

Curso: Compiladores

Tema 1. Panorama general

Objetivo.

El alumno identificará el papel de los traductores como herramientas de uso y desarrollo de sistemas de software, además de distinguir las diferentes áreas de trabajo de éstos.

Contenido.

- 1.1 Reseña del origen y evolución de los lenguajes traductores
- 1.2 Clasificación de los traductores.
- 1.3 Etapas en el proceso de compilación
- 1.4 Ambientes de ejecución

1.1 Reseña del origen y evolución de los lenguajes traductores

Iniciaremos este tema con los orígenes de los lenguajes empleados en el desarrollo de los sistemas de software y aplicaciones.

Seguramente has utilizado varios lenguajes en tus cursos de programación o de temas relacionados con el área de computación.

Actividad 1.1.1 Haz una lista de 3 lenguajes que hayas utilizado en dichos cursos o que conozcas de su utilización. Indica, en cada uno, además de su uso académico, en dónde o para el desarrollo de qué aplicaciones se han utilizado.

En la actualidad, existe una infinidad de lenguajes con los que se desarrolla software de varios niveles, es decir, desde desarrollo de sistemas operativos (bajo nivel) hasta aplicaciones de comunicación (alto nivel). Esto se debe a que las tecnologías de la información y comunicación (TIC) están en nuestro quehacer cotidiano para realizar muchas de nuestras actividades.

Los orígenes de los lenguajes para el desarrollo de software o aplicaciones, se dan en varias generaciones, de acuerdo a los avances tecnológicos de hardware y software:

Periodo	Lenguajes ejemplo
1950-1955	De máquina, ensamblador
1956-1960	Fortran, Algol 58, Algol 60, Cobol, Lisp
1961-1965	Cobol 61, Snobol, Jovial, APL
1966-1970	Fortran 66, Cobol 65, Algol 68, Basic
1971-1975	Pascal, C, Scheme, Prolog
1976-1980	Smalltalk, Ada, Fortran 77, ML
1981-1985	Turbo Pascal, Smalltalk 80, Postscript
1986-1990	C++, Fortran 90
1991-1995	Perl, Ada 95, TCL

Actividad 1.1.2 Completa la tabla mostrada, que llega hasta el periodo en el que nos encontramos, colocando uno o dos lenguajes para cada generación:

Generación	Lenguaje(s)
1996-2000	
2001-2005	
2006-2010	
2011-2015	
2016-2022	

Podrás observar que los lenguajes van surgiendo a la par de los avances de la ingeniería de software, del hardware y de las TIC.

Por ejemplo, en la década de 1960 cuando surge Basic, hace que la programación sea más sencilla y al alcance de más público; por ello, cuando se crean las microcomputadoras en la década de 1970, las cuales contaban con el lenguaje Basic, estudiantes y profesionales de cómputo ya podían desarrollar software en sistemas de cómputo más accesibles.

Actividad 1.1.3 A partir del surgimiento de Internet, qué lenguajes se han creado para desarrollar software y aplicaciones sobre la red.

1.2 Clasificación de los traductores.

Los lenguajes se pueden clasificar de acuerdo a diferentes factores:

- Por su nivel de complejidad, es decir, qué tanto se debe conocer el hardware del equipo donde se empleará. Es así que se tienen:
 - **Lenguajes de bajo nivel.** Ejemplos: lenguaje de máquina, lenguaje ensamblador.
 - **Lenguajes de medio nivel.** Ejemplo: Lenguaje C, ya que tiene algunas operaciones de bajo nivel utilizando instrucciones de alto nivel.
 - **Lenguajes de alto nivel.** Ejemplos: Fortran, Pascal.
 - **Lenguajes de muy alto nivel.** Ejemplos: SQL, Visual Basic
- Por su paradigma de programación, es decir, bajo qué modelo se programa para desarrollar un software. He aquí algunos paradigmas:
 - **Imperativo.** Los programas se componen de un conjunto de sentencias que cambian su estado. Son secuencias de comandos que ordenan acciones a la computadora.
 - **Declarativo.** Opuesto al imperativo. Los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.
 - **Lógico.** El problema se modela con enunciados de lógica de primer orden.
 - **Funcional.** Los programas se componen de funciones, es decir, implementaciones de comportamiento que reciben un conjunto de datos de entrada y devuelven un valor de salida.
 - **Orientado a objetos.** El comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamiento.

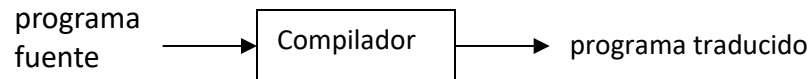
Actividad 1.2.1 ¿Puedes mencionar algún otro paradigma de programación?

Varios estudiosos de los lenguajes consideran que éstos se pueden clasificar en dos paradigmas básicos de programación, que a su vez contienen estilos de programación propios. Además, un lenguaje de programación puede incluir varios paradigmas.



- De acuerdo a cómo se procesan/traducen los lenguajes, es decir, el tipo de procesamiento al que se somete un programa o un archivo fuente para ser traducido y pueda ejecutarse.

- **Lenguaje compilado.** Los programas fuente (escritos en algún lenguaje de programación) pasan por un proceso de traducción para obtener un programa traducido u objetivo y que este último se pueda procesar-traducir nuevamente, o ya ejecutar.



- **Lenguaje interpretado.** Los archivos fuente (escritos en algún lenguaje) pasan por un proceso de traducción y ejecución "simultánea" aparente utilizando las entradas proporcionadas por el usuario.



