Tema 1
Introducción.

Desde el punto de vista de un informático, prácticamente todas las acciones que se va a ver obligado a desarrollar en el transcurso de su carrera profesional, tendrá que ver con traductores: la programación, la creación de ficheros batch, la utilización de un intérprete de comando, etc.

Por ejemplo ¿ Que ocurre si nos dan un documento de Word que procede de una fusión con una base de datos y se quiere, a partir de él, obtener la B.D. original?.
Pues se puede:
a) Convertirla a texto.
b) Procesarla con un traductor para quitar el texto superfluo y dar como resultado un texto en el que cada campo está entre comillas.
c) El texto anterior se importa con cualquier SGBD.

Otros ejemplos para los que necesitaremos utilizar traductores son:

* Conversión del carácter 10 ASCII (LF) en
 de HTML para pasar texto a la web.

* Creación de preprocesadores para lenguajes que no lo tienen . Por ejemplo para trabajar fácilmente con SQL en C, se puede hacer un preprocesador para meter SQL inmerso.

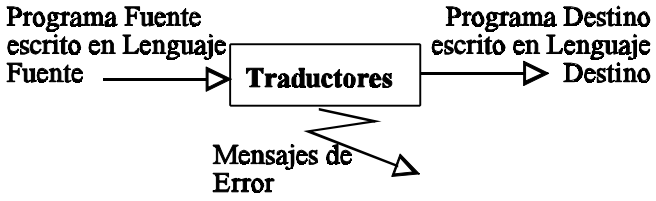
En este capítulo, se introduce el tema de la compilación escribiendo los componentes de un compilador, el entorno en el que trabajan los compiladores y algunas herramientas de software que facilitan la construcción de compiladores.

✯ ¿Qué es un traductor?

Un traductor es un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa escrito en un lenguaje destino produciendo, si cabe, mensajes de error.

* Los traductores engloban tanto al compilador como al intérprete.

* Esquema inicial para un traductor



* Es importante destacar la velocidad en la que hoy en día se hacen. En la década de 1950, se consideró a los traductores como programas notablemente difíciles de escribir. El primer compilador de FORTRAN, por ejemplo, necesitó para su implementación 18 años de trabajo en grupo. Hasta que apareció la teoría de autómatas no se pudo acelerar ni formalizar la creación de traductores.

Tipos de traductores

Desde sus orígenes, ha existido un "hueco" entre la forma de expresarse de las personas y de las máquinas. Los traductores han intentado acortar este hueco para facilitarle las cosas a las personas, lo que ha llevado a aplicar la teoría de autómatas a diferentes campos y áreas concretas de la informática:

- **Traductores del idioma** : Traducen de un idioma dado a otro, por ejemplo, un traductor de Inglés a Español..

  * Problemas:

  - Inteligencia Artificial y problemas de las frases hechas: El problema de la inteligencia artificial es que tiene mucho de artificial y poco de inteligencia. Por ejemplo una vez se tradujo del Ingles al Ruso (por lo de la guerra fría) : "El espíritu es fuerte pero la carne es débil" que, de nuevo, se pasó al Inglés, y dio: "El vino está bueno pero la carne está podrida" (En inglés *spirit* significa tanto espíritu como alcohol). Otro ejemplos de frases hechas son "Piel de pollo", "Piel de gallina"
  - Falta de formalización en la especificación del significado de las palabras.
  - Cambio del sentido de las palabras según el contexto. Ej: "Por decir aquello, se llevó una galleta".
  - Sólo un subconjunto del lenguaje.

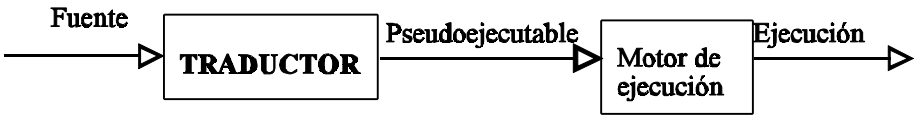
- **Compiladores** : Es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, hace una traducción de alto nivel a código máquina.

- **Intérpretes** : Es como un compilador, solo que la salida es una ejecución. El programa de entrada se interpreta y ejecuta a la vez.

  * Hay lenguajes que solo pueden ser interpretados.
  Ej: SNOBOL (StriNg Oriented SimBOlyc Language),
  LISP (LISt Processing)
  BASIC (Beginner's All ...)

