesionales de la informática para el desarrollo de aplicaciones.

Para ejecutar la compilación de un programa, éste debe estar en memoria central simultáneamente con el compilador. El resultado de la compilación puede dar lugar a la aparición de errores, en cuyo caso no se genera el programa objeto sino que se realiza lo que se denomina Listado de compilación, el cual permite ver dichos errores para volver al programa editor, corregirlos y empezar de nuevo el proceso de compilación.

Obtenido el programa objeto es necesario someterlo a un proceso de montaje (Linkage) donde se enlazan los distintos módulos, en caso de programas que poseen subprogramas. Además se incorporan las denominadas rutinas de librería en caso de solicitarlas el propio programa. Como ya se mencionó anteriormente, este montaje lo realiza un programa denominado LINKER o Montador.

La siguiente figura muestra el esquema general del proceso de compilación de un programa, donde en primer lugar se crea el mencionado programa fuente, normalmente mediante un programa de utilidad denominado Editor o con cualquier procesador de Texto de que se disponga en la computadora.



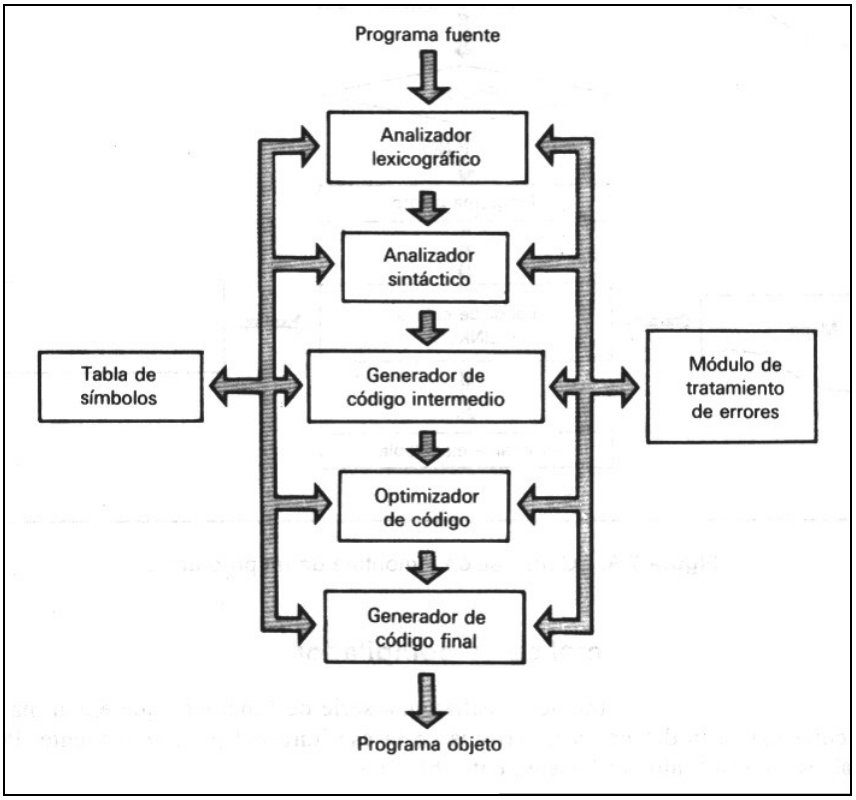
Estructura General de un Compilador.-

Un compilador, además de traducir, realiza una serie de funciones que en su mayoría están enfocadas a la detección de errores en la escritura del programa fuente. Por lo general, está constituido por los siguientes bloques:

- Analizador Lexicográfico (Scanner)
- Analizador Sintáctico (Parser)
- Generador de Código intermedio
- Optimizador de código
- Generador de código final
- Tabla de símbolos
- Módulo de tratamiento de errores

El compilador utiliza internamente una tabla de símbolos para introducir determinados datos que necesita y que está relacionada con todos sus elementos; asimismo, posee un módulo de tratamiento de errores también relacionado con todos sus elementos para determinar las reacciones que se deben producir ante la aparición de cualquier tipo de error.