De la última clasificación de los lenguajes, referida a cómo se procesan/traducen, vamos a revisar un poco más a detalle cómo trabaja cada uno de ellos.

Intérpretes.

Recordemos la representación esquemática de un intérprete.

- Podemos observar que el intérprete va produciendo salidas o resultados conforme va haciendo "simultáneamente" la traducción y ejecución del archivo fuente.
- También vemos que además del archivo fuente, el intérprete recibe una "entrada" que requiere para ejecutarlo. Esta entrada comúnmente son datos.
- En sus orígenes, los intérpretes interrumpían su ejecución al encontrar un primer error ya sea de traducción o de ejecución. En la actualidad, existen intérpretes que se recuperan de errores que no sean graves para continuar su proceso de traducción y ejecución.

Compiladores.

La representación esquemática de un compilador es:

- Observa que tiene exclusivamente como entrada, un **programa** fuente; el cual está escrito en algún **lenguaje de programación**.
- De la traducción del programa fuente, el compilador genera un programa traducido (objetivo o destino), siempre y cuando no encuentre errores graves en la traducción.
- Aunque encuentre errores al realizar la traducción, sigue revisando el programa fuente hasta terminar. Generalmente reporta los errores encontrados.
- Si el programa traducido está escrito en lenguaje de máquina puede pasar a un proceso independiente de ejecución. Por el contrario, el programa traducido deberá pasar por otro proceso de traducción que puede ser por un intérprete o bien, por otro compilador.

Comparativo entre intérpretes y compiladores.

Al diseñar un lenguaje, ¿qué proceso de traducción se debe elegir? Se puede interpretar o compilar, pero se puede preferir un intérprete a un compilador dependiendo del modelo de lenguaje y de la situación en la cual se presenta la traducción.

- Por ejemplo, el lenguaje Basic surgió como interpretado para facilitar el entendimiento de la programación, de hecho fue el seleccionado para trabajar en las primeras microcomputadoras. Este mismo caso se refleja en los lenguajes funcionales como Lisp.
- Si lo que importa es la velocidad de ejecución, se opta por un lenguaje compilado ya que se ejecuta el programa que está traducido, el cual siempre es más rápido que el código fuente interpretado.
- Los intérpretes comparten muchas operaciones con los compiladores, incluso hay traductores híbridos en su proceso de traducción y ejecución que ocupan ambos tipos de traductores. Un claro ejemplo de esto es el lenguaje Java.
- Si notamos, la entrada de un compilador es un programa, mientras que en un intérprete la entrada es un archivo escrito en un lenguaje que no necesariamente es de programación.

Tipos de lenguajes que se traducen.

Posiblemente hasta este momento, has supuesto que las entradas a los traductores son **programas**, es decir archivos que contienen código en algún lenguaje de programación. Sin embargo no siempre es así.

Si observas, en el caso del intérprete, tiene como entrada un **archivo fuente**, el cual no necesariamente contiene código en algún lenguaje de programación, es decir, no es un programa.

¿Entonces, en qué están escritos? Hay otros tipos de lenguajes que al ser traducidos muestran sus resultados de forma simultánea; esto es, son interpretados. Ejemplos claros son los lenguajes de marcado, intérpretes de comandos, entornos de programación y los scripts.

Actividad 1.2.2 Menciona un ejemplo de lenguaje de marcado y otro de script

Por lo que, todos los lenguajes que son interpretados, sean de programación o no, al ser traducidos y ejecutados simultáneamente, se dice que son **archivos ejecutables** aunque no estén escritos en lenguaje de máquina.

Actividad 1.2.3 Elabora un *cuadro comparativo* de las características de los intérpretes vs los compiladores. Puedes ampliar tu cuadro con características no mencionadas en estas notas.

En este curso, como su nombre lo indica, nos enfocaremos al diseño y construcción de compiladores. Sin embargo veremos que varias etapas del proceso de traducción que se realizan en un compilador, se aplican en un intérprete.

La elaboración de un compilador está muy asociado al diseño del nuevo lenguaje, por lo que se debe considerar lo siguiente para que se logre con éxito su construcción.

- Definición de objetivos.
 - Comunicación humana
 - Prevención y detección de errores
 - Usabilidad
 - Programación eficaz
 - Compilable
 - Independiente de la máquina
 - Simplicidad
 - Eficiente
- Filosofías.
 - Uniformidad. Lenguaje predecible.
 - Ortogonalidad. Todas las características del lenguaje se deben poder combinar.
 - Generalización y especialización.
 - Entornos de ejecución. En qué ambientes se ejecutará el código para mostrar resultados.
 - Procesamiento por lotes (batch). Se reemplazan los dispositivos de e/s por archivos.
 - Procesamiento interactivo. Manejo de dispositivos de e/s; por ejemplo: procesadores de texto, hojas de cálculo, intérpretes de consulta y juegos de video, entre otros.
 - Sistemas embebidos. Uso de dispositivos de e/s especiales, no maneja archivos de manera ordinaria. Son críticos si llegan a fallar.
 - Entornos de programación. Incluye herramientas de apoyo como editores, depuradores, verificadores y generadores de datos de prueba.
- Problemas de traducción.
 - Recursividad no controlada
 - Generación de ciclos infinitos
 - Ambigüedad

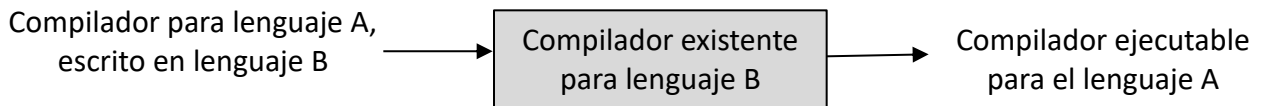
Lenguajes en el desarrollo de los compiladores.