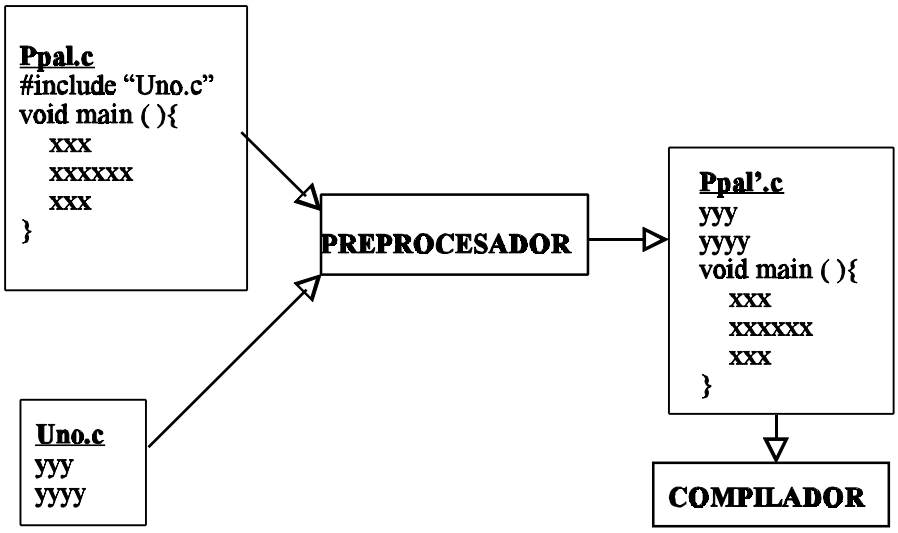
  La principal ventaja es que permiten una fácil depuración. Los inconvenientes son, en primer lugar la lentitud de ejecución , ya que si uno ejecuta a la vez que traduce no puede aplicarse mucha optimización, además si el programa entra en un bucle tiene que interpretar y ejecutar todas las veces que se realice el bucle. Otro inconveniente es que durante la ejecución, es necesario el intérprete en memoria por lo que consumen más recursos.

* Hay lenguajes que son pseudointerpretados.
Son aquellos lenguajes en los que el programa fuente pasa por un pseudocompilador que genera un pseudoejecutable. Este pseudoejecutable lo sometemos a un motor de ejecución. Esto tiene la ventaja de la portabilidad, ya que basta con tener el motor de ejecución en cualquier máquina para poder ejecutar cualquier pseudoejecutable.



Ej: Java, Cobol.
Ventajas, permite trabajar con lenguajes que se pueden modificar a sí mismo. En la etapa de traducción optimiza el programa acercándolo a la máquina.

✯ **Preprocesadores** : Permite modificar el programa fuente antes de la verdadera compilación. Hacen uso de macroinstrucciones y directivas.
Ej:



El preprocesador sustituye la instrucción "#include Uno.c" por el código que tiene "Uno.c", cuando el compilador empieza se encuentra con el código ya incluido en el programa fuente.
Ejemplos de algunas directivas de procesador (Clipper, C): #fi, #ifdef, #define, #ifndef, #define, ... que permiten compilar trozos de códigos opcionales.

Tipos de traductores

✯ **Intérpretes de comandos** : Lo que hace es traducir sentencias simples a llamadas a programas de una biblioteca. Son especialmente utilizados por Sistemas Operativos.
Ej: El shell del DOS o del UNIX. Desencadenan la ejecución de programas que pueden estar residentes en memoria o encontrarse en disco.

Por ejemplo, si ponemos en MS-DOS el comando "copy" se ejecuta la función "copy" del sistema operativo.

✯ **Ensambladores y Macroensambladores** : Son los pioneros de los compiladores, ya que en los albores de la informática, los programas se escribían directamente en código máquina, y los ensambladores establecen una relación biunívoca entre cada instrucción y una palabra mnemotécnica, de manera que el usuario escribe los programas haciendo uso de los mnemotécnicos, y el ensamblador se encarga de traducirlo al código máquina puro.

Obtener un código ejecutable es obtener un código máquina.

- Ensamblador : Es un compilador sencillo, en el que el lenguaje fuente tiene una estructura simple que permite una traducción de una sentencia fuente a una instrucción en código máquina.
El lenguaje que utiliza se llama lenguaje ensamblador y tiene una correspondencia uno a uno entre sus instrucciones y el código máquina.

Ej: Código máquina	65h.00h.01h
Ensamblador	LD HL, #0100

- Macroensamblador : Hay ensambladores que tienen macroinstrucciones que se suelen traducir a varias instrucciones máquinas, pues bien, un macroensamblador es un ensamblador con un preprocesador delante.

✯ **Conversores fuente - fuente** : Pasan un lenguaje de alto nivel a otro lenguaje de alto nivel, para conseguir mayor portabilidad.
Por ejemplo en un ordenador sólo hay un compilador de PASCAL, y queremos ejecutar un programa escrito en COBOL; Un conversor COBOL –> PASCAL nos solucionaría el problema.