La siguiente figura muesta el esquema de la estructura general de un compilador donde se pueden ver sus relaciones y secuenciamiento.



Analizador Lexicográfico.- También denominado scanner. Examina en el programa fuente las unidades básicas de información pertenecientes al lenguaje. Estas unidades básicas se denominan unidades léxicas o tokens. En el caso de no existir errores en este primer análisis, se realiza una primera traducción a un código propio del compilador, eliminando a su vez toda información superflua como pueden ser los comentarios y espacios en blanco no significativos del programa fuente.
Un token es un elemento o cadena con significado propio en el programa fuente, como por ejemplo pueden ser las palabras reservadas del lenguaje, los identificaciores, los operadores, etc. El resultado de este análisis y traducción es lo que se denomina tira de tokens que es la información que recibe el siguiente elemento del compilador.
Un ejemplo de error lexicográfico puede ser una palabra reservada mal escrita, un identificador no permitido, etc.

Analizador Sintáctico.- También denominado parser. Recibe la tira de tokens del scanner e investiga en ella los posibles errores sintácticos que aparezcan. Estos errores suelen ser de formatos de instrucciones, duplicidad de identificadores de distintas variables, etc.
Tanto en el analizador lexicográfico como en el sintáctico, todo error detectado es comunicado al programador por medio del listado de compilación, en el que se indica donde está situado y qué tipo de error es. En ocasiones, se indican al programador determinados errores que pueden haber sin que éstos perjudiquen el resto del proceso de compilación e incluso pueden permitir el funcionamiento del programa final. Estos mensajes de error se denominan advertencias o warnings.

Generador de código intermedio.- Traduce el resultado del análisis anterior (en caso de ausencia de errores) a un código intermedio propio del compilador, para con él permitir la portabilidad del lenguaje (posibilidad de utilización en distintas computadoras).
Debido a que el lenguaje máquina es distinto entre una computadora y otra, para un lenguaje de alto nivel determinado, se hace común todo el proceso de análisis, con lo que además se abaratan los costos de construcción de un compilador al aplicar el mismo lenguaje a computadoras de distinta naturaleza. Este código intermedio a continuación se particularice para cada familia de procesadores.

Optimizador de código.- La misión del optimizador de código consiste en tomar el código intermedio y optimizarlo, adaptándolo a las características del procesador al que va dirigido.

Es conveniente tener en cuenta que el código intermedio está enfocado a que un programa en dicho código pueda ser, con algunas modificaciones interpretado por cualquier procesador. Por tanto, podemos decir que el optimizador de código, además de su mision, realiza la de adaptar el código a un procesador concreto.

Generador de código final.- Su misión es la de traducir el código intermedio optimizado en el código final, es decir, en el lenguaje máquina del procesador al que el compilador va dirigido.

Módulo de tratamiento de errores.- Su misión es facilitar la detección y, en algún caso, la recuperación de errores en las distintas fases de la compilación. Un compilador, cuando detecta un error, trata de buscar su localización exacta y la posible causa del mismo para ofrecer al programador un mensaje de diagnóstico que será incluido en el listado de la compilación. Los tipos de errores que puede tener un programa son los siguientes:

- **Errores lexicográficos.-** Son errores que se producen por la aparición de tokens no reconocibles. Los detecta el scanner en el tiempo en que se ejecuta el analizador lexicográfico.
- **Errores sintácticos.-** Son los que no cumplen las reglas de sintaxis del lenguaje, detectándolos el parser al no reconocer una tira completa de tokens como un formato válido de sentencia.
- **Errores semánticos.-** Se detectan en alguna fase de la compilación y en algún caso en la ejecución y son aquellos que no interrumpen el proceso, pero se detecta alguna incorrección. Son los denominados Warnings.
- **Errores lógicos.-** Son los debidos a la utilización de un algoritmo o expresion incorrecta para el problema que se trata de resolver. Se detectan en la fase de pruebas de un programa por medio de "prueba y error".
- **Errores de ejecución.-** Son errores relacionados con desbordamientos, operaciones matemáticamente irresolubles, etc. Ejemplos de este tipo de errores son: division por cero, cálculo de la raiz cuadrada de un número negativo, desbordamientos , leer de un archivo no abierto o sin información, salir o exceder el rango de una tabla, bucles infinitos, etc.