Es claro entonces, que un compilador es un *programa ejecutable* que fue escrito en algún lenguaje de programación y pasó por un proceso de compilación para obtener un programa que traduce un programa fuente a un programa traducido (objetivo o destino).

Por lo tanto en la construcción de compiladores se manejan tres lenguajes (que pueden ser diferentes los tres, sólo dos o todos el mismo):

- Lenguaje con el que está escrito el programa fuente a compilar.
- Lenguaje en el que se escribió el compilador
- Lenguaje en el que el compilador genera el programa traducido.

En la actualidad, cuando se crea un nuevo lenguaje que será compilado, el programa fuente del correspondiente compilador se escribe en un lenguaje para el cual exista un compilador. Si el compilador existente ya se ejecuta en la máquina objetivo, entonces sólo necesitamos compilar el programa fuente del nuevo compilador utilizando el compilador existente para obtener su programa ejecutable:



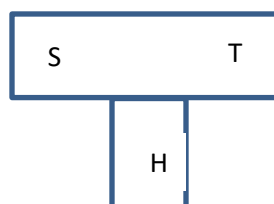
Si el compilador existente para el lenguaje B se ejecuta en una máquina diferente de la máquina objetivo, entonces la compilación produce un compilador cruzado, el cual genera código traducido u objetivo para una máquina diferente en la que pueda ejecutarse.

Actividad 1.2.4 ¿Conoces algún compilador cruzado? ¿Lo has usado? Describe brevemente la funcionalidad de un ejemplo de compilador cruzado.

Diagramas T en la especificación de la construcción de los compiladores.

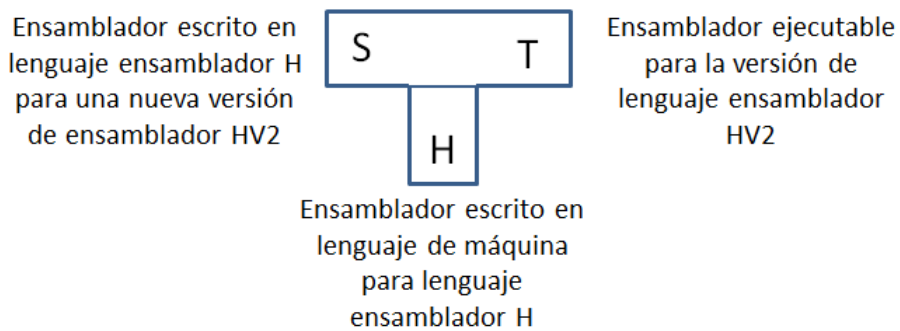
Son usados para especificar, desde un alto nivel, el diseño y construcción de un compilador. Por medio de estos diagramas se pueden observar los pasos por los que pasa el desarrollo de un compilador, desde que se escribe su código fuente, hasta que se obtiene el programa ejecutable del compilador.

Un compilador escrito en el lenguaje H (de host, anfitrión) que traduce lenguaje S (de source, fuente) en lenguaje T (de target, objetivo), se dibuja como el siguiente diagrama T:



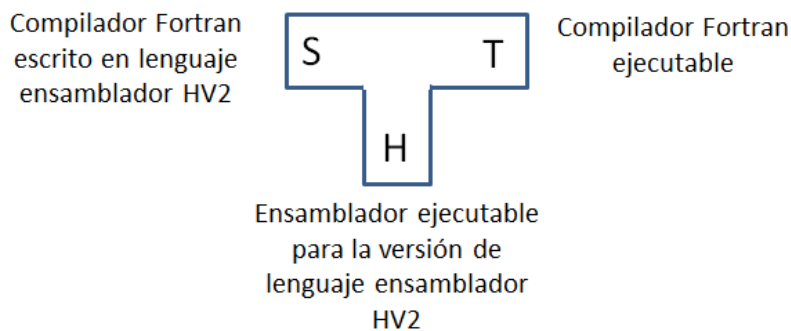
Ahora, utilizaremos los diagramas T para comprender cómo fueron escritos los primeros compiladores.

Como ya hemos visto en el curso, los primeros lenguajes para comunicarse con la computadora fueron los de máquina y los ensambladores. Siendo así, podríamos pensar que el primer ensamblador (programa que traduce un programa fuente escrito en lenguaje ensamblador a lenguaje de máquina) fue escrito en lenguaje de máquina directamente para que fuera ejecutable:



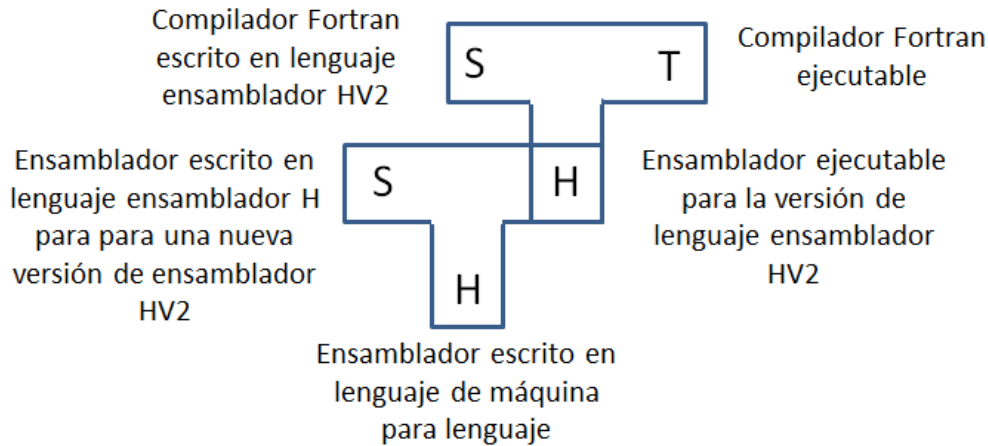
Una vez obtenido un ensamblador con más riqueza en operaciones, los lenguajes de alto nivel fueron diseñados para realizar dichas operaciones con mayor facilidad, viendo los componentes hardware como máquinas virtuales.