* Los resultados pueden requerir retoques por dos motivos:

. Si el lenguaje destino no tiene las mismas características que el origen. Por ejemplo un conversor de JAVA a C, necesitaría retoques ya que C no tiene recolector de basura.

. Si la traducción no ha sido inteligente y el programa destino no es eficiente.

✯ **Compilador cruzado** : Es un compilador que obtiene código para ejecutar en otra máquina. Se utilizan en la fase de desarrollo de nuevos ordenadores.

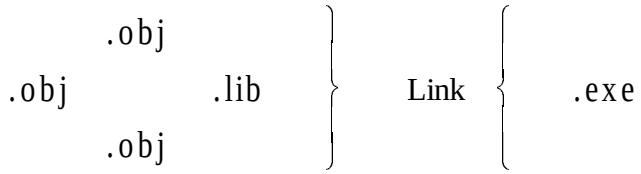
Conceptos

✯ **Compilar-linkar-ejecuta** : Estas son las tres fases básicas de un computador. Nosotros nos centraremos en la primera fase a lo largo de la asignatura.

* El compilador obtiene un código objeto, junto con una tabla de símbolos.

* ¿Porqué no hace directamente un fichero ejecutable?
Para permitir la compilación separada, de manera que puedan fusionarse diversos ficheros OBJ en un solo ejecutable

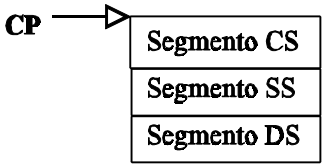
* Un fichero OBJ es un fichero que posee una estructura de registros. Estos registros tienen longitudes diferentes. Unos de estos registros tienen código máquina, otros registros van a tener información. También incluye información sobre los objetos externos. P.ej: Variables que están en otros ficheros declaradas (EXTERN)



* El enlazador resuelve las referencias cruzadas, o externas, que pueden estar o en otros OBJ, o en librerías LIB, y se encarga de generar el ejecutable final.

* Se obtiene un código reubicable, es decir, un código que en su momento se podrá ejecutar en diferentes posiciones de memoria, según la situación de la misma en el momento de la ejecución.

* El fichero ejecutable es un código dividido en segmentos.



Conceptos

* ¿como se hace un programa reubicable?

Cuando se crea el .EXE suponemos que va a empezar desde la dirección 0. Como el programa va a estar dividido en segmentos, las direcciones dentro de cada segmento, no se tratan como absolutas, sino que son direcciones relativas a partir del correspondiente segmento. Por tanto, conociendo la dirección absoluta del inicio de cada segmento es suficiente para poder acceder a su dirección.
Por ejemplo para acceder a la variable x, se hará :

dir absoluta segmento + dir relativa variable = dir absoluta de x

* El cargador, pone el código ejecutable en la memoria disponible, y asigna los registros bases a sus posiciones correctas, de manera que las direcciones relativas funcionen correctamente.

✯ **Pasadas de compilación** : Es el número de veces que se lee el programa fuente. Hay algunas situaciones en las que, para realizar la compilación, no es suficiente con leer el fichero fuente una sola vez. Por ejemplo:

*¿Que ocurre si tenemos una recursión indirecta?

A llama a B
B llama a A

Cuando se lee el cuerpo de A, no se sabe si B va a existir o no, y no se sabe su dirección de comienzo, luego en una pasada posterior hay que rellenar estos datos.

* Para solucionar el problema
1.- Hacer dos pasadas de compilación.
2.- Hacer una sola pasada de compilación utilizando la palabra reservada FORWARD.

```
FORWARD B( )
A( )
```

* Algunos compiladores dan por implícito el FORWARD. Si no encuentra aquello a que se hace referencia, continúan, esperando que el linkador resuelva el problema, o emita el mensaje de error.

* En C se puede hacer una única pasada de compilación, no se pone FORWARD B( ) y C asume una función sin parámetros que devuelve un entero.

Conceptos

✯ **Compilación incremental** : Es aquella que compila un programa en el que si después se descubren errores, en vez de corregir el programa fuente y compilarlo por completo, se compilan solo las modificaciones. Lo ideal es que solo se recompilen aquellas partes que contenían los errores, y que el código generado se reinserte con cuidado en el OBJ generado cuando se encontraron los errores. Sin embargo esto es muy difícil.

* En general se puede hacer a muchos niveles:

Por ejemplo, si se nos olvida un ';' se genera un OBJ transitorio. Si se pone el ';' que falta y se recompila, un compilador incremental puede funcionar a varios niveles

- A nivel de carácter : Se recompila el ';'
- A nivel de sentencia : Si el ';' faltaba en la línea 100 sólo se compila la línea 100.
- A nivel de bloque : Si el ';' faltaba en un procedimiento o bloque solo se compila ese bloque.
- A nivel de Fichero : Si tenemos 15 ficheros y solo se modifica 1(al que le faltaba el ';'), compilo ese fichero y luego se enlazan todos juntos.