Tabla de símbolos.- Esta tabla almacena todos los datos correspondientes a las variables y estructuras de datos del programa que se está compilando. Estos datos suelen ser: tipo de cada variable o estructura, dimensiones, características, etc.

Ejemplo 1 : Realizar el recorrido del compilador para la siguiente instrucción:

DO WHILE (A <=50)

ANALIZADOR LEXICO: PR1 PR2 CE1 ID OR CN CE2
PR1 → DO
PR2 → WHILE
CE1 → (
ID → A
OR → <=
CN → 50
CE2 →)

ANALIZADOR SINTACTICO: CORRECTO
CORRECTO → ID OR CN
CORRECTO → CE1 CORRECTO CE2
CORRECTO → PR1 PR2 CORRECTO

ANALIZADOR SEMANTICO: CHECA LOS TIPOS DE DATOS (CORRECTO)

CODIGO INTERMEDIO:
T1 = ID OR CN
T2 = CE1 T1 CE2
T3 = PR1 PR2
T4 = T3 T2

OPTIMIZACION DE CODIGO:
T1 ID OR CN
T2 CE1 T1 CE2
T3 PR1 PR2 T2

Ejemplo2: Realizar el recorrido del compilador para la siguiente instrucción:

P = I + V * 60

ANALIZADOR LEXICO: ID1 OA ID2 OA ID3 OA CN
ID1 → P
ID2 → I
ID3 → V
OA → + | - | * | * | =

ANALIZADOR SINTACTICO: CORRECTO
CORRECTO → ID OA CN
CORRECTO → ID OA CORRECTO

ANALIZADOR SEMANTICO: CHECA LOS TIPOS (CORRECTO)

GENERADOR DE CODIGO INTERMEDIO:
T1 = ID3 OA CN
T2 = ID2 OA T1
T3 = ID1 OA T2

OPTIMIZACION DE CODIGO:
T1 = ID3 OA CN
T2 = ID1 OA ID2 OA T1

NOCIONES BASICAS DE LENGUAJES, GRAMATICAS Y AUTOMATAS

Lenguajes.- Un lenguaje de programación es una notación formal para describir algoritmos o funciones que serán ejecutadas en una computadora.
Alfabeto o vocabulario es un conjunto finito de símbolos.
Cadena es una secuencia finita de símbolos de un determinado alfabeto.
Lenguaje es un conjunto de cadenas de símbolos de un alfabeto determinado.

Gramáticas.- Una gramática describe de forma natural la estructura jerárquica de muchas construcciones de los lenguajes de programación; es decir, *la definición formal de la sintaxis de un lenguaje de programación se llama gramática.* Una gramática se puede definir formalmente con los siguientes elementos:

- Terminales.- Son símbolos básicos con que se forman las cadenas. No pueden ser divididos nuevamente. Todas las sentencias del lenguaje están formadas con simbolos del vocabulario terminal. Por ejemplo, las palabras reservadas son Terminales.
- No Terminales.- Son símbolos que pueden ser divididos nuevamente en Simbolos Terminales y/o No terminales. Los No-Terminales son un Conjunto de simbolos auxiliares para la definición de las producciones de la gramática y que no figuran en las sentencias del lenguaje.
- Simbolo inicial.- Es un no-Terminal de una gramática a partir del cual pueden obtenerse todas las sentencias del lenguaje generado por la gramática aplicando las reglas de ésta.
- Reglas de producción (Producciones).- Transformaciones de cadenas de símbolos que se expresan mediante una pareja de expresiones separadas por una flecha. Especifican como se pueden combinar los terminales y no-terminales para formar cadenas.