Entonces podemos representar con diagrama T qué lenguajes se utilizaron para construir, por ejemplo, Fortran en el año 1957, después de haber necesitado 18 años-persona para su creación.



En 1960, un compilador FORTRAN extendido, ALTAC, estaba también disponible en el Philco 2000, por lo que es probable que un programa FORTRAN fuera compilado para ambas arquitecturas de computadores a mediados de los años 60.

Uniendo los dos diagramas T, aquí elaborados, para ver desde un inicio cómo se llegó a la creación del compilador para el lenguaje Fortran, resulta de la siguiente forma:



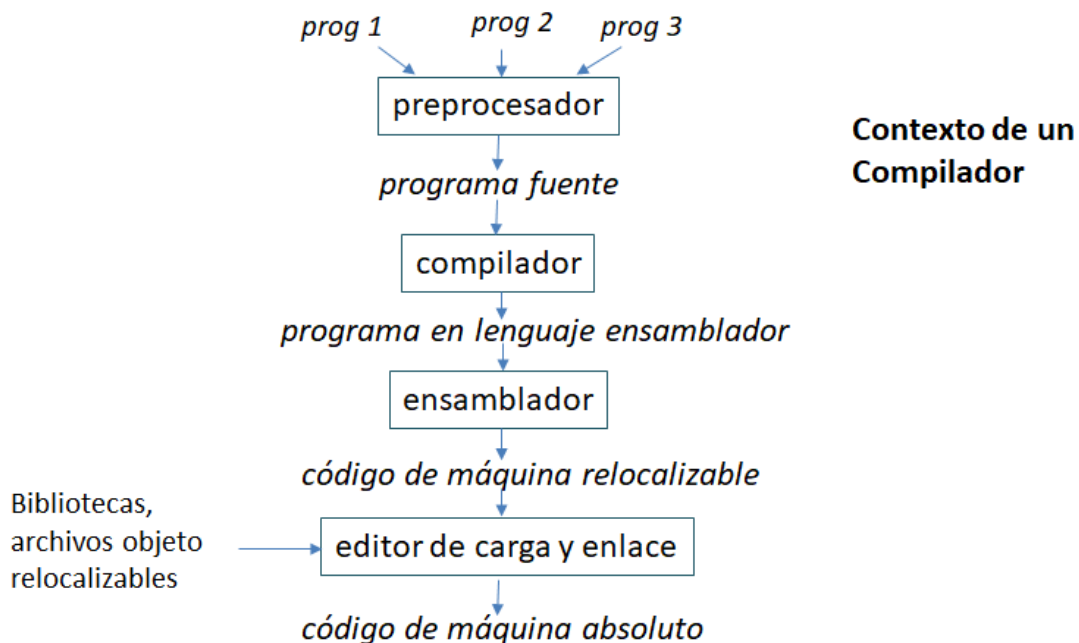
El primer lenguaje de alto nivel multiplataforma demostrado fue COBOL. En una demostración en diciembre de 1960, un programa COBOL fue compilado y ejecutado en el UNIVAC II y el RCA 501.

El compilador COBOL para el UNIVAC II fue probablemente el primero en ser escrito en un lenguaje de alto nivel, llamado FLOW-MATIC, por un equipo dirigido por Grace Hopper.

Actividad 1.2.5 De acuerdo a lo indicado en el párrafo anterior, indaga y representa con diagramas T, cómo fue que finalmente se obtuvo el compilador de COBOL para el UNIVAC II.

1.3 Etapas en el proceso de compilación.

Una vez que ya entendimos, a groso modo, el funcionamiento de un compilador, antes de describir de manera detallada su composición y operación, revisaremos el contexto en el cual trabaja un compilador clásico.



De la figura anterior, observamos que además de un compilador, se requieren de otros programas traductores para crear un programa ejecutable en código de máquina.

Revisemos cada elemento de la figura.

➤ Preprocesador

Un preprocesador es un programa que es invocado por el compilador para conformar un programa fuente para ser compilado. Un preprocesador de este tipo puede eliminar los comentarios, integrar otros archivos y ejecutar sustituciones de macro.

Los preprocesadores pueden ser requeridos por el lenguaje (como en C) o poder ser agregados posteriores para que proporcionen facilidades adicionales (como el preprocesador Ratfor para FORTRAN).

Actividad 1.3.1 Menciona qué directivas del preprocesador de C son las que integran archivos y ejecuta sustituciones macro. De igual manera, qué comando se utiliza en Linux para solamente pre-procesar un programa escrito en C y genere un programa en C sin directivas al preprocesador.

➤ Compilador

En un compilador típico, como el del lenguaje C, generalmente genera código en ensamblador, para no manejar directamente las complejidades operativas de bajo nivel, además de que el lenguaje ensamblador es particularmente más fácil de traducir.