* Lo ideal es que se hiciese a nivel de instrucción, pero lo normal es encontrarlo a nivel de fichero.

* El TopSpeed tiene un compilador incremental incluido. A veces en lugar de ponerlo en el compilador se suministra una herramienta externa como RMAKE en la que el programador indica las dependencias entre ficheros, de manera que si se recompila uno, se recompilan todos los que dependan de aquél.

✯ **Autocompilador** : Es un compilador escrito en el mismo lenguaje que compila.

* Cuando se extiende entre muchas máquinas diferentes el uso de un compilador, y éste se desea mejorar, el nuevo compilador se escribe con el antiguo, de manera que pueda ser compilado por todas esas máquinas diferentes, y dé como resultado un compilador más potente de ese mismo lenguaje.

Conceptos

✯ **Metacompilador** : Es un programa que acepta la descripción de un lenguaje y obtiene el compilador de dicho lenguaje, es decir, acepta como entrada una gramática de un lenguaje y genera un autómata que reconoce cualquier sentencia del lenguaje . A este autómata podemos añadirle código para realizar el compilador.

* Por ejemplo LEX y YACC, FLEX, Bison, JavaCC, PCCTS, MEDISE, etc.

* Unos metacompiladores pueden trabajar con gramáticas de contexto libre y otros trabajan con gramática regular. Los que trabajan con gramáticas de contexto libre se dedican a reconocer la sintaxis del lenguaje y los de gramática regular trocean la entrada y la dividen en palabras.

* El PCLEX es un metacompilador cuya función es generar un programa que es la parte del compilador que reconoce las palabras reservadas.

* El PCYACC es un metacompilador cuya función es generar un programa que es la parte del compilador que indica si una sentencia del lenguaje es válida o no.

✯ **Descompilador** : Pasa de un código máquina (o programa de salida) al lenguaje que lo generó (o programa fuente). Cada descompilador trabaja con un lenguaje de alto nivel concreto.

* Es una operación casi imposible, porque al código máquina casi siempre se le aplica una optimización. Por eso lo que hay suelen ser desensambladores, ya que existe una biyección entre cada instrucción máquina y cada instrucción ensamblador.

* Se utilizan especialmente cuando el código máquina ha sido generado con opciones de depuración, y contiene información adicional de ayuda a la depuración de errores (puntos de ruptura, opciones de visualización de variables, etc)
También se emplea cuando el compilador original no generó código máquina puro, sino pseudocódigo (para ejecutarlo a través de un pseudointérprete)

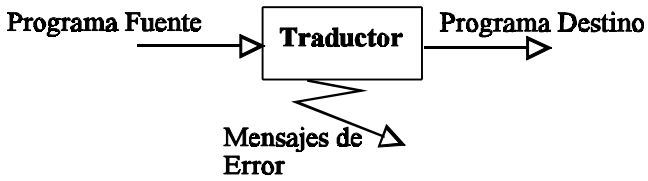
Estructura de un compilador. Fases

✯ Un compilador se divide en dos fases : Una parte que analiza la entrada y genera estructuras intermedias y otra parte que sintetiza la salida. En base a tales estructuras intermedias

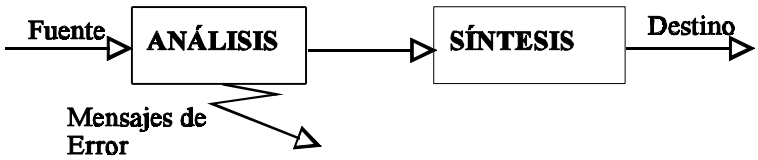
ANÁLISIS
- Lexicografico
- Sintactico
- Semantico

SÍNTESIS
- Generacion de codigo intermedio
- Optimizacion de codigo
- Generacion de codigo maquina

✯ Hasta ahora el esquema de un traductor era:



✯ Depuramos este esquema



✯ Básicamente los objetivos de la fase de Análisis son los enunciados

* Controlar la corrección del programa fuente
* Generar estructuras necesarias para comenzar la síntesis.

Para llevar esto a cabo el Análisis consta de las siguientes tareas:

*** Análisis Lexicográfico** : Divide el programa fuente en los componentes básicos: números, identificadores de usuario (variables, constantes, tipos, nombres de procedimientos,...), palabras reservadas, signos de puntuación. A cada componente le asocia la categoría a la que pertenece.

Estructura de un compilador. Fases

*** Análisis Sintáctico** : Comprueba que la estructura de los componentes básicos sea correcta según ciertas reglas gramaticales.