Una gramática desde el punto de vista lingüistico e informático debe cumplir dos objetivos:

- Definir si una oración pertenece o no al lenguaje
- Describir estructuralmente las sentencias.

Una sentencia pertenece a un lenguaje si:

- Está compuesta de símbolos terminales
- Puede derivarse del símbolo inicial mediante las reglas de producción de la gramática.

Autómatas.- El comprobar si una sentencia pertenece o no a un determinado lenguaje es tarea de los autómatas. La palabra autómata evoca algo que pretende imitar las funciones propias de los seres vivos, especialmente relacionadas con el movimiento, por ejemplo el típico robot antropomorfo. En el campo de los procesadores de lenguaje lo fundamental no es la simulación del movimiento sino la simulación de procesos para tratar información.

A un autómata se le presenta una cadena de entrada y produce otra cadena de símbolos de salida. Recibe los símbolos a la entrada secuencialmente. El símbolo de salida que produce no sólo depende del de entrada en un determinado momento sino también de toda la secuencia recibida hasta ese momento. En el estudio de los compiladores e intérpretes, los autómatas son utilizados como reconocedores de lenguajes. Una cadena pertenecerá a un lenguaje si el autómata la toma como entrada y partiendo del estado inicial transita a través de varias configuraciones hasta alcanzar un estado final.

Metalenguajes.- Son herramientas para la descripción formal de los lenguajes, facilitando no solo la comprensión del lenguaje, sino también el desarrollo del compilador correspondiente. Ejemplos: Expresiones regulares, notación BNF, notación EBNF.

Expresiones Regulares.- Una expresión regular es una notación que permite definir de manera precisa los patrones para obtener los componentes léxicos. Una expresion regular se construye de expresiones regulares mas pequeñas utilizando un conjunto de reglas que definen el lenguaje. Las expresiones regulares utilizan los siguientes tipos de operadores para definir los componentes léxicos:

- ( ) .- Se permite el uso de paréntesis para agrupar símbolos.
- * .- Cero o mas casos
- + .- Uno o mas casos
- ? .- Cero o un caso
- | .- Alternativa "O"

Ejemplo 1: Realizar la expresión regular para un identificador (Nombre de variable) que inicie con letra y le sigan letras o digitos o nada.

$$\text{Letra (Letra | Digito)}^{*}$$

Ejemplo 2: Expresión regular para una Constante entera sin signo

$$\text{Digito}^{+}$$

Ejemplo 3: Expresión regular para una constante entera con signo requerido

$$\text{(- | +) Digito}^{+}$$

Ejemplo 4: Expresión regular para operadores relacionales (<> , = , <= , >=, < , >)

$$\text{< (= | >)}^{?} \text{ | > (=)}^{?} \text{ | =}$$

Ejemplo 5: Expresión regular para la palabra reservada "PRINT"

$$\text{(P)(R)(I)(N)(T)}$$

Definición Regular.- Una definición regular es una expresión regular a la cual se le asignó un nombre.

Ejemplo 6: Definición regular para un identificador que inicie con letra y le sigan letras o digitos o nada.

$$L \rightarrow A \mid B \mid C \mid \ldots \mid Z$$
$$\# \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$
$$ID \rightarrow L (L \mid \#)^{*}$$

Ejemplo 7: Definición regular para operadores aritméticos

$$OA \rightarrow = \mid + \mid - \mid * \mid / \mid \wedge$$

Ejemplo 8: Escribir una definición regular para constantes numéricas con las siguientes reglas:
Puede Tener signo o no. Puede tener decimales o no. Puede comenzar con punto. Si tiene decimales debe tener por lo menos 2 digitos después del punto. Puede tener exponente E. El digito del exponente puede tener signo y debe ser entero.

$$\# \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$
$$S \rightarrow + \mid -$$
$$D \rightarrow .\#\#^{+}$$
$$EX \rightarrow ES^{?}\#^{+}$$
$$CN \rightarrow S^{?} (D \mid \#^{+} D^{?}) EX^{?}$$