Actividad 1.3.2 Menciona qué comando se utiliza en Linux para que un programa escrito en C genere un programa en ensamblador y se pueda editar o ver.

➤ Ensamblador

El ensamblador es un traductor para el lenguaje ensamblador de un tipo de sistema en particular.

En el contexto que estamos revisando, el ensamblador genera código de máquina relocizable, es decir, que cuyas principales referencias de memoria se hacen relativas a una localidad de arranque indeterminada que puede estar en cualquier sitio de la memoria.

Actividad 1.3.3 Menciona qué comando se utiliza en Linux para que un programa escrito en ensamblador genere un programa ejecutable escrito en código de máquina

➤ Editor de carga y enlace

En realidad, esta etapa es realizada por dos procesos: enlazado o ligado y la carga. Revisemos cada uno.

- *Enlazador o ligador.*

Recopila, en un solo código, programas con código de máquina relocizable, que se encuentran separados en diferentes archivos (incluyen a las bibliotecas).

Estos archivos con código de máquina relocizable, pueden ser las bibliotecas de un lenguaje o archivos objeto relocizables como resultado de haber pasado por un proceso de compilación o de ensamblado.

Los archivos objeto relocizables se forman cuando los programas extensos se diseñan por partes o módulos y se compilan por separado; éstos y las bibliotecas propician la reusabilidad de código.

Actividad 1.3.4 Refiriéndose al lenguaje de programación C, ¿en qué lenguaje están escritos los archivos de cabecera (por ejemplo `stdio.h`, `math.h`)? ¿Por qué se les llama de cabecera (header)? ¿Los puedes editar? Haz la prueba de editar alguno.

Actividad 1.3.5 Las funciones de la biblioteca estándar del lenguaje C, ¿qué lenguaje es el código que contiene? ¿Por qué cuando se usa la biblioteca matemática de C tienes que usar `#include math.h` en el programa fuente y también enlazarlo con `-lm`?

Actividad 1.3.6 Un programa escrito en C que no contenga la función *main*, ¿se puede compilar? ¿Puede obtenerse el ejecutable? ¿Por qué?

- *Cargador.*

El cargador recibe el código unificado del enlazador y convierte todas las direcciones relocizables relativas a una dirección base, o de inicio dada, a código de máquina con direcciones absolutas, donde realmente se almacenará en memoria para su ejecución.

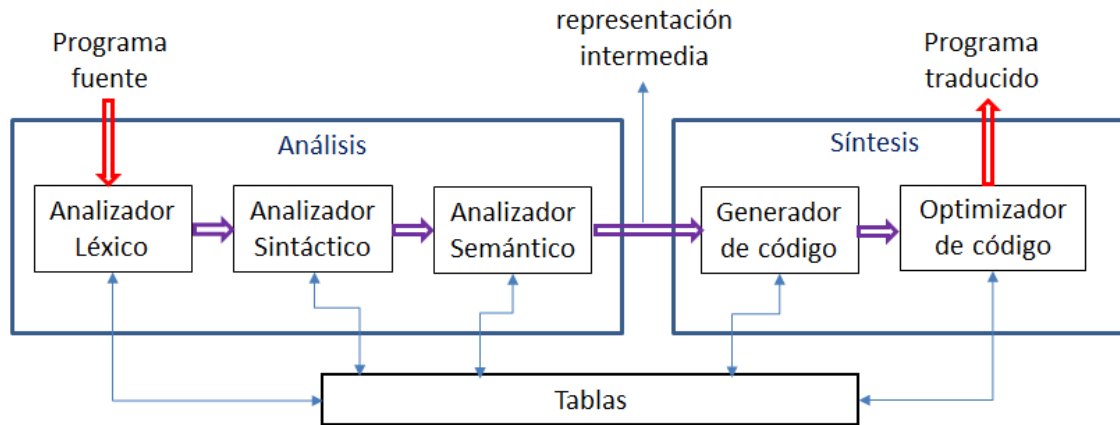
El uso de un cargador hace más flexible el código ejecutable, pero el proceso de carga con frecuencia ocurre en segundo plano (como parte del entorno operacional, o sea lo realiza el sistema operativo) o conjuntamente con el enlazado.

Rara vez un cargador es en realidad un programa por separado.

Actividad 1.3.7 De la revisión del contexto de un compilador clásico, podemos decir que el preprocesador y el ensamblador son traductores de tipo compilador. ¿Puedes sustentar esta aseveración?

Una vez que ya revisamos los procesos en el contexto de un compilador, ahora veremos cómo funciona de manera interna un compilador.

Revisemos la siguiente figura que muestra los componentes en el proceso de compilación.



Componentes en el proceso de Compilación.

Podemos observar que los componentes se agrupan en dos fases:

- **Análisis:** Divide el programa fuente en sus elementos componentes y crea una representación intermedia.
- **Síntesis:** Construcción del programa traducido a partir de la representación intermedia.

Etapas de la Fase de Análisis.

- **Analizador Léxico. También se le nombra Analizador Lineal o Escáner.**

En esta etapa se lee el programa fuente carácter por carácter para formar palabras o cadenas que tienen un significado colectivo. Por cada palabra o cadena identificada, la cual denominaremos **componente léxico**, genera un **token**.

Existen diferentes tipos o clases de componentes léxicos, definidos por el propio lenguaje del programa fuente a compilar; por lo que el analizador léxico deberá conocer esta clasificación para decir si una cadena es identificada como un componente léxico válido.