*** Análisis semántico** : Comprueba todo lo demás posible, es decir ,todo lo relacionado con el significado, chequeo de tipos, rangos de valores, existencia de variables, etc.

* En cualquiera de los tres análisis puede haber errores.

✯ El objetivo de la fase de síntesis consiste en:

* Construir el programa objeto deseado a partir de las estructuras generadas por la fase de análisis. Para ello realiza tres tareas fundamentales.

*** Generación de código intermedio** : Genera un código independiente de la máquina. Ventajas, es fácil hacer pseudocompiladores y además facilita la optimización de código.

* **Generación del código máquina** : Crea un fichero '.exe' directamente o un fichero '.obj'. Aquí también se puede hacer optimización propia del microprocesador.

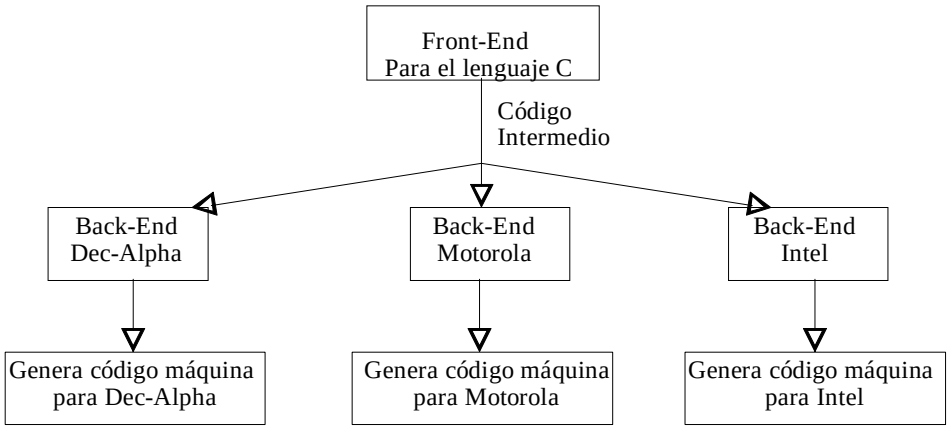
* **Fase de optimización**: La optimización puede realizarse durante las fases de generación de código intermedio y/o generación de código máquina y puede ser una fase aislada de éstas, o estar integrada con ellas.
La optimización del código intermedio debe ser independiente de la máquina.

✯ Con frecuencia, las fases se agrupan en una *etapa inicial (Front-End)* y una *etapa final (Back- End)*. La *etapa inicial* comprende aquellas fases, o partes de fases que dependen principalmente del lenguaje fuente y que son en gran parte independientes de la máquina objeto. Ahí normalmente se introducen los análisis léxicos y sintácticos, la creación de la tabla de símbolos, el análisis semántico y la generación de código intermedio. La etapa inicial también puede hacer cierta optimización de código e incluye además, el manejo de errores correspondiente a cada una de esas fases.
La *etapa final* incluye aquellas partes del compilador que dependen de la máquina objeto y, en general, esas partes no dependen del lenguaje fuente, sino sólo del lenguaje intermedio. En la etapa final, se encuentran aspectos de la fase de optimización de código, además de la generación de código, junto con el manejo de errores necesario y las operaciones con la tabla de símbolos.
Se ha convertido en rutina el tomar la etapa inicial de un compilador y rehacer su etapa final asociada para producir un compilador para el mismo lenguaje fuente en una máquina distinta. También resulta tentador compilar varios lenguajes distintos en el mismo lenguaje intermedio y usar una etapa final común para las distintas etapas iniciales, obteniendose así varios compiladores para una máquina. Veamos ejemplos:

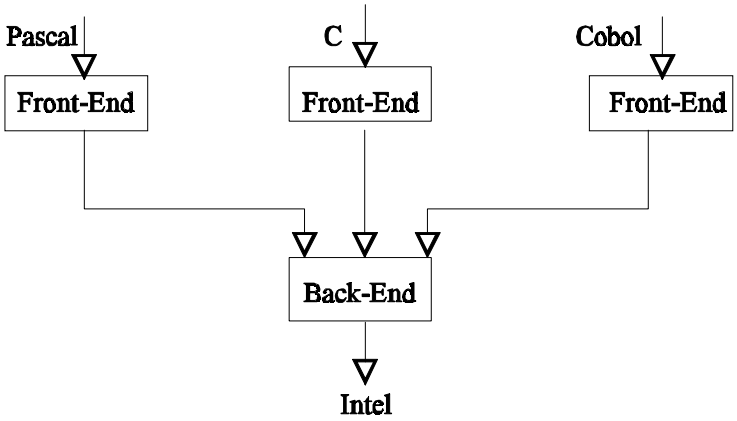
Estructura de un compilador. Fases

* Ejemplo :Suponemos que queremos crear un compilador del lenguaje C para tres máquinas diferentes.

Compilador C para { Dec - Alpha (UNIX) / Motorola (MacOs) / Intel (MS - DOS)

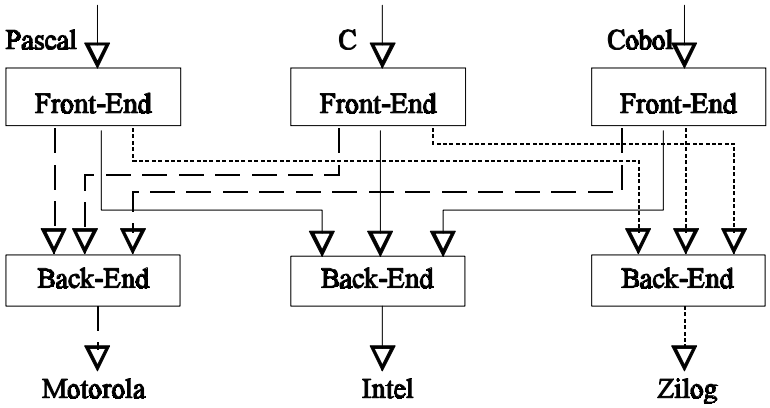


* También podemos crear tres compiladores para la misma máquina.



Estructura de un compilador. Fases

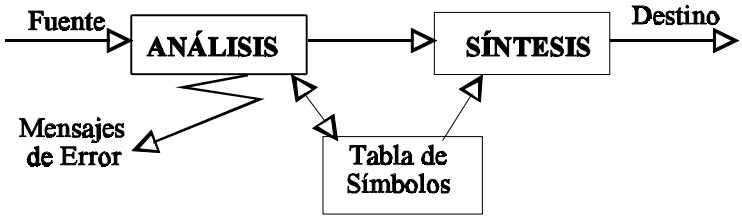
* Si queremos tres compiladores para tres máquinas. (9 compiladores en total)



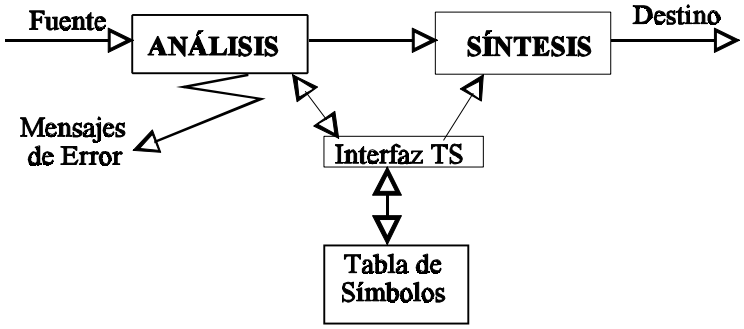
☆ Una función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, su tipo, su ámbito (la parte del programa dende tiende validez),...

Tabla de símbolos : Posee información sobre los identificadores definidos por el usuario, ya sean constantes, variables o tipos. Dado que puede contener información de diversa índole, debe hacerse de forma que no sea uniforme. Hace funciones de diccionario de datos y su estructura puede ser una tabla hash, un árbol binario de búsqueda, etc.

☆ Esquema definitivo de un traductor :