Por ejemplo, en lenguaje C, las siguientes clases de componentes léxicos son algunas que tiene definidas: palabras reservadas, identificadores, operadores aritméticos, caracteres o símbolos especiales, etc.

Por cada clase de componentes léxicos puede existir una tabla, identificadas en la figura como "Tablas", las cuales se usarán en todo el proceso de compilación.

Veamos un caso práctico para entender, a groso modo, lo que hace el analizador léxico: De la siguiente línea de código en C, mostraré qué componentes léxicos identifica y de qué clase.

area = base * altura/2;



Componente léxico	Clase
area	Identificador
=	Operador de asignación
base	Identificador
*	Operador aritmético
altura	Identificador
/	Operador aritmético
2	Constante entera
;	Símbolo especial

Reconocimiento de los componentes léxicos de una sentencia.

Conforme va identificando un componente léxico, el analizador léxico genera el correspondiente **token** que lo entregará a la siguiente etapa: el Analizador Sintáctico.

Son varias las funciones del analizador léxico, las cuales se estudiarán en el siguiente tema del curso.

El analizador léxico es en sí, un **Autómata Finito Determinístico** que reconoce las cadenas que pertenecen al lenguaje definido por todos los componentes léxicos de un lenguaje (en el que se escribe el programa fuente).

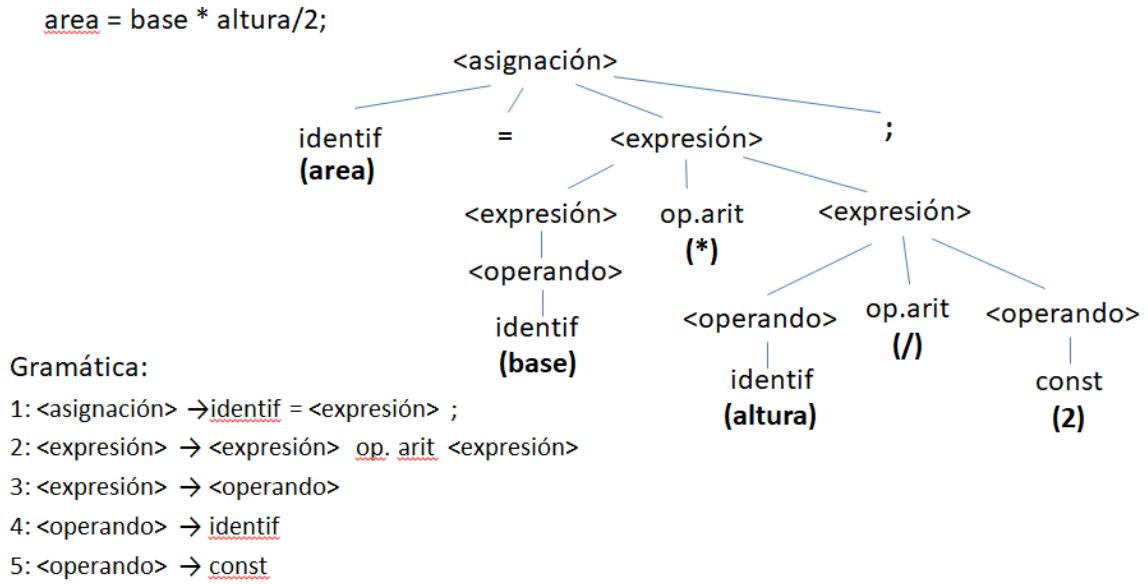
Actividad 1.3.8 Elabora un pequeño programa en C, que al compilar marque un error léxico. Es decir que no reconozca una cadena o un carácter por no pertenecer a ninguna clase de componentes léxicos en C.

- **Analizador Sintáctico. También se le nombra Analizador Jerárquico o Parser.**

Agrupar los componentes léxicos, recibidos del analizador léxico, en frases gramaticales. Recordemos que las frases gramaticales son el resultado de aplicar las reglas o producciones de una gramática, en este caso, libre de contexto.

Estas gramáticas definen la sintaxis del lenguaje y estructura de los programas fuente escritos en dicho lenguaje. Las frases gramaticales del programa fuente, el analizador sintáctico las representa mediante un árbol de análisis sintáctico.

Por ejemplo, de la línea de código analizada en la etapa anterior, construyamos su árbol sintáctico:



Árbol sintáctico y gramática asociada de una sentencia de asignación

El analizador sintáctico es en sí, un **Autómata de Pila o Push Down** que reconoce si la cadena de tokens, recibida del analizador léxico, es una frase gramatical perteneciente al lenguaje definido por la gramática libre de contexto correspondiente.

Actividad 1.3.9 Elabora un pequeño programa en C, que al compilar marque un error sintáctico. Es decir que la estructura del programa o la sintaxis de una instrucción esté mal.

▪ Analizador Semántico.

La semántica de un programa es su “significado”, en oposición a su sintaxis o estructura. La semántica de un programa determina su comportamiento durante el tiempo de ejecución, pero la mayoría de los lenguajes de programación tienen características que se pueden determinar **antes de la ejecución**. A tales características se revisan como **semántica estática** por un **Analizador semántico**.

La semántica “dinámica” de un programa, es decir, aquellas propiedades del programa que solamente se pueden determinar al ejecutarlo, no se pueden determinar mediante un compilador porque éste no ejecuta el programa.

Hagamos un ejercicio para entender qué es una revisión semántica estática versus una revisión semántica dinámica.