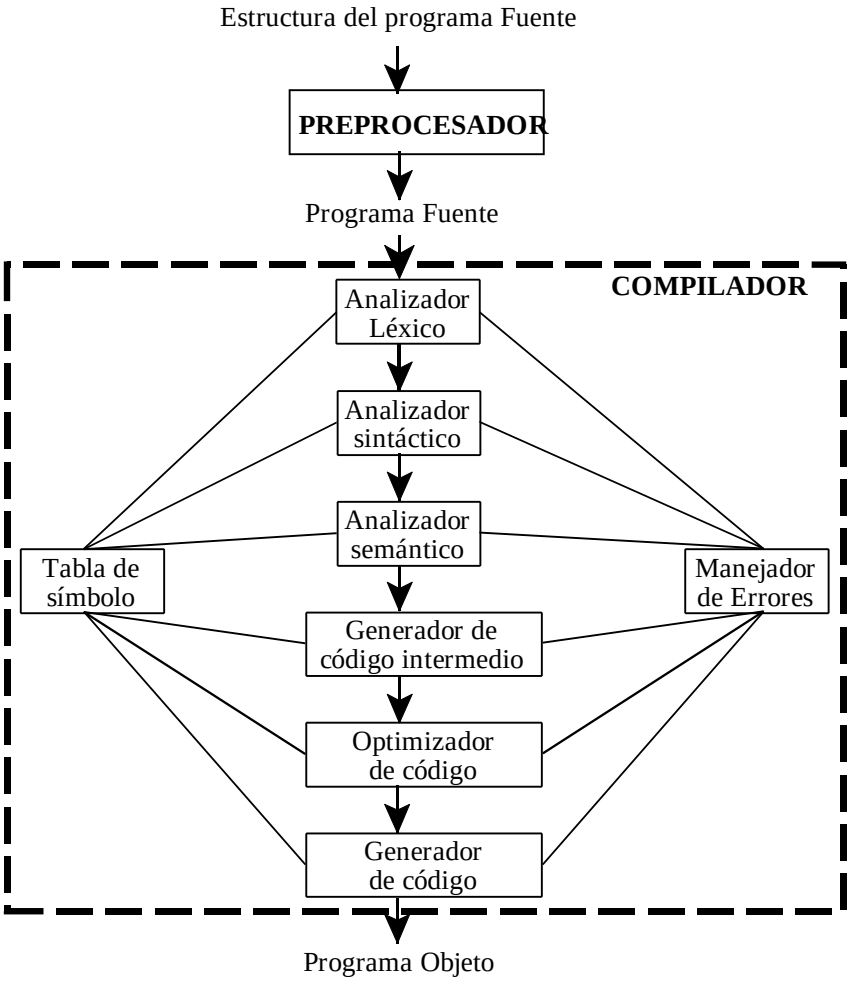
* Esta tabla debida a su complejidad es tratada como un objeto. Como en C no existen los objetos nosotros la trataremos como un TAD.



Ejemplo

El objetivo de este ejemplo es ver una introducción de los procesos que se van a estudiar posteriormente con mayor profundidad. Aunque, como veremos, esto puede hacerse de muchas formas posibles, aquí veremos una de ellas.

Como ya hemos dicho un compilador se divide en dos fases, y cada una de estas fases tiene una serie de tareas fundamentales, las cuales transforma al programa fuente de una representación en otra.



Ejemplo

✯ Suponemos que queremos compilar el siguiente programa.

```
#define TASA 8
descuento = fijo + valor * TASA;
```

* *Preprocesador* : Un programa fuente se puede dividir en módulos almacenados en archivos distintos. La tarea de reunir el programa fuente a menudo se confía a un programa distinto, llamado preprocesador. El preprocesador también puede espandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente. En nuestro ejemplo sustituye la constante TASA por su valor

```
descuento = fijo + valor * 8 ;
```

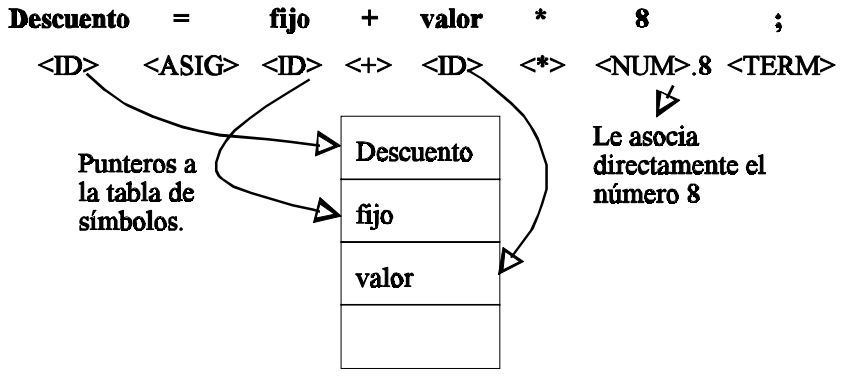
Fase de Análisis

* *Lexicográfico* : Análisis en el que la cadena de caracteres que constituye el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos, que son secuencias de caracteres que tienen un significado colectivo, además trabaja con la tabla de símbolos.
En nuestro ejemplo los caracteres de la proposición de asignación

Descuento = fijo + valor * 8 ;

se agruparían en los componentes léxicos siguientes:

1.- El identificador *Descuento.*
2.- El símbolo de asignación.
3.- El identificador *fijo*.
4.- El signo de suma.
5.- El identificador *valor*.
6.- El signo de multiplicación.
7.- El número 8.
8.- El símbolo terminal ';'.

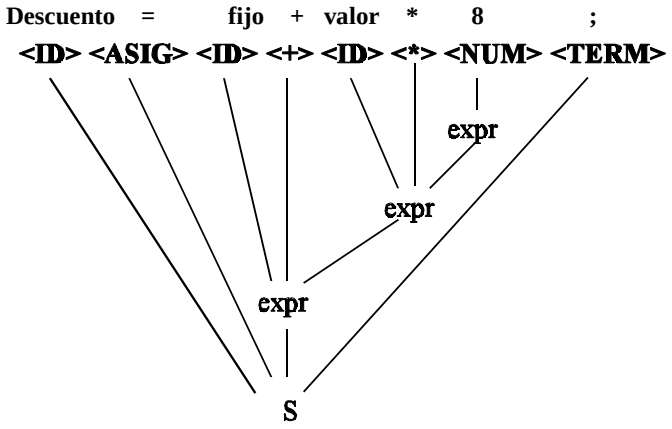


Los espacios en blanco que separan los caracteres de estos componentes léxicos normalmente se eliminan durante el análisis léxico.

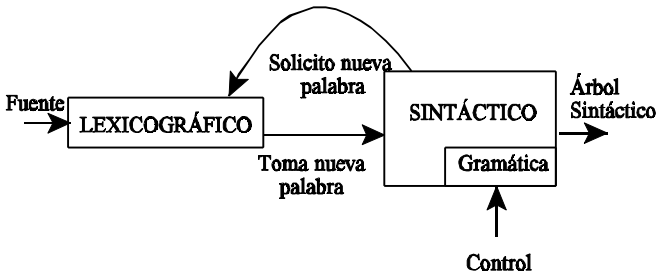
Ejemplo

* *Sintáctico* : Trabaja con una gramática de contexto libre y genera el árbol sintáctico que reconoce esa sentencia

S → <ID> <ASIG> expr <TERM>
expr → <ID>
| <ID> <+> expr
| <ID> <*> expr
| <NUM>



* Las fases se van haciendo según la demanda, es decir no se espera a que acabe una fase para empezar la otra, sino que el sintáctico va generando el árbol conforme el lexicográfico le va proporcionando sentencias .



* Compilación dirigida por sintaxis :El control lo lleva el sintáctico, y todas las demás fases están sometidas a él.

* El árbol sintáctico es la entrada para el análisis semántico.

Ejemplo

* *Semántico* :Esta fase revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza el árbol sintáctico del análisis anterior para identificar los operadores y operandos de expresiones y proposiciones.
Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del leguaje fuente.

Supongamos que nuestro lenguaje solo trabaja con números reales, la salida sería el mismo árbol pero donde teníamos <NUM>.$_8$ pondríamos <NUM>. $_{8.0}$

Ahora pasaríamos a la fase de síntesis.

* *Generación de código intermedio* : Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.
La representación intermedia puede tener diversas formas. Más adelante trabajaremos con una forma intermedia llamada “código de tres direcciones”, que es como el lenguaje ensamblador para una máquina en la que cada posición de memoria puede actuar como un registro. El código de tres direcciones consiste en una secuencia de instrucciones, cada una de las cuales tiene como máximo tres operandos. En nuestro ejemplo el código de tres direcciones sería:

```
t1 =int-to-real(8)
t2 = valor * t1
t3 = fijo + t2
descuento = t3
```

Esta representación intermedia tiene varias propiedades:
- Cada instrucción de tres direcciones tiene a lo sumo un operador, además de la asignación.
- El compilador debe generar un nombre temporal para guardar los valores calculador por cada instrucción.
- Algunas instrucciones de “tres direcciones” tienen memos de tres operandos, por ejemplo, la primera y la última instrucciones del ejemplo

Ejemplo

* *Optimización* : La fase de optimización de código, trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar. Algunas optimizaciones son triviales. Por ejemplo hay una forma mejor de realizar los mismos cálculos que hemos realizado en el código anterior:

```
t2 =  valor * 8.0
descuento = fijo + t2
```

El compilador puede deducir que la conversión de 8 de entero a real se puede hacer de una vez por todas en el momento de la compilación, de modo que la operación int-to-real se puede eliminar. Además ‘t2' se utiliza sólo una vez, para transmitir su valor a ‘descuento’. Entonces resulta seguro sustituir ‘descuento’ por ‘t3', a partir de lo cual se elimina otra de las líneas de código.

* *Generación de código máquina*: La fase final de un compilador es la generación de código objeto, que por lo general consiste en código máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.
Por ejemplo, utilizando los registros R1 y R2, la traducción del código optimizado podría ser:

```
MOVE [1Ah], R1
MULT #8.0, R1
MOVE [15h], R2
ADD R1, R2
MOVE R2, [10h]
```

El primero y segundo operandos de cada instrucción especifican una fuente y un destino, respectivamente. Este código traslada el contendido de la dirección [1Ah] al registro R1, después lo multiplica por la constante real 8.0. La tercera instrucción pasa el contenido de la dirección [15h] al registro R2. La cuarta instrucción le suma el valor previamente calculado en el registro R1. Por último el valor del registro R2 se pasa a la dirección [10h].