Ejercicio 1.3.1 Del siguiente código en C, indica qué línea tiene error semántico estático y cuál error semántico dinámico.

```
void main( )  
{  
    int a=5, b, d;  
  
    d= a/0;  
    scanf("%d",&b); // al solicitar valor de b se le da cero  
    d=a/b;  
}
```

Respuesta:

El analizador semántico detectará que hay error en la línea:

d= a/0;

debido a que el compilador puede detectar que esta operación da un resultado indeterminado. Éste es **error semántico estático**.

Por otro lado, cuando se ejecute el programa, si se le da a la variable 'b' el valor de 0, marcará error semántico en la línea

d=a/b;

éste es **error semántico dinámico**.

Una de las tareas típicas del analizador semántico es revisar, en los lenguajes de programación "tipificados", las declaraciones y verificación de tipos. Recordemos que los lenguajes de programación tipificados son aquellos que antes de utilizar variables se tienen que declarar. En dicha declaración, además, se indica el tipo de valor que puede contener.

Por lo anterior, podemos enumerar qué errores semánticos estáticos relacionados con lenguajes tipificados, reconoce un analizador semántico:

- Variable no declarada.
- Variable declarada dos veces en la misma función o procedimiento.
- Operaciones no válidas sobre variables. Por ejemplo: no cumplir con la aritmética de apuntadores en C, o asignar una cadena a una variable tipo char (en el lenguaje C).

Actividad 1.3.10 Elabora un programa en lenguaje C, que al compilarlo indique qué errores semánticos encontró en relación a la tipificación de variables. Deberás incluir todos los errores indicados en la lista arriba mencionada.

Otra tarea del analizador semántico es aplicar ajustes para realizar operaciones o instrucciones sobre variables.

Por ejemplo, de la línea de código que hemos analizado en las etapas anteriores:

```
area = base * altura/2;
```

Hagamos una revisión de lo que haría el analizador semántico del lenguaje C en cada caso:

- a) Si *area*, *base* y *altura* son declaradas de tipo entero, el resultado será entero porque el analizador semántico revisa que todas las variables de la asignación y la constante sean del mismo tipo. Si *base*altura* fuera impar, entonces, al momento de dividirlo entre 2 el resultado sería un entero y en *area* se asignaría ese valor entero.
- b) Si *base* y *area*, fueran de tipo int, y *altura* fuera de tipo float, el analizador semántico haría un ajuste al valor de *base* como flotante para hacer la multiplicación de *base * altura*; posteriormente, como el resultado anterior es flotante, convierte a la constante 2 a flotante 2.0 para realizar la división con operandos del mismo tipo. Sin embargo como *area* es entera, el valor flotante resultado, se trunca.
- c) Para este caso, invirtamos las operaciones de la asignación analizada para entender más cómo actuaría el analizador semántico:

```
area = altura/2 * base;
```

Recordemos que en el lenguaje C, para realizar las operaciones aritméticas, se basa en la precedencia y asociatividad de operadores; por lo que para la evaluación de la asignación arriba indicada, la secuencia de operaciones a seguir es:

Paso 1: *altura/2* y se guarda en la variable *temp1*

Paso 2: *temp1 * base* y se guarda en la variable *temp2*

Paso 3: *area = temp2*.

Ahora definamos el tipo de variables: *area* y *base* son float y *altura* es int.

Les damos valores: *altura* = 5, *base* = 2.3

Entonces *area* recibirá un valor de 4.6

Actividad 1.3.11 Justifica el valor que recibe la variable *area* del caso c), indicando qué ajustes hace el analizador semántico a los valores de las variables (incluye las temporales) y a la constante (en caso dado) en cada paso.

Es importante hacer notar, que estos ajustes que realiza el analizador semántico a los valores de los operandos (variables y constantes), los realiza para hacer la preparación de la generación del código, por ejemplo, en lenguaje ensamblador. **De ninguna manera son errores semánticos estáticos**; y acerca de los resultados “erróneos”, es responsabilidad del programador revisar si ese resultado es el que desea obtener (a veces desea obtener un valor truncado en una división entre enteros, por ejemplo).

En la revisión semántica, el analizador semántico, consulta las “Tablas” y actualiza algunas de ellas.

Optimización de código en el Análisis Semántico.

Generalmente, los compiladores generan una representación intermedia en el análisis semántico, inclusive haciendo una primera optimización de código, donde puede hacer mejoras del código que dependerá sólo del código fuente.

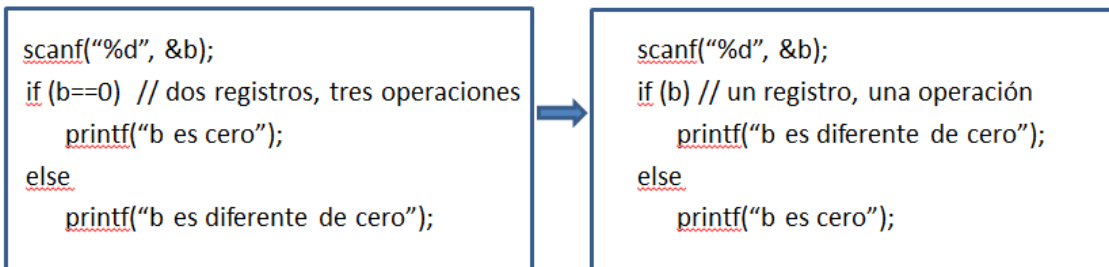
Los compiladores pueden mostrar una amplia variación en los tipos de optimizaciones realizadas en esta etapa. Por ejemplo, en el compilador de C, el analizador semántico optimiza sentencias que reducen el número de operaciones y/o el número de registros que ocuparía el procesador.

Revisemos los siguientes casos:

- a) La sentencia: $a = a + 1$; \longrightarrow ocupa dos registros y dos operaciones
se puede optimizar a: $a += 1$; \longrightarrow ocupa dos registros y una operación
y ésta a su vez, en: $a++$; \longrightarrow ocupa un solo registro y una operación

- b) La sentencia: $valor = 5 + 7$;
Se puede optimizar a: $valor = 12$;

- c) El siguiente código



Actividad 1.3.12 Menciona qué otro caso puede optimizar el analizador semántico del compilador de C

Una representación intermedia de la sentencia:

$area = altura/2 * base$; \longrightarrow $area$, $base$ y $altura$ definidas de tipo float

Pasando por las etapas de análisis léxico y sintáctico, la representación intermedia que genera el analizador semántico pudiera ser:

$temp1 = \text{intofloat}(2)$ \longrightarrow convierte el 2 a 2.0, lo guarda en temp1
 $temp2 = \text{ident1}/temp1$ \longrightarrow divide $altura/2.0$ y lo guarda en temp2
 $temp3 = \text{ident2} * temp2$ \longrightarrow multiplica $(altura/2.0)*base$ y lo guarda en temp3
 $\text{ident3} = temp3$ \longrightarrow finalmente guarda en $area$ el valor de la expresión

Etapas de la Fase de Síntesis


▪ Generador de Código

El generador de código toma la información generada en la Fase de Análisis, que incluye la que se encuentra en las “Tablas” y/o en la representación intermedia, y genera el código traducido u objetivo.

En muchos casos el código traducido es en código de máquina relocable o código ensamblador. (Dato curioso: tanto lex/flex como yacc/bison, que estudiaremos en este curso, son lenguajes de tipo compilador que generan código en lenguaje C, por lo que son ejemplos de lenguajes que no generan código de máquina o de ensamblador).

Si el lenguaje objetivo es código de máquina, se seleccionan registros y localidades de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. En este caso, un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.

Por ejemplo, la traducción en un cierto ensamblador del código intermedio de la sentencia estudiada sería:

Código intermedio		Código en ensamblador
temp1 = <u>intfloat</u> (2)		LDI R1,2 carga en R1 el valor 2
temp2 = ident1/temp1		ITF R1 convierte a flotante R1
temp3 = ident2 * temp2		LDF R2, id1 carga el valor de id1 en R2
ident3 = temp3		DIVF R2, R1, R3 división flotante de R1/R2 en R3
		LDF R4, id2 carga el valor de id2 en R4
		MULF R3, R4, R5 multiplica flotante R3*R4 en R5
		MOV id3, R5 mueve el valor de R5 a id3

Podemos observar que se ocupan 5 registros, del R1 al R5.

▪ Optimizador de Código

En esta etapa, el compilador intenta mejorar el código objetivo generado en la etapa anterior. Las mejoras incluyen la selección de modos de direccionamiento para mejorar el rendimiento, reemplazando las instrucciones lentas por otras rápidas, y eliminando las operaciones redundantes o inmersas.

Del código ensamblador obtenido, del caso de estudio, por el generador de código, podemos realizar las siguientes mejoras:

Código del generador de código

```
LDI R1, 2  
ITF R1  
LDF R2, id1  
DIVF R2, R1, R3  
LDF R4, id2  
MULF R3, R4, R5  
MOV id3, R5
```



Código optimizado

```
LDF R1, id1      carga el valor de id1 en R1  
DIVF R1, #2.0, R2 divide flotante R1/2.0 en R2  
LDF R1, id2      carga valor de id2 en R1  
MULF R2, R1, R3  multiplica R2*R1 en R3  
MOV id3, R3      mueve el valor de R3 a id3
```

Podemos observar que se redujo el número de registros y el número de operaciones.

Administración de las Tablas.

La búsqueda y la inserción rápida en el manejo de tablas es muy importante para la eficiencia del compilador, es por esto que la buena administración de ellas es clave para el adecuado funcionamiento del compilador. Revisemos a groso modo algunas de las tablas empleadas en el proceso de compilación.

- **Tabla de símbolos**

Una función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información sobre distintos atributos de cada identificador.

Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, su tipo, su ámbito (la parte del programa donde tiene validez) y en el caso de nombres de procedimientos, cosas como el número y tipos de argumentos, el método de pasar cada argumento y el tipo que devuelve, si lo hay.

- **Tabla de literales**

La tabla de literales generalmente almacena las cadenas utilizadas en el programa; algunos compiladores también almacenan las constantes. Una tabla de literales necesita impedir las eliminaciones porque sus datos se aplican globalmente al programa y una cadena o constante aparecerá una sola vez en esta tabla.

La tabla de literales es importante en la reducción del tamaño de un programa en la memoria al permitir la reutilización de constantes y cadenas. También es necesaria para que el generador de código construya direcciones simbólicas para literales y para introducir definiciones de datos en el archivo de código objetivo.

- **Catálogos**

Son tablas estáticas que contienen información definida por el lenguaje como palabras reservadas, operadores de relación, operadores de asignación, etc. Estos elementos son los mismos sin importar si se encuentran o no en el programa fuente.

Generalmente se maneja un catálogo por clase de componente léxico fijo. Es decir, una tabla o catálogo de palabras reservadas, otro para operadores relacionales, etc.

Con esto doy por concluido el primer tema del curso de compiladores: Tema 1. Panorama general.

Como se habrán dado cuenta, en los siguientes temas del curso, se estudian a detalle cada uno de los componentes en el proceso de compilación.